Introduction to Cryptography with Maple

José Luis Gómez Pardo

Introduction to Cryptography with Maple

Springer

José Luis Gómez Pardo
Departamento de Alxebra
Universidade de Santiago
Santiago de Compostela
Spain

ISBN 978-3-642-32165-8 ISBN 978-3-642-32166-5 (eBook)
DOI 10.1007/978-3-642-32166-5
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2012944964

ACM Codes: E.3, D.3, G.4

© Springer-Verlag Berlin Heidelberg 2013
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The origins of cryptography are rooted in antiquity but in the last 40 years the subject has experienced explosive growth which has led to deep changes both in its foundations and methodology, and also in the way it is used in the real world, where many new applications have arisen that were not even dreamed of in the middle of the twentieth century.

On the foundational side, we have seen the emergence of key concepts, such as that of one-way function, which now occupies a central role not only in public-key cryptography, where it originated, but also in private-key cryptography. Methodologically, randomness and complexity have surged to the forefront and, with their help, security has been rigorously defined in such a way that makes it possible to obtain reductionist proofs which show that some cryptographic schemes are secure on the assumption that certain mathematical problems are computationally hard. These "security reductions" do not guarantee the security of a scheme in an absolute sense because, in addition to the fact that no computational problems have been proven to be hard so far, the security definitions are formalizations that may not capture all the real-world complexities of the concept and there are other issues that have to be taken into account (implementation issues, side-channel attacks…). However, the reductions put the security of cryptographic schemes on firmer ground since they make explicit the relation with well-studied problems that are widely believed to be hard.

In addition, cryptography is now well beyond its original goal of transmitting information secretly and, to mention just one example, it has been argued that authentication, as provided by digital signatures, is even more important than the secrecy provided by encryption schemes. On top of all this, cryptography is currently being used by almost everyone on a daily basis when, say, connecting to a WiFi network, using a Web browser to make a secure connection, making a payment with a credit card, and so on. This has led to the development of cryptographic schemes that, besides enjoying good security properties, are sufficiently efficient for real-world use.

This book is an introduction to cryptography that, in addition to discussing the relevant theoretical constructions, employs a programing-oriented approach to

study the most important cryptographic schemes in use today, as well as the main cryptanalytic attacks against these schemes. It is based on undergraduate courses in both Mathematics and Computer Science, taught at the University of Santiago de Compostela, where only a fraction of the contents included here was covered. The book is aimed at advanced undergraduate or beginning graduate students, the main prerequisite being a basic knowledge of linear algebra and elementary calculus. In addition, some knowledge of probability and abstract algebra—modular arithmetic, groups, polynomials, fields, etc.—would be helpful but is not required and the necessary material is included in the text.

The focus of this book is on cryptographic algorithms and the cryptographic schemes built from them, as well as on the cryptanalytic algorithms used to attack these schemes, rather than on the higher level interactive protocols that require the cooperation of at least two parties for their execution. We treat the Diffie–Hellman key agreement protocol in detail in Chap. 7, because of its historical and practical importance, and we also occasionally give a brief description of other protocols, but we refer to [65, Chap. 4] for a good introduction to several important classes of interactive protocols, including protocols for user identification, for electronic voting, and for digital cash. More detailed descriptions of some of these and other protocols are given in [172, 134, 189] and in the more theoretically oriented reference [99].

This book includes cryptanalytic algorithms even when the goals of cryptanalysis are, in principle, opposite to those of cryptography because, while the latter tries to design secure schemes, cryptanalysis tries to break them. Despite these conflicting goals, cryptanalysis is one of cryptography's best friends. Checking that a cryptographic scheme resists the known cryptanalytic attacks may serve to increase confidence in its security properties and, even when the scheme has a security reduction to some presumably hard problem, attacks on this problem provide the tools to evaluate security in practical terms and to estimate, for example, which key sizes should be used. In addition to this, new attacks are constantly being developed—and existing attacks are being refined—and the possibility that some of them might not be prevented by the standard security definitions should not be discarded.

A distinctive feature of this book is the inclusion of Maple implementations of most cryptographic schemes—and also of the main cryptanalytic attacks—mentioned in the text. These implementations are not limited to the basic algorithms underlying the schemes, i.e., to the cryptographic primitives, but include all the relevant algorithms that build up each scheme (for example, depending on the kind of scheme, key generation, encryption, decryption, signing, verification) and are able to deal with real messages given either as text strings or files using keys of realistic size. This allows us to include numerous examples which, together with the Maple code, are an integral part of the text and serve to illustrate the theoretical constructions.

Frequently, cryptography books that rightfully warn the reader against the dangers of plain RSA (or "textbook RSA" as it is often called; see Sect. 8.3 for the reasons why it is insecure), when poised to give a concrete RSA example present …

what? … plain RSA, of course, and often with a 4- or 5-digit modulus that, paraphrasing Knuth, would have been factored by Fermat in minutes—if not in seconds—on the back of an envelope.

While appearing somewhat paradoxical, this is fair enough. The example serves to illustrate the weaknesses of the basic construction and may be taken as a starting point from which the reader can imagine what the real construction would look like. Moreover, small-sized examples can be manipulated by hand and, by using them, quite a few lines of text are spared. Even assuming the use of a computer, it would not be realistic to try to include full real-world examples in an introductory textbook, for there are myriads of technical details that would have to be taken into account and that would probably obscure the basic important concepts with the danger that the reader would not see the forest for the trees. In addition, one usually has to deal with some resource limitations; for example, in our setting it is virtually impossible to use Maple to obtain the random seeds required by many schemes.

We feel, however, that there is real added value in giving examples that go beyond the very basic algorithms and that also have realistic size. While they are still simplified models, they come much closer to the look and feel of real-world cryptography and, in our experience, students like them better because they serve to encrypt real messages and, in contrast to the toy examples mentioned, cannot be broken by anyone with access to a computer in a matter of seconds. Also, many crucial details are lost in the examples when one limits them to dealing with cryptographic primitives. For instance, in the case of RSA, we can use an implementation of RSA-OAEP to help appreciate that a secure—in the reductionist sense—version of RSA is much more than the modular exponentiation that is embodied in the RSA primitive.

We must warn the reader that the Maple implementations of cryptographic schemes given here are not intended for standard cryptographic use but, rather, for the purpose of experimenting and learning. In several cases, the value of these implementations stems precisely from the fact that they let us to better appreciate the reasons why some schemes are insecure, but, even when the underlying schemes have a reductionist security proof, there may be some missing ingredients in the Maple version. For example, many of these functions require as input random keys or seeds that cannot be generated within Maple. However, assuming that some additional conditions—like having access to a source of true randomness—hold, these implementations may give a reasonably simple approximation to the real thing within the more restricted Maple environment.

The Maple code and the programming examples try to build a bridge between the world of theoretical cryptography, where security is the main concern, and the world of practitioners and implementers where efficiency is often the focus, sometimes even at the expense of security. We hope that some of these examples will help to show that schemes with good security properties are not much difficult to understand and implement than other less secure schemes.

The Maple constructions given in the text are not only limited to cryptographic schemes but also include number-theoretic algorithms used in many cryptographic

primitives, as well as some of the most important cryptanalytic algorithms. The latter includes, for example, algorithms for factoring integers and for solving discrete logarithms. Since these algorithms are generally much less efficient than those involved in the implementation of the schemes, it is difficult to give examples of realistic size in this case but we try, nonetheless, to present examples that allow the reader to experiment and get a feeling for the behavior of the algorithms in real-world situations. The use of a computer algebra package greatly facilitates the implementation of schemes and algorithms because many of the mathematical and, more specifically, number-theoretic algorithms underlying these constructions are already built-in and immediately available to us. For example, algorithms for modular exponentiation or extraction of modular square roots are extensively used in cryptography and we discuss them in the text but, when they are required to implement other algorithms, we use the versions in Maple's kernel, which are fast and convenient.

The choice of Maple as a programming environment was prompted by the fact that this was the computer algebra system used in the courses which provided the initial motivation for the book but there are several other systems that could be used perfectly well for the same purpose. Among them, we mention the open source computer algebra package Sage (downloadable from [171]) which integrates a set of about 100 open source packages under a common interface and is emerging as a very good alternative to the more established systems. Other computer algebra systems with the capability to develop all these examples are Magma and Mathematica. In principle, an instructor familiar with any of these systems should not have much trouble to translate the Maple functions[1] included in the text into an alternative language. Except for some input–output and conversion routines, as well as some built-in functions which are specific to a given environment, the bulk of the programs rely only on basic programing constructions, such as loops, which have a similar structure in different programing languages. It is clear that having some programing background—as will be the case with Computer Science students and also with most Mathematics or Engineering students—will allow readers to modify the programs included or to write their own programs but we feel that, even in the absence of this background, some minimal guidance will be sufficient to explore the examples and to play and experiment with them. In addition, we refer to the extensive help facilities included in the Maple package, to the online help page at the Maplesoft web site [135] which also contains introductory and advanced programing guides, and to the many books of introduction to Maple, among which we specifically mention [100].

The structure of the book generally proceeds by discussing some cryptographic problem—and, in particular, the all-important security aspects—presenting the algorithms involved and then giving Maple implementations and examples of

[1] The term *function* is used in this book with two different meanings. One of them refers to the standard mathematical concept and the other refers to a *Maple function* and is essentially a synonym for *Maple procedure*, so that no confusion should arise.

these algorithms. Thus, the examples are not relegated to the end but are an integral part of the text flow. Although it is perfectly possible to read the book skipping the programing sections, this is far from ideal and we recommend going through them if at all possible. On the other hand, we have tried to include proofs of all the basic results but, when dealing with some more advanced results, we sometimes give only a proof outline or even, in other cases, a reference to an appropriate source. This is more often the case in relation with theoretical security results because the focus of this book is on algorithms rather than on these results. In all cases, however, we try to present the main ideas on which the proofs are based and to discuss them in context, and in these discussions we include, where appropriate, abundant references to the literature.

The contents of the book may be summarized as follows. After presenting, in the Introduction, the basic ideas that serve to define cryptography and the associated concepts of cryptanalysis and cryptology, we give in Chap. 1 an introduction to classical ciphers that requires little background and serves to introduce the basic cryptographic and cryptanalytic techniques and to appreciate some conditions that an encryption scheme must satisfy to meet even minimal security requirements. Chapter 2 provides the necessary background to read most of the rest of the book, with special attention devoted to modular arithmetic and number-theoretic algorithms that are extensively used throughout.

Chapter 3 gives an introduction to private-key cryptography, discussing perfect secrecy and the one-time pad and introducing one-way functions and pseudorandom number generators, as well as private-key encryption schemes and the important notions of CPA and CCA security. Then, in Chap. 4, block ciphers and their modes of operation are studied, with special attention devoted to AES, the Advanced Encryption Standard, which is implemented in Maple along with some of the more interesting modes of operation. Chapter 5 is devoted to message authentication and discusses several kinds of message authentication codes (MACs) as well as cryptographic hash functions. It includes Maple implementations of CMAC, GCM/GMAC, SHA-256 and HMAC, and also a simulation of the birthday attack.

Chapter 6 is entirely devoted to a study of more advanced number-theoretic algorithms that are relevant for cryptography, including algorithms for primality testing and prime generation, as well as some of the most important algorithms for the integer factorization problem—such as the quadratic sieve, discussed with the help of Maple—and the discrete logarithm problem.

In Chap. 7 we introduce the basic ideas behind public-key cryptography and discuss the Diffie–Hellman protocol and the hard problems underlying it, as well as some cryptanalytic attacks. Chapter 8 is devoted to public-key encryption and, after introducing the appropriate security notions for this setting, some of the most important public-key encryption schemes are studied. They include RSA, Rabin, and Elgamal, and the more secure variants RSA-OAEP, Rabin-SAEP$^+$, and Cramer–Shoup, as well as Paillier encryption. The reductionist security proof of Cramer–Shoup is presented and Maple implementations of all these schemes are given. Chapter 9 is devoted to the study of digital signatures, including RSA and

Elgamal signatures and the Digital Signature Algorithm (DSA), as well as schemes with security reductions, such as the PSS—also implemented in Maple—and Cramer–Shoup signatures. This chapter ends with a discussion of public-key infrastructures and certificates.

In Chap. 10 we introduce the recent subject of identity-based (IB) cryptography, including IB signature schemes and IB encryption (IBE) and give an implementation of the Cocks IBE scheme as well as a description of the Boneh–Franklin IBE scheme. The final chapter is devoted to an introduction to elliptic curve cryptography and includes a discussion of the elliptic curve discrete logarithm problem and related algorithms, as well as a study of elliptic curve-based signatures and encryption. This study is carried out with the help of Maple which is also used to implement the Elliptic Curve Digital Signature Algorithm, ECDSA. Appendix A contains some useful Maple conversion functions which are used throughout the text.

As can be seen from the summary above, we have included recent developments such as secure schemes (relative to some hard problem) and identity-based cryptography and, although we have not included a study of some advanced subjects such as, for example, fully homomorphic encryption or elliptic curve pairings and their cryptographic applications, we feel that someone who has read the book is well equipped to proceed with this study.

Finally, let us note that, as already mentioned, most of the schemes studied as well as many of the algorithms used for cryptanalytic attacks are implemented in Maple, giving the reader the opportunity to experiment with them. The implementations of schemes often rely on the standards contained in NIST's (the US National Institute of Standards and Technology) and other bodies' publications and, in the case of the more recent schemes, in the papers describing them. The Maple code is available from the book's website:

http://sites.google.com/site/cryptomaple

and includes, for each chapter of the book, a Maple language file called ChapterXX.mpl and a Maple worksheet called ChapterXX.mw. The language file contains all the Maple code pertaining to the chapter in question and can be read from the corresponding worksheet which contains all the chapter examples; other examples that will work with this code may be easily generated. All the code included should work in Maple 13 or later and, in fact, most of it will work also in Maple 11 and probably also in earlier versions, the exception being some functions that make use of the element-wise operator '~' (used to distribute an operation over a data structure such as a list or an array). The use of this operator may be avoided by using the commands 'map' or 'zip' instead and this can be an exercise for the reader as has been pointed out at appropriate places in the text. The book Web page also contains a list of errata and the author would be grateful for any comments or corrections, which may be sent to the email address: gomez.pardo@usc.es.

Suggestions for course organization and teaching. This book contains more than enough material for developing several kinds of courses depending on course duration and background requirements and offering different course orientations and flavors. We outline here some of these possibilities:

- A basic introductory course with the minimal background requirements mentioned above and that could have two variants according to course length. For a one-semester course, the material to be covered could be: a brief discussion of classical ciphers and their shortcomings as presented in Chap. 1, skipping some of the cryptanalytic details; a review of asymptotic notation, the basic number-theoretic algorithms (Euclidean algorithm, Euler's theorem, modular exponentiation), and the construction of finite fields, taken from Chap. 2; most of Chap. 3 (skipping some of the more advanced material); the description of AES and the modes of operation from Chap. 4; the basic properties of MACs and the construction of CBC-MAC/CMAC (Sects. 5.1–5.3); hash functions (Sect. 5.6 through Sect. 5.6.3 and Sect. 5.7); the basics of primality testing (Sect. 6.2) and a quick overview of factorization and discrete log algorithms extracted from the first subsections of Sects. 6.4 and 6.5; an introduction to public-key cryptography (Chap. 7) focusing on the DH protocol and the DH problems; the basics of public-key encryption, including the definitions in Sects. 8.1 and 8.2, a discussion of RSA after Sect. 8.3 (through Sect. 8.3.5); and, finally, digital signatures with the material in Sects. 9.1 and 9.2, the first part of Sect. 9.3 including hashed RSA signatures, and Sect. 9.7. For a longer two-semester introductory course, some of the preceding topics would be studied in greater detail and, moreover, the following additional material could be covered: Sects. 5.4, 6.3, 8.3, 8.5, 9.3, 9.4, and 9.6, and a selection of topics from Chap. 11 up to a discussion of the advantages of elliptic curve cryptography.
- A theoretically oriented course whose focus is on the precise definition of security concepts and on the cryptographic schemes which have reductionist security proofs. The precise contents for this course might vary according to the students' background but, in addition to some of the material mentioned for the basic course, could also include Sects. 5.4 and 5.5 on authenticated encryption, Sects. 8.4 and 8.6 on CCA secure encryption schemes, Sect. 8.8 on homomorphic encryption, Sect. 9.5 on CMA secure signatures, and, possibly, an introduction to indentity-based cryptography as presented in Chap. 10. 'Theoretically oriented' here does not mean that practical aspects would not be considered and, in particular, the practical challenges involved in implementing secure schemes could be explored with the help of some of the Maple implementations included.
- A mathematically oriented course which emphasizes number-theoretic ciphers and their cryptanalysis. Starting with the basics from Chap. 2, it would include some material from Chap. 3 (one-way functions and pseudo-random generators) and Chap. 5 (hash functions) and then focus on the study of number-theoretic algorithms (including most of Chap. 6) and their applications to building and

attacking public-key cryptosystems in Chap. 7–9, (specifically, Diffie–Hellman, RSA, Rabin, Elgamal, Goldwasser–Micali and Paillier) as well as a study of elliptic curve cryptography based on Chap. 11.

- A practice-oriented course requiring little mathematical background and with emphasis on applications and, specifically, on the Maple implementations of schemes that are defined in several standards, in particular, AES plus modes of operation from Chap. 4, CMAC, GCM/GMAC, SHA-256 and HMAC from Chap. 5, RSAES-OAEP from Chap. 8, RSASSA-PSS from Chap. 9 and ECDSA from Chap. 11.
- A mathematically advanced course addressed to students with a strong mathematical background, with focus on public-key cryptography and on cryptanalysis. This would include material from Chap. 3 on one-way functions and pseudorandom generators and from Chap. 5 on hash functions, as well as the contents in Chaps. 6–11.

Acknowledgments

Part of this book was written while I was on sabbatical leave from the University of Santiago de Compostela in 2009–2010. I thank the University for granting this leave and, in particular, my colleagues at the Algebra Department for being generous and making it possible, as well as for their support during the preparation of the manuscript.

I also acknowledge the support provided by Project 04555/GERM/06 from Fundación Séneca of the Región de Murcia (partially financed by FEDER funds from the European Union), and by Project INCITE09E2R207117ES from Xunta de Galicia.

I would like to thank José Luis García and Ángel del Río for their genuine interest in the book and for their valuable corrections and suggestions.

I also thank Ronan Nugent and Ulrike Stricker-Komba at Springer for their assistance and support through the publication process.

Finally, I wish to thank my wife, Nieves, and my son, Carlos, for reading parts of the book and offering helpful suggestions, as well as for their support and understanding during the long hours devoted to this project.

Santiago de Compostela, October 2012 José Luis Gómez Pardo

Contents

1 Classical Ciphers and Their Cryptanalysis 1
1.1 The Caesar Cipher 1
1.2 Substitution Ciphers 7
1.3 The Vigenère Cipher. 12
1.3.1 The Vigenère Cipher in Maple. 14
1.3.2 Cryptanalysis of the Vigenère Cipher 16
1.4 The Hill Cipher 25
1.4.1 The Hill Cipher in Maple 27
1.4.2 Cryptanalysis of the Hill Cipher. 29
1.5 Some Conclusions 32

2 Basic Concepts from Probability, Complexity, Algebra and Number Theory. 35
2.1 Basic Probability Theory. 35
2.2 Integers and Divisibility 39
2.2.1 Representation of Integers. 40
2.3 Basic Computational Complexity 42
2.3.1 Asymptotic Notation. 43
2.3.2 Efficient Computation and $\mathcal{P}$ Versus $\mathcal{NP}$ 45
2.3.3 Running Times of Some Simple Algorithms 51
2.3.4 Probabilistic Algorithms 58
2.3.5 Final Remarks on Complexity 60
2.4 The Euclidean Algorithm 61
2.5 Groups, Rings and Fields. 68
2.5.1 Basic Concepts. 68
2.5.2 Congruences and the Residue Class Ring 73
2.6 The Chinese Remainder Theorem. 76
2.6.1 The Chinese Remainder Theorem and the Residue Class Ring. 78

2.7 Euler's Theorem and Modular Exponentiation 80
2.7.1 Euler's Theorem 80
2.7.2 Modular Exponentiation 83
2.7.3 Finding Generators in $\mathbb{Z}_p^*$ 89
2.8 Finite Fields 95
2.8.1 A Field of 4 Elements 95
2.8.2 The Polynomial Ring 99
2.8.3 The Field of p^n Elements 104
2.8.4 The Field of 256 Elements 111
2.8.5 The Multiplicative Group of a Finite Field 113
2.9 Quadratic Residues and Modular Square Roots 115
2.9.1 Quadratic Residues and the Legendre and Jacobi Symbols 115
2.9.2 Computing Modular Square Roots 123

3 Private-Key Encryption 131
3.1 Perfect Secrecy 131
3.2 The One-Time Pad 135
3.3 From Unconditional Security to Computational Security: Pseudo-Random Generators and One-way Functions 139
3.3.1 Pseudo-Random Generators 139
3.3.2 One-Way Functions 144
3.3.3 From One-Way Functions to Pseudo-Random Generators: The Blum–Blum–Shub PRG 148
3.4 PRGs and Related Constructions in Maple 155
3.4.1 The Blum–Blum–Shub PRG in Maple 156
3.4.2 An Approximation to the One-Time Pad in Maple . . . 158
3.4.3 Practical Security Aspects 162
3.5 Private-Key Encryption Schemes and Their Security 167
3.5.1 Private-Key Encryption Schemes 168
3.5.2 Security Definitions for Private-Key Encryption Schemes 169
3.5.3 CPA Security and CCA Security 174
3.5.4 Concluding Remarks 178

4 Block Ciphers and Modes of Operation 181
4.1 Block Ciphers and Pseudo-Random Functions 181
4.2 The Advanced Encryption Standard 186
4.2.1 The Data Encryption Standard 187
4.2.2 Introducing AES 189
4.2.3 AES Decryption 198
4.2.4 Remarks on AES Design and AES Security 200
4.3 Modes of Operation 202
4.3.1 Confidentiality Modes 202
4.3.2 A CPA Secure Encryption Scheme 211

4.4 AES in Maple 216
4.4.1 AES Operations in Maple 216
4.4.2 AES Encryption and Decryption. 219
4.5 Some Modes of Operation in Maple 222
4.5.1 OFB and CTR in Maple 222
4.5.2 Encryption and Decryption with OFB and CTR 226

5 Message Authentication 231
5.1 Confidentiality Versus Authenticity 231
5.2 Message Authentication Codes. 233
5.2.1 Defining MACs 233
5.2.2 Security for MACs 234
5.3 Constructing MACs 236
5.3.1 MACs from Pseudo-Random Functions. 236
5.3.2 CBC-MAC 238
5.3.3 CMAC and Its Maple Implementation. 240
5.4 CCA Security and Authenticated Encryption 244
5.4.1 A CCA Secure Encryption Scheme. 244
5.4.2 Obtaining Authenticated Encryption 246
5.5 MACs Based on Universal Hashing 250
5.5.1 GCM 251
5.5.2 GMAC 256
5.6 Collision Resistant Hash Functions. 258
5.6.1 A Couple of Applications 260
5.6.2 The Merkle–Damgård Construction 261
5.6.3 SHA-256. 263
5.6.4 SHA-256 in Maple 264
5.6.5 MACs with Hash Functions: HMAC. 268
5.7 The Birthday Attack on Hash Functions 271
5.7.1 The Birthday Paradox 271
5.7.2 The Birthday Attack 274

6 Algorithmic Number Theory for Cryptography and Cryptanalysis: Primality, Factoring and Discrete Logarithms. . . . 283
6.1 Large Primes and How to Find Them 283
6.1.1 Searching for Large Random Primes. 284
6.1.2 The Distribution of Prime Numbers 286
6.2 Primality Testing 292
6.2.1 The Fermat Test and Pseudoprimes 292
6.2.2 The Strong Probable Prime Test. 299
6.2.3 The Miller–Rabin Test 304
6.2.4 Other Primality Tests 311
6.3 Generating Random Primes 314
6.3.1 Generating Safe Primes. 317
6.3.2 Generating Pseudo-Random Primes with Maple 319

6.4 The Integer Factorization Problem 320
6.4.1 Trial Division as a Factoring Algorithm 321
6.4.2 Pollard's Rho Method and Its Maple Implementation 323
6.4.3 Fermat's Factorization Method 326
6.4.4 Fermat's Factorization Method in Maple 329
6.4.5 Factor Bases 334
6.4.6 The Factor Base Method in Maple 340
6.4.7 The Quadratic Sieve 353
6.4.8 The Basic QS Algorithm in Maple 358
6.4.9 Some Improvements on the Basic QS 364
6.4.10 The Current Status of Factorization 366
6.5 The Discrete Logarithm Problem 371
6.5.1 The Baby-Step Giant-Step Algorithm and Its Maple Implementation 372
6.5.2 Pollard's Rho Method for Discrete Logarithms 376
6.5.3 The Rho Method for Discrete Logarithms in Maple 379
6.5.4 The Pohlig–Hellman Algorithm and Its Maple Implementation 382
6.5.5 The Index Calculus Method for Discrete Logarithms 385
6.5.6 The Index Calculus Method in Maple 389
6.5.7 Extensions of the Index Calculus Method 395
6.5.8 Final Remarks on the Discrete Logarithm Problem . . . 397

7 Introduction to Public-Key Cryptography: The Diffie–Hellman Protocol 399
7.1 From Private-Key to Public-Key Cryptography 399
7.2 The Diffie–Hellman Key Agreement 403
7.2.1 The DH Protocol and the DH Problems 403
7.2.2 Man-in-the-Middle Attacks 407
7.2.3 Groups for the DH Protocol 408
7.2.4 Attacking the Diffie–Hellman Protocol with Maple 412
7.2.5 Concluding Remarks on the Diffie–Hellman Protocol 416

8 Public-Key Encryption 419
8.1 Public-Key Encryption Schemes 419
8.2 Security for Public-Key Encryption 422
8.2.1 Definitions of Security 423
8.2.2 Hybrid Encryption and Its Security 427

8.3 RSA . . . 429
8.3.1 The RSA Assumption . . . 429
8.3.2 Plain RSA . . . 434
8.3.3 Plain RSA in Maple . . . 440
8.3.4 Security of RSA . . . 447
8.3.5 RSA with Probabilistic Encryption . . . 458
8.3.6 RSA-OAEP . . . 461
8.3.7 RSAES-OAEP in Maple . . . 469
8.4 Rabin Encryption . . . 475
8.4.1 Plain Rabin Encryption . . . 475
8.4.2 Plain Rabin Encryption in Maple . . . 480
8.4.3 CCA Secure Rabin Encryption . . . 483
8.4.4 Rabin-SAEP$^+$ in Maple . . . 489
8.5 The Elgamal Encryption Scheme . . . 495
8.5.1 Security of Elgamal . . . 497
8.5.2 Elgamal on the Group of Quadratic Residues Modulo a Safe Prime . . . 501
8.6 The Cramer–Shoup Encryption Scheme . . . 505
8.6.1 Cramer–Shoup Encryption and Its Security . . . 505
8.6.2 A Variant of the Cramer–Shoup Encryption Scheme in Maple . . . 513
8.7 A Speed Comparison Among Implementations of CCA Secure Schemes . . . 520
8.8 Homomorphic Encryption . . . 523
8.8.1 The Goldwasser–Micali Encryption Scheme . . . 523
8.8.2 The Paillier Encryption Scheme . . . 526
8.8.3 The Paillier Encryption Scheme in Maple . . . 532
8.9 Final Remarks on Public-Key Encryption . . . 533
8.9.1 Fully Homomorphic Encryption . . . 534
8.9.2 Lattice-Based Cryptography . . . 535

9 Digital Signatures . . . 537
9.1 Digital Signature Schemes . . . 538
9.1.1 Definition of Signature Schemes . . . 538
9.1.2 Security of Signature Schemes . . . 539
9.2 Some Early Signature Schemes . . . 541
9.2.1 Plain RSA Signatures . . . 541
9.2.2 Elgamal Signatures . . . 543
9.3 The “Hash-then-Decrypt” Paradigm . . . 544
9.3.1 Hashed RSA Signatures . . . 545
9.3.2 Hashed Elgamal Signatures . . . 546
9.4 The Digital Signature Algorithm . . . 548
9.4.1 The DSA Signature Scheme . . . 549
9.4.2 DSA Security . . . 551
9.4.3 DSA in Maple . . . 553

9.5 CMA Secure Signature Schemes 560
9.5.1 FDH Signatures 560
9.5.2 PSS Signatures 564
9.5.3 RSASSA-PSS from PKCS #1 v2.1 567
9.5.4 RSASSA-PSS in Maple 570
9.5.5 Cramer–Shoup Signatures 576
9.6 Signatures with Added Functionality 578
9.6.1 Blind Signatures 578
9.6.2 Other Signatures with Added Functionality 580
9.7 Public-Key Infrastructures 581
9.7.1 Certificates 581
9.7.2 Multiple Certification Authorities 583

10 Identity-Based Cryptography 587
10.1 Introducing Identity-Based Cryptography 587
10.2 Identity-Based Signatures 590
10.2.1 IBS Schemes 590
10.2.2 From Signature Schemes to IBS Schemes 591
10.3 Identity-Based Encryption 593
10.3.1 IBE Definition 593
10.3.2 Applications of IBE 594
10.3.3 The Cocks IBE Scheme 595
10.3.4 The Cocks IBE Scheme in Maple 599
10.4 The Boneh–Franklin IBE Scheme 605
10.4.1 Pairings 605
10.4.2 The Boneh–Franklin Scheme 607
10.5 Final Remarks on Identity-Based Cryptography 610

11 An Introduction to Elliptic Curve Cryptography 611
11.1 Elliptic Curves and Their Group Structure 613
11.1.1 Definition of Elliptic Curve 613
11.1.2 The Group Structure on an Elliptic Curve 616
11.2 Elliptic Curves Over Finite Fields 624
11.2.1 Some Small Examples 624
11.2.2 Elliptic Curve Elementary Computations 626
11.2.3 The Orders of Elliptic Curve Groups 632
11.2.4 Elliptic Curve Groups Over Prime Fields in Maple . . . 639
11.3 The Elliptic Curve Discrete Logarithm Problem 648
11.3.1 The Rho Method and Pohlig–Hellman for the ECDLP in Maple 649
11.3.2 The Current State of the ECDLP 653
11.3.3 Reduction Attacks Against the ECDLP 655
11.3.4 Final Remarks on the ECDLP 660

11.4 Elliptic Curve Schemes 662
11.4.1 ECC Schemes and Their Domain Parameters 662
11.4.2 The Elliptic Curve Digital Signature Algorithm (ECDSA) 665
11.4.3 ECDSA in Maple 668
11.4.4 Elliptic Curve Encryption 673
11.5 Final Remarks on Elliptic Curve Cryptography 676

Appendix A: Some Maple Conversion Functions 677

References 685

Index 695

Acronyms

AAD	Additional Authenticated Data
AES	Advanced Encryption Standard
AKS	Agrawal–Kayal–Saxena
AMD	Advanced Micro Devices
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
ATM	Automatic Teller Machine
BBS	Blum–Blum–Shub
BDH	Bilinear Diffie–Hellman
BPP	Bounded-error Probabilistic Polynomial time
BSGS	Baby-Step Giant-Step
CA	Certification Authority
CBC	Cipher Block Chaining
CCA	Chosen Ciphertext Attack
CCRA	Computational Composite Residuosity Assumption
CDH	Computational Diffie–Hellman
CFB	Cipher Feedback
CMA	Chosen Message Attack
CMAC	Cipher-based MAC
CPA	Chosen Plaintext Attack
CPP	Cyclotomy Primality Proving
CPU	Central Processing Unit
CRL	Certificate Revocation List
CRT	Chinese Remainder Theorem
CTR	Counter
DCRA	Decisional Composite Residuosity Assumption
DDH	Decisional Diffie–Hellman
DEM	Data Encasulaption Mechanism
DES	Data Encryption Standard
DH	Diffie–Hellman
DL	Discrete Logarithm

DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
EC	Elliptic Curve
ECB	Electronic Codebook
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve DLP
ECDSA	Elliptic Curve DSA
ECIES	Elliptic Curve Integrated Encryption Scheme
ECM	Elliptic Curve Method
ECPP	Elliptic Curve Primality Proving
ECRYPT	European Network of Excellence in Cryptology
EME-OAEP	Encoding Method for Encryption-OAEP
EMSA-PSS	Encoding Method for Signature with Appendix-PSS
EPFL	École Polytechnique Fédérale de Lausanne
ERH	Extended Riemann Hypothesis
FDH	Full Domain Hash
FEDER	Fonds Européen de Développement Régional
FIPS	Federal Information Processing Standard
FNISA	French Network and Information Security Agency
FNMT	Fábrica Nacional de Moneda y Timbre
FR	Frey–Rück
gcd	greatest common divisor
GCHQ	Government Communications Headquarters
GCM	Galois/Counter Mode
GCTR	Galois Counter
GF	Galois Field
GGNFS	GPL'd implementation of the General Number Field Sieve
GHASH	Galois Hash
GMAC	Galois Message Authentication Code
GMP	GNU Multiple-Precision Arithmetic Library
GNFS	General Number Field Sieve
GNU	GNU's not Unix
GPL	General Public License
HMAC	Hash-based Message Authentication Code
IBC	Identity-Based Cryptography
IBE	Identity-Based Encryption
IBM	International Business Machines
IBS	Identity-Based Signatures
IC	Index of Coincidence
ICB	Initial Counter Block
IEC	International Electrotechnical Commission
IFP	Integer Factorization Problem
IND-CCA	Indistinguishable under CCA
IND-CCA1	Indistinguishable under non-adaptive CCA
IND-CCA2	Indistinguishable under adaptive CCA

IND-CPA	Indistinguishable under CPA
IND-EAV	Indistinguishable to Eavesdroppers
INT-CTXT	Ciphertext Integrity
INT-PTXT	Plaintext Integrity
IPSec	Internet Protocol Security
ISO	International Organization for Standardization
IV	Initialization Vector
KDF	Key Derivation Function
KEM	Key Encasulaption Mechanism
lcm	least common multiple
LSB	Least Significant Bit
MAC	Message Authentication Code
MD5	Message Digest, version 5
MGF	Mask Generating Function
MIME	Multipurpose Internet Mail Extensions
MIRACL	Multiprecision Integer and Rational Arithmetic C/C++ Library
MIT	Massachusetts Institute of Technology
MOV	Menezes–Okamoto–Vanstone
MPQS	Multiple-Polynomial Quadratic Sieve
MSB	Most Significant Bit
NBS	National Bureau of Standards
NFS	Number Field Sieve
NIST	National Institute of Standards and Technology
NM	Non-Malleable
NP	Non-deterministic Polynomial time
NSA	National Security Agency
NTRU	Number Theorists R Us
OAEP	Optimal Asymmetric Encryption Padding
OFB	Output Feedback
OMAC	One-key MAC
OTP	One-Time Pad
OW-CCA	One-Way secure against CCA
OW-CPA	One-Way secure against CPA
P	Polynomial time
PGP	Pretty Good Privacy
PIN	Personal Identification Number
PKCS	Public-Key Cryptography Standard
PKG	Private-Key Generator
PKI	Public-Key Infrastructure
PNT	Prime Number Theorem
POTP	Pseudo-Random One-Time Pad
PPT	Probabilistic Polynomial Time
PRF	Pseudo-Random Function
PRP	Pseudo-Random Permutation
PRG	Pseudo-Random Generator

PSS	Probabilistic Signature Scheme
PSW	Pomerance–Selfridge–Wagstaff
QR	Quadratic Residuosity
QS	Quadratic Sieve
RFC	Request for Comments
RKA	Related-Key Attack
RP	Randomized Polynomial time
RREF	Reduced Row Echelon Form
RSA	Rivest–Shamir–Adleman
RSAES-OAEP	RSA Encryption Scheme-OAEP
RSASSA-PSS	RSA Signature Scheme with Appendix-PSS
SAEP	Simple OAEP
SEA	Schoof–Elkies–Atkin
SEC	Standards for Efficient Cryptography
SECG	Standards for Efficient Cryptography Group
SHA	Secure Hash Algorithm
SHS	Secure Hash Standard
SIQS	Self-Initializing Quadratic Sieve
S/MIME	Secure MIME
SS-2-IBS	Standard Signatures to Identity-Based Signatures
SSL	Secure Sockets Layer
SUF-CMA	Strongly Unforgeable under CMA
SVP	Shortest Vector Problem
TLS	Transport Layer Security
UF-CMA	Unforgeable under CMA
UMAC	Universal MAC
XCBC	eXtended CBC
Xor	eXclusive or
XSL	eXtended Sparse Linearization
ZPP	Zero-error Probabilistic Polynomial time

Introduction

The historical origin of cryptography is tied to the necessity of permitting two parties to exchange information in a 'secret' manner. The etymology of the word *'cryptography'*—from the Greek words *kryptós* (hidden, secret) and *gráphein* (writing)—reflects this fact pretty accurately. Thus one could render the meaning of *cryptography* as 'secret writing', as it was indeed done by D. Kahn in his monumental history of the subject [108]. Keeping in line with the original meaning but taking a more modern perspective one could define cryptography as *'the science of secure communication'*. This definition is rather vague and does not capture all the goals of modern cryptography—which will subsequently be explained in detail—but serves as a good first approximation to the subject.

We will take as starting point the basic goal of allowing two parties to communicate through an *insecure channel* (that may even be public), without allowing an adversary, with access to the channel, to understand the information being transmitted. After the publication of the seminal article [164] in which the first system capable of public-key encryption was introduced, it is customary to name the parties exchanging information Alice and Bob (or, briefly, A and B). On the other hand, the adversary is commonly given the name Eve or, sometimes, Oscar or Mallory or a variety of other names.[2] The information that Alice wants to send to Bob is called the *message* or the *plaintext*. To describe a plaintext we will use an *alphabet*, which is simply a nonempty finite set Σ. The elements of the alphabet are usually called *characters*, *symbols*, or *letters* and the plaintext will usually be a *string* (i.e., a finite sequence) of characters. The most common alphabet is the *binary* alphabet $\Sigma = \{0, 1\}$ and other common alphabets are the *hexadecimal* (or *hex*) alphabet $\{0, 1, \ldots 9, a, b, c, d, e, f\}$, the alphabets corresponding to natural languages, or the one given by the 128 symbols of the 7-bit ASCII code.

[2] More specifically, Eve usually refers to a passive adversary, also called an eavesdropper, but we will often use the name to refer to any adversary whatsoever.

In the traditional cryptographic setting, when Alice wants to send Bob some information, she first uses a secret key they both share to encrypt the plaintext by means of a cryptographic algorithm belonging to a scheme whose structure we will describe below. The result of encrypting the plaintext, called the *ciphertext*, is also a string of characters over an alphabet (which need not be the same as the one used for the plaintext although this will often be the case). Bob knows the secret key that Alice used for encryption and uses it to recover the plaintext from the ciphertext. On the other hand, Eve gets hold of the ciphertext by eavesdropping on the channel but, as she does not know the key used, she is (hopefully) unable to recover the plaintext. In this setting, Alice and Bob use a *private-key encryption scheme*, also called a *symmetric encryption scheme*.[3] We will later give a detailed definition but for now we just remark that such a scheme consists of three efficient algorithms (**Gen**, **Enc**, **Dec**) called, respectively, the key-generation algorithm, the encryption algorithm, and the decryption algorithm. The ingredients of the scheme are then the following:

(i) **Gen** outputs a randomly chosen key k of a given length belonging to a finite key space $\mathcal{K}$.
(ii) **Enc** takes as input a key k and a plaintext m in the plaintext space and outputs a ciphertext $c := \mathbf{Enc}(k, m)$ (often also written $c := \mathbf{Enc}_k(m)$). The plaintext space is a finite set $\mathcal{M}$ whose elements are also called messages or *cleartexts*. The set of all ciphertexts obtained from $\mathcal{K}$ and $\mathcal{M}$ is denoted by $\mathcal{C}$.
(iii) **Dec** takes as input a key k and a ciphertext c and outputs a plaintext $m := \mathbf{Dec}(k, c)$ (also written $m := \mathbf{Dec}_k(c)$).
(iv) For the scheme to work as intended it must be *correct*, i.e., it must be the case that, for every key k output by **Gen** and every plaintext m, $\mathbf{Dec}(k, \mathbf{Enc}(k, m)) = m$. We also write $\mathbf{Dec}_k(\mathbf{Enc}_k(m)) = m$, where $\mathbf{Enc}_k$ and $\mathbf{Dec}_k$ are often referred to as the encryption and decryption functions, respectively.

Remarks 0.1. The notation := which we will use for the assignment operation is exactly the same as Maple uses for this purpose. The concept of efficient algorithm will be made precise in Chap. 2 and we will more formally introduce encryption schemes in Definition 3.20. In this definition, **Enc** will be a probabilistic algorithm which may output different ciphertexts when run multiple times with the same plaintext and the same key (denoted $c \leftarrow \mathbf{Enc}(k, m)$) but, until then, we will assume that encryption algorithms are deterministic, as is the case in the classical ciphers studied in Chap. 1.

The encryption scheme is used as follows. Alice and Bob agree on a secret key that they must share by means of a *secure channel* (this is a serious drawback of private-key encryption schemes because secure channels are often not readily available in the real world, but we will see later how this inconvenience can be overcome by using a different kind of encryption scheme). Once the key $k \in \mathcal{K}$ has

[3] Classically, such a scheme is called a *cipher* and another very common name is *cryptosystem* but encryption scheme is more precise as it leaves out cryptosystems not devoted to encryption.

been chosen, Alice takes the plaintext m and computes the ciphertext $c = \mathbf{Enc}_k(m)$ which she then sends to Bob. Upon receiving c, Bob computes $\mathbf{Dec}_k(c) = \mathbf{Dec}_k(\mathbf{Enc}_k(m))$, recovering the plaintext. The scheme is also called symmetric because the encryption and decryption operations are 'symmetric' in the sense that both Alice and Bob use the same key.

As mentioned before, Eve can eavesdrop on the channel and get c but she cannot do the same computation as Bob as she does not know the key. This is only a necessary condition for the system to be secure but is not sufficient in general as Eve might find out ways to obtain information about the plaintext without knowing the key. The security of encryption schemes will be discussed in more detail later on.

Besides confidentiality, there are more requirements that an encryption scheme should meet to be useful and that have not been explicitly stated. For example, there should exist efficient algorithms to compute both the encryption and decryption functions assuming the key is known. On the other hand, one should bear in mind that, although privacy was the primary use of cryptography until very recently, this is by no means the only goal of modern cryptography. Other very important goals are:

1. Authentication. This means that, when Bob receives a message from Alice, he knows that it is really Alice who sent the message.
2. Integrity. Bob obtains assurance that the message sent by Alice has not been altered by a third party.
3. Non-repudiation. After sending a message to Bob, Alice cannot claim that she did not send it.

The importance of these goals is easy to grasp if one considers, for example, the exchange of messages originated by the fact that Alice buys something from Bob using the Internet and paying by credit card. To achieve all these goals it is necessary to have *digital signatures*, which are the electronic analog of written signatures. A symmetric encryption scheme such as the one described above does not provide digital signatures. If Bob wants to convince a third party—for example, a judge—that a certain message came from Alice, he cannot do it in this setting because both Alice and Bob have the same ability to encrypt and decrypt and hence Bob himself might have constructed the ciphertext and attributed it to Alice. However, as we will see later, digital signatures can be obtained by means of appropriate schemes.

Finally, let us stress that, for an encryption scheme to be useful, it needs to be *secure*. The ultimate goal would be that an adversary who obtains the ciphertext should be unable to learn anything about the plaintext except, perhaps, its length, which is usually recoverable. The exact meaning of this requirement is not obvious and giving a good definition of security is one of the goals of modern cryptography which we will examine later. Historically, this concept has been very elusive and, until the middle of the twentieth century, there were no objective criteria to evaluate the security of an encryption scheme. Often, a scheme was thought secure when some time had passed without anyone being able to 'break' it. But history has

shown that this approach is dangerous and, as we will see when we review some symmetric encryption schemes that have been extensively used in the past, it has happened that a certain scheme was broken precisely at the moment when people were more confident that it was secure because no one had yet been able to break it!

As indicated in the Preface, the fundamental methodological tools that have permitted precise security definitions to be given are randomness and complexity. Both concepts are involved when we require that an adversary can only succeed with 'negligible probability' which, as we will see, is a crucial component of all security definitions. Moreover, the use of randomness is essential to attain security because, as we also will see, an encryption scheme must have a randomized (or probabilistic) encryption algorithm to satisfy the most important security definitions. On the other hand, complexity is what allows a precise meaning to be given to the term 'hard problem', another key ingredient to obtain secure cryptographic schemes. The rigorous security definitions will be given in subsequent chapters, starting with Chap. 3, but, until then, we will use a more naive approach in our study of some classical ciphers and of the cryptanalytic attacks against them.

As mentioned before, the security of a symmetric encryption scheme depends on the assumption that Eve does not know which key Alice and Bob are using. For the system to be secure one must also assume that, without knowing the key, Eve is unable to recover (information about) the plaintext from the ciphertext. This would be the main goal of Eve but it is by no means the only possibility: for instance, Eve could try to modify the ciphertext sent by Alice, in order to deceive Bob. In either case, what Eve is doing goes beyond the limits of cryptography and receives the name of *cryptanalysis*. Cryptanalysis can thus be regarded as a discipline which one might say is the opposite of cryptography itself for, while the latter deals with the design of 'good' encryption schemes, the main goal of the former is to 'break' these schemes, either by recovering (total or partial) information about the plaintext or the key, or by other means.

Thus, what is usually studied in cryptography texts such as the present one is not exactly cryptography but a broader discipline called *cryptology*[4] which can be succinctly defined by the following equation:

$$\text{Cryptology} = \text{Cryptography} + \text{Cryptanalysis}$$

The history of cryptology can be seen as a long battle between cryptographers and cryptanalysts and, contrary to what is often the popular belief, cryptanalysts have been victorious most of the time (at least until the arrival of modern cryptography in the last quarter of the twentieth century). This is apparent in Kahn's book [108] and we will give several examples of broken encryption schemes later. There is a basic cryptological principle which is generally embraced by cryptographers:

[4] The term cryptography is, however, often used in this broader sense, as is the case in the title of this book.

Kerckhoffs' Principle. An encryption scheme should be secure even if everything about the system, with the exception of the key, is public knowledge. (A. Kerckhoffs, *La cryptographie militaire*, 1883).

So the cryptographer assumes that the cryptanalyst knows everything about the scheme except the key or, in Claude Shannon's words: *The enemy knows the system*. The philosophy underlying this principle is applicable, beyond cryptology, to security in general and it can be summarized as follows: the fewer the things one has to keep secret, the easier it is to maintain the security of a system (the rationale is that these things can be easily changed if inadvertently disclosed). The opposite philosophy is called *security through obscurity* as it relies also on the secrecy of the design and/or the implementation of the algorithm. The history of cryptography shows that Kerckhoffs' principle is a very sensible one, as it is very difficult to keep the algorithms secret and, more importantly, the advantages of open discussion and examination of the algorithms are widely perceived to far outweigh the hypothetical advantages of keeping them secret.

We will close this introduction by briefly mentioning the different kinds of cryptanalysis. First of all, the cryptanalyst can be passive (an eavesdropper) or active (for example, she may tamper with the ciphertexts in the channel). Both passive and active cryptanalysts may try to recover the plaintext or the key (and hence all plaintexts). But there are other goals that require an active cryptanalyst. For example, Eve can change or corrupt the message and deceive Bob into thinking that the changed message originates with Alice. Or she may impersonate Alice and communicate with Bob while he thinks that he is communicating with Alice. If one bears in mind the resources available to the cryptanalyst, there are four main types of attack that can be carried out:

1. **Ciphertext-only.** In this case, Eve has only ciphertext at her disposal. This requirement is easily met and these are the weakest attacks.
2. **Known plaintext.** Eve knows one or several plaintext-ciphertext pairs and uses this knowledge to try to find the plaintext corresponding to a different ciphertext. This is a very common situation that can happen, for example, if the same plaintext is routinely encrypted every day and Eve knows it. It is often mentioned that a small German army unit in the Sahara desert during World War II sent each day virtually the same message, essentially stating "Nothing to report". This provided a plaintext–ciphertext pair that was enough for the Bletchley Park group in England, devoted to cryptanalyzing the Enigma machine used by the Germans, to find the daily key and decrypt other, more interesting messages.
3. **Chosen plaintext.** Eve can choose several plaintexts and obtain the corresponding ciphertexts or, equivalently, she has temporary access to the encryption machine as a black box (as we will see, this is always the case in public-key encryption). A trivial example that illustrates the advantages of this situation is the following. If Eve knows that there are only two possible plaintexts (for example yes/no) and the encryption algorithm is deterministic, she could encrypt both to find out which one corresponds to the ciphertext she

is trying to decrypt. In general, the situation will not be so favorable for Eve but, using her knowledge of the chosen plaintexts and the corresponding ciphertexts, she can try to recover the plaintext corresponding to another ciphertext or even the key. A historical example of an attack of this type against a private-key encryption scheme occurred during World War II in the Pacific Theater and is mentioned in Sect. 3.5.3.

4. **Chosen ciphertext.** The situation is similar to the preceding one with the roles of plaintext and ciphertext reversed, that is, Eve has temporal access to the decryption machine (as a black box) and so she can find out the plaintexts corresponding to several ciphertexts that she chooses. With this knowledge she tries to decrypt a different ciphertext or even to recover the key. While this situation might seem very unlikely at first sight, we will see in Sect. 8.3.5 that a successful attack of this kind has been devised—although not implemented in practice—against a certain implementation of the RSA public-key encryption scheme. In this attack, some partial information obtained about the result of decrypting the chosen ciphertexts was, in principle, sufficient to recover the plaintext and, in fact, all these types of attacks have variants in which Eve is only assumed to obtain some partial information about ciphertexts or plaintexts.

Chapter 1
Classical Ciphers and Their Cryptanalysis

The object of this chapter is to examine some symmetric encryption schemes that are of historical interest. From this analysis it will turn out that all of them are easy to cryptanalyze. Thus their interest is not because they can be used in practice but, rather, because their cryptanalysis gives us valuable information about some kinds of attacks that have been tried and that should be prevented when designing an encryption scheme. This chapter has an introductory nature and in it we use only very basic mathematical notions, mainly from modular arithmetic, elementary probability and statistics, and linear algebra. Modular arithmetic and probability are reviewed in Chap. 2 in more detail than is required to understand the ciphers presented here, and we refer to that chapter for further background. At the same time, we use Maple to implement and to cryptanalyze these schemes and this also serves as an introduction to the Maple programming environment that we will be using throughout. We start with one of the most ancient encryption schemes known: the Caesar cipher, also known as the shift cipher.

1.1 The Caesar Cipher

The Caesar cipher gets its name from Julius Caesar who, according to Suetonius, used it to communicate with his generals. The Caesar cipher over the 26-letter English alphabet can be described as follows. First, to make the description easier, we identify the alphabet with $\mathbb{Z}_{26} = \{0, 1, \ldots, 25\}$ in the standard way (i.e., the letter 'a' corresponds to 0, 'b' corresponds to 1 and so on, until 'z' which corresponds to 25). Then we define:

- $\mathcal{M} = \mathcal{C} = \mathcal{K} = \mathbb{Z}_{26}$ (hence both the plaintext and the ciphertext space, as well as the key space, are identified with the alphabet).
- $\mathbf{Enc}(k, i) := (i + k) \bmod 26$, for $k, i \in \mathbb{Z}_{26}$.
- $\mathbf{Dec}(k, i) := (i - k) \bmod 26$, for $k, i \in \mathbb{Z}_{26}$.

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_1, © Springer-Verlag Berlin Heidelberg 2013

(here, the notation 'mod 26' means that the remainder of integer division by 26 is taken).

It is then easy to check that $\mathbf{Dec}(k, \mathbf{Enc}(k, i)) = i$ for $i, k \in \mathbb{Z}_{26}$, so that we have indeed defined a symmetric encryption scheme except for one thing: we have not defined the key generation algorithm. In the case of classical ciphers this algorithm was usually left unspecified, which is very dangerous regarding security. An extreme example of this is provided by Caesar himself who, according to Suetonius, used the fixed value $k = 3$ as key. This is not very important in this case because, as shown below, the cipher is extremely easy to break anyway. However, even if the cipher were strong in other aspects, using a fixed key would make it very weak. The sensible way to choose keys is to do it uniformly at random, which means that all keys have the same probability of being chosen. In the following we will assume that, unless otherwise specified, keys are chosen this way.

Observe that we are not restricted to encrypting only one-letter plaintexts because any string of text can be encrypted by concatenating the encryptions of individual letters and the same applies when decrypting a ciphertext. Observe also that, once a key k has been chosen, the Caesar cipher merely encrypts a text string by shifting—hence the name 'shift cipher'—each character k positions to the right, i.e., by replacing it by the one which is located k positions to the right in the alphabet (here a cyclic shift is used, in the sense that when one arrives at the last letter one goes back to the first; this cyclic shifting corresponds to the operation of adding k (modulo 26) to an element of $\mathbb{Z}_{26}$). The decryption function merely undoes this by shifting the letter k positions to the left.

As a first, easy example, suppose that we want to encrypt the word 'caesar' using the Caesar cipher over the English alphabet (with lower case letters only) with $k = 3$. We obtain $\mathbf{Enc}_3(caesar) = fdhvdu$ and, of course, $\mathbf{Dec}_3(fdhvdu) = caesar$ (here we use the shorter form $\mathbf{Enc}_k$ instead of $\mathbf{Enc}(k, -)$ and similarly for decryption).

Example 1.1 Maple has an implementation of the Caesar cipher in the functions `Encode` and `Decode` of the package `StringTools`. We first load `StringTools` using the `with` command, which makes all the functions in the package immediately accessible by just keying in their name. Then, to encrypt a text string over the English alphabet, we use the command `Encode(string, encoding = alpharot[n])`, where `n` is an integer in the 0..25 range[1] (i.e., the key for the Caesar cipher). For example, we have:

```
> with(StringTools):

> Encode("caesar", encoding = alpharot[3]);
                                  "fdhvdu"

> Decode("fdhvdu", encoding = alpharot[3]);
                                  "caesar"
```

There is a variant of the above functions that performs a cyclic shift modulo 256 over the 8-bit ASCII (or ISO Latin 1) code using the format `Encode(string,`

[1] Here and in the sequel we use Maple's notation for an integer range.

`encoding = rot[n])`, where `string` consists of characters with ASCII code in the 1..255 range (so that the *null character* with ASCII code 0 is excluded) and `n` is an integer in the 0..255 range. This variant is impractical because many of the ASCII-8 characters are rarely used in texts and some of them are even non-printable.

The version of the Caesar cipher in the previous example has the inconvenience of not allowing the user to select the alphabet except for the two variants mentioned. We are going to give another simple implementation that can be used with any alphabet obtained as a subset of the characters in the ASCII 1..255 range. At the same time, this will allow us to give a gentle introduction to some of the Maple functions we will be using. This version also takes advantage of Maple's package `StringTools` which contains many utilities for the manipulation of text strings. Since we will often work with messages given as text strings (later we will also use messages given as files), we will make extensive use throughout the book of some of the routines in this package and hence we recommend the reader to look up `StringTools` in Maple's help and examine the functions and the code examples given there.

As before, we first load the package `StringTools`. An alternative way to use the functions in the package is to call them by means of their 'long form calling sequence', which does not require the previous loading of the package. For example, the function `SearchText` below (which searches for the position of a substring within a text string) can be called by inputting `StringTools:-SearchText`.

```
> with(StringTools):
```

Next, we define the alphabet and assign its value to a global variable called `alphabet`:

```
> alphabet := convert([$97..122], bytes);
                          "abcdefghijklmnopqrstuvwxyz"
```

Here we are using the English (lower case) alphabet, but any other alphabet, given by a string consisting of distinct characters in the ASCII code, may be used. The function `convert` is another extremely useful function in Maple's kernel that is used to convert between different data formats and that we will often use, for example, to convert between different number bases. Here we have used another variant of the `convert` function, namely, `convert/bytes`, which converts between strings and lists of integers by mapping characters to their ASCII codes (and conversely) and will also often be used in our examples.

Next we define two mutually inverse bijections between the alphabet and $\mathbb{Z}_n$, where n is the alphabet length (in our example they particularize to bijections between the English alphabet and $\mathbb{Z}_{26}$). The function `code` assigns to each alphabet letter its corresponding residue class—represented as an integer in the 0..25 range—and the function `char` is the inverse of `code`.

```
> code:= letter -> SearchText(letter,alphabet)-1:

  char := i -> alphabet[i+1]:
```

Now, we define a global variable `modulus` to which we assign the alphabet length (26 for the English alphabet) and define a cyclic shift modulo this

length corresponding to some key which, in this case, must be an integer in the 0..25 range:

```
> modulus := Length(alphabet):

  shift := proc(key);
  i -> (i+key) mod modulus;
  end:
```

We are now ready for the encryption and decryption functions corresponding to the Caesar cipher. To encrypt a text string we first convert it into a list formed by elements of $\mathbb{Z}_{26}$, then we shift these elements by using the key and, finally, we transform the resulting list into the corresponding text string, which is the ciphertext. To convert a string into a list of its individual characters we use the function `Explode` from the package `StringTools` and we use its inverse `Implode` to convert back to a string. Another very useful Maple function that we are using here is `map`, which is used to apply a procedure to the operands of an expression and that we often use—as happens here—to apply a function to all the elements of a list. Thus, for example, `map(code,Explode(message))` in the function `CaesarEnc` below, first builds a list with the individual characters of the message and then applies the function `code` to the elements in this list, obtaining a list of elements of $\mathbb{Z}_{26}$ that represents the message. Then we map the function `shift(key)` to this list to obtain another list of elements of $\mathbb{Z}_{26}$ that represents the ciphertext. Finally, we apply the function `char` to this list to obtain the list of the corresponding alphabetic characters and we apply `Implode` to convert this list to a text string which is the ciphertext returned by the function.

As an aside, let us mention that in recent versions of Maple, the use of the function `map` can often be replaced by the use of the element-wise operators obtained by appending the symbol '~' to a function name, which act element-wise on a list (or another data structure such as Array, Vector, Matrix, etc.), just as if one had used `map`. In later chapters we often use this symbol because it allows for shorter code but in the present chapter we stick to using `map`. We advise readers to become familiar with these functions as they are most useful and are used very often in this book. In the following we will be using many Maple built-in functions like these ones without further explanation. We refer to Maple's help for details and also to [100] for an introduction to Maple and its programming language. On the other hand, we think that a good way to explore and to learn how the schemes and algorithms studied here work is to go through their implementations step by step, running their components in order and looking at the intermediate results.

The encryption and decryption functions for the Caesar cipher are then the following, where decryption is obtained from encryption just by replacing the key by its negative value.

```
> CaesarEnc := (key,message) ->
               Implode(map(char,map(shift(key),map(code,Explode(message))))):

  CaesarDec := (key,message) -> CaesarEnc(-key,message):
```

Example 1.2 Suppose that we want to encrypt the sentence "The die is cast" with key $k = 19$. Since our alphabet does not contain spaces or upper case letters (additional

symbols can be easily included, see exercises below and also the implementations of encryption schemes in the following chapters) we will use the string “thedieiscast” instead. We obtain:

```
> c := CaesarEnc(19, "thedieiscast");
                                    c := "maxwbxblvtlm"
```

Now, this ciphertext can be decrypted as follows:

```
> CaesarDec(19, c);
                                   "thedieiscast"
```

Exercise 1.1 Write Maple functions that check that the above defined functions `code` and `char` are mutually inverse.

Exercise 1.2 Modify the alphabet and the above functions implementing the Caesar cipher to allow for upper case letters and punctuation signs in messages.

Cryptanalysis of the Caesar Cipher

An analysis of the different types of cryptanalytic attack gives the following possibilities:

1. *Ciphertext-only.* If Eve only has a ciphertext c, she can mount a *brute-force attack*, consisting of an exhaustive search to determine which of the possible 26 keys is the correct one. This can be accomplished by decrypting c with each key in turn, to find out which one gives a meaningful text. A brute-force attack can always be attempted but, as we shall see in the case of other encryption schemes, it does not always work. For example, the key space may be so large as to make the search infeasible within a reasonable time. In the case of the Caesar cipher this does not happen and, in fact, it is child’s play (literally!) to try all possible keys. Shannon defined, in statistical terms, using the concept of *entropy*, the *unicity distance* of an encryption scheme, which can be informally described as the minimum length of ciphertext (here the length is just the number of characters) for which one expects that there will be only one meaningful plaintext when decrypting it with all possible keys. The theoretical estimates for the unicity distance of the Caesar cipher over the English language suggest that only two letters are needed to recover the key (and hence the plaintext). This number should only be viewed as a lower bound (see Exercise 1.4 below for the case of length 1) but, even if there are some longer English words that have meaningful shifts (such as, for example, ‘river’ and ‘arena’, which are shifts of each other and hence can produce the same Caesar ciphertext with appropriate keys), these words are very few in number. Thus the brute-force attack will most likely recover the key given a ciphertext of, say, length 4, with key recovery being almost certain for ciphertexts of length $\geq$6. Anyway, in contrast with the idea that the ciphertext should not leak any information about the plaintext, not having a uniquely recoverable key is a very weak security property which will not be of much interest to us.

2. *Known plaintext.* In this case it suffices that Eve knows a plaintext of just one letter and its corresponding ciphertext. Indeed, if $\mathbf{Enc}_k(i) = j$, then $k = (j - i) \bmod 26$.
3. *Chosen plaintext.* Eve just chooses 0 as plaintext and computes $k = \mathbf{Enc}_k(0)$.
4. *Chosen ciphertext.* Eve chooses 0 as ciphertext and computes $k = -\mathbf{Dec}_k(0) \bmod 26$.

We see that cryptanalysis of the Caesar cipher is so easy that it can be easily carried out by hand even assuming ciphertext-only. The remaining cases are included just to illustrate the different situations that might arise. As a curiosity, we can mention a recent episode that serves to illustrate the pitfalls of using cryptography while being completely ignorant of the available cryptanalytic techniques. In April 2006 it was reported that Italian police had arrested the 'boss of bosses' of the Sicilian Mafia, Bernardo Provenzano. It appears that one of the things that led to his arrest was the fact that the police found in his hideaway about 350 *pizzini* (cryptograms in Sicilian) which, once deciphered, revealed in some cases the names of his collaborators. These cryptograms were enciphered by means of the *Binnu code* which encrypts the letter A as 4, B as 5 and so on up to Z which goes to 24 and hence is essentially the Caesar cipher with key 3 over a 21-letter alphabet. It is no wonder that the police easily decrypted these cryptograms. But it is also interesting to note a remark attributed to a 'mathematics expert' in the news report. According to this expert, '*In the Provenzano code the key is the +3 shift ... the code might have been made more secure by changing the +3 key with other shift characters (+5, +7, +8) from time to time'*.

Exercise 1.3 Write a brief comment on the 'expert remarks' on the Binnu code.

Exercise 1.4 Suppose that the Caesar cipher, with a randomly chosen key, is used to encrypt a message which is either 'y' or 'n' (we can interpret these messages as 'yes' and 'no'). An adversary knows this and observes the ciphertext 'q'. Does this ciphertext give the adversary any additional information about the plaintext?

Example 1.3 We have just seen that cryptanalyzing the Caesar cipher is an easy task that can be carried out by hand with little effort. Of course, with a computer, the brute-force attack is carried out almost instantly. For example, a Maple function that does it (assuming the same alphabet as before) is the following:

```
> CaesarCryptanalysis :=  ciphertext ->
                          table([seq(CaesarDec(k+1, ciphertext), k=0..modulus-1)]):
```

If we apply this function to the ciphertext previously obtained, we have:

```
> CaesarCryptanalysis(c);
table([1 = "lzwvawakuskl", 2 = "kyvuzvzjtrjk", 3 = "jxutyuyisqij",
5 = "hvsrwswgqogh", 4 = "iwtsxtxhrphi", 7 = "ftqpuqueomef",6 = "gurqvrvfpnfg",
10 = "cqnmrnrbljbc", 11 = "bpmlqmqakiab", 8 = "espotptdnlde", 9 = "dronsoscmkcd",
15 = "xlihmimwgewx", 14 = "ymjinjnxhfxy", 13 = "znkjokoyigyz", 12 = "aolkplpzjhza",
21 = "rfcbgcgqayqr", 20 = "sgdchdhrbzrs", 23 = "pdazeaeoywop", 22 = "qebafbfpzxpq",
16 = "wkhglhlvfdvw", 17 = "vjgfkgkuecuv", 18 = "uifejfjtdbtu", 19 = "thedieiscast",
26 = "maxwbxblvtlm", 24 = "oczydzdnxvno", 25 = "nbyxcycmwumn"])
```

A quick glance at this table shows that the only meaningful text is item number 19, which also reveals the key.

Exercise 1.5 Decrypt the ciphertext `"tnlxptdlhtnzybfpcpo"`, encrypted with the Caesar cipher. If you only had the sub-ciphertext `tdlh` would you still be able to decrypt it?

1.2 Substitution Ciphers

The Caesar cipher is an example of a *substitution cipher*, namely a cipher that acts by replacing each letter in the alphabet by another (possibly the same) one fixed by the key. In this case, the substitutions used (which correspond to certain permutations of the alphabet) are shifts, and the reason for the extreme weakness of the cipher is the fact that the number of shifts, and hence of possible keys, is just the same as the number of symbols (26 in the case of the English alphabet). The fact that this number is very small is what makes the brute-force attack successful against shift ciphers and so an obvious idea to increase security is to keep the substitutions but do not restrict them to being shifts, in order to have a much larger key space. If the substitution cipher acts upon single letters as is usual (there are variants in which what one replaces are not single letters but words or, perhaps, n-grams for a certain fixed value of n) and if the same alphabet is used for both plaintext and ciphertext, then the encryption functions must be permutations of the alphabet. This is because, in order to allow unique decryption, these functions must be injective and, since the alphabet Σ is a finite set, they must necessarily be bijections from Σ to itself or, in other words, permutations of the alphabet. Therefore, the size of the key space (and the difficulty of a brute-force attack) is maximized by taking all the permutations of the alphabet as encryption functions as follows.

Definition 1.1 Let Σ be an n-letter alphabet (in particular, Σ can be the 26-letter English alphabet which can also be identified with $\mathbb{Z}_{26}$ in the standard way). The (most general) substitution cipher over Σ is defined as follows:

- $\mathcal{M} = \mathcal{C} = \Sigma$.
- $\mathcal{K} = S_n$ (where S_n denotes, as usual, the set of permutations of a set of n letters which we may identify with Σ).
- For each $\sigma \in S_n$, $\mathbf{Enc}_\sigma := \sigma$, $\mathbf{Dec}_\sigma := \sigma^{-1}$.

Let us now use Maple to implement this cipher. We will use the lower case English alphabet as before. We will again use Maple's package `StringTools` to manipulate strings, so we start with:

```
> with(StringTools):
  alphabet := convert([$97 .. 122], bytes):
```

We will use the function `randperm` from Maple's combinatorics package `combinat` in order to generate the pseudo-random permutations that will be our keys.[2] A key will be a permutation of $\mathbb{Z}_{26}$ given as a list. We initialize the pseudo-random number generator[3] by means of the function `randomize` and then we generate a permutation:

```
> Randomize():
  k := combinat:-randperm(26);
[9, 5, 20, 12, 10, 8, 24, 1, 26, 15, 18, 17, 23, 19, 16, 13,
 25, 11, 21, 4, 3, 7, 22, 2, 6, 14]
```

Note that the permutation is given as a list containing the integers from 1 to 26 instead of the elements of $\mathbb{Z}_{26}$, namely the integers 0 to 25 which have been identified with the alphabet symbols. Thus it is a permutation of the set $\{1, 2, \ldots, 26\}$ (mapping 1 to 9, 2 to 5, and so on) rather than of $\mathbb{Z}_{26}$. We could use the earlier representation and regard this list as a permutation of $\mathbb{Z}_{26}$ by just subtracting one from each element in the list but this will be not necessary because we will use the command `Permute` from the `StringTools` package that works with the current representation. The encryption function is now given by:

```
> SubstitutionEnc :=
  (key, message) -> CharacterMap(alphabet, Permute(alphabet, key), message):
```

As an example, let us encrypt a message with the previously generated key `k`:

```
> c := SubstitutionEnc(k, "sellthesharesbeforetomorrow");
                          "ujqqdajuaikjuejhpkjdpwpkkpv"
```

For the decryption function, we use the inverse of the permutation corresponding to the key. This inverse is computed by the command `invperm` from the `group` package, which we call here by using the 'long form' `group:-invperm`. In order to apply this function, the permutation must be in 'disjoint cycle notation' and we use the `convert/disjcyc` function to obtain this format (see Maple's help for details):

```
> SubstitutionDec := (key, ciphertext) -> SubstitutionEnc(
    convert(group:-invperm(convert(key, 'disjcyc')), 'permlist', 26), ciphertext):
```

Now, we can decrypt the ciphertext:

---

[2] The precise meaning of the term *pseudo-random* is defined in Chap. 3; for now we can informally think of pseudo-random as something that, while possibly being non-random, looks random to a computationally bounded adversary.

[3] Pseudo-random number generators should always be initialized—or seeded—before using them because, otherwise, the sequence of generated values will always be the same.

```
> SubstitutionDec(k, c);
                          "sellthesharesbeforetomorrow"
```

The Caesar cipher is a particular case of the substitution cipher. It can be recovered as follows (where the function `sshift` below takes care of converting a Caesar key to a permutation given as a list of integers in the 1..26 range):

```
> sshift := k -> map(i->(i+k mod 26)+1,[$0..25]):
  SCaesar := (key, message) ->
              CharacterMap(alphabet, Permute(alphabet, sshift(key)), message):

  SCaesar(19, "thedieiscast");
                                  "maxwbxblvtlm"

  SCaesar(-19, %);
                                  "thedieiscast"
```

(the symbol % is the 'ditto operator' in Maple and takes the value of the previous output).

Cryptanalysis of the Substitution Cipher.

Let us now examine the security of the substitution cipher. What we have gained by replacing the shift cipher by a general substitution cipher is that now $|\mathcal{K}| = n!$ is much larger than n. When Σ is the English alphabet, the number of possible keys is $26! > 4 \cdot 10^{26}$ so trying all possible keys is definitely out of the question for now although it may become feasible in a not-too-distant future.

To get a more precise idea about the size of this task one can have a look at one of the largest successful brute-force attacks so far, the one against the 'Data Encryption Standard' (DES)[4] carried out in 1999 in response to the 'DES Challenge III' (see Sect. 4.2.1). This attack was carried out by a custom-built machine called *Deep Crack* which contained about 1500 processors, working cooperatively with many other computers coordinated through a web site called *distributed.net*. The search rate was 245 billion keys per second and the plaintext was recovered in just over 22 h. We can see that, if the same computing power were applied to a brute-force search of a substitution cipher over the English alphabet and if one assumes that the key is found after searching one half of the key space (as would be expected on average), then the required time would be more than $2.6 \cdot 10^7$ years. Even taking into account the increase of computing power since 1999, breaking the cipher this way is not yet feasible as of this writing.

Exercise 1.6 Assuming that computational power doubles every two years as a consequence of *Moore's law* ([145]) and the use of parallel computing, and taking as reference the 1999 DES cracking effort, compute the year in which one could expect the brute-force attack on the substitution cipher to succeed within one month.

In addition to the previous remarks, one should also bear in mind that the computational cost of breaking a substitution cipher by exhaustive search grows steeply (or,

[4] DES is a private-key encryption scheme that is briefly discussed in Sect. 4.2.1.

to be more precise, exponentially, a term whose meaning will be defined later on) as one increases the size of the alphabet. The amount of work is increased by a factor of n when one passes from $n-1$ to n letters and hence the mere addition of one letter to the English alphabet (as would happen, for example, if one uses the Spanish alphabet which contains the additional character $\tilde{n}$) would multiply the required time by a factor of 27.

Although a general substitution cipher cannot be currently broken by a brute-force attack, it would be very naive to conclude that this makes these ciphers secure. The fact that it appears to be so to the layman has been exploited many times in the literature. Perhaps the best known literary example of cryptanalysis of a substitution cipher occurs in Edgar Alan Poe's tale *The Gold Bug*, where his main character, Will Legrand, successfully recovers a plaintext containing the description of the place where a pirate's treasure is hidden. Poe underscores the triumph of the cryptanalyst by putting the following words in Legrand's mouth: ... *it may well be doubted whether human ingenuity can construct an enigma of the kind which human ingenuity may not, by proper application, resolve.*

Another prime literary example of cryptanalysis occurs in a tale by Sir Arthur Conan Doyle entitled *The Adventure of the Dancing Men*, where the protagonist, none other than Sherlock Holmes himself, solves a murder mystery by breaking a substitution cipher, having at his disposal only very little ciphertext. Yet another example, this time not fictional but something that really happened, can be found in the memoirs of a well-known eighteenth century personage and is quoted in [108]. The story is about an encrypted manuscript that contained the formula to transmute base metals into gold, which had been given to the writer by a Madame d'Urfé who believed no one would be able to decrypt it. But our personage not only broke the substitution cipher—which was more difficult since it was a polyalphabetic substitution, see Sect. 1.3—to recover the plaintext but also found the key (which happened to be 'Nabucodonosor') and told it to Madame d'Urfé. She was deeply surprised as she was the only person that knew the word and she had never written it down. Then the cryptanalyst proceeded to tell her that a genie had revealed the key to him. He then goes on to write *'This false disclosure fettered Madame d'Urfé to me. That day I became the master of her soul, and I abused my power'*. It appears very unlikely that the writer could make gold because of this episode but he certainly took advantage of it to seduce Madame d'Urfé, which is not at all surprising bearing in mind that he was none other than the famous Venetian adventurer and womanizer Giacomo Casanova.

So, what is the fact that makes substitution ciphers insecure? It is the inherent redundancy of all natural languages, as a consequence of which the alphabetic characters appear with different frequencies, which can be computed by analyzing a sufficient quantity of text. For example, an analysis of English text shows that the three most frequent letters are, in decreasing order, e, t, and a, with approximate relative frequencies of 0.127, 0.091 and 0.082, respectively. On the opposite extreme of the frequency spectrum we find q, x, and z, all of them with frequency close to 0.001.

Of course, the frequency distribution of characters depends on the language itself and, for example, the three most frequent letters in Spanish are *e*, *a*, and *o*, with frequencies about 0.137, 0.125 and 0.087, respectively. Moreover, natural languages also have characteristic frequency distributions for *n*-grams (which are very uneven), and, in particular, those corresponding to digrams and trigrams can be easily computed by counting their occurrences in a sufficient amount of text. Now, these frequency distributions are preserved when the text is encrypted by means of a substitution cipher. Since each occurrence of a given character in the plaintext is always encrypted in the same manner, the frequency distribution of characters in the ciphertext will be the same as the one in the plaintext, albeit with the frequencies corresponding to different letters. Thus, a substitution cipher can be cryptanalyzed, assuming ciphertext-only, by comparing the frequency distribution of characters in the ciphertext with the known frequency distribution of characters in the plaintext language (which is known by Eve according to Kerckhoffs' principle). This is a trial-and-error process because the frequency distribution of characters in the plaintext need not be exactly the same as the expected one for the language in question and the process will be easier the longer the ciphertext Eve has at her disposal.

There are extreme cases, such as the book *Gadsby* by Wright [199], which does not contain the letter *e*. But even this *ad hoc* rarity would not prevent a cryptanalyst from deciphering an encrypted version of this book, because the distribution of the remaining letters will be close to the expected one. Summing up, the cryptanalysis of the simple substitution cipher is very easy, as the literary examples given by Poe and Conan Doyle show. Of course, the attack is even easier if Eve has known plaintext (or chosen plaintext or chosen ciphertext) as then it is only a matter of comparing the letters in the plaintext with the corresponding letters in the ciphertext, which will reveal (portions of) the key.

We will now sketch an example of cryptanalysis of a substitution cipher. Suppose we are given the following ciphertext which we know (Kerckhoffs' principle!) was encrypted with a substitution cipher over the English alphabet. Here we will write the ciphertext in upper case while we assume the plaintext is in lower case, which is convenient to handle partially decrypted ciphertexts. Note that spaces and punctuation symbols were not encrypted.

```
c := "Z MPLAO XIDG FPAO XGW FXG FWLFX,FXIF FXG BIYG MIAMLAIFZPU SXZMX XIO BGWDGO YG
KPW OGMZEXGWZUJ FXG YIULBMWZEF XIO GUINAGO YG FP AGIWU FXG SPWO, NLF PU I MIEWZMG
ZF BFWLMR YG FP FGAA XGW FXIF I JGUZG XIO WGDGIAGO ZF FP YG. FXZB KIABG OZBMAPBLWG
KGFFGWGO YIOIYG OLWKG FP YG. FXIF OIV Z NGMIYG FXG YIBFGW PK XGW BPLA, IUO Z INLBGO
YV EPSGW. GDGWV FZYG Z FXZUR PK ZF, Z IY OZBFWGBBGO IUO IBXIYGO, IUO Z OP EGUIUMG
UPS ZU FXG PNAZJIFZPU LUOGW SXZMX Z EAIMG YVBGAK PK FGAAZUJ FXG FWLFX ZU SWZFZUJ YV
YGYPZWB.":
```

The first step to cryptanalyze this would be to do a frequency analysis of the ciphertext. This is easy with Maple, as the function `CharacterFrequencies` of the package `StringTools` (which we assume is already loaded) does precisely this.

```
> freq := CharacterFrequencies(c);
freq := " " = 97, "," = 5, "." = 4, "A" = 17, "B" = 17, "D" = 4, "E" = 6, "F" = 40,
  "G" = 56, "I" = 33, "J" = 5, "K" = 8, "L" = 12, "M" = 14, "N" = 5, "O" = 24, "P" =
  21, "R" = 2, "S" = 6, "U" = 20, "V" = 5, "W" = 24, "X" = 27, "Y" = 19, "Z" = 31
```

To make the task easier, we sort these frequencies in decreasing order:

```
> sort([freq], (u, v) -> evalb(rhs(u) >= rhs(v)));
 [" " = 97, "G" = 56, "F" = 40, "I" = 33, "Z" = 31, "X" = 27, "O" = 24, "W" = 24,
   "P" = 21, "U" = 20, "Y" = 19, "A" = 17, "B" = 17, "M" = 14, "L" = 12, "K" = 8,
   "E" = 6, "S" = 6, "," = 5, "J" = 5, "N" = 5, "V" = 5, "." = 4, "D" = 4, "R" = 2]
```

Having these frequencies at our disposal we can compare them with the known frequency distribution of the English language. This distribution is available from many sources (e.g., from [130]; see next section for a list of these frequencies) and can also be approximated by using a sufficiently long English text. We see that the three most frequent characters in the above ciphertext are, in decreasing order, `G`, `F`, and `I`. We can start by replacing these characters by the three most frequent characters in English texts, which are `e`, `t`, and `a`. We use Maple's function `CharacterMap`, from the `StringTools` package for this purpose and we obtain:

```
> CharacterMap("GFI", "eta", c);
"Z MPLAO XaDe tPAO XeW tXe tWLtX,tXat tXe BaYe MaAMLAatZPU SXZMX XaO BeWDeO Ye KPW
  OeMZEXeWZUJ tXe YaULBMWZEt XaO eUaNAeO Ye tP AeaWU tXe SPWO, NLt PU a MaEWZMe Zt
  BtWLMR Ye tP teAA XeW tXat a JeUZe XaO WeDeaAeO Zt tP Ye. tXZB KaABe OZBMAPBLWe
  KetteWeO YaOaYe OLWKe tP Ye. tXat OaV Z NeMaYe tXe YaBteW PK XeW BPLA, aUO Z
  aNLBeO YV EPSeW. eDeWV tZYe Z tXZUR PK Zt, Z aY OZBtWeBBeO aUO aBXaYeO, aUO Z OP
  EeUaUMe UPS ZU tXe PNAZJatZPU LUOeW SXZMX Z EAaMe YVBeAK PK teAAZUJ tXe tWLtX ZU
  SWZtZUJ YV YeYPZWB."
```

We have to bear in mind that it is possible that the three most frequent letters in the plaintext are not `e`, `t`, and `a` and, as we have mentioned, this is a trial-and-error process. However, once some letters are fitted into their correct place, the task becomes easier. In this case we see that, for example, the trigram `tXe` appears several times, from which it is easy to conjecture that the ciphertext character `X` corresponds to the plaintext character `h`. Other short words are also easy to guess. In this particular case the task is very easy—we leave it as an exercise—because we kept the spaces but even without this help the ciphertext would not be too difficult to decrypt. We refer to [189] for a detailed example of cryptanalysis of the substitution cipher.

Exercise 1.7 Decrypt the following ciphertext, obtained by encrypting an English plaintext with the substitution cipher:

```
"DSSDNUJLQQVNUYVRNOMNWMSLPVFMPLPKFDMNPHNPNDONBFMJJFUIPFGLOKHLXFQKJSXNSPOSPPNUISDWN
DONDSPMSLMMLOTUYISDJFUJSDSYPLDWPNIYMLMFSDNGNPKSDNJLUMJNPFEJMMSMJNIPSMNOMFSDSHMJNQL
BLELFDUMUYOJFDMNPHNPNDONSPLMMLOTU"
```

1.3 The Vigenère Cipher

The substitution ciphers we considered in the preceding section are also known as *monoalphabetic substitution ciphers* because they use only a substitution (a permutation of the alphabet) to encrypt the whole message. As we have just seen, their vulnerability comes from the fact that the substitution preserves the frequency distribution (with changed characters). To remedy this fact, the idea arose of using

several substitutions—thus obtaining what is often called a *polyalphabetic substitution cipher*—to encrypt a given message, in such a way that different characters can be encrypted as the same character because different substitutions are used and also, conversely, different occurrences of the same character can be encrypted differently for the same reason. This setup destroys the one-to-one correspondence between plaintext characters and ciphertext characters and so it seems that it renders frequency analysis useless. However, things are not so simple, as we are going to see by looking at perhaps the most famous polyalphabetic cipher, which was used for several centuries and eventually became regarded as *'le chiffre indéchiffrable'*. This is the Vigenère cipher, which was introduced by Blaise de Vigenère in the sixteenth century and reigned supreme in the crypto world until it was cryptanalyzed, in the nineteenth century, by Babbage and Kasiski. The Vigenère cipher can be described as follows. Suppose that we have an alphabet Σ, with $|\Sigma| = n$. Then we identify Σ with $\mathbb{Z}_n$ and we set:

- $\mathcal{M} = \mathcal{C} = \mathcal{K} = \mathbb{Z}_n^l$ for some positive integer l.
- If $k = (k_1, \dots k_l) \in \mathcal{K}$ then $\mathbf{Enc}_k((i_1, \dots i_l)) := (i_1 + k_1, \dots, i_l + k_l) \bmod n$ and $\mathbf{Dec}_k((i_1, \dots i_l)) := (i_1 - k_1, \dots, i_l - k_l) \bmod n$.

(here we denote by $(x_1, \dots, x_l) \bmod n$ the l-tuple $(x_1 \bmod n, \dots, x_l \bmod n)$). Thus, to use this cipher we divide the plaintext into blocks of length l and each character in each block is encrypted with the shift given by the corresponding character of the key. The key is usually presented as the word of length l over Σ whose characters correspond to $k_1, \dots, k_l$ (frequently, as is the case with natural language alphabets, the alphabet has a standard ordering which is used to identify it with $\mathbb{Z}_n$ as we have done before by means of the inverse bijections `code` and `char`). It is also customary to use a so-called *Vigenère table*, which is nothing more that the operation table corresponding to the sum in $\mathbb{Z}_n$ given in terms of the characters of Σ. In this way, the first plaintext character, together with those in positions $l + 1, 2l + 1$, ... (i.e., the characters whose position in the plaintext is congruent to 1 modulo l) are encrypted with the shift corresponding to the first character of the key. Similarly, the characters whose position is congruent to 2 modulo l are encrypted with the shift corresponding to the second letter of the key, and so on.

Example 1.4 Let us give an example of a Vigenère encryption using the English alphabet. Suppose the plaintext is the text string `vigenereexample` and the key is `word` (the key should be a random string over the English alphabet but never mind as we shall see that the scheme is easy to break anyway). Then we encrypt the first plaintext character, `v`, by adding it to the first key character `w`, where both letters are regarded as elements of $\mathbb{Z}_{26}$ as we saw when dealing with the Caesar cipher (identifying letters with elements of $\mathbb{Z}_{26}$ by means of the mutually inverse bijections `code` and `char`). Since `v` corresponds to 21 and `w` to 22, the result of adding them is $(21 + 22) \mod 26 = 43 \bmod 26 = 17$. The character that corresponds to 17 is `char(17) =r` and so `r` is the first ciphertext letter. Then we would add the second letters, i.e., `i` to `o`, and so on. When the fifth plaintext letter, `n`, is reached, there are no key letters left since the key has

length 4 in this case. Then we start again with the first letter of the key, i.e., the fifth ciphertext character is obtained by adding the fifth plaintext character `n` to the first (since $5 \bmod 4 = 1$) character of the key, namely `w`. This process is continued until all the plaintext characters are encrypted and the result is summarized in the following table:

Plaintext	`vigenereexample`
Key	`wordwordwordwor`
Ciphertext	`rwxhjsihalrplzv`

1.3.1 The Vigenère Cipher in Maple

We next implement in Maple the encryption and decryption functions of the Vigenère cipher. If we want to be able to handle plaintexts in different languages, we may use the procedure `alph` below to generate the corresponding alphabets. As a sample, we include the English alphabet (also valid for other languages with Latin-based alphabets if diacritics and ligatures are disregarded) and the Spanish alphabet, which has the letter 'ñ' as an additional character (it also has accented vowels but they are not regarded as separate characters, so we will disregard the accents). The procedure takes as input the designation of the language (`en` for English, `es` for Spanish) and produces as output the corresponding alphabet. An obvious modification may allow the procedure to include other languages and their alphabets if they are based on the Latin alphabet or if their character set is a subset of the ASCII-8 code; this is left as an exercise for the interested reader. Specifying the language is a convenient way to select the alphabet—which will only contain lower-case letters and no punctuation symbols—but the procedures for encryption and decryption that follow do not depend on the language, only on the underlying alphabet. In contrast with this, the cryptanalytic functions that we will give later are dependent on statistical properties of the language itself.

```
> alph := proc(language)
    if language = en then convert([$97 .. 122], bytes)
        elif language = es then Insert(convert([$97 .. 122], bytes), 14,  "ñ")
        else error "Unrecognized language"
    end if;
 end proc:
```

We discuss in Chap. 3 how Maple's pseudo-random number generators can be used to generate keys. For now we leave as an exercise to write a Maple function that generates a pseudo-random word of specified length over a given alphabet in order to use it as a key for the Vigenère cipher:

Exercise 1.8 Write a Maple function that uses the functions `Randomize()` and `Random`, from the package `StringTools`, to pseudo-randomly generate a key for the Vigenère cipher.

The following Maple functions—which will be called from the higher-level functions that carry out encryption and decryption of text strings—perform encryption and decryption in $\mathbb{Z}_n^l$, i.e., when messages and keys are regarded as lists of elements of $\mathbb{Z}_n$. The third parameter, `language` is used to allow the functions to deal with different languages and different alphabets.

```
> evig := proc(keylist, textlist, language)
  local l, alphabet, modulus;
  l := nops(keylist);
  alphabet := alph(language);
  modulus := StringTools:-Length(alphabet);
  [seq((textlist[i] + keylist[(i-1) mod l + 1]) mod modulus, i=1..nops(textlist))];
  end proc:

  dvig := proc(keylist, textlist, language)
  evig(-keylist, textlist, language)
  end proc:
```

We can now give the encryption and decryption algorithms and, since they are very similar, we will use only one function for both tasks in order to save space. In the following chapters we will see how such a function can be easily modified to make it able to encrypt/decrypt messages given as files but for now we limit ourselves to messages given as text strings. The function `Vigenere` has three required input parameters, namely `key` (for the key given as a word over the alphabet), `plaintext` (for the plaintext string) and `action` for the action to be performed. The last parameter accepts only two possible values which are either `enc` if encryption is to be performed or `dec` for decryption (note that in this function, as will often be the case with other functions, we include type declarations to specify the type that the argument values must have). There is also an optional keyword parameter (see Maple's help for a description of these parameters) `language`, which is used to specify the plaintext language (since now we are not using global variables as we did before). The default value for this parameter is `en` (corresponding to the English language), so that no argument needs to be passed to this parameter when dealing with English text.

```
> Vigenere := proc(key::string, message::string, action::identical(enc, dec),
                   {language := en})
  local alphabet, keycodes, messagecodes, act, resultcodes;
  alphabet := alph(language);
  keycodes := map(x -> SearchText(x,alphabet)-1, StringTools:-Explode(key));
  messagecodes := map(x -> SearchText(x,alphabet)-1, StringTools:-Explode(message));
  if action = enc then
    act := evig
  elif action = dec then
    act := dvig
  else "Incorrect action"
  end if;
  resultcodes := act(keycodes, messagecodes, language);
  StringTools:-Implode(map(x -> alphabet[x+1], resultcodes))
  end proc:
```

Example 1.5 Let us consider the plaintext and the key used in Example 1.4. Then the ciphertext is obtained as follows:

```
> Vigenere("word", "vigenereexample", enc);
                                "rwxhjsihalrplzv"
```

Now, the ciphertext is decrypted as follows:

```
> Vigenere("word", %, dec);
                                "vigenereexample"
```

Exercise 1.9 Modify the previous functions so that any text over the ASCII-8 character set (excepting the null character) may be encrypted/decrypted (the alphabet would be the string `convert([$1..255],bytes)`).

Exercise 1.10 Consider the encryption algorithm obtained by first encrypting the message using the Vigenère cipher with a key k_1 of length l_1 and then encrypting the resulting ciphertext again with the Vigenère cipher using a key k_2 of length l_2. Show that this is equivalent to a Vigenère encryption with a key k_3. Explain how k_3 is obtained from k_1 and k_2 and determine its length in terms of l_1 and l_2. Find the key word that corresponds to encryption with key 'key' followed by encryption with key 'word'.

1.3.2 Cryptanalysis of the Vigenère Cipher

The size of the key space of the Vigenère cipher is 26^l, where l is the length of the key, assuming that a 26-letter alphabet is used. Thus an exhaustive key search is not viable if a sufficiently long key—say, 20 characters or more—is used. However, we are going to see that this cipher is also easy to cryptanalyze given a sufficient amount of ciphertext, even if longer keys are used. The basic idea to cryptanalyze the Vigenère cipher is that it is enough to determine the length of the key. Once this is done, the rest is easy. Indeed, if the key length is l, then the characters whose position in the plaintext is congruent to 1 modulo l are encrypted with the same shift (the one corresponding to the first letter in the key). Similarly, the characters whose position is congruent to 2 modulo l are encrypted with the second shift, and so on. Thus, after determining the key length, the ciphertext can be decomposed into l sub-ciphertexts, each one consisting of the characters whose position belongs to the same congruence class modulo l, and then one breaks the cipher by carrying out frequency analysis on each of these sub-ciphertexts separately.

The first idea to determine the key length was due to Babbage and Kasiski and is nowadays known as the *Kasiski test*. If the same sequence of three or more letters appears in different places of the ciphertext, it is very likely (the more likely the longer the sequence) that the cause is the fact that a certain repeated substring in the plaintext has been encrypted repeatedly with the same part of the key. Thus, if this happens, the attacker measures the distances between the different appearances of each of these repeated substrings in the ciphertext and expects that the key length is a divisor of all these distances. If the attacker has a sufficient quantity of ciphertext at her disposal she can determine the key length by this method.

We will give below some examples of the use of the Kasiski test but now we will look at an alternative, a more powerful method based on the use of the *index of coincidence* which was introduced by the American cryptanalyst of Russian origin William Friedman in the 1920s. Friedman was the first person who used mathematical methods in cryptology in a nontrivial way and, besides introducing the index of coincidence, he is famous because of his success in cryptanalyzing the Japanese cipher *Purple* in 1939.

The index of coincidence (IC) of a text string $x = x_1, \ldots, x_n$ of length n, denoted $I_c(x)$, is defined as the probability that two elements of x taken at random are the same. It is intuitively clear that the IC of a text string will be larger when the frequency distribution of the characters is less uniform. For example, if we consider the binary alphabet, it is completely obvious that the IC of a string consisting of 99 'zeros' and 1 'one' is much larger than that of a string with the same number of zeros and ones. To compute the IC of a string x of length n over an alphabet of r symbols, suppose that the frequencies (i.e., the number of appearances in the string) of these characters (assuming the characters are enumerated from 0 to $r-1$) are, respectively, $f_0, f_1, ..., f_{r-1}$. Since there are $\binom{n}{2}$ ways to choose two elements in x and, moreover, there are $\binom{f_i}{2}$ ways to choose them so that both coincide with the ith character of the alphabet, we have that:

$$I_c(x) = \frac{\sum_{i=0}^{r-1} \binom{f_i}{2}}{\binom{n}{2}} = \frac{\sum_{i=0}^{r-1} f_i(f_i-1)}{n(n-1)} = \frac{\sum_{i=0}^{r-1} f_i^2 - n}{n(n-1)} \tag{1.1}$$

We are going to develop the tools to cryptanalyze the Vigenère cipher with Maple. From version 11 onwards, Maple has the built-in function `StringTools:-IndexOfCoincidence` to compute the IC of a text string but we are going to give an alternative function to compute the IC that also works on earlier versions of Maple. First, the function `frequencies` below calls Maple's function `CountCharacterOccurrences`, from the package `StringTools`, to count the number of occurrences of each character in a given string over a given alphabet (the English alphabet by default). Then the function `Ic` calls `frequencies` to compute the IC using Eq. 1.1:

```
> frequencies := proc(text, language := en)
  local alphabet;
  alphabet := alph(language);
  [seq(StringTools:-CountCharacterOccurrences(text, l), l in alphabet)]
  end proc:

> Ic := proc(text, language :=en )
  local n, f;
  n := StringTools:-Length(text);
  f := frequencies(text, language);
  evalf((ListTools:-DotProduct(f,f)-n)/(n*(n-1)));
  end:
```

Natural languages have, as we have mentioned, a characteristic frequency distribution of letters. If one knows this distribution, one can compute the IC of the language, which amounts to an approximation of the average IC of all texts in the given

language. Suppose now that we have a language such that the relative frequencies of the alphabet characters are $p_0, p_1, ..., p_{r-1}$. Then, starting with Eq. 1.1 above, we have for a text of length n with frequencies $f_0, f_1, ..., f_{r-1}$:

$$I_c(x) = \frac{\sum_{i=0}^{r-1} f_i^2 - n}{n(n-1)} = \frac{\sum_{i=0}^{r-1} f_i^2 - n}{n^2(1-\frac{1}{n})} = \frac{\sum_{i=0}^{r-1} (\frac{f_i}{n})^2 - \frac{1}{n}}{1-\frac{1}{n}}$$

Now, for n large, each of the relative frequencies $\frac{f_i}{n}$ in the text is close to p_i and we obtain the following approximation:

$$I_c \approx \sum_{i=0}^{r-1} \left(\frac{f_i}{n}\right)^2 \approx \sum_{i=0}^{r-1} p_i^2$$

This value is the probability that two characters chosen at random in a text in the given language are the same. We may apply a similar method to compute the IC of the language consisting of random texts over a given alphabet. For such a text (a text over the given alphabet, whose characters are randomly generated with uniform probability distribution), the IC has the following value:

$$I_c \approx r\left(\frac{1}{r}\right)^2 = \frac{1}{r}$$

We see that, as was to be expected, in this case the IC depends only on the size of the alphabet. For example, it is approximately 0.0384 for the 26-letter English alphabet and 0.037 for the 27-letter Spanish alphabet.

Exercise 1.11 Use the functions `Generate` and `AddFlavor` from Maple's `RandomTools` package to generate pseudo-random text strings of specified length over a given alphabet. Compute the IC of these text strings using the function `Ic` (or `IndexOfCoincidence`) and check that they are close to the expected value for a random string over the considered alphabet.

As we observed before, the IC increases when the frequency distribution of characters is more uneven (that is, when the language in question has more redundancy) and so the IC of a natural language with an alphabet of r symbols is larger than $\frac{1}{r}$. We can use the above formula to compute the indices of coincidence of English and Spanish from their frequency distributions. The frequency distribution of the languages we are considering is given in Maple by the following procedure. The list of frequencies in English is taken from [130] and that for Spanish was compiled from a collection of texts of various kinds which included around 850,000 characters.

```
> freqlist := proc(language)
    if language = en then
      [0.0817, 0.0149, 0.0278, 0.0425, 0.1270, 0.0223, 0.0202, 0.0609, 0.0697,
      0.0015, 0.0077, 0.0403, 0.0241, 0.0675, 0.0751, 0.0193, 0.0010, 0.0599,
```

```
    0.0633, 0.0906, 0.0276, 0.0098, 0.0236, 0.0015, 0.0197, 0.0007]
  elif language = es then
    [0.1174, 0.0117, 0.0436, 0.0474, 0.1389, 0.0073, 0.0096, 0.0098, 0.0713,
    0.0031, 0.00024, 0.0535, 0.0294, 0.0713, 0.001, 0.0922, 0.0265, 0.0115,
    0.0624, 0.0781, 0.0453, 0.043, 0.0109, 0.00006, 0.0021, 0.0085, 0.0033]
  end if
end proc:
```

Exercise 1.12 Use literary works in text format, taken from the Internet, to compute with Maple the frequency distribution of different languages.

The following procedure computes the index of coincidence of a language from the frequency list:

```
> Ind := proc(language)
  add(i^2, i in freqlist(language)))
  end proc:
```

In particular, we see that the indices of coincidence of English and Spanish, computed from the frequency distributions given above, are the following:

```
> Ind(en);
                                0.06552226
> Ind(es);
                               0.0749428912
```

Of course, these indices are only approximations and one may obtain slightly different values by using different texts as sources. For example, the IC of 'Don Quijote', in the version available at the 'Project Gutenberg' web page, [161] with 1, 640, 590 characters, is just 0.0746771. Other sources give the index of coincidence of Spanish as 0.078 and that of English as 0.067. We will see that approximations given in the Maple code above are precise enough to allow for an accurate cryptanalysis of the Vigenère cipher.

The difference between the value corresponding to a random text and that characteristic of the language can be exploited to find the length of the key used in a Vigenère encryption. Observe that, in a monoalphabetic cipher, the ICs of the plaintext and the corresponding ciphertext are exactly the same and hence the IC of the ciphertext should be close to the one corresponding to the language used in the plaintext. In contrast with this, the ciphertext obtained from applying a polyalphabetic cipher such as Vigenère's should have an IC which is closer to that of a random text over the alphabet (the longer the key the closer). This remark can be quantitatively analyzed in order to derive a formula that gives the key length in terms of the IC of the ciphertext along with the IC of the plaintext language and the length of the corresponding alphabet (which are known assuming that the plaintext language is known as postulated by Kerckhoffs' principle). However, these formulas are not very precise, especially when the key word has repeated characters and, for this reason, we will sketch a different method (also based on the above remarks) to determine the key length, which is much more precise in practice. To illustrate it, suppose that we have the following ciphertext, corresponding to an English plaintext:

```
> c := "oltmhvurseaebwzchltyqwncnhouyqquvwaibcnwgwvofmrplupaujpoxlogautayzkmxzevec\
  wbbsvmifdfucgmmshtijtyngepyyqohasfontvnwtrivwzcbepyatbsfejbiwmnavbioikqwlmmsttlw\
  nfkfpjlmvuwzakqnztxqqugsejjlhimsobmlifdmtnclyevwsfrnhimsirmwvfdmqiodvarvgirygwok\
  nylstylupaujcnwoeinxtb":
```

In order to cryptanalyze it we first determine the key length—let's call it l—as follows. We partition the ciphertext into a collection of t sub-ciphertexts (for $t = 1, 2, 3, ...$) in such a way that the first one consists of the characters whose position in the ciphertext is congruent to 1 modulo t, the second one those with position congruent to 2 modulo t and so on, until the tth one, which consists of the characters whose position is congruent to 0 modulo t. We do this up to a value of t which is as large as the key length we are trying to find (we have to guess this value but if, after completing the process, the key length is not within the considered interval, we can try with larger values of t). For each of those sub-ciphertexts, we compute the IC and then we compute, for each value of t, the mean of the ICs of the t sub-ciphertexts. We expect that, if l is the (still unknown) key length, the mean of these ICs for $t = l$ should be very close to the IC of English, that is, to 0.066. The reason is that, when t equals the key length, each of the sub-messages we are considering has been encrypted with the same shift cipher and so its expected IC is that of the language used. On the contrary, for values of t different from l, the characters of each sub-message were not encrypted with the same substitution and so the corresponding ICs will be lower, closer to that of a random text. The next function partitions the ciphertext (given as a list of elements in $\mathbb{Z}_m$, where $m =$ `modulus = 26` in our case) into sub-ciphertexts as indicated above, with $1 \leq$ `t` $\leq$ `long`.

```
> partit:= (lis, long) -> [seq([seq(lis[(i-1)*long+t], i =
  1..floor(((nops(lis)-t)/long)+1))],t = 1..long)]:
```

Now, the following function computes the mean IC for the sub-ciphertexts corresponding to each value of t:

```
> indiceslist := proc(text, min, max)
  uses StringTools;
  local i, ind, ic, t;
  i := min;
  while i <= max do
    ind := map(IndexOfCoincidence, map(Implode, partit(Explode(text), i)));
    ic := evalf((sum(ind[t], t = 1 .. nops(ind)))/nops(ind));
    i := i+1;
    printf("%2d %9f\n", i-1, ic);
  end do;
  end proc:
```

We may look at the list of indices obtained when we apply this function to our sample ciphertext `c` above, with t ranging from 1 to 15. The list is the following:

```
> indiceslist(c, 1, 15);
 1  0.039338
 2  0.041093
 3  0.044679
 4  0.049851
 5  0.039578
 6  0.049834
 7  0.037323
 8  0.047631
 9  0.041010
10  0.044026
11  0.035723
12  0.070599
13  0.035762
14  0.036218
15  0.045370
```

This shows quite clearly that the key length should be $l = 12$ and, as we shall soon see, this is indeed the case. In fact, if we continue computing mean ICs up to $t = 24$, we see that all of them are below 0.058 except the one corresponding to $t = 24$, which is 0.070286. One sees that the mean ICs are somewhat higher for values of t which are not relatively prime to 12 (Exercise: why?) and that, for multiples of 12 such as 24, the value is very close to the value for 12. This could lead us to think that the key length is 24 but this would not be a serious problem, for the Vigenère cipher with one keyword is equivalent to that with the keyword concatenated several times with itself.

The process of finding the key length can be automated in Maple using, for example, the following function which, on input a Vigenère ciphertext and a maximum length, outputs the probable length of the key. The function applies the method explained above, computing the indexes of coincidence corresponding to the different ways of partitioning the ciphertext, until finding one which is very close to the IC of the language. The parameter `max` is optional and, if no argument is supplied for it, the function will use a maximum value equal to the text length divided by 15 (if the key is longer than this, i.e., if we have fewer than 15 ciphertext characters for each key character, then the process becomes more and more unreliable):

```
> keylength := proc(text, max:=floor(StringTools:-Length(text)/15), {language:=en})
  uses StringTools;
  local i, j, k, l, m, t, ind;
  j := 1;
  k := 0;
  m := 0;
  i := 1;
  ind := [];
  l := max;
  while i <= l and m < Ind(language)-0.002 do
    ind := map(IndexOfCoincidence, map(Implode, partit(Explode(text), i)));
    m := evalf((sum(ind[t], t = 1 .. nops(ind)))/nops(ind));
    if k < m then
      k := m;
      j := i
    end if;
    i := i+1
  end do;
  j
  end proc:
```

In our case, this gives:

```
> keylength(c);
                                12
```

Now, the final part of the attack consists in doing the frequency analysis for each one of the l sub-ciphertexts (where l is the key length) into which we divided the ciphertext, which is very easy because each one of the ciphers used for each corresponding sub-message is actually a shift. To do this, we determine the frequency distribution of each of the l (in our example, 12) sub-ciphertexts. For each of these texts, we have a list of frequencies, one of whose shifts should be close to the list of frequencies corresponding to the language, as this sub-ciphertext has been obtained from the corresponding sub-message of the plaintext by means of a shift cipher. To determine which shift is closest to the expected list of frequencies we use the

chi-squared statistic (see any introductory statistics text such as, for example, [168] for an explanation of the chi-squared goodness-of-fit test). If $(f_0, f_1, \ldots, f_{m-1})$ is the vector corresponding to the expected frequency distribution of the alphabetic characters in the language (English in this case) and $(g_0, g_1, \ldots, g_{m-1})$ is the observed frequency distribution of one of the l sub-ciphertexts into which the given ciphertext is partitioned (where l is the key length), then for each of the m shifts $(g_{0j}, g_{1j}, \ldots, g_{(m-1)j})$ of $(g_0, g_1, \ldots, g_{m-1})$, with $j = 0, \ldots, m-1$ and $g_{0j} = g_j$, we will compute the chi-squared statistic $\chi^2 = \sum_{i=0}^{m-1} \frac{(g_{ij}-f_i)^2}{f_i}$. Then we choose the shift j for which this value is lowest, which will give, with high probability, the key letter corresponding to this sub-ciphertext. In our situation, this process is repeated for each of the 12 sub-ciphertexts to reveal the 12 characters of the key. We will make use of the following function to determine the position of the minimum element in a list:

```
> findmin := lis -> ListTools:-Search(min(op(lis)), lis):
```

Next we give the function `frequencyanalysis` which actually computes the key letter corresponding to one of the sub-ciphertexts following the method outlined above:

```
> frequencyanalysis := proc(lis, {language := en})
  uses ListTools;
  local f, s, alphabet, modulus;
  f := freqlist(language);
  alphabet := alph(language);
  modulus := StringTools:-Length(alphabet);
  s := seq(DotProduct(map(i -> 1/i, f), map(i -> i^2, x-f)),
           x = seq(Rotate(lis, i), i = 0 .. modulus-1));
  alphabet[findmin([s])]
  end proc:
```

The function `VigenereKeyFind` below uses the preceding ones to find the key that has been used. The input is the ciphertext and, optionally, the maximum value that will be tested as possible length of the key and the language used. As in the case of the function `keylength` above, and for the same reasons, the default value for `max` is taken to be equal to $\frac{1}{15}$th of the length of the ciphertext. A little experimentation shows that 25 ciphertext characters—or even 20 or fewer in many cases—for each key character provides for successful key recovery most of the time.

```
> relativefrequencies := proc(text, language)
  evalf(frequencies(text, language)/StringTools:-Length(text))
  end proc:

  VigenereKeyFind := proc(ciphertext, max:=floor(StringTools:-Length(ciphertext)/15),
                          {language:=en})
  uses StringTools;
  local freqs;
  freqs := map(x -> relativefrequencies(x, language),
               map(Implode, partit(Explode(ciphertext),
                   keylength(ciphertext, max, ':-language'=language))));
  Implode(map(li -> frequencyanalysis(li, ':-language'=language), freqs))
  end proc:
```

For the ciphertext given above as example, we obtain the following key:

```
> VigenereKeyFind(c);
                              "juliuscaesar"
```

Now we can go a step further, and give a Maple function which will cryptanalyze a ciphertext and output the corresponding plaintext:

```
> VigenereCryptanalysis := proc(ciphertext,
                       max:=floor(StringTools:-Length(ciphertext)/15),{language:=en})
  Vigenere(VigenereKeyFind(ciphertext, max, ':-language'=language), ciphertext, dec,
                            ':-language'=language)
  end proc:
```

The result of applying this function to the sample ciphertext above is the following:

```
> pt := VigenereCryptanalysis(c);
"friendsromanscountrymenlendmeyourearsicometoburycaesarnottopraisehimtheevilthatmen\
dolivesafterthemthegoodisoftinterredwiththeirbonessoletitbewithcaesarthenoblebrutus\
hathtoldyoucaesarwasambitiousifitweresoitwasagrievousfaultandgrievouslyhathcaesaran\
sweredit"
```

Let us experiment a bit to see if the length of the ciphertext given above may be enough to cryptanalyze the Vigenère cipher even if a longer key is used. We may encrypt again the plaintext above, using a longer key, and still be able to cryptanalyze the resulting ciphertext. For example, let us encrypt it with key `"willshakespeare"`, without showing the ciphertext on screen, and let us try to recover the key used from the ciphertext:

```
> c2 := Vigenere("willshakespeare", pt, enc):
  VigenereKeyFind(%);
                              "willshakespeare"
```

In this case, a ratio between ciphertext length and key length below 20 is enough for the successful recovery of the key. This is often the case but in some cases it may not be enough. For example, if the key is a word with many repetitions, then it may be more difficult to find. Let us now see what happens if we use a still longer key with the above plaintext:

```
> Vigenere("williamshakespeare", pt, enc):
  VigenereKeyFind(%);
                                   "zilypeqrd"
```

In this case, the result is not even remotely accurate but this is due to the fact that the key used is longer than the maximum length examined by the procedure `keyword` (there are fewer than 15 ciphertext characters for each key character). We can avoid this difficulty by specifying a longer maximum length, for example, 20:

```
> Vigenere("williamshakespeare", pt, enc):
  VigenereKeyFind(%, 20);
                              "williamshakkspuare"
```

Now the result obtained is still incorrect but is much closer to the correct one and, by using this key, one could easily recover the plaintext as only a few scattered characters (just two out of every 18) would be incorrect and the redundancy of the language would easily compensate for it.

Remark 1.1 We have been using English words (or names) as keys for the Vigenère cipher in our examples but we stress the fact that, as a general rule, for actual cryptographic use randomly generated keys should be used. Indeed, the number of English words of a given length is tiny in comparison with the number of all possible words of the same length over the English alphabet and a similar remark applies in general to other kinds of structured (and hence non-random) keys vis-a-vis random keys. Using English words as keys greatly reduces the size of the key space and makes it possible to use *dictionary attacks*, which are exhaustive searches restricted to words or phrases in a dictionary. In the case of the Vigenère cipher this does not matter much because the cipher is easily broken by the index of coincidence method, which does not exploit special characteristics of keys and works equally well with random keys.

Example 1.6 We do an example with a Spanish text:

```
> st:="esteesunmensajedepruebadecriptoanalisisenespaña olusandoelindicedecoincidencia":
  Vigenere("wxyz", st, enc, language = es);
  Vigenere("wxyz", %, dec, language = es);
  VigenereKeyFind(%%, language = es);
  "aprdapsmiblrwgccanptayycazphmqnzjxjhofqdjbqowlnkqpymzmckekbhybbdymgmyfbdjzgz"
  "esteesunmensajedepruebadecriptoanalisisenespaña olusandoelindicedecoincidencia"
                                        "wxyz"
```

Exercise 1.13 Modify the Maple functions used to cryptanalyze the Vigenère cipher in order to make them able to deal with more languages. In order to do it, compute the frequency distributions of characters in the desired languages—or learn them from the literature or the Web—and make the procedures work with them.

Exercise 1.14 Consider Vigenère encryptions with different ratios of ciphertext length to key length and use the Maple functions `VigenereKeyFind` and `VigenereCryptanalysis` to explore the minimum ratio needed to obtain reliable decryptions by the index of coincidence method.

We will now have a brief look at the Kasiski test, which can be used as a complement to the method based on the index of coincidence. Maple has a function `Kasiski` in the `StringTools` package but it does not do what we need, so we give a simple function that computes the greatest common divisor of the distances between repeated substrings of length at least 4 (length 3 substrings are often considered for this purpose but this way the method is more reliable):

```
> KasiskiTest := proc(ciphertext)
  local positions;
  positions := select(x -> evalb(3 < x[3]), [StringTools:-Repeats(ciphertext)]);
  igcd(op(map(x -> x[2]-x[1], positions)))
  end proc:
```

Example 1.7 Let us apply the function `KasiskiTest` to some of the previously computed ciphertexts. For instance, for ciphertext `c` above, we have:

```
> KasiskiTest(c);
                                        24
```

The result is 24 and, as we have seen before, the key length is actually 12 which is, as expected, a divisor of 24. Now, if we apply the test to ciphertext `c2` above we obtain:

```
> KasiskiTest(c2);
                                    30
```

which, again, is in consonance with the fact that the key used in this case had 15 characters. Of course, for some key lengths it may happen that the Kasiski test does not give any information (this is less likely the longer the ciphertext is in comparison with the key length). We can look, for example, at what happens if we encrypt plaintext `pt` with shorter keys:

```
> KasiskiTest(Vigenere("juliusca", pt, enc));
                                    8
> KasiskiTest(Vigenere("juliusc", pt, enc));
                                    0
> KasiskiTest(Vigenere("julius", pt, enc));
                                    6
> KasiskiTest(Vigenere("juliu", pt, enc));
                                    5
> KasiskiTest(Vigenere("juli", pt, enc));
                                    8
> KasiskiTest(Vigenere("jul", pt, enc));
                                    3
```

We see that for keys of length 3, 5, 6 and 8 the test reveals the exact length of the key, while for a key of length 4 the test gives 8, of which 4 is a divisor. For a key of length 7 with this plaintext, the test returns 0 and gives no information. The Kasiski test may be used previously to applying the method based on the index of coincidence to select the probable length(s) of the key. Once we suspect that the key is a divisor of a given integer, we can use the index of coincidence to select which divisor is the probable length of the key.

Exercise 1.15 Write a Maple function that combines the Kasiski test with the index of coincidence and, given a Vigenère ciphertext, returns (with high probability if the ciphertext is sufficiently long) the key used.

1.4 The Hill Cipher

The encryption schemes we have studied so far are easy to break, mainly because they encrypt the alphabet characters one at a time, so that changing one letter in the plaintext changes only one letter in the ciphertext. This fact is what makes these systems vulnerable to attacks based on statistical frequency analysis. A powerful idea to render statistical analysis more difficult consists of encrypting blocks of symbols simultaneously, in such a way that changing one symbol in a plaintext block can potentially change all the symbols in the corresponding ciphertext block. This is, in essence, what *block ciphers* do and, in fact, things can get even more difficult for the cryptanalyst since, as we will see, there are encryption modes in which the result of encrypting one block also depends on the preceding ones. We study block ciphers

from a more general point of view in Chap. 4 but for now we restrict ourselves to the situation that arises when we consider an encryption scheme that acts on blocks of alphabet symbols. Such an encryption scheme is often also called a block cipher and is characterized by the fact that both the plaintext space $\mathcal{M}$ and the ciphertext space $\mathcal{C}$ are the set of words Σ^n of fixed length n over an alphabet Σ. The positive integer n is called the *block length* or the *block size*.

Observe that this concept is very general and includes, for example, monoalphabetic substitution ciphers, which are just block ciphers of length 1. However, the preceding remarks about resistance to statistical cryptanalysis apply only to block ciphers with larger block sizes. Similarly, the Vigenère cipher may be regarded as a block cipher with block size equal to the key length but the fact that it is built from Caesar ciphers makes it vulnerable to statistical cryptanalysis even if the block length is large.

In order to make statistical cryptanalysis hard, one may try to consider a cipher in which each block is potentially encrypted in as many ways as possible, so that the key space is also as large as possible. Observe now that, since encryption functions must be injective, they must also be permutations of Σ^n. Thus, in a similar way to the case of monoalphabetic substitution ciphers, we can easily describe the more general block cipher of block length n, which is obtained by taking as encryption functions all the possible permutations of Σ^n:

- $\mathcal{M} = \mathcal{C} = \Sigma^n$.
- $\mathcal{K} = S(\Sigma^n)$ (the set of all permutations of Σ^n).
- For each $\sigma \in S(\Sigma^n)$, $\mathbf{Enc}_\sigma := \sigma$, $\mathbf{Dec}_\sigma := \sigma^{-1}$.

This encryption scheme has a very large key space since there are $t^n!$ possible keys if the alphabet has t symbols. It is not commonly used, however, for efficiency reasons, as it is not clear how to represent and evaluate efficiently an arbitrary permutation of Σ^n. For this reason, in practice block ciphers only use a subset of the set of all possible permutations. However, one must be very careful with the way these permutations are chosen. They should look like "random permutations" (as we will see when we discuss block ciphers in detail in Chap. 4) because any "regularity" could conceivably be exploited by an attacker.

To get a feeling of what may happen if the encryption functions for a block cipher have a nice structure, we will look at the Hill cipher, invented by L. Hill in 1929. The Hill cipher is obtained from the general block cipher described above by restricting the encryption functions to be linear. This makes sense because of the fact that, if the alphabet Σ has t symbols, then it may be identified with $\mathbb{Z}_t$ which, as explained in Chap. 2, is a ring and even a field when t is prime.

The plaintext (and ciphertext) space $\mathbb{Z}_t^n = \{(x_1, x_2, \ldots, x_n) | x_i \in \mathbb{Z}_t\}$ then consists of the n-component vectors over $\mathbb{Z}_t$ and has the structure of a $\mathbb{Z}_t$-module (it is actually a 'free module', the exact analogue over a ring of the concept of vector space over a field, cf. [45, 5.2]) and the maps that preserve this structure are called linear (or module homomorphisms). The linear maps from $\mathbb{Z}_t^n$ to itself are given by $n \times n$ square matrices with coefficients in $\mathbb{Z}_t$ (the image of a (row) vector of $\mathbb{Z}_t^n$ is obtained

by multiplying it by the matrix as is standard in linear algebra, i.e., using matrix multiplication). The linear applications that are invertible (i.e., the permutations) are those given by invertible matrices, i.e., matrices A such that $gcd(\det(A), t) = 1$. This is because the determinant and the concept of adjoint matrix can be defined exactly as in the case of matrices over fields and then it is easy to see that a square matrix is invertible if and only if its determinant is invertible in $\mathbb{Z}_t$. The inverse matrix is computed in a way entirely similar to when the underlying ring is a field, namely by multiplying the inverse of the determinant by the adjoint matrix. We will not go into these details and will simply let Maple carry out this computation. The Hill cipher is given by:

- $\mathcal{M} = \mathcal{C} = \mathbb{Z}_t^n$.
- $\mathcal{K} = \{A \in M_{n\times n}(\mathbb{Z}_t) | \gcd(\det(A), t) = 1\}$.
- For each $A \in \mathcal{K}$ and each $v \in \mathbb{Z}_t^n$, $\mathbf{Enc}_A(v) := vA$, $\mathbf{Dec}_A(v) := vA^{-1}$.

1.4.1 The Hill Cipher in Maple

Let us see how the Hill encryption scheme can be implemented in Maple. We start by loading the package `StringTools`, which is needed by several of the functions used below, and we define the alphabet that will consist just of the 26 letters of the English alphabet followed by one space, so that it will have exactly 27 symbols (we could also use the standard 26-letter alphabet but the presence of spaces will make texts more legible and, moreover, we will use spaces to pad the messages; another good possibility is to use an alphabet of 29 symbols in which case $\mathbb{Z}_{29}$ is a field and the invertible matrices are those with nonzero determinant). We store the alphabet in a global variable `alphabet` and we also use a variable `modulus` for the modulus t (i.e., the number of symbols in the alphabet) which, in this case, will be 27.

```
> with(StringTools):
  alphabet := cat(convert([$97..122], bytes)," "):
  modulus := Length(alphabet);
```

We recall the functions `code` and `char` that we have used to establish inverse bijections between the alphabet and $\mathbb{Z}_t$ (here $\mathbb{Z}_{27}$).

```
> code:= letter -> SearchText(letter,alphabet)-1:
  char := i -> alphabet[i+1]:
```

The next two functions are used to pad the message (with the last character of the alphabet) in order to make its length a multiple of the block length, and to build the blocks of plaintext that will be encrypted. The plaintext is converted to a list of elements of $\mathbb{Z}_t$ ($\mathbb{Z}_{27}$ if our 27-letter alphabet is used) and pads the list at the end with items equal to $t - 1$ (26 in our case), which corresponds to padding the text with the last character of the alphabet. Then the blocks (the vectors of $\mathbb{Z}_t^n$) are organized in a matrix, where each row corresponds to one block:

```
> pad:= proc(message, blocklength)
  local n, m, lis, p;
  m := message;
  n := blocklength;
  lis := map(code, Explode(m));
  p := n - (length(m) mod n);
  lis := [op(lis),seq(modulus-1, j=1..p)];
  lis;
  end:

> blocks:= (message, blocklength) ->
           Matrix([seq([op(i..i+blocklength-1, pad(message,blocklength))],
           i = seq(blocklength*k+1,k=0..floor((Length(message)-1)/blocklength)))]):
```

In order to define the encryption and decryption functions we will first load the package `LinearAlgebra` because we will make use of several functions in this package and in the sub-package `LinearAlgebra:-Modular` (so that functions in this sub-package will be called simply in the form `Modular:-functionname`):

```
> with(LinearAlgebra):
```

The encryption and decryption functions are given below. The encryption function `HillEnc` proceeds by first building the plaintext blocks by means of the function `blocks` which produces the blocks matrix with coefficients in $\mathbb{Z}_t$. Then this n-column matrix is multiplied, using Maple's function `LinearAlgebra:-Modular:-Multiply`, by the key matrix. This way the encrypted blocks matrix is obtained and, finally, the ciphertext (given as a text string) is recovered from this matrix. Observe that having the blocks organized as a matrix is very convenient because this way the whole encryption is carried out with just one matrix multiplication (this is essentially the same as multiplying each one of the blocks—which are the rows of the block matrix—by the key matrix). Note that the decryption function is just the encryption function applied to the inverse matrix of the key matrix. To compute the inverse matrix, the function `LinearAlgebra:-Modular:-Inverse` is used and the 'row reduction method' `RREF` is specified because other methods may not be able to compute the inverse when the modulus t is composite (even if the inverse exists, see Maple's help for details).

```
> HillEnc := (key, message) ->
             Implode(map(char, ListTools:-Flatten(convert(Modular:-Multiply(
             modulus, blocks(message, RowDimension(key)), key), list, nested)))):

> HillDec := (key,ciphertext) ->
             HillEnc(Modular:-Inverse(modulus, key, RREF), ciphertext):
```

To give an example of how these functions work, we need to use as key an invertible square matrix over $\mathbb{Z}_{27}$. We will generate the key using one of Maple's pseudo-random generators (which are discussed in more detail in Chap. 3) by means of the following procedure, which accepts as input the size of the matrix (i.e., the number of rows):

```
> PseudoRandomInvertibleMatrixMod:= proc(dimension)
  local q, d;
  randomize();
  d := 0;
  while igcd(d,modulus) <> 1 do
```

```
    q := Modular:-Random(modulus,dimension,dimension,integer);
    d := Modular:-Determinant(modulus,q)
  end do;
  q
  end:
```

Now, we use the preceding procedure to generate a 4×4 invertible matrix over $\mathbb{Z}_{27}$:

```
> k: = PseudoRandomInvertibleMatrixMod(4);
```

$$k := \begin{bmatrix} 17 & 10 & 6 & 19 \\ 15 & 16 & 3 & 5 \\ 12 & 22 & 8 & 5 \\ 4 & 21 & 13 & 18 \end{bmatrix}$$

Using this matrix as key, we encrypt the plaintext: “this text is the message”:

```
> HillEnc(k, "this text is the message");
                          "ctbedbeyr esrtyvna hqstm"
```

Finally, we decrypt the ciphertext:

```
> HillDec(k,%);
                          "this text is the message"
```

1.4.2 Cryptanalysis of the Hill Cipher

Let us now look at the cryptanalysis of the Hill cipher. First of all, observe that changing one letter of plaintext can potentially change n letters of ciphertext, namely, all the letters in the ciphertext block corresponding to the plaintext block with the changed letter. In the above example, look at what happens if we change just one letter in the plaintext:

```
> HillEnc(k, "thin text is the message");
                          "jwrwdbeyr esrtyvna hqstm"
```

We see that all four characters in the first ciphertext block have changed. This is in contrast with the previous encryption schemes and makes frequency analysis ineffective if the block size n is not too small. The situation can be best described by referring to the concepts of *diffusion* and *confusion*, which were introduced by Shannon in his 1949 paper [176]. These concepts are formulated in the context of *information theory* and refer to properties that a good encryption scheme should have to make statistical cryptanalysis difficult or even impossible. We will not give a precise definition of these terms—which would require an excursion into information theory—since an informal one will be sufficient for our purposes:

- An encryption scheme has high diffusion when changing one plaintext character (or bit) produces the change of many ciphertext characters (or bits). In a block cipher, diffusion propagates small changes in a plaintext block to a large part of the corresponding ciphertext block. This is the effect we saw in the above example.
- The confusion of an encryption scheme is high if the relation between key and ciphertext is complex so that, in particular, each ciphertext character depends on several parts of the key.

We have already observed that the Hill cipher has the high diffusion property. Similarly for confusion, because if we change a character of ciphertext, keeping the original plaintext, then a whole column of the key matrix may change. These properties make the cipher difficult to break by using a ciphertext-only attack. This, however, does not mean that the Hill cipher is secure and we are going to show that it is, in fact, highly vulnerable to a known plaintext attack (and hence unsuitable for most applications). Such an attack succeeds because the encryption functions, being linear, are far from being random permutations. By standard linear algebra, they are uniquely determined as soon as the attacker knows the image of a basis of $\mathbb{Z}_t^n$ and hence a successful known plaintext attack can be mounted if the known plaintext contains n linearly independent (when regarded as vectors of $\mathbb{Z}_t^n$) blocks. This is not difficult to achieve since n will be relatively small (compared, for example, to the size of the key space) and, even if the known plaintext does not contain a basis, useful information about the encryption function can be obtained from its action on a smaller set of linearly independent vectors.

We now give a Maple function to cryptanalyze the Hill cipher. The first two input parameters are `knownplaintext`, for a text string containing the known plaintext, and `kciphertext` for another text string containing the ciphertext obtained by encrypting `knownplaintext`. The third parameter is for the block length—supposed to be known by Kerckhoffs' principle—and the last, `ciphertext`, for the text string containing the target ciphertext we are trying to decrypt (not to be confused with `kciphertext`, whose decryption is already known).

The function computes the block matrices corresponding to the known plaintext and the corresponding ciphertext. Then it searches in the known plaintext block matrix for n rows which make an $n \times n$ (where n is the block length) invertible matrix over $\mathbb{Z}_t$—so, for the algorithm to work as intended, the known plaintext should contain at least n blocks. The search is carried out by using Maple's function `combinat:-choose` to generate all the combinations of n rows from the known plaintext matrix and, for each sub-matrix determined by one of these combinations, checking whether the matrix is invertible. If no invertible sub-matrix is found (which, in other terms, means that the known plaintext does not contain n independent blocks) then an error is returned stating that there are not enough linearly independent blocks to complete cryptanalysis.[5] If an invertible $n \times n$ sub-matrix X is found, then we have the matrix equation $XA = Y$, where A is the key matrix and Y is the submatrix of the ciphertext matrix obtained from the known plaintext whose rows correspond

[5] Of course, even if this happens, ciphertext decryption may still be possible, but not by this automated method.

to those of X. Since X is invertible, the key A may be computed as $A = X^{-1}Y$ and then the target ciphertext is decrypted by using the inverse of A. Finally, the function returns the decryption of the target ciphertext, given as a text string.

```
> HillCryptanalysis := proc(knownplaintext, kciphertext, blocklength, ciphertext)
  local blt, blc, rd, det, c, z, q, rows, gcd, invq, k, x;
  if StringTools:-Length(ciphertext) mod blocklength <> 0 then
    error "Erroneous blocksize"
  end if;
  blt := blocks(knownplaintext, blocklength);
  blc := blocks(kciphertext, blocklength);
  rd := RowDimension(blt);
  gcd := 0;
  c := combinat:-choose([$1..rd], blocklength);
  for z in c while gcd <> 1 do
    q := SubMatrix(blt, z, [$1..blocklength]);
    det := Modular:-Determinant(modulus, q);
    gcd := igcd(det, modulus);
    rows := z;
  end do;
  if gcd <>1 then
    error "There are not enough linearly independent blocks to complete
           cryptanalysis"
  end if;
  invq := Modular:-Inverse(modulus, q, RREF);
  x := SubMatrix(blc, rows, [$1..blocklength]);
  k := Modular:-Multiply(modulus, invq, x);
  HillDec(k, ciphertext);
  end proc:
```

Let us present a specific example of how this procedure works. Let us take as known plaintext the following string:

```
> kpt := "this sentence is the known plaintext that we will use in our attempt to
  break this block cipher":
```

Next, we generate the key. In order to make the example nontrivial, we will use a 10×10 matrix (i.e., we will use 10 as block length) which will not be explicitly printed here. This way the key space (consisting of all the invertible 10×10-matrices over $\mathbb{Z}_{27}$) is sufficiently large to ensure that a brute-force attack is infeasible.

```
> k := PseudoRandomInvertibleMatrixMod(10):
```

We compute the ciphertext that corresponds to the above known plaintext:

```
> kct := HillEnc(k,kpt);
  kct := "sqwtwoz h sxoniizxtcmthsuxgasjfloxyryfktcnyptyzpmlhycsubnxyvhakozjunptrc\
  bbgtraeunqvkoayzsjzql  notgb"
```

Now, suppose that we have a ciphertext obtained by encrypting an unknown plaintext with the same key:

```
> ciphertext := "tt msflnikw vrgzncxftqwzeeebal":
```

To recover the plaintext without using k (which now we assume is unknown to us), we use the `HillCryptanalysis` function above:

```
> HillCryptanalysis(kpt, kct, 10, ciphertext);
                    "this cipher is easy to break"
```

Note that the recovered plaintext has some additional spaces at the end. This is due to the fact that we used the last character in the alphabet (the space) to pad the messages whose length is not a multiple of the block size. Observe also that the function `HillCryptanalysis` actually recovers the key that is being used, and can be modified in an obvious way to produce this key as output.

Exercise 1.16 Modify the function `HillCryptanalysis` to make it output the key used. In this case only three arguments are necessary: the known plaintext, the known ciphertext and the size of the key.

Exercise 1.17 Consider the following plaintext written in the 27-character English alphabet that includes the space: "the hill cipher is highly vulnerable to a known plaintext attack". Suppose that this message is encrypted using the encryption algorithm above and an invertible $n \times n$ matrix over $\mathbb{Z}_{27}$ as key, and the adversary knows both this plaintext and the corresponding ciphertext. Determine the maximum size of n that allows the adversary a full recovery of the key using only the linear algebra attack we have described, and for this maximum key size find the shortest plaintext length that allows full key recovery by this method. Is there another text of the same length that allows the full key recovery for a larger value of n?

1.5 Some Conclusions

The history of cryptography is a fascinating subject and in this chapter we have looked at several encryption schemes whose interest is mainly of a historical nature, as they are far from secure and hence no longer used in serious cryptographic applications. But we have not studied the history of the subject per se and, for a detailed account, the interested reader may consult David Kahn's encyclopedic work [108] and also [183], which is a less extensive but more up-to-date reference.

These systems are also often used in puzzles but the main reason why we have studied them is not their recreational value but, rather, the fact that they teach us valuable lessons from which we should learn. One of the first things we have observed is that a large key space is a necessary, but not sufficient, condition for the security of an encryption scheme. It is not sufficient because, among other possibilities, an adversary can use statistical methods to find connections between the ciphertext and the plaintext, such as in the case of monoalphabetic substitution ciphers and also of the Vigenère cipher. Even when statistical methods do not work so well, there may be structural connections that make a cipher insecure as we have seen in the case of the Hill cipher, when these connections allow the recovery of the key with relatively little effort.

One of the things that stands out from the analysis of these schemes bearing in mind the previous remarks is that, speaking from the cryptographer's point of view,

ciphertext should look like random text to an adversary. Even if the attacker has at her disposal a piece of plaintext and the corresponding ciphertext, she should not be able to find patterns connecting them that could be exploited to analyze other ciphertexts.

As we have noticed, the systems studied in this chapter are all easy to cryptanalyze by elementary means. There are many other classical ciphers that we have not studied and most of them are also easy to break. There are also historical ciphers whose cryptanalysis requires a much greater effort. This is the case, for example, for the cipher implemented by the *Enigma machine*, used by the Germans during World War II and cryptanalyzed by Polish and British cryptanalysts (see, for example, [184] for details).

The cryptanalysis of these historical ciphers is a reflection of the fact that all the publicly known encryption schemes originated before World War II—with the exception of the one-time pad described in Chap. 3—have essentially been broken. After the introduction of public-key cryptography in the 1970s and the development of encryption schemes based on hard computational problems, cryptanalysis has become a much more difficult task but new cryptanalytic methods and attacks are constantly being developed and refined; some of them are studied in subsequent chapters. The recent publication of several books integrally devoted to cryptanalysis bears witness to this development, see for example [9, 106, 186, 190, 194, 201].

Finally, we mention the fact that our examples show that the classical ciphers we have considered can be totally broken, i.e., the discussed attacks allow full recovery of the plaintext or even the key. It is then clear that these ciphers are extremely insecure but one should not draw the conclusion that an encryption scheme is secure whenever such a total break is not possible. Far from it, we shall see in the following chapters that much stricter conditions are required for an encryption scheme to be regarded as secure, which essentially amount to demanding that no resource-bounded adversary can obtain any information whatsoever about the plaintext.

Chapter 2
Basic Concepts from Probability, Complexity, Algebra and Number Theory

In this chapter we study the basic notions from mathematics and computer science that we will be using throughout the book. Its purpose is not to give a comprehensive account of these concepts but, rather, to study them from the point of view of their applications to cryptography, emphasizing the algorithmic aspects. For this task we will be using Maple as a tool to implement the relevant algorithms and to experiment with them. Bearing in mind these premises, we include here the required concepts and results from probability, complexity theory and modular arithmetic, including quadratic residues and modular square roots, and we also give a brief introduction to the relevant aspects of group theory and finite fields. Thus, for readers already familiar with these subjects, this chapter will hopefully serve as a refresher of the topics mentioned in the Preface (with the exception of linear algebra which is not included here) and, at the same time, as an introduction to the use of Maple to explore the algorithms discussed. Even for readers not familiar with these subjects we hope that sufficient information is included here to allow them to profitably read the rest of the book. We include proofs of most of the algorithmic and mathematical results mentioned, with pointers to the literature for the reader who wishes to go deeper on these topics and, at the same time, we will be introducing here a great deal of the notation used throughout the book.

2.1 Basic Probability Theory

Randomness plays a crucial role in cryptology and in order to deal with this concept we need some basic notions from probability theory. We assume the reader already knows the elements of this theory but, in order to fix the notation, we recall here some of the most important probability-related concepts that we will use. We refer to any introductory text on (discrete) probability such as, for example, [94, 167] for more detailed information on these topics.

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_2, © Springer-Verlag Berlin Heidelberg 2013

Definition 2.1 A finite *probability space* is a pair $(\Omega, \Pr)$, where Ω is a finite set (called the *sample space* and whose elements are called *elementary events*) and

$$\Pr : \Omega \to \mathbb{R}$$

is a *probability distribution*, i.e., a function that satisfies the following conditions:

1. $0 \leq \Pr(\omega) \leq 1$ for every $\omega \in \Omega$,
2. $\sum_{\omega \in \Omega} \Pr(\omega) = 1$.

If we run a random experiment, then every elementary event of the sample space represents a possible outcome. The most important probability distribution used in cryptography is the *uniform distribution* where all the elementary events occur with the same probability, i.e., $\Pr(\omega) = \frac{1}{|\Omega|}$ for every $\omega \in \Omega$.

For example, rolling a die can be interpreted as an experiment on the sample space $\Omega = \{1, 2, \ldots, 6\}$ and, assuming that the die is fair, each possible outcome is equally likely and hence the probability of each outcome is $\Pr(n) = \frac{1}{6}$ for each $n \in \Omega$ (in other words, the distribution is uniform).

The assignment of probabilities can be extended to subsets $E \subseteq \Omega$, which are called *events*, by setting $\Pr(E) = \sum_{\omega \in E} \Pr(\omega)$. For the uniform distribution, $\Pr(E) = \frac{|E|}{|\Omega|}$ is just the ratio of the number of "favorable cases" to the number of all "possible cases" as in Laplace's original definition of probability.

The complement of the event $E \subseteq \Omega$ is the set $\bar{E} = \Omega - E$ and it is clear that $\Pr(\bar{E}) = 1 - \Pr(E)$. From Definition 2.1 have that $\Pr(\Omega) = 1$ and then $\Pr(\emptyset) = 0$. More generally, if we have two events $E, F \in \Omega$, then the probability of the *union event* $E \cup F$ is given by

$$\Pr(E \cup F) = \Pr(E) + \Pr(F) - \Pr(E \cap F).$$

For example, if Ω consists of the possible outcomes of rolling a die, then the event E described by "an even number is rolled" has probability $\frac{1}{2}$ and the event F given by "a multiple of 3 is rolled" has probability $\frac{1}{3}$. The probability of $E \cap F$ is just the probability of rolling a 6 and hence equal to $\frac{1}{6}$, so that the probability of the event $E \cup F$ is $\frac{1}{2} + \frac{1}{3} - \frac{1}{6} = \frac{2}{3}$.

The next definition captures the idea that two events may be such that the probability of one of them does not influence the probability of the other:

Definition 2.2 Two events $E, F \in \Omega$ are said to be *independent* if

$$\Pr(E \cap F) = \Pr(E) \cdot \Pr(F).$$

In other words, E and F are independent when the probability of them both occurring is the product of their individual probabilities of occurring.

Exercise 2.1 Suppose that two fair dice are rolled in succession and let E be the event that the first die is a 3 and F the event that the sum of the dice is 5. Determine whether these events are independent.

It is often necessary to compute the probability of an event F occurring when it is known that another event E has occurred. This is called the *conditional probability of F on E*:

Definition 2.3 If E, F are events such that $\Pr(E) > 0$ then the *conditional probability of F on E*, $\Pr(F|E)$, is defined by:

$$\Pr(F|E) = \frac{\Pr(F \cap E)}{\Pr(E)}.$$

The intuitive idea behind this definition is that, if we assume that E occurs, our sample space is now E instead of Ω and in this smaller space we compute the ratio between the probability of the part of F included in E and the probability of E itself. Observe that, in particular, the independence of E and F is equivalent to $\Pr(F|E) = \Pr(F)$ and also to $\Pr(E|F) = \Pr(E)$.

In our earlier example where Ω consists of the possible outcomes of rolling a die, E is the event "an even number is rolled" and F the event "a multiple of 3 is rolled" we have that $\Pr(F|E) = \frac{1}{3}$, while $\Pr(E|F) = \frac{1}{2}$.

As an immediate consequence of the previous definition we have:

Lemma 2.1 (Bayes' formula) *Let $E, F \in \Omega$ be events such that $\Pr(E) > 0$, $\Pr(F) > 0$. Then*

$$\Pr(E|F) = \frac{\Pr(F|E)\Pr(E)}{\Pr(F)}.$$

Proof From the definition of conditional probability we have that $\Pr(F|E)\Pr(E) = \Pr(F \cap E) = \Pr(E|F)\Pr(F)$. □

The following proposition gives another version of Bayes' formula which exploits the fact that an event is the union of the two disjoint events given by its intersection with another event and its complement:

Proposition 2.1 *Let $E, F \subseteq \Omega$ be two events such that $\Pr(E) > 0$, $\Pr(F) > 0$. Then*

(i) $\Pr(F) = \Pr(F|E)\Pr(E) + \Pr(F|\bar{E})(1 - \Pr(E))$.

(ii) $\Pr(E|F) = \dfrac{\Pr(F|E)\Pr(E)}{\Pr(F|E)\Pr(E) + \Pr(F|\bar{E})(1 - \Pr(E))}$.

Proof For (i), we have that $F = (F \cap E) \cup (F \cap \bar{E})$, where the union is disjoint. Thus we see that $\Pr(F) = \Pr(F \cap E) + \Pr(F \cap \bar{E}) = \Pr(F|E)\Pr(E) + \Pr(F|\bar{E})(1 - \Pr(E))$, where the last equality follows from the definition of conditional probability and from the fact that $\Pr(\bar{E}) = 1 - \Pr(E)$. (ii) follows from (i) using Bayes' formula. □

When running a random experiment on a finite sample space we might be more interested in some values that depend on the outcome of the experiment (for example, in the net gain in a game of chance) than in the outcome itself. This leads to the

definition of a real-valued *random variable* as a function defined on the sample space:

Definition 2.4 Let $(\Omega, \Pr)$ be a finite probability space. A *random variable* is a function $X : \Omega \to \mathbb{R}$.

Since the sample spaces we are using are finite, a random variable takes only a finite number of values. Random variables can be used to define events related to a real number $x \in \mathbb{R}$ such as $\{\omega \in \Omega | X(\omega) \leq x\}$, $\{\omega \in \Omega | X(\omega) > x\}$ or $\{\omega \in \Omega | X(\omega) = x\}$. The probabilities of these events are denoted simply by $\Pr(X \leq x)$, $\Pr(X > x)$, and $\Pr(X = x)$, respectively. The expected value (or expectation, or mean) of a random variable is the average of its values weighted by their probability of occurrence:

Definition 2.5 Let X be a random variable taking $x_1, x_2, \ldots, x_n$ as values. The *expected value* or *mean* of X is defined as: $\mathrm{E}(X) = \sum_{i=1}^{n} x_i \cdot \Pr(X = x_i)$.

Exercise 2.2 Let X be the random variable defined by the sum of the numbers appearing on two rolled fair dice. Compute the expected value of X.

Let us see now how the expected value of a random variable may be used to estimate the number of trials in an experiment. For example, suppose that a randomized algorithm (an algorithm that accepts random bits as input, see Sect. 2.3; a typical example is an algorithm that searches for primes among randomly chosen integers) produces some desired result with probability p. A natural question is then: How many times must the algorithm be executed, on average, to obtain this result?

The following proposition shows that the number of times the algorithm must be run is $1/p$:

Proposition 2.2 *Suppose that an event $E \subseteq \Omega$ occurs with probability p. Then the expected value of the number of trials until E occurs is $1/p$.*

Proof Let $E \subseteq \Omega$ be an event and $p = \Pr(E)$. Let X be a random variable that gives the average number of independent trials until E occurs and $\mu = \mathrm{E}(X)$. We see that $\Pr(X = 1) = p$ and $\Pr(X > 1) = 1 - p$ so that, with probability p, E occurs already on the first trial and with probability $1 - p$ one needs the average value of μ trials after the first (unsuccessful) trial, i.e., a total of $1 + \mu$ trials. Therefore, $\mu = 1 \cdot p + (1 + \mu) \cdot (1 - p)$, which implies that $\mu = 1/p$. □

Example 2.1 A *Bernoulli* experiment is a random experiment such that the outcome of a single trial is either 1 (success) with probability p or 0 (failure) with probability $1 - p$. If a sequence of independent Bernoulli trials with success probability p is performed, then we are in the situation described in Proposition 2.2. Although we are not going to explain density and distribution functions here (see, for example, [167] for more details), we mention that the *geometric* distribution estimates the number of failures there will be before the first success is achieved in a sequence of independent Bernoulli trials (more generally, the Pascal distribution estimates the

number of failures before r successes). This and many other probability distributions are implemented in Maple's package `Statistics`. In particular, Maple gives us the result in the above proposition by setting:

```
> with(Statistics):
  X := RandomVariable(Geometric(p));
  ExpectedValue(X);
                                  (-p + 1)/p
```

Thus $(1/p) - 1$ failures are expected before obtaining the first success, i.e., the expected number of trials for the first success is $1/p$.

2.2 Integers and Divisibility

Let $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ be the set of integers. The usual operations of addition and multiplication are defined in $\mathbb{Z}$ and give it the structure of a commutative ring with identity (see the definition in Sect. 2.5 or in [45, 48]). In fact, $(\mathbb{Z}, +, \cdot)$ is the prototypical ring from which this concept arises by abstraction. Actually, $\mathbb{Z}$ is a special type of commutative ring called a (commutative) domain because it has no *zero divisors*, i.e., there are no nonzero elements $a, b \in \mathbb{Z}$ such that $ab = 0$. We assume that the basic properties of addition and multiplication (associativity, commutativity, distributivity, etc.) are well known and start by studying the concept of divisibility.

Definition 2.6 Let $a, b \in \mathbb{Z}$ such that $a \neq 0$. We say that *a divides b* (or, in other words, that *b is a multiple of a*) if there exists an integer q such that $b = aq$.

If a divides b then a is called a *divisor* of b and we write $a|b$. If a is not a divisor of b we write $a \nmid b$. Some of the most basic properties of divisibility are collected in the following proposition whose proof is left as an exercise.

Proposition 2.3

(i) If $a \mid b$ and $b \mid c$ then $a \mid c$ (transitivity).
(ii) If $a \mid b$ and $b \mid a$ then $a = \pm b$.
(iii) If $a \mid b$ and $a \mid c$, then $a \mid (bx + cy)$ for all $x, y \in \mathbb{Z}$.

Next, we recall a couple of important results whose proofs are straightforward and can be found in any introductory number theory textbook (see, for example, [77]). We start with the following fact, which shows that even if $b \nmid a$, we can divide b into a obtaining a quotient and a remainder:

Theorem 2.1 (Division algorithm) *Let $a, b \in \mathbb{Z}$ such that $b > 0$. Then there exist unique integers q and r such that $a = bq + r$ and $0 \leq r < b$.*

The integers q and r are, respectively, the *quotient* and the *remainder* when a is divided by b. The division algorithm is implemented in Maple by means of the functions `irem` and `iquo`. For example

```
> iquo(25,7,'r'), r;
                                    3, 4
```

shows that the quotient of dividing 25 by 7 is 3 and the remainder is 4. The quotient is the output of the function and the remainder is stored in the variable `'r'`. Similarly, if we compute

```
> irem(25,7,'q'), q;
                                    4, 3
```

we obtain as output the remainder and the quotient is stored in the variable `'q'`.

We recall the definition of a prime number, a concept that plays an important role in modern cryptography:

Definition 2.7 An integer p is called a *prime number* if $p > 1$ and the only positive integers dividing p are 1 and p. A *composite number* is an integer greater than 1 that is not prime.

The first primes are $2, 3, 5, 7, 11, 13, \ldots$ and there are arbitrarily large primes because, as Euclid already knew more than 2300 years ago (Elements, Book IX, Proposition 20), the number of primes is infinite [194]. Another important fact that Euclid also knew is:

Theorem 2.2 (The fundamental theorem of arithmetic) *Let $a > 1$ be an integer. Then a can be written as a product, $a = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, with p_i prime and $e_i \geq 1$ for $i = 1, \ldots, k$. Furthermore, this factorization in prime powers is unique if the primes are written in increasing order.*

As we shall see, the problem of finding the prime factorization of a large integer is thought to be computationally hard and this fact turns out to be very useful for cryptography.

2.2.1 Representation of Integers

The usual method that we use to represent integers is by means of their decimal expansion. But computers work naturally with the binary expansion and any integer $b > 1$ can be used as the *base* of a number system. These representations rely on the following theorem in which we use the notation $\lfloor x \rfloor$ to denote the *floor* of a real number x, i.e., the greatest integer less than or equal to x.

Theorem 2.3 (Positional representation of numbers) *Let $b > 1$ be an integer. Then each positive integer n has a unique representation in the form*

$$n = \sum_{i=0}^{k-1} d_i b^i,$$

where $k > 0$, the d_i are integers such that $0 \leq d_i \leq b-1$ and $d_{k-1} \neq 0$. Moreover, $k = \lfloor \log_b n \rfloor + 1$.

Proof We use the division algorithm (Theorem 2.1) to build the representation. We successively divide:

$$\begin{aligned} n &= bq_0 + d_0, \\ q_0 &= bq_1 + d_1, \\ q_1 &= bq_2 + d_2, \\ &\cdots \\ q_{i-1} &= bq_i + d_i, \end{aligned}$$

and note that $0 \leq d_i < b$ and $n > q_0 > q_1 \cdots \geq 0$. This strictly decreasing sequence of non-negative integers must reach the value 0, so let k be the first index such that $q_{k-1} = 0$. Then the above sequence of identities ends in $q_{k-2} = d_{k-1}$ which is a positive integer. Now, starting at the first identity and replacing each q_i by its value given by the next identity we obtain the required expression for n. The uniqueness of this expression is left as an exercise. Finally, note that $b^{k-1} \leq n < b^k$ and, taking base-b logarithms, we obtain the chain of inequalities $k - 1 \leq \log_b n < k$ which is clearly equivalent to $k = \lfloor \log_b n \rfloor + 1$. □

From this theorem it follows that each positive integer has a representation of the form $n = (d_{k-1}, d_{k-2}, \ldots, d_1, d_0)_b$ which, if there is no ambiguity, is also written as $d_{k-1}d_{k-2}\ldots d_1d_0$. It is called the *b-adic expansion* of n and its terms are called *digits* while its *length* is k. Notable special cases are the bases $b = 10$ (decimal), $b = 2$ (binary), $b = 8$ (octal) and $b = 16$ (hexadecimal or, simply, hex). If $b \leq 10$, then the base-b digits are represented by the usual decimal digits which are $<b$. In particular, the binary digits, or bits, are 0, 1, and we have the binary expansion $d_{k-1}d_{k-2}\ldots d_1d_0 \in \{0, 1\}^k \subset \{0, 1\}^*$, where $\{0, 1\}^k$ denotes the set of binary strings of length k and $\{0, 1\}^*$ the set of all (finite-length) binary strings. For other bases, additional symbols (usually letters) are used and, for example, hexadecimal digits are given, in increasing order, by 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f (sometimes upper-case letters are used instead of lower-case ones).

The proof of the preceding theorem provides an algorithm to compute the b-adic expansion of an integer n by means of successive integer divisions by b and this algorithm is used whenever we want to do a "base change", for example, for passing from the decimal expansion to the binary one. In some cases this is particularly easy. If $a = b^k$ then conversion from base a to base b is just a matter of replacing each base-a digit by its k-digit representation in base b, and conversely to pass from base b to base a. For example, to convert between hexadecimal and binary we only have to replace each hex digit by a corresponding 4-bit block (sometimes called a *half-byte* or a *nibble*), where 0 goes to 0000, 1 to 0001, and so on, until f which goes to 1111. For the inverse conversion one should group the bits in blocks of 4 and do the replacement, bearing in mind that if the first group from the left has fewer than 4 bits then it should be completed with zeros added to the left. So, for example, the binary

number 1011101 is converted to hexadecimal by replacing 0101 by 5 and 1101 by d, so that its hexadecimal representation is $5d$.

Maple has commands for converting between representations of integers in different bases. The generic conversion function is `convert/base` which converts between two bases a and b by taking the list of digits in base a (if $a = 10$ then the digits can also be given in the usual form instead of as a list) and converting it to the list of digits of the same number in base b. It should be noted that these Maple lists follow the *little-endian* convention in which the most significant digit is the one on the right (the last one on the list). For example,

```
> convert([1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1], base, 2, 16);
                                [9, 11, 4, 11, 3]
```

passes from the list of binary digits to the corresponding list of hex digits. Note that in this case the letters a, b, c, d, e, f are not used for the hex digits and their decimal form (with two decimal digits each) is used instead. These letters are used, however, when converting from decimal to hex by means of another variant of the `convert` function called `convert/hex`. We use this function together with `convert/binary` to convert the binary number whose bits are given by the list above to hex:

```
> convert(111011010010111001, decimal, binary);
                                     242873
> convert(%,hex);
                                     3B4B9
```

When using these functions in Maple, it should be taken into account that these representations of a number are not, in general, valid replacements for the number itself. For example, consider:

```
> x := convert(5, hex);
                                       5
> whattype(x);
                                    symbol
```

In this case, the result of converting the integer 5 to hex is a symbol, i.e., no longer a number and, for example, we could not perform arithmetic operations with it anymore. But even more caution is required with the following conversion:

```
> y := convert(5, binary);
                                      101
> whattype(y), irem(y, 5);
                                  integer, 1
```

Now, the result of converting 5 to binary is indeed an integer but this integer is no longer 5 but 101, i.e., the number whose decimal representation has the same digits as the binary representation of 5 …

2.3 Basic Computational Complexity

As we shall see later, the security of cryptosystems is often based on the assumed hardness of certain computational (mostly number-theoretic) problems. It is therefore important to have procedures to evaluate the hardness of computational problems

and to classify these problems according to their difficulty. An intrinsic estimation of the difficulty of a problem is hard to come by and so usually one has to settle for estimating the cost of solving the problem by means of a specific algorithm. Computational complexity theory classifies algorithms in terms of the resources (such as time, space, etc.) needed to run them. The most important of these resources is time, which is usually evaluated by means of a function that estimates how time varies as a function of input size. This function does not attempt to measure the real time the algorithm will spend on a given input but, rather, the number of *elementary operations* used, which will be roughly proportional to the time actually spent (with the proportionality constant depending on many additional factors, such as the power of the machine or machines running it, the specifics of the implementation, etc.). The important thing here is that this estimation should be precise enough to allow us to predict, given the time required to execute the algorithm with a given input on a concrete machine, how long will it take to run the algorithm with the same implementation and a larger input.

2.3.1 Asymptotic Notation

In order to achieve the estimation just mentioned it will be enough to know the *asymptotic behavior* of the running time function, i.e., the growth rate of this function as the input size gets large. This growth rate might be a complicated function as it will depend on many different factors, but the idea is to approximate it by simpler functions. For this, the following asymptotic notation is used:

Definition 2.8 Let $f, g : \mathbb{N} \to \mathbb{R}^+$ (where $\mathbb{R}^+$ denotes the positive reals; more generally, we can consider functions defined for sufficiently large positive integers that are eventually positive, i.e., $f(x), g(x) > 0$ for sufficiently large x). Then we say that:

1. $f = O(g)$ (f *is big-O of* g) if there exists $c \in \mathbb{R}^+$ and $x_0 \in \mathbb{N}$ such that $f(x) \leq cg(x)$ for all $x \geq x_0$. In this case we also say that f *is of order* g .
2. $f = \Omega(g)$ (f is *big-Omega* of g) means that there exists $c \in \mathbb{R}^+$ and $x_0 \in \mathbb{N}$ such that $f(x) \geq cg(x)$ for all $x \geq x_0$, equivalently, $g = O(f)$.
3. $f = \Theta(g)$ (f is *big-Theta* of g) means that there exist $c, d \in \mathbb{R}^+$ and $x_0 \in \mathbb{N}$ such that $cg(x) \leq f(x) \leq dg(x)$ for all $x \geq x_0$, equivalently, $f = O(g)$ and $g = O(f)$.
4. $f = o(g)$ (f *is little-o of* g) if $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$, equivalently, for every $\varepsilon > 0$, $f(x) \leq \varepsilon g(x)$ for every sufficiently large x. Intuitively, this means that g grows much faster than f and it clearly implies that $f = O(g)$.
5. $f \sim g$ if $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$. In this case we say that f is *asymptotically equal* (or *asymptotic*) to g.

Remarks 2.1

1. The big-O notation denotes a binary relation between the functions f and g that is not symmetric in general, i.e. $f = O(g)$ does not imply $g = O(f)$ or, in other words, $f = O(g)$ does not imply $f = \Omega(g)$. For example, we have that $x\sqrt{x} = O(x^2)$ (interpreting these expressions as functions in the obvious way) but it is not true that $x^2 = O(x\sqrt{x})$. In general, the idea of using the big-O notation is that, whenever we write $f = O(g)$, g should be a simpler function than f that gives us a good approximation to how fast f grows asymptotically.
2. The functions in Definition 2.8 can be functions of several variables, i.e., defined over subsets $X, Y \subseteq \mathbb{N}^k$. Then we write, for example, $f = O(g)$ if there is a positive constant c and an integer t such that for all $(x_1, \ldots, x_k) \in \mathbb{N}^k$, with $x_i \geq t$, $1 \leq i \leq k$, the following conditions hold:

   a. $(x_1, \ldots, x_k) \in X \cap Y$, i.e., both functions are defined on $(x_1, \ldots, x_k)$,
   b. $f(x_1, \ldots, x_k) \leq cg(x_1, \ldots, x_k)$.

Examples 2.1

1. If f is a polynomial of degree d with positive leading coefficient (see Sect. 2.8.2 for the definition of leading coefficient), then $f = O(x^d)$ and, in fact, $f = \Theta(x^d)$.
2. If f and g are polynomials of the same degree and with the same (positive) leading coefficient, then $f \sim g$.
3. If f and g are polynomials with positive leading coefficient and the degree of f is smaller than the degree of g, then $f = o(g)$.
4. When we write $O(g)$, this expression denotes an anonymous function f such that $f = O(g)$ and similarly for the other symbols in Definition 2.8. Thus $O(1)$ denotes a function bounded by a constant and $o(1)$ denotes a function that tends to zero as x tends to infinity. For example, $f(x) \sim g(x)$ is equivalent to $f(x) = (1 + o(1))g(x)$.
5. If ε is a positive constant, then $\ln x = O(x^\varepsilon)$ and, in fact, $\ln x = o(x^\varepsilon)$ too. This is because $\lim_{x\to\infty} \frac{\ln x}{x^\varepsilon} = 0$, i.e., the logarithmic function grows slower than any positive power.
6. If b is an integer ≥ 2 and $f(x)$ denotes the length of the b-adic expansion of x (i.e., the number of base-b digits of x), then $f(x) = O(\ln x)$. This is because, by Theorem 2.3, $f(x) = \lfloor \log_b x \rfloor + 1 \leq \log_b x + 1 = \frac{\ln x}{\ln b} + 1$. This can be interpreted as: "the number of digits grows like the logarithm" (and the logarithm base does not matter because logarithms to different bases are proportional). In fact, it is clear that we also have that $f(x) = \Theta(\ln x)$.
7. The length of an integer n, denoted $\text{len}(n)$ is the number of bits of n and, according to the preceding example, $\text{len}(n) = \Theta(\ln n)$, so that $\text{len}(n)$ and $\ln n$ can be interchanged when appearing within the big-O notation.
8. The big-O notation will be used to estimate the running time of algorithms as a function of the input size. When the input is an integer, the input size is, by definition, $\text{len}(n)$. Thus the running times of these algorithms will be estimated as functions of $\text{len}(n)$ or, equivalently, of $\ln n$.

9. If $\pi(x)$ denotes the number of primes $\leq x$, then $\pi(x) \sim \frac{x}{\ln x}$ and $\pi(x) \sim \mathrm{li}(x)$, where $\mathrm{li}(x) = \int_2^x \frac{dt}{\ln t}$ is the "logarithmic integral". This is the *Prime Number Theorem* (PNT) which is important for cryptography and is explained in more detail in Chap. 6 (see Theorem 6.2).

Exercise 2.3

(i) Prove that if f is a polynomial of degree d with positive leading coefficient, then $f = \Theta(x^d)$.
(ii) Prove that $\mathrm{len}(n) = \Theta(\ln n)$.
(iii) Using l'Hôpital's rule, prove that $\lim_{x\to\infty} \frac{\ln x}{x^\varepsilon} = 0$ for any positive ε, and hence that $\ln x = o(x^\varepsilon)$.
(iv) Prove that if $\lim_{x\to\infty} \frac{f(x)}{g(x)} = c$ where c is any non-negative constant, then $f = O(g)$ and hence that, in particular, $f \sim g$ implies $f = \Theta(g)$.

2.3.2 *Efficient Computation and $\mathcal{P}$ Versus $\mathcal{NP}$*

Here we give a brief overview of complexity theory, focusing on some aspects of cryptologic interest. For detailed analyses of the complexity of many number-theoretic algorithms we refer to [180] and for a general presentation of complexity theory to [6].

As already mentioned, we are going to use the asymptotic notation to express estimates of the running times of some number-theoretic algorithms which have important cryptologic applications. For this we need to count the number of "elementary operations" performed as a function of input size. The algorithms we are going to deal with will typically accept an integer (or several integers) as input, so the input size(s) will be the binary length(s) of the involved integer(s). We will assume our integers are given in binary and the basic operations will be the arithmetic operations with integers of one bit (see [118, 180] for details). This means that our estimates will be *bit complexity* estimates.

It is not difficult to see that performing addition of two integers n, m can be done in $O(\max\{\mathrm{len}(n), \mathrm{len}(m)\})$ bit operations (where $\mathrm{len}(n)$ denotes, as before, the binary length of n) which is the same as $O(\max\{\ln n, \ln m\})$. Indeed, this follows from the fact that to perform the addition, we scan the binary representation of both integers from right to left adding bit by bit with carry. The same running time estimate applies to subtraction and, for multiplication, note that in order to multiply a k-bit integer m by an l-bit integer n by means of the usual grade-school algorithm, we have to perform at most $l - 1$ additions each of which requires k bit operations, obtaining the estimate $O(kl)$ or, in terms of the logarithms, $O(\ln m \ln n)$.

We should mention here that there are faster algorithms for multiplication, although their advantage is only realized in practice when multiplying very large numbers. Thus, for example, the multiplication algorithm of Schönhage and Strassen (see, for example, [6, 180]) can multiply two k-bit numbers in time $O(k \ln k \ln\ln k)$ which is better than the previous estimate of $O(k^2)$ as it is even better than $O(k^{1+\varepsilon})$ for any $\varepsilon > 0$. The running time of division with remainder, where a k-bit integer is

to be divided by an l-bit integer with $k \geq l$ is $O(l(k-l+1))$ which is also $O(kl)$ but may be significantly less than the latter. Thus, if $a > b$, dividing a by b with quotient q and remainder r requires time $O(\text{len}(b)\text{len}(q))$.

We are now ready to introduce the important notion of *polynomial time* (or polytime as it is sometimes called for brevity).

Definition 2.9 A function $f : \mathbb{N} \to \mathbb{R}^+$ is called polynomial (or polynomially bounded) if $f(x) = O(x^c)$ for some positive integer c. An algorithm is said to be a *polynomial-time algorithm* if its running time is a polynomial function of the input size.

Since we will be mainly interested in algorithms that accept integers as input we remark that, for these algorithms, the previous definition may be reformulated as follows:

Definition 2.10 An algorithm that accepts as input integers $n_1, \ldots, n_t$ (given in positional notation) is said to *run in polynomial time* if there are non-negative integers $c_1, \ldots, c_t$ such that the running time of the algorithm is

$$O(\text{len}(n_1)^{c_1}\text{len}(n_2)^{c_2}\cdots\text{len}(n_t)^{c_t}) = O(\ln^{c_1} n_1 \ln^{c_2} n_2 \cdots \ln^{c_t} n_t).$$

In particular, if the input is a unique integer n, the running time is $O(\text{len}(n)^c) = O(\ln^c n)$ (where $\ln^c n$ is an abbreviated notation for $(\ln n)^c$).

We will sometimes use the notation polylog(n) to denote a generic polynomial function in $\ln n$, so that $O(\text{polylog}(n))$ just means $O(\ln^c n)$ for some constant c. Note that the definition of polynomial time just means that the running time is $O(f(\text{len}(n)))$, where f is a polynomial function.

We see that the algorithms corresponding to the basic arithmetic operations all run in polynomial time. Let us give an example of an algorithm that does not.

Example 2.2 The usual algorithm for computing $n!$ runs on time $O(n^2 \ln^2 n)$. Indeed, we do $n-2$ multiplications in which one of the factors is bounded by $n!$ and the other one bounded by n. The length of $n!$ (which is the product of fewer than n numbers each of which is less than or equal to n) is $O(n \ln n)$ so each of these $n-2$ multiplications requires $O(n \ln^2 n)$ bit operations. Thus the total time for computing $n!$ is $O(n) \cdot O(n \ln^2 n) = O(n^2 \ln^2 n)$. Notice that this algorithm is not a polynomial-time algorithm. If we express the running time as a function of the logarithm of n, we see that this time is $O(e^{2 \ln n} \ln^2 n)$.

Definition 2.11 An algorithm that accepts an integer n as input runs in *exponential time* if its running time is $O(n^c) = O(e^{c \ln n})$ for some constant $c > 0$.[1] More generally, we can say that an algorithm runs in exponential time when it has a time

[1] Thus the algorithm used to compute $n!$ is indeed exponential because $O(\ln^2 n) = O(n^\varepsilon)$ and so the running time of the algorithm is $O(n^2 \ln^2 n) = O(n^{2+\varepsilon})$.

estimate of the form $O(e^{ck})$, where k is the total binary length of the integers to which the algorithm is being applied [119].

Polynomial-time algorithms are considered efficient and computational problems that may be solved by these algorithms are considered "feasible" or even "easy". Of course, a specific polynomial-time algorithm might have a very large exponent and hence not be very efficient, but, in general, almost all polynomial-time algorithms that arise naturally have a low exponent and are really efficient. In addition, computational problems for which no polynomial-time algorithms are known are considered hard, and they are usually hard in practice. Thus the concept of polynomial time seems to capture the algorithms that are efficient in practice and both concepts are regarded as synonymous, as we will do in this book.

Besides the problems for which only exponential-time algorithms are known, there are other intermediate problems which are easier than exponential but harder than polynomial time and, in fact, some of the most important "hard problems" used in cryptography are of this type. In order to deal with algorithms whose complexity is intermediate between polynomial and exponential, the so-called L-notation is often used. Let n be a large positive integer that can be thought of as the input of an algorithm, γ a real number in the interval $[0, 1]$, and $c > 0$ a constant. We define:

Definition 2.12 The *subexponential function* with parameters $\alpha \in [0, 1]$ and $c > 0$ is

$$L_n[\alpha, c] = e^{c(\ln n)^{\alpha}(\ln \ln n)^{1-\alpha}}.$$

We will, in particular, write $L(n) = L_n[1/2, 1] = e^{\sqrt{(\ln n)(\ln \ln n)}}$.

We see that, in particular:

- $L_n[0, c] = e^{c(\ln \ln n)} = (\ln n)^c$,
- $L_n[1, c] = e^{c(\ln n)} = n^c$.

Thus an algorithm whose running time is $O(L_n[0, c])$ is a polynomial-time algorithm while an algorithm with running time $O(L_n[1, c])$ is exponential. This leads to the following definition:

Definition 2.13 An algorithm which takes as input a positive integer n runs in *subexponential time* if its running time is $O(L_n[\alpha, c])$ for some $0 < \alpha < 1$.

Thus a subexponential algorithm is one whose running time is somewhat intermediate between polynomial and exponential or, more precisely, it is both $\Omega((\ln n)^c)$ for every $c \in \mathbb{R}^+$ (and hence slower than polynomial) and $O(n^c)$, for every $c \in \mathbb{R}^+$ (and hence faster than exponential). We will see that the most efficient algorithms known for certain hard problems on which several public-key cryptographic schemes are based run in subexponential time[2] $O(L_n[1/3, (64/9)^{1/3} + o(1)])$.

We have mentioned that computational complexity theory tries to classify computational problems and the algorithms used to solve them but we have not given a

[2] This estimate is *heuristic* in the sense that it relies on some widely believed but unproven conjectures and it agrees well with the results obtained in practice.

precise definition of what a computational problem is. This requires the formulation of a mathematical model of computation and for this there are several possibilities, with the computational model of choice being based on *Turing machines*. Fortunately, it can be shown that the classification provided by complexity theory is largely independent of the computational model used. More precisely, the *Church–Turing Thesis* basically says that all computational models are equivalent in the sense that a function computable in one model is also computable in the other models and the computational complexities defined by the different models are "polynomially related" so that polynomial time, and hence efficient computation, can be defined independently of the adopted model. Thus we do not explicitly choose any one model and we simply rely on the asymptotic notation previously introduced to give estimates of the hardness of the problems. We refer the interested reader to [6] for more detailed complexity theory notions and to [169, 191] for a more specific study of the relation between complexity and cryptography.

When dealing with computational problems, it is easier to look at *decision problems* which are those that admit only a yes/no answer or, in other terms, are solved by an algorithm that produces as output a unique bit. This might seem too restrictive but, in practice, it is not so because the computational problems we are interested in can often be formulated as decision problems in such a way that an efficient algorithm for the latter gives an efficient algorithm for the former and vice versa.

When classifying computational problems in terms of their relative difficulty it is useful to include them in *complexity classes*, some of which we are now going to define. We will consider only decision problems but other kinds of problems may also be similarly classified.

Definition 2.14 A decision problem belongs to the class $\mathcal{P}$ if there exists a deterministic polynomial-time algorithm that solves it. We usually say that these problems are *solvable in polynomial time*.

There are some decision problems that, while seemingly non-solvable in polynomial time, have the property that a solution can be verified in polynomial time:

Definition 2.15 A decision problem belongs to the class $\mathcal{NP}$ (non-deterministic polynomial time) if a 'yes' answer can be verified in polynomial time given some extra information, called a *certificate* or a *witness*. Similarly, the decision problem belongs to the class co$\mathcal{NP}$ if a 'no' answer can be verified in polynomial time given a certificate.

Example 2.3 (**The integer factorization problem**)
We next give a brief summary of the *integer factorization problem* because of its great importance for cryptography; algorithms to solve this problem are studied in Chap. 6. This is the computational problem of, given a positive integer n as input, computing the prime factorization of n. It can be reduced to the *factorization search problem* which is the following: Given an integer $n > 1$, find a nontrivial factor a of n (i.e, an integer a such that $1 < a < n$ and there exists an integer b with $n = ab$) or determine that no such factor exists (i.e., that n is prime).

The factorization problem can be reduced to the corresponding search problem in the sense that there is a polynomial-time algorithm to transform an algorithm to solve the second problem into an algorithm to solve the first. In order to obtain the factorization of n we would repeatedly call the search algorithm to find factors of n until obtaining all the factors, i.e., the search algorithm would act as a subroutine of the factoring algorithm. The important thing is that, since clearly the number of prime factors of n is $O(\text{len}(n))$, the search algorithm must be called a number of times which is polynomial in the size of n (and the factoring algorithm also has to do the corresponding divisions by the factors found). This means that solving the factorization problem is no harder than solving the search problem, up to a polynomial-time amount of computation. As we shall see, the numbers for which the factorization problem has cryptologic interest are the product of just two primes and, in this case, the factorization algorithm has the same running time as the search algorithm.

Since the definitions of $\mathcal{P}$ and $\mathcal{NP}$ refer to decision problems, we will now consider the *integer factorization decision problem* (see, e.g., [119]) which asks, given positive integers n, k with $k < n$, if there exists a factor a of n such that $2 \leq a \leq k$. Again, we can see that the search factorization problem (and hence also the factorization problem) is reducible in polynomial time to the corresponding decision problem. In this case, the reduction process consists of applying a *binary search* to pinpoint a nontrivial factor (actually the smallest one) by calling the algorithm solving the decision problem $O(\text{len}(n))$ times. Indeed, suppose that $l = \text{len}(n)$, so that 2^l is the smallest power of 2 which is greater than n. We first call the decision problem algorithm with $k = 2^{l-1} - 1$. If the answer is 'no', i.e., if n does not have nontrivial factors $\leq k$, then we know that n is prime and we are done, otherwise we call the decision algorithm again with $k = 2^{l-2} - 1$. If the answer is 'yes' then we know that n has a factor whose binary length is $< l - 1$ whereas if the answer is 'no', then it has a factor of length $l - 1$. Thus we have determined the bit d_{l-2} of the binary expansion $d_{l-2}d_{l-3} \ldots d_1 d_0$ of the factor, which is 0 in the first case and 1 in the second. We continue calling the decision problem, each time halving the size of the interval where the nontrivial factor of n must lie by taking the midpoint of the previous interval as the value of k, so that the factor is in the lower half interval if the answer is 'yes' and in the upper half if the answer is 'no'. Thus the next bit, d_{l-3}, is determined by calling the decision algorithm again with $k = 2^{l-3} - 1$ (in case the previous application of the algorithm gave a 'yes' answer) and with $k = 2^{l-2} + 2^{l-3} - 1$ (in case the previous answer was 'no'). Again, a 'yes' answer means that $d_{l-3} = 0$ and a 'no' answer that $d_{l-3} = 1$ and, after calling the algorithm l times, we will have computed the binary expansion of the smallest nontrivial factor of n (in case n is not prime).

Observe that the first call to the decision algorithm (with $k = 2^{l-1} - 1$) is only to determine whether n is prime or not, and in this case one would rather apply a primality test which is usually much faster than a factoring algorithm: before trying to factor a number one should already know that the number is not prime. Finally, note that, since the factorization search algorithm requires $O(\text{len}(n))$ calls to the

decision problem algorithm, its complexity is equal to the complexity of the latter multiplied by $O(\text{len}(n))$.

The factorization decision problem belongs to the class $\mathcal{NP}$. This is because, if the answer is 'yes', i.e., n has a nontrivial factor $\leq k$, this can be checked in polynomial time using a factor a as a certificate by just dividing a into n. This problem is also in the class co$\mathcal{NP}$: if the answer is 'no', then the complete prime factorization of n can act as a certificate that, just by comparing all the prime factors with k, allows a polynomial time verification of the fact that none of them is $\leq k$. This requires checking that the provided factors are indeed prime (to ensure that there are no smaller factors) but, as we will see in Sect. 6.2, this can also be done in polynomial time.

The following Maple function may be used to simulate an algorithm for the factorization decision problem which, when asked whether an integer n has a factor a such that $2 \leq a \leq k < n$, answers either `true` if this is the case or `false` if there is no such factor. To do this, we will let Maple compute the complete factorization first, but we may use Maple's factoring algorithm as an oracle (or a black box to whose inner workings we do not have access) so that the function answers our queries without revealing the factors of n. The function is the following:

```
> DecisionFactor := proc(n, k)
  evalb(ifactors(n)[2][1][1] <= k)
  end proc:
```

Exercise 2.4 Use the method outlined in the previous example, together with the function `DecisionFactor`, to factor 247. What is the exact number of calls to the function in this case?

Exercise 2.5 Write a Maple function that automates the factorization method used in the previous exercise and, on input a positive integer n, outputs the least positive factor of n.

Example 2.4 ($\mathcal{P}$ **versus** $\mathcal{NP}$)
If a problem is in $\mathcal{P}$ then it is trivially in $\mathcal{NP}$ (and also in co$\mathcal{NP}$), so that we have the inclusion $\mathcal{P} \subseteq \mathcal{NP}$. However, it is not known whether this inclusion is proper or not and it is generally thought that $\mathcal{P} \neq \mathcal{NP}$. This open problem, called the $\mathcal{P}$ *versus* $\mathcal{NP}$ *problem*, is one of the so-called *Millennium Prize Problems* for the solution of each of which the *Clay Mathematics Institute* (see [50]) offers a prize of one million dollars. This problem is highly relevant for cryptography because, as we shall see later on, both private-key and public-key cryptography rely on the existence of *one-way functions*. That one-way functions exist is only a conjecture which, as we will see when discussing RSA, would be proved if we could show, for example, that no polynomial-time factorization algorithm exists. This conjecture implies that $\mathcal{P} \neq \mathcal{NP}$ so that, in other words, if $\mathcal{P} = \mathcal{NP}$ then public-key cryptography as we understand it today would not be possible (and neither would private-key cryptography).

More specifically, in Chap. 7 we will see that a necessary condition for the security of RSA encryption is the hardness of the factorization problem and hence that

$\mathcal{P} \neq \mathcal{NP}$. One might wonder whether $\mathcal{P} \neq \mathcal{NP}$ is also sufficient for the factorization problem to be difficult but this does not seem to be the case. There is a subclass of problems within $\mathcal{NP}$, called $\mathcal{NP}$-complete problems, which have the property that all other problems in $\mathcal{NP}$ can be reduced to any one of them in polynomial time and hence proving that an $\mathcal{NP}$-complete problem is in $\mathcal{P}$ is enough to show that $\mathcal{P} = \mathcal{NP}$. $\mathcal{NP}$-complete problems are the hardest problems in $\mathcal{NP}$ and there are a great number of them (for example, the *Traveling Salesperson problem* is one) but the factorization decision problem is not believed to be among them. See [6] for a more detailed discussion of these questions.

2.3.3 Running Times of Some Simple Algorithms

We are going to obtain estimates of the running time of a couple of very ancient algorithms and we will perform an experiment using Maple to check how close these estimates are to the actual running time of a specific implementation. As we will see, prime numbers play a crucial role in cryptography and hence it is very important to have efficient algorithms that allow us to distinguish between primes and composites. This is the task of a *primality test*, an algorithm that takes as input a positive integer and outputs `true` in case the integer is prime and `false` otherwise. The distribution of prime numbers and efficient primality tests are studied in Chap. 6 but here, as a preliminary example, we are going to study a couple of primality tests which, although inefficient, are important because they play an auxiliary role in many other algorithms: the *sieve of Eratosthenes* and *trial division*.

2.3.3.1 The Sieve of Eratosthenes

One of the oldest algorithms that can be used for primality testing is the sieve of Eratosthenes, which was known more than 2200 years ago and is still today the most efficient method to generate all the primes less than a given integer. The sieve takes as input a positive integer $n \geq 2$ and builds the list of all the primes $\leq n$ as follows. Start with the list of all the integers between 2 and n. The first number in the list is 2, which is prime, and we start by crossing off all proper multiples of 2. We repeat this process by picking, at each stage, the next number i in the list that is not crossed off (note that i is then necessarily a prime) and crossing off all proper multiples of i. This process is repeated while $i \leq \sqrt{n}$, at which point the numbers that remain without having been sieved out are the primes $\leq n$.

Next we give a Maple implementation of the sieve. To economize on memory only odd integers (plus the number 2) are considered and, in fact, they are not actually stored but the array is filled with zeros instead and the crossing off is implemented by replacing zeros by ones (for which we use the very efficient Maple function `ArrayTools:-Fill`). Moreover, the crossing off of multiples of each found prime number i can be started at i^2 since lower multiples have already been crossed off

during the previous steps. At the end, the positions corresponding to the remaining zeros give the primes in the list.

```
> Eratosthenes := proc(n)
  local s, f, a, i, j, k;
  s := floor((isqrt(n)+1)/2);
  f := floor((n+1)/2);
  a := Array(1..f);
  a[1]:=2;
  for i from 2 to s do
    if a[i] = 0 then
      j := 2*(i^2-i)+1;
      ArrayTools:-Fill(1,a,j-1,2*i-1);
    end if;
  end do;
  for i from 2 to f do
    if a[i] = 0 then
      a[i]:= 2*i-1
    end if;
  end do;
  convert(subs(1=NULL,a),list);
  end:
```

For example, the 10 primes preceding 3000000 (Gauss computed all the primes up to 3000000, so these were presumably the last ones computed by him) are the following:

```
> Eratosthenes(3000000)[-10 .. -1];
  [2999873,2999879,2999897,2999903,2999911,2999921,2999933,2999951,2999957,2999999]
```

Although the sieve can be used as a primality test, it is clearly not a very efficient one due, among other things, to the fact that it achieves much more than required: to test for the primality of n we have to compute all the primes $\leq n$ and check whether n is among them! It is also clear that the sieve requires a lot of memory (more than 20 MB for the preceding Maple computation, where n is rather small). The complexity of the algorithm can be estimated as follows, assuming that all arithmetic operations take constant time which is reasonable in view of the fact that the numbers involved will be machine-size integers—otherwise the memory requirements would be enormous. The number of crossings off for each prime p is $\lfloor \frac{n}{p} \rfloor$, so that the total number is $\sum_{p \leq \sqrt{n}} \frac{n}{p}$. From [180, Theorem 5.10] it follows that

$$\sum_{p \leq n} \frac{n}{p} = n \ln \ln n + O(n),$$

and this gives for the algorithm a running time estimate of $O(n\text{len}(\text{len}(n)))$. This complexity is exponential but, on the other hand, the space requirements of the algorithm are even more daunting. Without going into details, we remark that, by the PNT (see Examples 2.1 or Theorem 6.2), the number of primes less than a given integer can be approximated by the logarithmic integral, given in Maple by the function `Li` so that, for example, there are approximately

```
> Li(2.^1024);
                              2.536315701*10^305
```

primes of 1024 bits or less. Thus this algorithm is very far from being able to deal with, say, 1024-bit primes (as we will see, primes of this size are often used in cryptography). A practical trick to economize memory usage when sieving is to segment the array of numbers from 2 to n. There are many variations on the basic algorithm which, for example, can be modified to output all the primes in an interval $[a, b]$. Moreover, generalizations of the sieve of Eratosthenes are used in many other algorithms, including the most powerful factoring algorithms that we shall discuss later: the quadratic sieve and the number field sieve. For more information about this see [60].

2.3.3.2 Trial Division

Trial division is another old primality test that, in contrast with the sieve, has the advantage that it requires very little memory. The algorithm proceeds by trying to divide n successively by the integers between 2 and $\lfloor\sqrt{n}\rfloor$. If $\sqrt{n}$ is reached without any of them dividing n, then n is declared prime, otherwise, n is composite. In fact, it suffices to trial divide by the primes between 2 and $\sqrt{n}$, for which one has to generate the list of these primes. As in the case of the sieve of Eratosthenes, this algorithm also does more than primality testing for it is actually a factoring algorithm that can output all the prime factors of n.

The following implements trial division (as a primality test) in Maple for integers $\leq 10^{14}$:

```
> primelist := Eratosthenes(10^7):

> TrialDivision := proc(n)
  local s, p, r;
  if n < 2 then
    return false
  end if;
  if 10^14 < n then
    error "number too big"
  end if;
  s := isqrt(n);
  r := 1;
  for p in primelist while r <> 0 and p <= s do
    r := modp(n, p)
  end do;
  evalb(r <> 0)
  end proc:
```

For example, the primes among the last 50 integers preceding 10^{14} are:

```
> select(TrialDivision, [$10^14-50 .. 10^14]);
            [99999999999959, 99999999999971, 99999999999973]
```

The bit complexity of trial division can be easily estimated as follows using the PNT (Theorem 6.2). By this theorem, there are approximately $\frac{n^{1/2}}{\ln n^{1/2}} = O(\frac{n^{1/2}}{\ln n})$ primes less than $\sqrt{n}$. If n is prime, then one division for each of those primes must be performed, with cost $O(\ln^2 n)$. This gives a total cost of $O(n^{1/2} \ln n)$ bit operations and, since $\ln n = O(n^{\varepsilon})$ for $\varepsilon > 0$, the running time of the algorithm can be estimated

as $O(n^{1/2+\varepsilon})$. The running time is, therefore, exponential, and using this test with a 1024-bit number would approximately require—in case the number is prime—

```
> evalf(Li(sqrt(2^1024)));
                              3.788709545*10^151
```

divisions, which is far from feasible. It is not necessary to resort to such big numbers to reach the practical limits of this algorithm. In fact, it has been estimated in [60] that testing the primality of a 50-digit integer this way would require a computational effort equivalent to all the computation done until the beginning of the twenty-first century, i.e., around 10^{24} bit operations. It must be pointed out, however, that trial division is often useful as a first step in factoring algorithms, where it serves to get rid of small factors quickly.

Exercise 2.6 Modify the function `TrialDivision` so that the function itself computes the primes used and does not require `primelist` to be computed independently.

2.3.3.3 An Experiment on Trial Division Timing

We are now going to do an experiment—that may serve as a model for other experiments with different algorithms—to test the running time of trial division in practice and check how well it agrees with the theoretically expected one. To approximate the complexity of the algorithm we want to work with numbers as large as possible and so, to minimize the time and memory required to build the list of primes by means of the sieve of Eratosthenes, we will use an array instead of a list. Another advantage of using arrays throughout is that this will allow us to write functions that are compilable by the external C compiler included with Maple, with a considerable gain in speed; thus this experiment will also serve to introduce new features of Maple as a programming language. In order to achieve these goals we will modify the previous procedure `Eratosthenes` in which, after running the sieve, the initial array contains zeros in the positions corresponding to primes and ones in the positions corresponding to composites. We want an array containing only the sequence of prime numbers less than or equal to n and the next function, `ReduceArray`, converts the larger array to this form.

```
> ReduceArray := proc(n::integer[4],
                      a::Array(datatype=integer[4]), b::Array(datatype=integer[4]))
  local i, j;
  j := 1;
  for i from 2 to n do
    if a[i] = 0 then a[i] := 2*i-1 end if
  end do;
  for i to n do
    if a[i] <> 1 then
      b[j] := a[i];
      j := j+1
    end if
  end do
  end proc:
```

To make it compilable, the preceding function has explicit type declarations of integers as hardware integers. We compile the function, obtaining the function `CompReduceArray`[3]:

```
> CompReduceArray := Compiler:-Compile(ReduceArray):
```

Next we give the new function, called `Sieve`, that implements the sieve and builds the array of prime numbers up to n. Note that in order to build this array we need to know in advance the number of primes $\leq n$. We did not use this knowledge when giving the first version of the sieve as it seems inappropriate to use a much more sophisticated algorithm in order to implement it. Now, however, we are not interested in a function to sieve different intervals but rather in a function that allows us to build an array of primes as large as possible to use in trial division. In this case it makes sense to assume that we know the number of primes as given by the Maple function `numtheory:-pi`. `Sieve` is just a modification of `Eratosthenes`, and now we use `CompReduceArray` in the final stage to produce as output an array containing the primes.

```
> Sieve := proc(n)
  local s, f, a, b, i, j;
  s := floor((isqrt(n)+1)/2);
  f := floor((n+1)/2);
  a := Array(1 .. f, datatype = integer[4]);
  a[1] := 2;
  for i from 2 to s do
    if a[i] = 0 then
      j := 2*i^2-2*i+1;
      ArrayTools:-Fill(1, a, j-1, 2*i-1)
    end if
  end do;
  b := Array(1 .. numtheory:-pi(n), datatype = integer[4]);
  CompReduceArray(f, a, b);
  b
  end proc:
```

We are going to build an array containing all the primes up to 2^{28}. The number of primes in this array is

```
> numtheory:-pi(2^28);
                                14630843
```

i.e., more than 14 million primes. The array containing the primes is then:

```
> primearray := Sieve(2^28):
```

Now we give the function that performs trial division based on this array; we do not compile it because we want it to work, even in 32-bit computers, with integers longer than 32 bits:

```
> TrialDivide := proc(n)
  local s, p, r;
  if n < 2 then
```

[3] Compiled functions like this one are not included in the Maple language files provided on the book's website and must be generated by calling `Compiler:-Compile` from the corresponding examples worksheet, where the appropriate command is already included.

```
    return false
  end if;
  if 2^56 < n then
    error "number too big"
  end if;
  s := isqrt(n);
  r := 1;
  for p in primearray while r <> 0 and p <= s do
    r := modp(n, p)
  end do;
  evalb(r < > 0)
  end proc:
```

The next function measures the time taken by `TrialDivide` when acting on a given integer:

```
> TrialDivideTiming := proc(value)
  local t;
  t := time();
  TrialDivide(value);
  time()-t
  end proc:
```

Now, in order to time the function `TrialDivide`, we build up a sample of primes (we use Maple's function `prevprime` for speed but, of course, we could use `TrialDivide` as well). The reason of taking prime numbers is that these are the integers that require the most time when trial divided (most composite numbers have small prime factors and hence require very little time to be tested).

```
> plist := [seq(prevprime(i*2^52), i = 1 .. 16)];
  plist := [4503599627370449, 9007199254740881, 13510798882111483, 18014398509481951,
            22517998136852473, 27021597764222939,31525197391593467, 36028797018963913,
            40532396646334423, 45035996273704937, 49539595901075453, 54043195528445869,
            58546795155816433, 63050394783186917, 67553994410557429, 72057594037927931]
```

Next we time `TrialDivide` acting on these primes:

```
> tlist := map(TrialDivideTiming, plist);
tlist := [8.047, 11.172, 13.531, 15.531, 17.234, 18.751, 20.266, 21.609, 22.859,
  23.938, 25.109, 26.172, 27.173, 28.171, 29.157, 30.047]
```

We now approximate the time function. We expect this function to be proportional to a power of n, where the exponent should be close to 0.5. Observe that the running time will be really exponential when given as a function of the size of the number (or of its logarithm) although it is just a power when given as a function of n. Because of this we use Maple's function `PowerFit` from the `Statistics` package, which applies the least-squares method to build a model function of the form $a \cdot n^b$ that fits these data points:

```
> t := Statistics:-PowerFit(plist, tlist, n);
                2.91575852460995 10^-7 n^0.475299605601998410
```

The function obtained is approximately $2.92 \cdot 10^{-7} \cdot n^{0.476}$ and we see that the exponent obtained is close to 0.5 as expected. It is reasonable to expect that, when performing these timings with Maple, there is some overhead time. If we subtract a small constant value from the times in `tlist` above and repeat the computation of the model function, we will see that the resulting function is even closer to the theoretically expected one (i.e., the exponent is closer to 0.5).

Exercise 2.7 Experiment by using the `TrialDivideTiming` function with different data sets of integers in order to obtain the timing functions that fit these data. In each case, compare the function obtained with the expected one and check that subtracting some small constant value from the times list gives a timing function which is closer to the theoretically expected one.

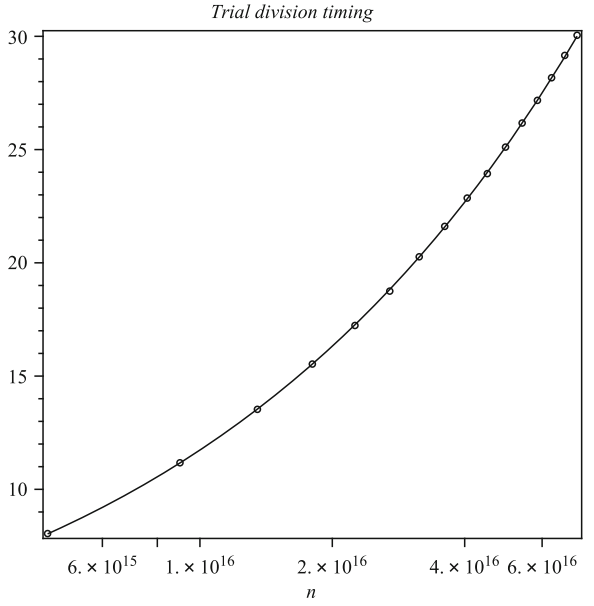
In order to be able to plot the data points we have used, we build the list of these points, where each point will be given as a list of length two, whose first element is an integer and whose second element is the time taken by `TrialDivide` on this integer:

```
l := [ListTools:-LengthSplit(ListTools:-Interleave(plist, tlist), 2)]:
```

Now we plot the curve t together with the set of data points we have obtained; this lets us graphically appreciate that the exponential function t does fit the data well. To plot the list of points we use the function `listplot` in the `plots` package and, to take into account the fact that the running time function grows exponentially in $\mathrm{len}(n)$ (or in $\ln n$) we plot t with the `semilogplot` command (also from the `plots` package), which gives the horizontal axis a logarithmic scale.

```
> with(plots):
  display(title='Trial division timing', {semilogplot(t, n=2^52..2^56, color=black),
  listplot(l, color=black, style=point, symbol=circle, symbolsize=10)}, axes=boxed)
```

The output of this command is the following plot:



2.3.4 Probabilistic Algorithms

The concrete algorithms we have considered so far are *deterministic*, in the sense that their behavior is completely determined by their input, so that if the algorithm is executed multiple times with the same input, it always produces the same output. But, as we shall see, in cryptology one often uses *probabilistic* (*or randomized*) *algorithms*. These algorithms have access to a source of randomness that yields random bits[4] with uniform probability distribution (i.e., where 0 and 1 each have probability $\frac{1}{2}$). Consequently, a probabilistic algorithm may produce a different output each time it is executed with the same input. Examples of algorithms of this type used in cryptography are:

- Algorithms to randomly generate a key of a given size.
- Randomized encryption algorithms which produce different ciphertexts when repeatedly used with the same plaintext and the same key (as we shall see, the use of probabilistic encryption algorithms is crucial to obtain good security).
- Probabilistic algorithms used to test primality, such as the *Miller–Rabin test*, to be discussed later.
- Algorithms to randomly choose a prime number of a given size.

We have already mentioned the relation between polynomial-time and efficient (or, at least, tractable) computation and now we want to point out that probabilistic polynomial-time algorithms (which are generally faster than deterministic ones) are actually the ones usually taken to define efficiency. There are several interesting classes of probabilistic algorithms, among which we mention:

- *Monte Carlo algorithms*. These run in polynomial time but may not always give the correct result. In the case of a decision problem a Monte Carlo algorithm is *yes-biased* if a 'yes' answer is correct with some probability $> 1/2$ but a 'no' answer is always correct. Similarly, *no-biased* Monte Carlo algorithms are defined by interchanging the roles of 'yes' and 'no'. The two-sided versions of Monte Carlo algorithms, which may err on either the 'yes' or the 'no' side, are also called *Atlantic City* algorithms.
- *Las Vegas algorithms*. These algorithms run in *expected* polynomial time, meaning that their running time is a random variable whose expected value is polynomial (see Sect. 2.1) but, in contrast, their output is always correct. However, Las Vegas algorithms may not even provide an output for some inputs.

It is sometimes said that Monte Carlo algorithms are always fast but only probably correct while Las Vegas algorithms, on the contrary, are always correct but only probably fast. As we shall see later, examples of both classes of algorithms are known for the decision problem of determining whether a given integer is prime (in fact, there is also a deterministic polynomial-time algorithm for this problem, but it is significantly less efficient than the probabilistic ones).

[4] It is often metaphorically said that the algorithm tosses "random coins".

Definition 2.16 A decision problem belongs to the class $\mathcal{RP}$ (*randomized polynomial time*) if there exists a no-biased Monte Carlo algorithm to solve the problem. Similarly, the problem is in co$\mathcal{RP}$ if there exists a yes-biased Monte Carlo algorithm that solves it. The intersection $\mathcal{RP} \cap \text{co}\mathcal{RP}$ defines the class $\mathcal{ZPP}$ (*zero-error probabilistic polynomial time*).

It is not difficult to see [6, 191] that the class $\mathcal{ZPP}$ actually consists of the decision problems which can be solved by a Las Vegas algorithm.

Example 2.5 Let us consider the decision problem that asks, given a positive odd integer n, whether n is composite. As we will see, by Theorem 6.7 the Miller–Rabin test is a no-biased Monte Carlo algorithm for this problem and hence the decision compositeness problem is in $\mathcal{RP}$. By the same reason the decision primality problem (given a positive integer n, is n prime?) belongs to $co\mathcal{RP}$. In fact, since the 1990s it has been known that this problem also belongs to $\mathcal{RP}$ and hence to $\mathcal{ZPP}$ but, from a theoretical point of view, all these results have been superseded by the proof in 2002 [3] that the decision primality problem is in $\mathcal{P}$ (taking into account that, clearly, $\mathcal{P} \subseteq \mathcal{ZPP} \subseteq \mathcal{RP}$).

The classes $\mathcal{RP}$ and co$\mathcal{RP}$ are defined by probabilistic polynomial-time algorithms which make only one-sided errors. We can consider also algorithms that can make two-sided errors, i.e., what we referred earlier to as Atlantic City algorithms. This idea leads to the definition of the class $\mathcal{BPP}$:

Definition 2.17 A decision problem belongs to the class $\mathcal{BPP}$ (*bounded-error probabilistic polynomial time*) if there exists a constant $\varepsilon > 0$ and a probabilistic polynomial-time algorithm that produces a 'yes' or 'no' answer which is correct with probability $> 1/2 + \varepsilon$.

We remark that if a probabilistic algorithm gives a correct answer with probability greater than a constant $> 1/2$, then one may obtain from it an algorithm that, for any $\varepsilon > 0$ produces a correct answer with probability greater than $1 - \varepsilon$. In the case of a Monte Carlo algorithm it suffices to take k independent iterations of the original algorithm, where k satisfies $2^{-k} < \varepsilon$. In the case of an Atlantic City algorithm, one considers a sufficient number of iterations of the algorithm and then takes the "majority answer" (see, e.g., [191, Proposition 4.14]). This last remark has the consequence that, although the inclusion $\mathcal{RP} \subseteq \mathcal{BPP}$ is probably proper (and it is suspected that $\mathcal{BPP}$ may be much larger than $\mathcal{RP}$), the problems in $\mathcal{BPP}$ are considered tractable.

Example 2.6 Consider the following problem: Given a positive integer k, find a prime of (at least) k bits in time polynomial in k. We will see that many cryptographic algorithms use large primes so the problem has a definite cryptographic interest. However, no (deterministic) polynomial-time algorithm is known so far to solve this problem! (see [157] for a discussion and references). Despite this apparently unfortunate situation, there is no problem to find large primes for cryptographic use, which can be done efficiently by using probabilistic algorithms. The idea is simply to

choose integers of the required size at random and submit them to a primality test until a prime is found. The primality test used can be either probabilistic or deterministic (with the former being the more efficient by far) but the prime-finding algorithm will be probabilistic anyway because we have to start by choosing random integers. We will see in Chap. 6 that this allows us to find primes in expected polynomial time, the reason being essentially that primes are sufficiently dense among the integers so that if p is the probability that a randomly chosen integer of a given size is prime, then $1/p$ is a polynomial function on the size. Since the expected number of trials to find a prime is equal to $1/p$ by Proposition 2.2 and, since each trial involves a polynomial-time primality test, the algorithm does indeed run in expected polynomial time.

2.3.5 Final Remarks on Complexity

In this section we briefly discuss some aspects of complexity that, although not required to read the rest of the book, are relevant for cryptography and are also of considerable interest from the point of view of theoretical computer science. We also include some bibliographic references for the reader who wishes additional information on the subject.

All the notions of complexity discussed so far are based on evaluating the *worst case* of the computational problems involved. For example, the idea behind the concept of $\mathcal{NP}$-completeness is that an algorithm that can efficiently solve all instances of the problem, including the most difficult ones, would lead to an efficient algorithm to solve all the problems in $\mathcal{NP}$. But there are $\mathcal{NP}$-complete problems that are easy to solve most of the time and this is perfectly compatible with the hypothesis that $\mathcal{P} \neq \mathcal{NP}$, which only implies that there is no efficient algorithm to solve these problems in the worst case [6]. Public-key cryptography is based, as we shall see, on the hypothesis that some computational problems are difficult, but problems that are hard in the worst case but not on average are not good for this purpose and, in fact, one would like to use problems which are hard for most instances. For example, the security of RSA is based on the assumption that most integers that are the product of two randomly chosen large primes are hard to factor. The plausibility of this assumption makes the factorization problem adequate for cryptographic purposes although, as we have already mentioned, it is not thought to be $\mathcal{NP}$-complete. In contrast with this, even if we assume that $\mathcal{P} \neq \mathcal{NP}$, we do not have an a priori guarantee that an $\mathcal{NP}$-complete problem will be hard on average, and hence such a problem may not be suitable for cryptographic use. On the other hand, we will see that the hard problems of cryptographic interest are usually *random self-reducible* (see Definition 7.6) in the sense that any instance can be reduced in polynomial time to a random instance and hence, if the problem is hard in the worst case, then it is also hard on average.

These considerations lead to the study of *average-case complexity*, and it is by no means obvious how to precisely define what it means for a problem to be "intractable

on average". The idea is that to capture the notion of "being efficient on typical instances", one should allow an algorithm to take a large amount of time on inputs that occur with low probability, in such a way that if we have inputs that require increasingly large running time, these inputs should occur with proportionally decreasing probability. This is the idea underlying the definition of average-case complexity given by Levin [129] in the 1980s, which was later modified by Impagliazzo [104]. We refer to these papers and also to [9] for the details.

We have mentioned the fact that computational efficiency can be defined independently of the computational model used. This is true for classical models such as Turing machines or Boolean circuits but does not apply to *quantum computers*. These are computational devices based on the principles of quantum mechanics and there is a celebrated result due to Peter Shor [177] that shows that quantum computers can factor integers (and also solve other related problems) in polynomial time. This poses a lethal threat to RSA and, in fact, to most public-key cryptosystems currently in use. But it is not clear whether full-sized quantum computers—which, so far, exist only as experimental devices with very little capacity—will ever be built and, on the other hand, there are also public-key cryptosytems based on problems (mainly lattice-theoretic ones) which have not been shown to be easy for quantum computers. We say a few more words about this when discussing the current status of integer factorization in Sect. 6.4.10.

2.4 The Euclidean Algorithm

Definition 2.18 Given $a, b \in \mathbb{Z}$ not both zero, the *greatest common divisor* of a and b, denoted $\gcd(a, b)$ is the largest integer that divides both a and b. We say that a and b are *relatively prime* if $\gcd(a, b) = 1$.

The computation of the greatest common divisor is an interesting problem for several reasons. In the first place, this computation is a basic component in many number-theoretic algorithms, as we will see when implementing them. But it is also interesting because it is not at all obvious how to compute, for example, the gcd of two integers that have several hundreds of decimal digits each (as we will see later on, integers of this size are commonly used in cryptography).

A straightforward procedure to obtain the gcd is to list all the common positive divisors of a and b. Both lists will be finite and the largest number on both lists will be $\gcd(a, b)$. But it is intuitively clear that this method will not work with large numbers and we could try a related but more sophisticated method instead, using the fundamental theorem of arithmetic (Theorem 2.2). From this theorem we can derive the following method to compute the gcd of two integers a and b which we can assume, without loss of generality, to be greater than 1. Suppose that $a = \prod_{i=1}^{k} p_i^{e_i}$ and $b = \prod_{i=1}^{k} p_i^{f_i}$, where the primes p_i are written in increasing order and we allow the exponents e_i and f_i to be zero. Then it is obvious that

$$\gcd(a,b) = \prod_{i=1}^{k} p_i^{\min(e_i, f_i)}$$

However, this method is, again, unable to find, in general, the gcd of two integers of several hundred digits each. The reason is that no algorithms are known that can factor integers of this size in reasonable time (the largest 'difficult' integer factored so far is one of 232 decimal digits and it took about 2.5 years to factor it using many workstations; this factorization is discussed in more detail in Sect. 6.4.10). Fortunately, there is a very efficient algorithm to compute the gcd which is able to deal with very big numbers—even numbers much larger than the one just mentioned. This is the Euclidean algorithm, the oldest nontrivial algorithm and hence, according to Knuth [115], "*the granddaddy of all algorithms*". In describing it (and throughout the book) we will denote by $a \mod b$ the remainder in the set $\{0, 1, \ldots b-1\}$ of dividing a by b, also called the *least non-negative residue* of a modulo b.

The basic fact underlying the Euclidean algorithm is the following:

Theorem 2.4 *Let* $a, b \in \mathbb{Z}$ *such that* $a \geq b > 0$. *Then* $\gcd(a,b) = \gcd(b, a \bmod b)$.

The proof is a straightforward consequence of the obvious fact that d is a common divisor of a and b if and only if d is a common divisor of b and $a \mod b$. If $b < a$ (otherwise $b = a$ and $\gcd(a,b) = a$) then, since $a \mod b < b$, this theorem reduces the computation of $\gcd(a,b)$ to computing the gcd of two strictly smaller numbers. Setting $a = r_0$ and $b = r_1$ and repeating this process by computing the remainders $r_i = r_{i+1}q_{i+1} + r_{i+2}$, with $0 \leq r_{i+2} < r_{i+1}$, we obtain a strictly decreasing sequence of non-negative integers $a = r_0 > r_1 > r_2 > \cdots \geq 0$ and hence $r_{n+1} = 0$ for some index $n \leq a$. If n is the least such index we have $\gcd(a,b) = \gcd(r_0, r_1) = \gcd(r_1, r_2) = \ldots \gcd(r_n, 0) = r_n$.

We have just described the Euclidean algorithm, which can be given in pseudocode as a recursive function that calls itself:

Algorithm 2.1. Euclidean algorithm (recursive version).

```
recgcd:
Input: Integers a ≥ b ≥ 0, not both 0.
Output: gcd(a, b).

if b = 0 then
    return a
else
    recgcd(b, a mod b)
end if.
```

An alternative to the recursive version is the iterative version given in Algorithm 2.2.

Maple has a built-in function called `igcd` for computing the gcd of two integers. This function is fast and we use it in many of our functions throughout the book.

Algorithm 2.2. Euclidean algorithm (iterative version).

Input: Integers $a \geq b > 0$.
Output: $\gcd(a, b)$.

```
while b > 0
    r := a mod b
    a := b
    b := r
end while
return a.
```

However, for illustrative purposes, we next give simple implementations of the two versions of the Euclidean algorithm we have just seen. The recursive version of the Euclidean algorithm is implemented in the Maple function `recgcd` below, which is an almost verbatim transliteration of Algorithm 2.1. In order to visualize the numbers to which the algorithm is being applied in the successive steps, we include an optional parameter `verbose` which, if set to `true`, makes the function print these pairs of integers on screen:

```
> recgcd := proc(a::nonnegint, b::nonnegint, verbose := false)
  if verbose then
    print(a,b)
  end if;
  if b = 0 then
    return a
  end if;
  recgcd(b, a mod b, verbose);
  end proc:
```

Example 2.7 Let us use `recgcd` to compute the gcd of two numbers:

```
> recgcd(387349873729876, 4896209096872);
                                        4
```

We will see that the number of steps in applying the Euclidean algorithm to two integers is relatively small and, because of this fact, the algorithm is very efficient. This claim will be made precise when studying the algorithm's complexity but now we use the numbers above to give intuition about these steps. We visualize the recursive calls on screen as follows:

```
> recgcd(387349873729876, 4896209096872, true);
                      387349873729876, 4896209096872
                        4896209096872, 549355076988
                         549355076988, 501368480968
                         501368480968, 47986596020
                          47986596020, 21502520768
                          21502520768, 4981554484
                           4981554484, 1576302832
                           1576302832, 252645988
                            252645988, 60426904
                             60426904, 10938372
                             10938372, 5735044
                              5735044, 5203328
                              5203328, 531716
                               531716, 417884
```

```
417884, 113832
113832, 76388
 76388, 37444
 37444, 1500
  1500, 1444
  1444, 56
    56, 44
    44, 12
    12, 8
     8, 4
     4, 0
      4
```

It is also straightforward to implement the iterative version of Euclid's algorithm in Maple, closely following Algorithm 2.2:

```
> itergcd := proc(a, b, verbose := false)
  local x, y, r;
  x := a;
  y := b;
  while y <> 0 do
    if verbose then
      print(x,y)
    end if;
    r := x mod y;
    x := y;
    y := r;
  end do;
  x;
  end proc:
```

We are going to analyze the complexity of the Euclidean algorithm and, as we are going to see, the worst case is provided by consecutive Fibonacci numbers as was proved by Lamé already in the mid-nineteenth century. TheFibonacci numbers are defined by the linear recurrence

$$F_n = F_{n-1} + F_{n-2},$$

where $F_0 = 0$ and $F_1 = 1$. They can be computed in Maple by means of the command `fibonacci` in the `combinat` package. To simplify the notation, we define an alias for this command and we may use it as follows:

```
> alias(F = combinat:-fibonacci):

> F(1), F(10), F(100);
                        1, 55, 354224848179261915075
```

Let us identify the worst case in the complexity of the Euclidean algorithm:

Theorem 2.5 *Assume $a > b > 0$ and let r_n be the last nonzero remainder in the sequence of integer divisions $r_0 = a, r_1 = b, \ldots, r_{i-1} = r_i q_i + r_{i+1}, \ldots$ Then $a \geq F_{n+2}$ and $b \geq F_{n+1}$.*

Proof Note that, since the sequence of remainders is strictly decreasing, $r_{i-1} > r_i > r_{i+1}$ for $1 \leq i \leq n$ and so $q_i \geq 1$. Moreover, since $r_{n+1} = 0$, we have that $r_{n-1} = r_n q_n$ and so $q_n \geq 2$ (for otherwise we would have $r_{n-1} = r_n$). Thus, working backwards through the sequence of divisions we have that:

$$\begin{aligned} r_n &\geq 1 = F_2, \\ r_{n-1} &\geq 2r_n \geq 2F_2 = F_3, \\ r_{n-2} &\geq r_{n-1} + r_n \geq F_3 + F_2 = F_4, \\ &\vdots \\ r_2 &\geq r_3 + r_4 \geq F_{n-1} + F_{n-2}, \\ b = r_1 &\geq r_2 + r_3 \geq F_n + F_{n-1} = F_{n+1}, \\ a = r_0 &\geq r_1 + r_2 \geq F_{n+1} + F_n = F_{n+2}. \end{aligned}$$

□

Note also that computing $\gcd(F_{n+2}, F_{n+1}) = 1$ by means of Euclid's algorithm takes exactly n steps. Indeed, from the fact that $F_{i+1} = F_i + F_{i-1}$ it follows that all quotients are equal to 1 in this case and the remainders are the Fibonacci numbers in reverse order. Let's do an example with Maple:

Example 2.8 We compute the greatest common divisor of F_{12} and F_{11} and we display the sequence of remainders used in the successive divisions:

```
> itergcd(F(12), F(11), true);
                                144, 89
                                 89, 55
                                 55, 34
                                 34, 21
                                 21, 13
                                 13, 8
                                  8, 5
                                  5, 3
                                  3, 2
                                  2, 1
                                    1
```

We see that the computation indeed took 10 division steps and the output (the gcd) is 1. Next, we compute the sequence of Fibonacci numbers from F_1 to F_{12}:

```
> map(F, [$1 .. 12]);
              [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

We see that the sequence of remainders is indeed the sequence of Fibonacci numbers in reverse order.

We can now give a bound for the number of steps in the Euclidean algorithm.

Theorem 2.6 (Lamé's theorem) *Assume that $a > b > 0$ and let $\varphi = (1+\sqrt{5})/2$ be the golden ratio constant. Then the number of iterations in applying the Euclidean algorithm to a and b is bounded above by $1 + \log_\varphi b$.*

Proof First we show that, for all $n \geq 3$, $\varphi^{n-2} < F_n$. We use induction on n and we start from the fact that this inequality holds for $n = 3$ and $n = 4$ as can easily be checked with Maple:

```
> phi := (1+sqrt(5))*(1/2):

> evalb(evalf(phi) < F(3));
                                    true
> evalb(evalf(phi^2) < F(4));
                                    true
```

Now φ is the positive root of the quadratic equation $x^2 - x - 1 = 0$, so that it satisfies $\varphi^2 = \varphi + 1$ which, multiplying by φ^{n-4} gives the identity $\varphi^{n-2} = \varphi^{n-3} + \varphi^{n-4}$. Our induction hypothesis is then that both $\varphi^{n-4} < F_{n-2}$ and $\varphi^{n-3} < F_{n-1}$ hold. Adding these inequalities we obtain $\varphi^{n-3} + \varphi^{n-4} = \varphi^{n-2} < F_{n-2} + F_{n-1} = F_n$, completing the inductive proof. If applying the Euclidean algorithm to a and b takes n steps, we see from Theorem 2.5 that $b \geq F_{n+1}$ and, by the preceding inequality, we have that $b > \varphi^{n-1}$. Taking logarithms we obtain $\log_\varphi b > n - 1$. □

Exercise 2.8 Show that, for $n \geq 3$, $\varphi^{n-2} < F_n < \varphi^{n-1}$ (use the same induction as in the proof of the preceding theorem, where the first of these two inequalities has already been proved).

Lamé's theorem is also sometimes stated as follows: *the number of division operations needed by the Euclidean algorithm to find the* gcd *of two positive integers is no more than five times the number of decimal digits of the smaller of the two numbers* [194]. This follows from Theorem 2.6 which tells us that if the number of steps is n then $n - 1 < \log_\varphi b$ (where b is the smaller of the two numbers). We also have that $\log_\varphi b = \log_{10} b / \log_{10} \varphi$ and, since $\log_{10} \varphi > 0.2$ we obtain $n - 1 < 5 \log_{10} b$. Then, if b has k decimal digits, we have that $\log_{10} b < k$ and so we get $n - 1 < 5k$ from which it follows, taking into account that both n and k are integers, that $n \leq 5k$.

Now, the complexity of the Euclidean algorithm is an easy consequence of Lamé's theorem:

Theorem 2.7 *The complexity of the Euclidean algorithm applied to integers a, b is* $O(\text{len}(a)\text{len}(b))$.

Proof We can assume that $a > b > 0$ (if this is not the case, then it will be after just one iteration). With the notation previously used, let $(r_i)_{2 \leq i \leq n+1}$ be the remainder sequence and $(q_i)_{2 \leq i \leq n}$ the corresponding quotient sequence. The computation of r_{i+1} by means of an integer division such that $r_{i-1} = r_i q_i + r_{i+1}$ (with $1 \leq i \leq n$) requires time $O(\text{len}(r_i)\text{len}(q_i))$ as we have already remarked. Since $r_i \leq b$ we have that $\text{len}(r_i) \leq \text{len}(b)$ for $1 \leq i \leq n+1$ and, on the other hand, $\text{len}(q_i) \leq 1 + \log_2 q_i$. Thus the time taken by the Euclidean algorithm on input (a, b) can be estimated as

$$O\left(\sum_{i=1}^{n} (\text{len}(b)(1 + \log_2 q_i))\right) = O\left(\text{len}(b)\left(n + \sum_{i=1}^{n} \log_2 q_i\right)\right).$$

Now, it follows from Theorem 2.6 that $n = O(\text{len}(b))$ and, on the other hand, $a = r_0 = r_1 q_1 + r_2 \geq r_1 q_1 = (r_2 q_2 + r_3) q_1 \geq r_2 q_2 q_1 \geq \cdots \geq q_n q_{n-1} \ldots q_1$, which

implies $\sum_{i=1}^{n} \log_2 q_i = O(\text{len}(a))$. Replacing these equalities in the time estimate above we obtain the estimate $O(\text{len}(b)\text{len}(a))$. □

From the Euclidean algorithm one easily obtains that $\gcd(a, b)$ is a linear combination of a and b, namely:

Theorem 2.8 *Let $a, b \in \mathbb{Z}$ such that $a \geq b > 0$. Then there exist $s, t \in \mathbb{Z}$ such that $\gcd(a, b) = sa + tb$.*

To prove this result it suffices to use the Euclidean algorithm to compute $\gcd(a, b)$, keeping all the intermediate results of the divisions and then backtrack by substituting each remainder in terms of the two previous ones until reaching $r_0 = a$ and $r_1 = b$, at which moment one has the required linear combination. A more efficient way to carry out this computation is by using the following algorithm in which, following Klappenecker, we use matrix notation:

Algorithm 2.3. Extended Euclidean algorithm.

Input: Integers $a \geq b > 0$.
Output: Integers d, s, t such that $d = \gcd(a, b) = sa + tb$.

$$\begin{pmatrix} s & u \\ t & v \\ d & x \end{pmatrix} := \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ a & b \end{pmatrix};$$

`while` $x \neq 0$
  $q := \lfloor \frac{d}{x} \rfloor$;

$$\begin{pmatrix} s & u \\ t & v \\ d & x \end{pmatrix} := \begin{pmatrix} s & u \\ t & v \\ d & x \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix};$$

`end while`;
`return` (d, s, t).

To see that the algorithm works as expected note that at the start of the loop we have $d = a$ and $x = b$ and, after the first step the new values of d and x are

$$(d, x) = (a, b) \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} = (b, a \bmod b) = (r_1, r_2),$$

where $q := \lfloor \frac{a}{b} \rfloor$ and we have used our earlier notation for the remainders. After the next step we similarly have $(d, x) = (r_2, r_3)$ and upon termination of the loop we have that $(d, x) = (r_n, 0) = (\gcd(a, b), 0)$, so that $d = \gcd(a, b)$.

Furthermore, at the start of the loop, the following equations hold: $as + bt = d$, $au + bv = x$. Subtracting the second equation q times from the first we obtain the equation $a(s - qu) + b(t - qv) = d - qx$ which, taking into account the assignments to the variables u, v and x made in the while loop, means that after each iteration of the loop we still have $au + bv = x$. After the assignments $s := u$, $t := v$, $d := x$ in

the last iteration, the equation $au + bv = x$ becomes $as + bt = d$, giving the desired result.

The extended Euclidean algorithm is implemented in Maple's function `igcdex`. This function accepts as arguments, besides the two integers whose gcd is to be computed, two names `'s'` and `'t'` where the values of the integers s, t mentioned above are to be stored. For example:

```
> igcdex(387349873729876, 48962090968, 's', 't');
                                          4

> s, t, 387349873729876*s + 48962090968*t;
                            -2790500723, 22076265153864, 4
```

As we will see, the Euclidean algorithm plays an important role in cryptology because it is an essential step in many other interesting algorithms. It is also very important that it is a very efficient algorithm, not only in theoretical terms but also in practice. By working backwards in the Euclidean algorithm we may similarly show that the complexity of the extended Euclidean algorithm is also $O(\text{len}(a)\text{len}(b))$.

2.5 Groups, Rings and Fields

In this section we briefly recall some basic algebraic concepts that will be useful in what follows. For more detailed accounts we refer to [45, 48].

2.5.1 Basic Concepts

Recall that a *group* $(G, \cdot)$ is a set G together with a binary operation $G \times G \stackrel{\cdot}{\to} G$ which satisfies the following properties:

- *Associative law*. For all $x, y, z \in G$, $x \cdot (y \cdot z) = (x \cdot y) \cdot z$.
- *Identity element*. There is a (necessarily unique) element $e \in G$ such that, for every $x \in G$, $x \cdot e = e \cdot x = x$.
- *Inverse element*. Each element $x \in G$ has a unique inverse $x^{-1} \in G$, namely, an element x^{-1} that satisfies $x \cdot x^{-1} = x^{-1} \cdot x = e$.

A group is called *abelian* or *commutative* when, in addition, the operation satisfies:

- *Commutative law*. For all $x, y \in G$, $x \cdot y = y \cdot x$.

When the binary operation is understood, we often commit abuse of language and refer to the set G as a group. The symbols most frequently used to denote the group operation are $\cdot$ (multiplicative group) and $+$ (additive group) although, of course, the operation may not be related to the addition or the multiplication of numbers. Additive notation is frequently used for abelian groups and when multiplicative notation is used, the symbol $\cdot$ is often omitted and we write xy to denote the result of

operating x with y. The identity element of a multiplicative group is usually denoted by 1 and the identity element of an additive group is denoted by 0. Moreover, in an additive group, the inverse of an element x is called the *opposite* element of x (or also the *negative* of x) and is denoted by $-x$. For generic groups, we will always use multiplicative notation.

If G and H are groups, then a map $f : G \to H$ is called a *homomorphism* if $f(xy) = f(x)f(y)$ for all $x, y \in G$. Clearly, a homomorphism maps the identity of G to the identity of H and the inverse of an element to the inverse of its image. A bijective homomorphism is called an *isomorphism*. In this case, the inverse $f^{-1} : H \to G$ is also an isomorphism and G and H are said to be *isomorphic*, denoted $G \cong H$. This means that G and H are essentially the "same" group, in the sense that they share the same algebraic properties.

The group G is finite when G is a finite set, and the number of elements of G, denoted $|G|$, is called the *order* of the group G. A *subgroup* of the group G is a subset $H \subseteq G$ such that the operation of G induces by restriction a group structure in H. If G is a finite group and $x \in G$, then the *order* of x is the smallest positive integer i such that $x^i = 1$. The subgroup *generated by* x, i.e., the smallest subgroup of G containing x, consists of the elements $1, x, x^2 \ldots, x^{i-1}$, where i is the order of x. We will denote this subgroup by $\langle x \rangle$ and we say that x is a generator of $\langle x \rangle$. A finite group is called *cyclic*[5] when it has a generator, i.e., an element $g \in G$ such that $G = \langle g \rangle$, which happens if and only if the smallest subgroup of G containing g is G itself. As is easily seen, such groups are always abelian and, if $G = \langle g \rangle$, the order of the element g is equal to the order of the group G.

Examples 2.2

1. Every one-element set $\{x\}$ with the obvious operation is a group (in which x is the identity and its own inverse). This group is isomorphic to the subgroup $\{1\}$ of any group G.
2. $\mathbb{Z}$ is a group with ordinary addition of integers. It is an infinite abelian group and is also cyclic in the sense that the smallest subgroup containing 1 (or -1) is $\mathbb{Z}$ itself but $\mathbb{Z}$ is not a group under multiplication because the only elements that have inverse with respect to this operation are 1 and -1.
3. $\mathbb{R}$ and $\mathbb{C}$ are groups with ordinary addition but they are not groups with respect to multiplication because 0 has no inverse. However, $\mathbb{R}^* = \mathbb{R} - \{0\}$ and $\mathbb{C}^* = \mathbb{C} - \{0\}$ are multiplicative groups. All these groups are abelian but none of them are cyclic.
4. If X is a set, then the set $S(X)$ (also denoted S_X) of all permutations of X is a group under the composition of functions, called the *symmetric group on X*.

Exercise 2.9 Show that if G is a group and H a nonempty subset of G, then H is a subgroup if and only if $xy \in H$ for every $x, y \in H$ and $x^{-1} \in H$ for all $x \in H$. If H is finite then the first condition suffices, i.e., H is a subgroup if and only if $xy \in H$ for all $x, y \in H$.

[5] There are also infinite cyclic groups and they are all isomorphic to the additive group of the integers $\mathbb{Z}$ but in this book we shall be mainly interested in finite groups.

Next we give a couple of elementary properties of the order of an element.

Proposition 2.4 *Let G be a group, $x \in G$ and $t \in \mathbb{Z}$. Then $x^t = 1$ if and only if the order of x divides t.*

Proof If i is the order of x and $t = ik$ for some $k \in \mathbb{Z}$ then $x^t = x^{ik} = (x^i)^k = 1^k = 1$. Conversely, if $x^t = 1$ and i is the order of x, performing the division of t by i we obtain $t = iq + r$ with $0 \leq r < i$. Then $1 = x^t = x^{iq}x^r = x^r$ and, since i is, by definition, the smallest positive integer such that $x^i = 1$, we have that $r = 0$, so that $t = iq$. □

The order of a power may be computed as follows:

Proposition 2.5 *Let G a group, $x \in G$ an element of order i and k an integer. Then x^k has order $i/\gcd(i,k)$.*

Proof Observe that $(x^k)^{i/\gcd(i,k)} = (x^i)^{k/\gcd(i,k)} = 1$. Thus, to prove that $i/\gcd(i,k)$ is the order of x^k it suffices to show that it divides any integer n such that $(x^k)^n = 1$. But if $(x^k)^n = x^{kn} = 1$ we see, by the preceding proposition, that i divides kn. This in turn implies that $i/\gcd(i,k)$ divides n and we are done. □

Next we show how to test whether a given integer n is the order of an element $x \in G$ where G is a group. This will also be useful in finding a generator of G in case G is cyclic, when one applies this to $n = |G|$, but observe that it requires knowledge of the prime factors of n:

Proposition 2.6 *Let n be a positive integer, G a group and $x \in G$. Then n is the order of x in G if and only if $x^n = 1$ and $x^{n/p} \neq 1$ for each prime divisor p of n.*

Proof If n is the order of x then n is the smallest positive integer such that $x^n = 1$ and hence $x^{n/p} \neq 1$ holds for all prime divisors p of n. Conversely, assume that these conditions hold. Since $x^n = 1$ we know from Proposition 2.4 that the order of x divides n. If the order is different from n then it divides some n/p and so $x^{n/p} = 1$, a contradiction. Thus n is the order of x. □

If H is a subgroup of a group G, then H defines an *equivalence relation* on G, namely, a relation which is *reflexive*, *symmetric*, and *transitive*, as follows. Given $x, y \in G$ we say that $x \sim_H y$ if $xy^{-1} \in H$. Then the relation $\sim_H$ is reflexive because $xx^{-1} = 1 \in H$ and symmetric because if $xy^{-1} \in H$ then $yx^{-1} = (xy^{-1})^{-1} \in H$. Transitivity is also clear because if $xy^{-1} \in H$ and $yz^{-1} \in H$ then $xz^{-1} = xy^{-1}yz^{-1} \in H$. The *equivalence class* of $x \in G$, i.e., the set of all elements which are related to x is $Hx = \{hx | h \in H\}$ and, in this case, these sets are called the *right cosets* of H in G. Since any equivalence relation on a nonempty set defines a partition of the set, namely, a decomposition of the set as a disjoint union of the equivalence classes, we have that G is the disjoint union of all the right cosets (this is also easy to prove directly but, for those not familiar with equivalence relations and quotient sets, we refer to [45] or [48]).

Theorem 2.9 (Lagrange's theorem) *If G is a finite group and H a subgroup of G then the order of H divides the order of G.*

Proof We have just noticed that G is the disjoint union of all the right cosets defined by H. Now, observe that all the right cosets have the same number of elements because if Hx and Hy are two different cosets (i.e., x and y are not equivalent in the equivalence relation defined by H) then the map from Hx to Hy that maps hx to hy is a bijection. Thus G is a disjoint union of sets which have $|H|$ elements each (note that H itself is a right coset, namely, the one corresponding to $1 \in G$) and hence $|H|$ divides $|G|$. □

Corollary 2.1 *If G is a finite group and $x \in G$ then the order of x divides the order of G. In particular, $x^{|G|} = 1$.*

Proof The order of $x \in G$ is, as we have seen, the order of the subgroup $\langle x \rangle$ generated by x, so that the order of x divides $|G|$ by Lagrange's theorem. Thus, if i is the order of x we have that $|G| = ti$ for some $t \geq 1$ and hence $x^{|G|} = x^{ti} = (x^i)^t = 1^t = 1$. □

Exercise 2.10 Prove that if H is a subgroup of G, then G is the disjoint union of the left cosets xH, all of which are in bijective correspondence with H.

Exercise 2.11 Show that each subgroup of a cyclic group is also cyclic.

Exercise 2.12 Show that if G is a finite cyclic group and $d||G|$, then there exists exactly one subgroup of G of order d.

If H is a subgroup of G, then the *index* of H in G, denoted $(G : H)$ is defined as the number of right cosets (or, equivalently, of left cosets) of H in G. If G is a finite group and $K \subseteq H \subseteq G$ are subgroups, then it follows from the preceding discussion that $(G : K) = (G : H)(H : K)$. It is also clear that $|G| = (G : \{1\})$ and that $|G| = (G : H)|H|$.

Suppose now that G is an abelian group and H a subgroup of G. Then it is easy to see that the set of cosets, denoted G/H, has the structure of a group with operation given by $HxHy = Hxy$. G/H is then called the quotient group of G modulo H and its order satisfies $|G/H| = |G|/|H|$.

Remark 2.1 Quotient groups may be defined even if the group G is not abelian, provided that H is a *normal* subgroup of G, which means that $xH = Hx$ for every $x \in G$, so that the left and the right cosets coincide. Of course, the subgroups of an abelian group G are all normal.

Let $f : G \to H$ be a homomorphism of groups. Then the *kernel* of f is defined as $\mathrm{Ker} f = \{x \in G | f(x) = 1\} \subseteq G$ and the *image* of f is $\mathrm{Im} f = \{f(x) | x \in G\} \subseteq H$. $\mathrm{Ker} f$ is a normal subgroup of G and $\mathrm{Im} f$ is a subgroup of H. Observe that f is injective if and only if $\mathrm{Ker} f = \{1\}$ (because $f(x) = f(y)$ if and only if $xy^{-1} \in \mathrm{Ker} f$) and that f is surjective if and only if $\mathrm{Im} f = H$.

Proposition 2.7 *Let G be a group and $f : G \to H$ a homomorphism. Then f induces an isomorphism $\bar{f} : G/\mathrm{Ker}\, f \to \mathrm{Im}\, f$.*

Proof Let $K = \mathrm{Ker}\, f$. We define a map $\bar{f} : G/K \to \mathrm{Im}\, f$ by $\bar{f}(Kx) = f(x) \in \mathrm{Im}\, f$, for every $x \in G$. f is well-defined because if $Kx = Ky$, then $xy^{-1} \in K$ and so $f(x)f(y)^{-1} = f(xy^{-1}) = 1$, and multiplying both sides of the equality by $f(y)$ we obtain that $f(x) = f(y)$. Moreover, $\bar{f}$ is clearly a homomorphism and is surjective. On the other hand, $\mathrm{Ker}\, \bar{f} = \{Kx \in G/K \,|\, f(x) = 1\} = \{Kx \in G/K \,|\, x \in K\} = \{K\}$ and hence $\mathrm{Ker}\, \bar{f}$ is the trivial subgroup (its unique element is the identity K of G/K). Thus we see that $\bar{f}$ is indeed an isomorphism. □

Remark 2.2 Note that from Proposition 2.7 it follows that if $f : G \to H$ is a homomorphism, then $(G : \mathrm{Ker}\, f) = |\mathrm{Im}\, f|$. In particular, if f is surjective and $K = \mathrm{Ker}\, f$, then $(G : K) = |H|$.

Next we recall the concept of ring. A *ring* is a set R with at least two elements $0 \neq 1$ and two operations $+, \cdot$, satisfying the following properties:

- $(R, +)$ is an abelian group with identity element 0.
- The multiplication is associative with identity element 1.
- *Distributive laws*. For all $x, y, z \in R$, $x(y + z) = xy + xz$, $(x + y)z = xz + yz$.

If the multiplication of the ring is, furthermore, commutative, then we say that R is a commutative ring. The multiplicative identity 1 is called the *identity* of the ring (sometimes rings without identity are considered but all the rings that we will consider are commutative and with 1). If R is a ring, an element $a \in R$ is a *unit* (or an *invertible element*) if and only if it has a multiplicative inverse, i.e., there exists an element x^{-1} such that $xx^{-1} = x^{-1}x = 1$. It is immediate to see that the set R^* of the units of R is a group with the multiplication induced by that of R (and with identity 1). If R is a commutative ring, an element $x \in R$ is called a *zero divisor* if $x \neq 0$ and there exists an element $y \in R$ such that $y \neq 0$ and $xy = 0$. If x is a zero divisor then x is not a unit of R (note that 0 is neither a unit nor a zero divisor). A commutative ring without zero divisors is called an *integral domain*.

Examples 2.3

1. $\mathbb{Z}$, with the usual addition and multiplication of integers, is a commutative ring. The only units of $\mathbb{Z}$ are 1 and -1 (hence the group of units is $\mathbb{Z}^* = \{-1, 1\} \subseteq \mathbb{Z}$) and there are no zero divisors, so that $\mathbb{Z}$ is an integral domain.
2. $\mathbb{R}$ and $\mathbb{C}$ are rings with ordinary addition and multiplication. All of their nonzero elements are units (a property that, as we shall see below, characterizes the commutative rings that are fields). In particular, these rings are integral domains.

The concepts of ring homomorphism and ring isomorphism are defined similarly to those for groups, as follows:

Definition 2.19 Let R and S be rings and $f : R \to S$ a function. f is a *ring homomorphism* in case it preserves the ring operations and the identity, namely:

1. For all $x, y \in R$, $f(x + y) = f(x) + f(y)$ (i.e., f is a group homomorphism between the additive groups of R and S).
2. For all $x, y \in R$, $f(xy) = f(x)f(y)$.
3. $f(1_R) = 1_S$ (where 1_R and 1_S denote the identity elements of R and S, respectively).

If, moreover, f is bijective, then f is said to be an *isomorphism*.

A class of rings which is especially important is the following:

Definition 2.20 A *field* is a commutative ring with identity in which every nonzero element is invertible.

Example 2.9 $\mathbb{Z}$ is a commutative ring which is not a field because the only invertible elements are 1 and -1. $\mathbb{R}$ and $\mathbb{C}$ are both fields.

Exercise 2.13 Prove that a commutative ring $(R, +, \cdot)$ is a field if and only if $(R - \{0\}, \cdot)$ is a group (in this case $R^* = R - \{0\}$ is called the multiplicative group of R).

Exercise 2.14 Consider the bitwise Xor operation defined on $\mathbb{N}$ as follows. Given two non-negative integers $a, b \in \mathbb{N}$, compute their binary expansions and make them of equal length by adding leading zeros if necessary to the expansion of the smaller number. Then $a \oplus b$ is the integer whose binary expansion is the bit string of the same length which in each position has a 1 if the corresponding bits in the expansions of a and b are different and a 0 if these bits are equal (i.e., the exclusive OR of the corresponding bits of the two expansions is computed). Prove that $(\mathbb{N}, \oplus)$ is an abelian group but show that $(\mathbb{N}, \oplus, \cdot)$ (where $\cdot$ is the multiplication of integers) is not a ring.

2.5.2 Congruences and the Residue Class Ring

All the examples of groups, rings and fields mentioned so far (excluding the trivial group $\{1\}$) are infinite but the examples of cryptographic interest are usually finite. In order to introduce some of the more important ones we start with the definition of congruence.

Definition 2.21 Let $a, b \in \mathbb{Z}$ and $n \in \mathbb{Z}$, $n \geq 2$. We say that *a is congruent to b modulo n*, written $a \equiv b \pmod{n}$, whenever $n | a - b$. If n does not divide $a - b$ then we say that a is not congruent to b modulo n and we denote this fact by $a \not\equiv b \pmod{n}$. The formula $a \equiv b \pmod{n}$ is called a *congruence* and the integer n is the *modulus* of the congruence.

We will not make an exhaustive listing of the properties of congruences, many of which can be found, for example, in [194]. But we recall that the congruence relation modulo n (when viewed as a binary relation on $\mathbb{Z}$) is reflexive, symmetric and transitive, and hence is an equivalence relation. Thus it defines a partition, namely,

it gives a decomposition of $\mathbb{Z}$ as a disjoint union of nonempty equivalence classes which, in this case, are called *congruence classes*. The congruence class or *residue class* of a modulo n is the set $\bar{a} = \{b \in \mathbb{Z} | a \equiv b \pmod{n}\} = \{a + nq | q \in \mathbb{Z}\} = a + n\mathbb{Z}$, where $n\mathbb{Z}$ is the ideal[6] of $\mathbb{Z}$ consisting of all multiples of n. From the definition of congruence, it readily follows that, for $a, b \in \mathbb{Z}$, $a \equiv b \pmod{n}$ if and only if a and b give the same remainder (in the set $\{0, 1, \ldots, n-1\}$) when divided by n, i.e., if and only if $a \bmod n = b \bmod n$.

It is clear then that the set of all congruence classes modulo n has exactly n members and is the set $\mathbb{Z}/n\mathbb{Z} = \{\bar{0}, \ldots, \overline{n-1}\}$. In this set, two binary operations called addition and multiplication are defined in a natural way by setting $\bar{a} + \bar{b} = \overline{a+b}$ and $\bar{a} \cdot \bar{b} = \overline{a \cdot b}$ for $a, b \in \mathbb{Z}$. It is easily checked that these definitions are correct (for the resulting class does not depend on the *representatives* a and b chosen to represent the classes $\bar{a}$ and $\bar{b}$). Moreover, these operations give the set $\mathbb{Z}/n\mathbb{Z}$ the structure of a *commutative ring with identity*, where the required ring properties are derived from those of the addition and the multiplication in $\mathbb{Z}$, of which $\mathbb{Z}/n\mathbb{Z}$ is a *quotient ring* (called the *residue class ring* modulo n). This is just the quotient group $\mathbb{Z}/n\mathbb{Z}$ which, since $n\mathbb{Z}$ is an ideal of the ring, inherits the ring structure of $\mathbb{Z}$.

There is an alternative way of looking at this ring which is perhaps more convenient for our purposes. Let us consider the set $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ (its elements are called the *least non-negative residues* modulo n). The remainder of division of a by n is equal to the unique representative of the congruence class of a modulo n in $\mathbb{Z}_n$, namely $a \bmod n$. Then it is easily checked that the map

$$\begin{array}{ccc} \mathbb{Z}/n\mathbb{Z} & \longrightarrow & \mathbb{Z}_n \\ \bar{a} & \longmapsto & a \bmod n, \end{array}$$

from $\mathbb{Z}/n\mathbb{Z}$ to $\mathbb{Z}_n$, which assigns to each residue class $\bar{a}$ the least non-negative residue of a modulo n, is a bijection. Moreover, if one defines operations in $\mathbb{Z}_n$ by $a + b = (a + b) \bmod n$ and $ab = ab \bmod n$ (we commit an obvious abuse of notation here), so that now addition and multiplication are just the ordinary addition and multiplication of integers followed by reduction modulo n, then the above map preserves these operations and hence we see that $\mathbb{Z}_n$ becomes a ring (commutative and with 1) which is, actually, isomorphic to the ring $\mathbb{Z}/n\mathbb{Z}$. This canonical isomorphism allows us to identify these two rings and to work with whichever of them is more convenient in a given setting. We shall usually work with $\mathbb{Z}_n$ and we remark that any nontrivial alphabet, i.e., any finite set with more than one element, can be identified with one of these sets and hence can be given the corresponding ring structure.

Remark 2.3 We will often use generic notation for the elements and operations of the ring $\mathbb{Z}_n$. For example, if $a, b \in \mathbb{Z}_n$, we may write $a + b$ for their sum in this ring and we only write $(a + b) \bmod n$ if we want to emphasize that the addition in $\mathbb{Z}_n$ proceeds by adding the elements as integers and then taking the least non-negative residue modulo n. A similar remark applies to the product and other standard notation,

[6] An ideal of a (commutative) ring is an additive subgroup which is closed under multiplication by elements of the ring.

like the one for the opposite of an element $a \in \mathbb{Z}_n$, which is denoted generically by $-a$ and is equal to $-a \bmod n = n - a \in \mathbb{Z}_n$. Also, if $a \in \mathbb{Z}_n^*$ is a unit of the ring $\mathbb{Z}_n$, we will denote its inverse by a^{-1}.

We have just seen that all the $\mathbb{Z}_n$ are rings but, in some cases, they have an additional structure which is useful in many cryptographic settings, namely, they are fields. This happens whenever every nonzero element of the ring has a multiplicative inverse. Thus it is easily seen that $\mathbb{Z}_n$ is a field for $n = 2, 3$ or 5 but it is not a field when $n = 4$ or $n = 6$. In the latter case we observe that, since n is not prime, it has a decomposition $n = rs$ where $r, s > 1$. Then we have that $r, s \in \mathbb{Z}_n$ and, in this ring, $rs = 0$. This means that r and s are zero divisors and it is obvious that a zero divisor cannot have a multiplicative inverse for, otherwise, multiplying both terms of the equation $rs = 0$ by r^{-1} we would obtain $s = 0$, a contradiction. This shows that n being prime is a necessary condition for $\mathbb{Z}_n$ to be a field and, using the preceding results, it is also easy to show that this condition is actually sufficient:

Theorem 2.10 *$\mathbb{Z}_n$ is a field if and only if n is a prime number.*

Proof We have already shown the necessity so, to complete the proof of the theorem, we just have to show that if n is prime then every $0 \neq a \in \mathbb{Z}_n$ has an inverse. By Theorem 2.8 there are integers r, s such that $1 = \gcd(a, n) = sa + tn$. Reducing modulo n we obtain that $1 \equiv sa \pmod{n} \equiv ((s \bmod n) \cdot a) \pmod{n}$ which shows that $s \bmod n$ is the inverse of a in $\mathbb{Z}_n$ and completes the proof. □

More generally we have the following theorem whose proof we leave as an exercise:

Theorem 2.11 *An element $a \in \mathbb{Z}_n$ is a unit if and only if a and n are relatively prime.*

Example 2.10 Modular inverses can be easily computed in Maple. A straightforward rendition of the above proof gives us the following function:

```
> modinv := proc(a::nonnegint, p::posint)
   local d, u;
   d:=igcdex(a, p, 'u') mod p;
   if d<>1 then
     error "%1 does not have an inverse modulo %2", a, p
   end if;
   u mod p;
 end:
```

But Maple already has ways to compute these inverses directly, using built-in functions. For example:

```
> modinv(3865,38984), modp(1/3865, 38984), 1/3865 mod 38984;
                                        817, 817, 817
```

2.6 The Chinese Remainder Theorem

The Chinese remainder theorem is a clever method to solve systems of linear congruences that goes back to the Chinese mathematician Sun Tsu (or Sun Zi) who lived sometime between 200 BC and 200 AD. It is said [162] that it was used by the Chinese generals to count the number of soldiers by lining them in multiples of 7, 10, etc. and counting the remaining soldiers in each case. In his classic three-volume book *Mathematics Manual*, Sun Tsu proposed the following problem: find a number that leaves a remainder of 2 when divided by 3, a remainder of 3 when divided by 5, and a remainder of 2 when divided by 7. This amounts to finding a positive integer satisfying the following system of congruences:

$$\left.\begin{array}{l} x \equiv 2 \pmod 3 \\ x \equiv 3 \pmod 5 \\ x \equiv 2 \pmod 7 \end{array}\right\} \tag{2.1}$$

The general version of the theorem about these systems is as follows:

Theorem 2.12 (Chinese remainder theorem, CRT) *Let $m_1, m_2, \ldots, m_n$ be pairwise relatively prime integers greater than 1 and let $a_1, a_2, \ldots, a_n$ be integers. Then the system of congruences*

$$\left.\begin{array}{l} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \qquad \vdots \\ x \equiv a_n \pmod{m_n} \end{array}\right\} \tag{2.2}$$

has a solution which is unique modulo $m = \prod_{i=1}^{n} m_i$.

Proof For each $i = 1, \ldots, n$, let $M_i = m/m_i = \prod_{j \neq i} m_j$. Since the m_i are pairwise relatively prime we have that $\gcd(m_i, M_i) = 1$ for each i such that $1 \leq i \leq n$. Thus each M_i is invertible in $\mathbb{Z}_{m_i}$ by Theorem 2.11 and, using the extended Euclidean algorithm, we may compute integers $y_i \in \mathbb{Z}$ such that for each $1 \leq i \leq n$,

$$y_i M_i \equiv 1 \pmod{m_i}$$

Then we set $x = (\sum_{i=1}^{n} a_i y_i M_i) \bmod m$ and we claim that x is the unique solution of the system (2.2) modulo m. Indeed, we observe that $y_i M_i \equiv 1 \pmod{m_i}$ implies $a_i y_i M_i \equiv a_i \pmod{m_i}$ for $1 \leq i \leq n$ and, since $m_i | M_j$ for each $j \neq i$, we obtain:

$$a_j y_j M_j \equiv 0 \pmod{m_i}$$

for $1 \leq i, j \leq n$, $i \neq j$. We see that $a_i y_i M_i$ is a solution of the ith congruence and does not contribute to the remaining ones, so the sum of all these terms satisfies

$$x \equiv a_i y_i M_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} a_j y_j M_j \equiv a_i \pmod{m_i}$$

for $1 \leq i \leq n$. This proves the existence and, for the uniqueness, let us note that if x' is another solution of the system, then $x \equiv x' \pmod{m_i}$ and, since the m_i are pairwise relatively prime, this clearly implies $x \equiv x' \pmod{m}$. □

Example 2.11 Let us consider Sun Tsu's problem (Eq. (2.1)). We compute:

$$\begin{aligned} m &= 3 \cdot 5 \cdot 7 = 105, \\ M_1 &= m/m_1 = 105/3 = 35, \quad y_1 = M_1^{-1} \bmod m_1 = 35^{-1} \bmod 3 = 2; \\ M_2 &= m/m_2 = 105/5 = 21, \quad y_2 = M_2^{-1} \bmod m_2 = 21^{-1} \bmod 5 = 1; \\ M_3 &= m/m_3 = 105/7 = 15, \quad y_3 = M_3^{-1} \bmod m_3 = 15^{-1} \bmod 7 = 1. \end{aligned}$$

Thus we obtain the solution:

$$x = \left(\sum_{i=1}^{3} a_i y_i M_i \right) \bmod m = (2 \cdot 2 \cdot 35 + 3 \cdot 1 \cdot 21 + 2 \cdot 1 \cdot 15) \bmod 105 = 23.$$

Of course, this can be also done with Maple, which has the CRT algorithm implemented in the function `chrem`. To solve Sun Tsu's problem we simply compute:

```
> chrem([2, 3, 2], [3, 5, 7]);
                                  23
```

Remarks 2.2

1. An important feature of the proof of the CRT given above is that it is constructive and it actually gives an algorithm for computing the solution(s).
2. The y_i and the M_i defined in the proof of Theorem 2.12 do not depend on the a_i (they depend only on the m_i). This is often useful in applications where the m_i are fixed, for then the y_i and the M_i can be pre-computed and one only has to substitute the a_i in the formula $x = (\sum_{i=1}^{n} a_i y_i M_i) \bmod m$ to obtain the solution in a faster way.
3. The modern interpretation of the CRT places the theorem in the setting of commutative ring theory as we will see in Theorem 2.14 below.

The complexity of the algorithm for the CRT given in the proof of Theorem 2.12 can be estimated as follows:

Theorem 2.13 *The CRT algorithm to solve the congruence system*

$$\left.\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_n \pmod{m_n} \end{aligned}\right\} \tag{2.3}$$

where $m_1, m_2, \ldots, m_n$ are pairwise relatively prime integers greater than 1 and $a_1, a_2, \ldots, a_n$ are integers, runs in time $O(\text{len}(m)^2)$, where $m = \prod_{i=1}^{n} m_i$.

Proof Computing $m = \prod_{i=1}^{n} m_i$ requires time $O(\text{len}(m)(\sum_{i=1}^{n} \text{len}(m_i)) = O(\text{len}(m)^2)$. The computation of all the M_i requires time $O(\text{len}(m)^2)$ as well. Computing the $y_i \in \mathbb{Z}$, $1 \leq i \leq n$, such that $y_i M_i \equiv 1 \pmod{m_i}$ requires, by Theorem 2.7 time $O(\sum_{i=1}^{n} \text{len}(m_i)\text{len}(M_i)) = O((\sum_{i=1}^{n} \text{len}(m_i))\text{len}(m)) = O(\text{len}(m)^2)$. Finally, the computation of $x = (\sum_{i=1}^{n} a_i y_i M_i) \mod m$ also requires time $O(\text{len}(m)^2)$, so the total time estimate is $O(\text{len}(m)^2)$. □

2.6.1 The Chinese Remainder Theorem and the Residue Class Ring

The Chinese remainder theorem has a natural interpretation in terms of the residue class ring $\mathbb{Z}_m$, where $m = \prod_{i=1}^{n} m_i$ is the product of the pairwise relatively prime moduli. First we define the *direct product* of a family of rings (note that the direct product of a family of groups can be similarly defined):

Definition 2.22 Let $\{R_i\}_{i=1}^{n}$ be a family of rings. The *direct product* of this family is the ring $\prod_{i=1}^{n} R_i = \{(x_1, x_2, \ldots, x_n) | x_i \in R_i\}$ with addition and multiplication defined component-wise, i.e., given two tuples $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$, their sum is $(x_1 + y_1, \ldots, x_n + y_n)$ and their product $(x_1 y_1, \ldots, x_n y_n)$.

An alternative notation for the direct product is $R_1 \times R_2 \times \cdots \times R_n$ and, in particular, the direct product of two rings R, S is commonly denoted by $R \times S$.

Example 2.12 Consider the rings $\mathbb{Z}_3$ and $\mathbb{Z}_4$. Then their direct product is the ring $\mathbb{Z}_3 \times \mathbb{Z}_4 = \{(0, 0), (0, 1), \ldots, (2, 2), (2, 3)\}$, which has 12 elements (in general, $|R \times S| = |R| \times |S|$). Note that the identity of this ring is the element $(1, 1)$.

Now, the Chinese remainder theorem can be reformulated as follows:

Theorem 2.14 (Chinese remainder isomorphism) *Let $\{m_i\}_{i=1}^{n}$ be a family of pairwise relative prime integers and $n = \prod_{i=1}^{n} m_i$. Then the map*

$$\begin{array}{ccc} \mathbb{Z}_m & \xrightarrow{f} & \mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_n} \\ x & \longmapsto & (x \bmod m_1,\ x \bmod m_2, \ldots, x \bmod m_n) \end{array}$$

is a ring isomorphism. Moreover, this map induces by restriction an isomorphism between the unit groups of these rings, namely an isomorphism

$$\mathbb{Z}_m^* \xrightarrow{f} \mathbb{Z}_{m_1}^* \times \cdots \times \mathbb{Z}_{m_n}^*.$$

Proof It is clear that f is a well-defined function. It is also easy to show that f is a ring homomorphism. For example, if $x, y \in \mathbb{Z}_m$ we have that $((x+y) \bmod m) \bmod m_i = (x+y) \bmod m_i = (x \bmod m_i + y \bmod m_i) \bmod m_i$ and a similar formula holds for the product. Now, in order to prove that f is bijective it suffices to show, for example, that f is surjective because both its domain and its codomain have the same number of elements. But, given an element $(x_1, x_2, \ldots, x_n) \in \mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_n}$, the Chinese remainder theorem (Theorem 2.12) implies that the system of congruences

$$\left.\begin{array}{l} x \equiv x_1 \pmod{m_1} \\ x \equiv x_2 \pmod{m_2} \\ \qquad \vdots \\ x \equiv x_n \pmod{m_n} \end{array}\right\} \tag{2.4}$$

has a solution which is unique modulo $m = \prod_{i=1}^n m_i$ and this gives an element of $\mathbb{Z}_m$ whose image is $(x_1, x_2, \ldots, x_n)$ (actually, the CRT gives also the injectivity of f because the uniqueness of this solution shows that the element of $\mathbb{Z}_m$ that maps to the given element of the product ring is unique).

For the final part observe that a ring isomorphism such as f has the property that $f(x)$ is a unit if and only if so is x. Indeed, if x is a unit with inverse x^{-1}, then $f(xx^{-1}) = f(1) = 1 = f(x)f(x^{-1})$, so that $f(x)$ is also a unit and the converse follows in a similar way using the inverse isomorphism. Thus the last claim in the statement of the theorem follows bearing in mind that the unit group of the product ring is $\mathbb{Z}_{m_1}^* \times \cdots \times \mathbb{Z}_{m_n}^*$ as it is clear that an element in the product ring is a unit if and only if each of its components is a unit in the corresponding factor ring. □

Exercise 2.15 Show that Theorems 2.12 and 2.14 are equivalent in the sense that not only is the latter a corollary of the former as we have seen but also the former can be deduced from the latter.

Exercise 2.16 Show that the ring $\mathbb{Z}_4$ is not isomorphic to the product $\mathbb{Z}_2 \times \mathbb{Z}_2$ and that their unit groups are not isomorphic either.

As we will see later, one interesting application of the CRT is the speed-up of RSA decryption. Another interesting application is in multiprecision arithmetic and in the construction of the so-called *residue computers* that use the CRT to subdivide large computations in $\mathbb{Z}_{\prod_{i=1}^n m_i}$ into several computations in the smaller rings $\mathbb{Z}_{m_i}$. This can provide a considerable efficiency gain because, on the one hand, the computations in

these rings can be done completely in hardware, without having to use multiprecision arithmetic and, on the other, they can easily be parallelized.

Exercise 2.17 Write a Maple function that computes $n!$ for a large integer n (say $n > 10000$) by computing the result modulo a sufficient number of primes preceding 2^{31} (which may be estimated by means of Stirling's formula, see [94]). These computations may be carried out with machine-size integers of type `integer[4]` and then the CRT may be used to compute the final result.

2.7 Euler's Theorem and Modular Exponentiation

2.7.1 Euler's Theorem

We have seen that the units of a commutative ring form an abelian group and, in particular, the group of units of $\mathbb{Z}_n$ is, by Theorem 2.11 , $\mathbb{Z}_n^*=\{a \in \mathbb{Z}_n | \gcd(a, n)=1\}$.

There is a result of Euler related to this group which, after being proved in the first half of the eighteenth century, found an interesting cryptographic application more than two centuries later for, as we shall see, it is crucial for the RSA encryption scheme. In order to state and prove this result we need the following concept:

Definition 2.23 Let $n \in \mathbb{Z}, n \geq 2$. The Euler ϕ-function (also called totient function) of n is defined by:

$$\phi(n) = |\{x \in \mathbb{Z} | 1 \leq x \leq n \text{ and } \gcd(x, n) = 1\}|.$$

In other words, $\phi(n)$ is the order of $\mathbb{Z}_n^*$, i.e.

$$\phi(n) = |\mathbb{Z}_n^*|.$$

Remark 2.4 Note that, although we do not consider congruences modulo 1, the definition of Euler's ϕ-function extends naturally to all positive integers by setting $\phi(1) = 1$.

Euler's theorem is then the following:

Theorem 2.15 (Euler's theorem) *Let a and n be positive integers such that $n \geq 2$ and $\gcd(a, n) = 1$. Then $a^{\phi(n)} \equiv 1$ (mod n).*

Proof Observe that the statement of this theorem may be formulated in terms of the group $\mathbb{Z}_n^*$. Saying that $\gcd(a, n) = 1$ is equivalent to saying that $a \bmod n$ is an element of $\mathbb{Z}_n^*$. The statement of the theorem is then equivalent to saying that, in the group $\mathbb{Z}_n^*$ we have $(a \bmod n)^{\phi(n)} = 1$, and this follows from Corollary 2.1. □

Exercise 2.18 Use Euler's theorem to compute the last (decimal) digit of 7^{2010}. (Hint: Apply Euler's theorem modulo 10.)

Observe now that, whenever p is a prime number, $\mathbb{Z}_p^* = \mathbb{Z}_p - \{0\}$ and hence $\phi(p) = p - 1$. This gives the following corollary of Euler's theorem:

Corollary 2.2 (Fermat's Little Theorem) *Let p be a prime number and a an integer such that p does not divide a. Then $a^{p-1} \equiv 1 \pmod{p}$.*

Corollary 2.3 *Let p be a prime and a, e, positive integers such that p does not divide a. Then $a^e \equiv a^{e \bmod (p-1)} \pmod{p}$.*

Proof We have that $e = (p-1)q + e \bmod (p-1)$ for some non-negative integer q. Then, by Fermat's little theorem, $a^{p-1} \equiv 1 \pmod{p}$ and we have:

$$a^e \equiv a^{(p-1)q} a^{e \bmod (p-1)} \equiv a^{e \bmod (p-1)} \pmod{p}.$$

□

Exercise 2.19 Use the previous corollary to find the last decimal digit of 7^{2574}. (Hint: Compute $7^{2574} \bmod 2$ and $7^{2574} \bmod 5$ and use these values to compute $7^{2574} \bmod 10$.)

Remark 2.5 We see that Euler's theorem is really a particular case of Lagrange's theorem for groups, obtained by applying the latter—or its Corollary 2.1—to the group $\mathbb{Z}_n^*$ and similarly for Fermat's little theorem and the group $\mathbb{Z}_p^*$, where p is prime.

Example 2.13 In Maple, Euler's ϕ-function is given by the command `numtheory:-phi`. For example, the values of ϕ on the first 20 positive integers are (`map` may be used instead of ~ in Maple versions prior to v13):

```
> numtheory:-phi~([$1 .. 20]);
      [1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16, 6, 18, 8]
```

We show that the ϕ-function gives the number of generators in a finite cyclic group:

Theorem 2.16 *Let G be a finite cyclic group of order n and $g \in G$ a generator. Then the generators of G are the elements g^t with $t \in \{1, \ldots, n-1\}$ such that $\gcd(t, n) = 1$. In particular, G has $\phi(n)$ generators.*

Proof Since $G = \langle g \rangle$ and $|G| = n$, we have that $G = \{g^t | 0 \leq t < n\}$. Among these elements, the generators are those with order n which, by Proposition 2.5 are the g^t such that $n/\gcd(n, t) = n$, which in turn is equivalent to $\gcd(n, t) = 1$. The number of generators is then equal to $\phi(n)$ by definition of ϕ. □

Exercise 2.20 Prove that, for each integer $n \geq 2$, the additive group $\mathbb{Z}_n$ is cyclic and its generators are the elements $g \in \mathbb{Z}_n$ such that $\gcd(g, n) = 1$.

To derive a formula to compute ϕ we first observe that it is a *multiplicative function*, namely:

Theorem 2.17 *If n_1 and n_2 are relatively prime positive integers ≥ 2 then $\phi(n_1 n_2) = \phi(n_1)\phi(n_2)$.*

Proof This follows from Theorem 2.14 which, in particular, gives a bijection between $\mathbb{Z}^*_{n_1 n_2}$ and $\mathbb{Z}^*_{n_1} \times \mathbb{Z}^*_{n_2}$. □

The preceding theorem, together with the fundamental theorem of arithmetic, reduces the computation of $\phi(n)$ to the case in which n is a prime power. If $n = p^k$, where p is prime and $k > 0$, the only numbers between 1 and n which are not relatively prime to n are $p, 2p, 3p, \ldots, p^{k-1}p = p^k$, so that $\phi(p^k) = p^k - p^{k-1}$. From this, using the multiplicativity of ϕ, we have:

Theorem 2.18 *If $n = \prod_{i=1}^k p_i^{e_i}$ is the prime factorization of n, then*

$$\phi(n) = n \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right).$$

Proof Bearing in mind the previous remark we have that $\phi(n) = \prod_{i=1}^k \phi(p_i^{e_i}) = \prod_{i=1}^k (p_i^{e_i} - p_i^{e_i - 1}) = \prod_{i=1}^k (p_i^{e_i} - \frac{p_i^{e_i}}{p_i}) = \prod_{i=1}^k p_i^{e_i}(1 - \frac{1}{p_i}) = n \prod_{i=1}^k (1 - \frac{1}{p_i})$. □

We next record a useful consequence of the multiplicativity of the ϕ-function:

Proposition 2.8 $\sum_{d|n} \phi(d) = n$.

Proof Let $f(n) = \sum_{d|n} \phi(d)$, where d runs on the set of the positive divisors of n. We have to show that $f(n) = n$ and, in order to do that, we first show that f is a multiplicative function, i.e., that $f(n_1 n_2) = f(n_1) f(n_2)$ whenever $\gcd(n_1, n_2) = 1$. Observe that the divisors of $n_1 n_2$ can be written in a unique way in the form $d_1 d_2$ where $d_1 | n_1$ and $d_2 | n_2$ and so, bearing in mind that ϕ is multiplicative we have:

$$f(n_1 n_2) = \sum_{d_1|n_1, d_2|n_2} \phi(d_1)\phi(d_2) = \left(\sum_{d_1|n_1} \phi(d_1)\right)\left(\sum_{d_2|n_2} \phi(d_2)\right) = f(n_1) f(n_2)$$

Now let $n = \prod_{i=1}^r p_i^{e_i}$ be the prime factorization of n. Then, by the multiplicativity of f we have that $f(n) = \prod_{i=1}^r f(p_i^{e_i})$, so that it suffices to prove that f is the identity on prime powers, i.e., that $f(p^e) = p^e$ whenever p is prime. But, as we have seen, $\phi(p^i) = p^i - p^{i-1}$ for $i \geq 1$ and hence we have:

$$f(p^e) = \sum_{i=0}^{e} \phi(p^i) = 1 + \sum_{i=1}^{e} (p^i - p^{i-1}) = p^e,$$

which completes the proof. □

2.7.2 Modular Exponentiation

One of the key ideas underlying the birth of public-key cryptography in the 1970s was to use as encryption function a trapdoor one-way function, which can informally be described as a function that is easy to compute but hard to invert (namely, a one-way function, which will be formally defined later) and, moreover, has a trapdoor, i.e., information whose knowledge allows the efficient inversion of the function. The first candidate for such a function was proposed in 1978 by Rivest, Shamir and Adleman as the basis of what would become the RSA cryptosystem. The underlying hypothesis on which RSA is based is the following:

- *Modular exponentiation with fixed exponent and modulus is a trapdoor one-way function*

Let us analyze the meaning of this assertion in some detail. The idea is that, if n is a large integer (where the meaning of 'large' will be made precise when discussing RSA) which is the product of two distinct primes p and q of approximately the same size, and e is a positive exponent such that $\gcd(e, (p-1)(q-1)) = 1$ (where the latter condition just means that $e \in \mathbb{Z}^*_{\phi(n)}$), then the function

$$\begin{array}{ccc} \mathbb{Z}_n & \longrightarrow & \mathbb{Z}_n \\ m & \mapsto & m^e \bmod n, \end{array}$$

is trapdoor one-way. This means that the following conditions should hold:

1. Computing $c = m^e \bmod n$ is easy (in the complexity-theoretic sense of the term).
2. The inverse problem of *extracting* eth *roots modulo* n, that is, 'given a random $c \in \mathbb{Z}_n$ and a positive integer $e \in \mathbb{Z}^*_{\phi(n)}$ (we may also assume that $1 < e < \phi(n) - 1$ to exclude trivial cases), compute m such that $c = m^e \bmod n$' is hard.
3. If the prime factorization of n is known, then the computation of eth roots modulo n is easy (so that the factorization of n would be the trapdoor).

As we shall soon see, conditions 1 and 3 above do indeed hold and the only thing that remains to be proved in order to ensure that modular exponentiation is a trapdoor one-way function is condition 2. Because of this, the assertion that the function is one-way is so far only a conjecture and, more generally, the existence of one-way functions—which, if true, would imply the truth of the $\mathcal{P} \neq \mathcal{NP}$ conjecture—has not been proved. But it is widely believed that some candidates such as modular exponentiation are indeed trapdoor one-way and this is the reason why they are currently used in cryptography.

We leave for later the discussion of conditions 2 and 3 above and we shall now examine condition 1, which is easily shown to hold. To prove 1 we have to show that modular powers can be efficiently computed and, as we will see when discussing RSA, this also implies 3. More precisely, in terms of complexity theory we will show that modular exponentiation can be done in polynomial time. It is not completely obvious how to do this and, to analyze the difficulties that may arise, let us try to compute an instance with Maple.

Example 2.14 Let us try to do a modular exponentiation by means of the naive algorithm consisting in multiplying the basis by itself $e - 1$ times (where e is the exponent):

```
> 21293^458976453419787627567139383 9 mod 29848327924656663721619585710 3;
Error, numeric exception: overflow
```

Here, Maple was trying to compute $21293^{458976453419787627567139383 9}$ before reducing it modulo $n = 29848327924656663721619585710 3$. But this power is an enormous number because, as can be seen by taking the logarithm, the size of a power grows proportionally to the exponent, or, in other words, exponentially. Thus this number does not fit in the computer's memory producing the overflow error. This problem is easily corrected because all operations can be carried out modulo n, i.e., after each multiplication, the remainder of the result divided by n can be computed and the multiplication result can be replaced by this remainder, so that no numbers greater than $n - 1$ are ever multiplied. Let us try this in Maple:

```
> z := 1:
  for i to 458976453419787627567139383 9 do
    z := 21293*z mod 29848327924656663721619585710 3
  end do:
  z;
Warning,  computation interrupted
```

Now memory was not a problem but we had to stop the computation because Maple would be unable to complete it in our lifetime. Actually, we were asking Maple to carry out 458976453419787627567139383 8 multiplications and the same number of divisions by 29848327924656663721619585710 3. Even assuming a computer powerful enough for Maple to carry out one billion of these multiplication/division pairs per second, this computation would take:

```
> 458976453419787627567139383 8./(365*24*3600*10^9);
                                1.455404786*10^11
```

years. Taking into account that the age of the universe is variously estimated at between 12 and 14 billion years (i.e., less than $14 \cdot 10^9$ years), we see that the entire life of the universe would not be enough for this computation. Moreover, it should be taken into account that the involved numbers in the example are relatively small and, at any rate, much smaller than the numbers used in cryptographic applications. The naive exponentiation algorithm requires a number of multiplications (and divisions) that is just one less than the exponent. Thus the total number of operations is proportional to the exponent (and not to its size!) and hence the algorithm runs in exponential time.

Fortunately, we do not have to use the algorithm in the preceding example since there is a much more efficient algorithm which runs in polynomial time, with the number of operations bounded by a multiple of the number of binary digits in the exponent. This algorithm is variously known as *fast exponentiation*, *modular exponentiation*, *binary exponentiation*, *square and multiply algorithm* … and the basic idea is the following. Suppose that we want to compute $m^e \bmod n$ and that the binary expansion of the exponent is $e = (e_k e_{k-1} \dots e_1 e_0)_2 = \sum_{i=0}^{k} e_i 2^i$ where $e_i \in \mathbb{Z}_2$. Then

$m^e \bmod n = m^{\sum_{i=0}^{k} e_i 2^i} \bmod n = \prod_{i=0}^{k} (m^{2^i})^{e_i} \bmod n = \prod_{0 \le i \le k,\ e_i = 1} m^{2^i} \bmod n$. This suggests the following algorithm:

Algorithm 2.4. Modular exponentiation algorithm.

Input: $n \ge 2$, $m \in \mathbb{Z}_n$, and $e \ge 0$.
Output: $m^e \bmod n$.

1. `Compute` the binary expansion of e, $e = (e_k e_{k-1} \dots e_1 e_0)_2 = \sum_{i=0}^{k} e_i 2^i$.
2. `Compute` the powers $m^{2^i} \bmod n$ for $0 \le i \le k$ by successive squarings (note that $m^{2^{i+1}} = (m^{2^i})^2$).
3. `Compute` $m^e \bmod n$ as the product of those $m^{2^i} \bmod n$ for which the corresponding bit is $e_i = 1$.
4. `return` $m^e \bmod n$.

As we are going to see, this algorithm is implemented in Maple and allows the computation of extremely large exponentiations with great efficiency. But, before discussing Maple's built-in function for this purpose, we are going to exhibit a straightforward implementation that makes it easy to understand the simplicity and elegance of the algorithm. The algorithm can be rendered in Maple as follows:

```
> ModExp := proc(m::nonnegint, e::nonnegint, n::posint)
  local z, y, t;
  if m = 0 and e = 0 then
    error "0^0 is undefined"
  end if;
  z := m;
  y := 1;
  t := e;
  while 0 < t do
    if irem(t, 2, 't') = 1 then
      y := irem(y*z, n)
    end if;
    z := irem(z*z, n)
  end do;
  y
  end proc:
```

In this function the operations described in Algorithm 2.4 are carried out in a slightly different order. In the while loop, the binary expansion of e is being computed by means of successive divisions by 2 and, after computing each bit, a multiplication by m followed by a squaring is done if the corresponding bit is 1, while if the bit is 0 only the squaring is performed; note that in this process the bits of e are being scanned in the same order as they are generated, i.e., from the least significant to the most significant one or, in other words, from 'right to left'. It is then clear that, discounting the initial iteration of the loop which is trivial as at most it only requires multiplying by 1 and squaring 1, if e has $k+1$ bits and exactly $r+1$ of these bits are equal to 1, then the total number of modular operations (either multiplications or squarings) performed is $k+r$. This amounts to a total of $k+r$ integer multiplications of numbers less than n and the same number of divisions to reduce modulo n. To estimate the complexity of this algorithm note that the number of iterations of the

while loop is the number of bits of e which is $O(\text{len}(e))$. Each modular operation requires $O(\ln^2 n) = O(\text{len}(n)^2)$ bit operations (since $m < n$). Furthermore, we will usually assume that $e < n$ (since, by Euler's theorem, if $\gcd(m, n) = 1$ then $m^e \bmod n = m^{e \bmod \phi(n)} \bmod n$) and hence $O(\text{len}(e)) = O(\text{len}(n))$, so that the running time of the algorithm is $O(\text{len}(n)^3)$.

We are now going to explore a couple of variants of the binary exponentiation algorithm. Since they are quite simple, we do not describe them in detail, and we just give Maple implementations from which the structure of these variants is easily deduced. We start with a recursive version of the algorithm which can be easily implemented in Maple as follows:

```
> RecModExp := proc(m::nonnegint, e::nonnegint, n::posint)
  if e = 0 then
    if m <> 0 then
      return 1
    else
      error "0^0 is undefined"
    end if
  end if;
  if m = 0 then
    return 0
  end if
  if irem(e,2)=0 then
    return RecModExp(m^2 mod n, iquo(e,2),n)
  end if;
  (m*RecModExp(m^2 mod n, iquo(e,2),n)) mod n;
  end proc:
```

A slightly different version of the algorithm is the 'left to right' version in which the bits of the exponent are scanned from the most significant to the least significant one (note that Maple uses the little-endian convention so that when using Maple's `convert/base` function the bits of the list produced by it are actually scanned from right to left). A Maple procedure implementing this variant is given in the next function `ExpMod`, where we assume that $n > 2$ and $m < n$. Here we use Maple's powerful left-fold operator `foldl` to perform the sequence of squarings and multiplications (see Maple's help for a description of this function) but it is easy to replace the use of `foldl` by a '`for` loop' performing the same operations:

```
> ExpMod := proc(m::nonnegint, e::nonnegint, n::posint)
  local bits, x, y;
  if e = 0 then
    if m <> 0 then
      return 1
    else
      error "0^0 is not defined"
    end if
  end if;
  bits := op(ListTools:-Reverse(convert(e, base, 2)[1 .. -2]));
  foldl((x, y) -> modp(modp(x^2, n)*m^y, n), m, bits)
  end proc:
```

Exercise 2.21 Write a version of the function `ExpMod` that, instead of using the command `foldl`, uses a `for` loop.

Of course, Maple also has a built-in function implementing the binary exponentiation algorithm, namely `Power`, the inert power function, which is also represented by

the 'infix operator' &^. As was to be expected, this function is very fast and we will use it extensively in the sequel as a component of many of the algorithms implemented.

Example 2.15 Let us consider again the exponentiation we tried to compute in Example 2.14. Now we see that any of the variants of the binary method is able to carry out this computation quickly:

```
> 21293&^458976453419787627567139383 9 mod 29848327924656663721619585710 3;
  ModExp(21293, 458976453419787627567139383 9, 29848327924656663721619585710 3);
  RecModExp(21293, 458976453419787627567139383 9, 29848327924656663721619585710 3);
  ExpMod(21293, 458976453419787627567139383 9, 29848327924656663721619585710 3);
                    157396720784600298623131794157
                    157396720784600298623131794157
                    157396720784600298623131794157
                    157396720784600298623131794157
```

2.7.2.1 Speeding up Modular Exponentiation with the Chinese Remainder Theorem

As we shall see, modular exponentiation is the basic operation one has to perform when doing RSA encryption or decryption. In the case of decryption, the modulus is a product of two known large primes and the speed of the binary exponentiation algorithm can be significantly increased. Indeed, given a modulus whose prime factorization is known, one can apply the Chinese remainder theorem to recover the final result from the smaller exponentiations corresponding to the different prime-power factors of the modulus. We next describe how to do this in Maple when the modulus is—as in the RSA case—a product of two different primes.

Suppose that we want to compute $x = m^e \bmod n$, where $n = pq$, with p and q distinct primes not dividing m. Let $x_p = x \bmod p$ and $x_q = x \bmod q$. Then we have that $x_p = (m^e \bmod n) \bmod p = (\text{since } p|n)\ m^e \bmod p = (\text{by Corollary 2.3})\ m^{e \bmod p-1} \bmod p$ and, similarly, $x_q = m^e \bmod q = m^{e \bmod q-1} \bmod q$. Thus the Chinese remainder theorem implies that $x = m^e \bmod n$ is the unique solution modulo n of the system of congruences:

$$\left.\begin{aligned} x &\equiv x_p \pmod{p} \\ x &\equiv x_q \pmod{q} \end{aligned}\right\} \tag{2.5}$$

The computations of x_p and x_q require $O(\text{len}(p)^3)$ and $O(\text{len}(q)^3)$ bit operations respectively. If we assume that p and q have approximately the same length, we will have $\text{len}(p) \approx \text{len}(q) \approx \frac{1}{2}\text{len}(n)$ and hence each of these two computations requires approximately one-eight of the time required by the direct computation of $x = m^e \bmod n$ in time $O(\text{len}(n)^3)$. Thus the time taken by the computations of x_p and x_q will be approximately one-fourth of the time required by computing x directly. Since the time taken to apply the CRT is $O(\text{len}(n)^2)$, which is faster, we see that the use of the CRT makes the process almost four times faster when n is large (in other words, the computation that uses the CRT will require time roughly

proportional to $4\text{len}(p)^2 + 2\text{len}(p)^3$ while the standard computation will require time proportional to $8\text{len}(p)^3$).

This algorithm is implemented in Maple as follows:

```
> ChineseModExp :=
  (m,e,p,q) -> chrem([Power(m,e mod (p-1)) mod p,Power(m,e mod (q-1)) mod q],[p,q]):
```

Example 2.16 We perform an experiment to test whether the running times of the two exponentiation algorithms—with and without the CRT—in Maple behave as expected. For this we will use two 1024-bit primes and an exponent chosen as follows:

```
> p := nextprime(2^1023):
  q := prevprime(2^1024):
  n := p*q:
  e := rand(1..(p-1)*(q-1))():
  e1:= e mod (p-1):
  e2:= e mod (q-1):
```

We remark that the primes `p` and `q` above are decidedly non-random and are not suitable for cryptographic use but they are OK for the purposes of this experiment. The next function performs `k` modular exponentiations (where `k` is a positive integer) and outputs the CPU time spent. Note that the bases `m` are chosen by means of Maple's default pseudo-random algorithm[7] in the range `2..n-2`. In fact, we do not seed this generator (this would be accomplished, for example, by the use of `randomize()`) so that, after calling `restart`, the same sequence of bases is always obtained. This way, these bases are far from random but we can use the same ones to compare the speed of the two methods.

```
> ChineseTimeTest := proc(k)
  local i, m, x, t;
  t := time();
  for i to k do
    m := (rand(2 .. n-2))();
    x := chrem([Power(m, e1) mod p, Power(m, e2) mod q], [p, q]);
  end do;
  time()-t
  end proc:
```

The next function is similar to the preceding one and records the time spent by Maple's modular exponentiation algorithm without using the CRT. As mentioned before, after a `restart` and the recomputation of the primes and the exponent, the same numbers will be used in both tests.

```
> TimeTest := proc(k)
  local i, m, x, t;
  t := time();
  for i to k do
    m := (rand(2 .. n-2))();
    x := Power(m, e) mod n;
  end do;
  time()-t
  end proc:
```

[7] Pseudo-random number generators and, in particular, those used by Maple, are discussed in Chap. 3.

Now we measure the time taken by `200` iterations of the test by the Chinese method:

```
> ChineseTimeTest(200);
                                  2.000
```

Next, after restarting Maple's kernel and recomputing all the parameters, we do the timing for the ordinary exponentiation:

```
> TimeTest(200);
                                  6.922
```

We see that the method that uses the CRT was, for these numbers, almost 3.5 times faster than the ordinary method. A quick check that both methods give the same result when these parameters are used is the following:

```
> a := rand(2..n-1)():
  evalb(Power(a,e) mod n = chrem([Power(a,e1) mod p,Power(a,e2) mod q],[p,q]));
                                  true
```

Exercise 2.22 Write a Maple procedure that extends the above method to any positive modulus whose prime factorization is known. The procedure should compute the exponentiation modulo each prime power dividing the modulus and then use the CRT to compute the final result.

2.7.3 Finding Generators in $\mathbb{Z}_p^*$

The binary exponentiation method, together with Proposition 2.6, gives an efficient algorithm to test whether a given integer, whose prime factors are known, is the order of an element of a group. In particular, this can be applied to find a generator of a cyclic group of order n (given the factorization of n). One of the more important classes of cyclic groups is given by the following proposition whose proof will be given later on, in Corollary 2.6:

Proposition 2.9 *If p is prime then the group $\mathbb{Z}_p^*$ is cyclic of order $p-1$.*

When the group $\mathbb{Z}_n^*$ is cyclic, a generator of this group is also called a *primitive root* modulo n (more generally, any integer a such that $\gcd(a,n)=1$ and $a \bmod n$ is a generator of $\mathbb{Z}_n^*$ is also called a primitive root modulo n). We will now consider the problem of finding a primitive root modulo a prime.

Example 2.17 Let us consider the following prime:

```
> nextprime(1000);
                                  1009
```

If $p = 1009$ then $p-1$ has the following factorization:

```
> ifactor(1008);
                              (2)^4 (3)^2 (7)
```

Using this information we try to find a generator of $\mathbb{Z}^*_{1009}$, i.e., an element of $\mathbb{Z}^*_{1009}$ of order 1008. By Proposition 2.6 we have that, since the quotients of 1008 by its three prime factors are 504, 336 and 144, we need to find an element g in the group such that $g^{504} \neq 1$, $g^{336} \neq 1$ and $g^{144} \neq 1$. We start by trying the element $2 \in \mathbb{Z}^*_{1009}$:

```
> 2&^504 mod 1009;
                              1
```

This already tells us that 2 is not a primitive root modulo 1009. The same happens with the successive values 3, 5, …(some values such as 4 need not be tested because once we know that 2 is not a generator we also know that $4 = 2^2$ is not a generator either) until 11 for which we find:

```
> 11&^504 mod 1009;
                           1008
> 11&^336 mod 1009;
                            374
> 11&^144 mod 1009;
                            935
```

Thus we see that 11 satisfies the conditions of Proposition 2.6 and hence is a primitive root modulo 1009.

The preceding example suggests the following algorithm to find a generator of $\mathbb{Z}^*_p$ assuming that the prime factors of $p-1$ are known:

- Choose an element $x \in \mathbb{Z}^*_p$ at random and check whether it satisfies $x^{(p-1)/q} \neq 1$ for each prime factor q of $p-1$.
- If for some prime factor q this does not hold, discard x and choose a new random element, repeating the process until a generator is found.

This method gives a probabilistic algorithm which runs in expected polynomial time on $\text{len}(p)$. The algorithm performs a sequence of Bernoulli trials (choosing elements of $\mathbb{Z}^*_p$ at random) with success probability $\phi(p-1)/(p-1)$ (since, by Theorem 2.16, there are $\phi(p-1)$ generators out of $p-1$ elements in the group) and the expected value of the random variable that estimates the number of trials for the first success is $(p-1)/\phi(p-1)$ by Proposition 2.2. Since checking whether a candidate element is a primitive root requires one modular exponentiation for each prime factor of $p-1$ and the number of these prime factors is clearly $O(\text{len}(p))$, we see that each trial requires polynomial time on $\text{len}(p)$ and hence to show that the generator-finding algorithm runs in expected polynomial time it suffices to see that the expected number of trials is also polynomial in $\text{len}(p)$. This follows from a theorem from analytic number theory due to Rosser and Schoenfeld (see, e.g., [7, Theorem 8.8.7] or also [60, Exercise 4.1]) which shows that, for $n \geq 3$, $n/\phi(n) < e^{\gamma}(\ln\ln n) + 3/(\ln\ln n)$, where γ is Euler's constant (cf. [7, p. 26]). Thus the expected number of trials is $(p-1)/\phi(p-1) = O(\ln\ln p)$.

Remarks 2.3

1. The algorithm just described requires knowledge of the prime factorization of $p-1$. If this factorization is not known, then we do not have an efficient algorithm to find a primitive root modulo p.
2. Note also that the previous algorithm does not make specific use of the fact that the cyclic group is $\mathbb{Z}_p^*$ and works also in expected polynomial time on any cyclic group of order n where the group operation can be efficiently computed.
3. The previous algorithm is probabilistic but, so far, no deterministic polynomial-time algorithm is known to find a primitive root modulo p. Such an algorithm exists if we assume the Extended Riemann Hypothesis (ERH), a widely believed but unproven conjecture that we describe in Chap. 6. If the ERH holds then, by a result of Shoup, the least primitive root modulo p is $O(\ln^6 p)$, and hence it suffices to check the least elements of $\mathbb{Z}_p^*$ in order until this primitive root is found. See [60, Research Problem 2.39] for a discussion and references.
4. There is a special case where finding a generator of a group of order n is extremely easy, namely, when the group order n is prime (which is actually the case for some groups of cryptographic interest such as elliptic curve groups of prime order). Then it follows from Corollary 2.1 that any element of G distinct from 1 is a generator.

Exercise 2.23 Write a Maple procedure that computes the smallest primitive root modulo a prime $p > 2$. Use this procedure to gather some statistical data about the size of this primitive root and examine these data in relation to a conjecture (mentioned in [60, Research Problem 2.39]) that the least primitive root is $O((\ln p)(\ln\ln p))$.

There is a variant of the preceding algorithm which is given in [180, 11.1], where it is shown that it runs in expected time $O((\text{len}(p))^4)$. Suppose that $p-1 := \prod_{i=1}^{r} q_i^{e_i}$. The algorithm proceeds as before but, instead of looking directly for a generator of $\mathbb{Z}_p^*$, it proceeds to find elements (which are chosen at random) of order $q_i^{e_i}$ for each $i = 1, \ldots, r$. Once these r elements are found, their product is a generator of $\mathbb{Z}_p^*$.

This algorithm works, more generally, for any finite cyclic group, and may be summarized as follows:

Algorithm 2.5. Computation of a generator of a finite cyclic group.

Input: A finite cyclic group G of order n and the prime factorization $n = \prod_{i=1}^{r} p_i^{e_i}$.
Output: A generator g of G.

1. `Choose`, for each $i \in \{1, \ldots, r\}$, elements in G at random until finding $x_i \in G$ such that $x_i^{n/p_i} \neq 1$.
2. `Set`, for each $i \in \{1, \ldots, r\}$, $y_i := x_i^{n/p_i^{e_i}}$.
3. `return` $g := \prod_{i=1}^{r} y_i$.

In order to see that the element g returned by this algorithm is indeed a generator of G, we can argue as follows. Consider an element $x_i \in G$ obtained in the first step of the algorithm and let n_i be the order of x_i. By Corollary 2.1, n_i divides n and, since $x_i^{n/p_i} \neq 1$, we see from Proposition 2.4 that n_i is a multiple of $p_i^{e_i}$. Now let $y_i = x_i^{n/p_i^{e_i}}$. By Proposition 2.5 the order of this element is equal to $\frac{n_i}{\gcd(n_i, n/p_i^{e_i})}$. As we have just seen, n_i is a multiple of $p_i^{e_i}$ and the cofactor of the latter is removed when dividing by the gcd, so the order of y_i is exactly $p_i^{e_i}$. Finally, since the integers $p_i^{e_i}$ are pairwise relatively prime, the order of the product g of the y_i is equal to $\prod_{i=1}^{r} p_i^{e_i} = n$ (and hence g is a generator of G) as a consequence of the following exercise:

Exercise 2.24 Let $(G, \cdot)$ be an abelian group and $x_1, x_2 \in G$ elements whose orders o_1, o_2 satisfy $\gcd(o_1, o_2) = 1$. Show that the order of $x_1 x_2$ is the product $o_1 o_2$. (Hint: Prove that the subgroup $\langle x_1 \rangle \cap \langle x_2 \rangle \subseteq G$ is the identity subgroup by checking that its order must divide the orders of x_1 and x_2. If $x_1 = x_2 = 1$ then there is nothing to prove, otherwise assume that $1 < d < o_1 o_2$ is the order of $x_1 x_2$ and show that $x_1^d, x_2^d \in \langle x_1 \rangle \cap \langle x_2 \rangle = 1$. Deduce that then d divides both o_1 and o_2, giving a contradiction.)

An implementation of the generator-finding Algorithm 2.5 in Maple, for the case of the cyclic group $\mathbb{Z}_p^*$, where p is prime, is given in the function `findgenerator` below. The first input parameter is for the prime p and there is an optional parameter `factors`, which is used for the factors of $p-1$ in the format $[[p_1, e_1], [p_2, e_2], \ldots, [p_r, e_r]]$ where the p_i are the prime divisors of $p-1$ and the e_i the corresponding exponents. The default value is precisely this list which, if not passed to the function, is computed by Maple by means of the built-in function `ifactors` applied to $p-1$.

The option of providing the factorization of $p-1$ as input is given because this will allow Maple to compute a primitive root modulo p if this factorization is known, even in cases when `ifactors` is not able to compute it. The output is a generator of $\mathbb{Z}_p^*$.

```
> findgenerator := proc(p, factors := ifactors(p-1)[2])
  local n, l, i, found, e, x, g;
  if `not`(isprime(p)) then
    error "%1 is not prime", p
  end if;
  n := nops(factors);
  RandomTools:-MersenneTwister:-SetState();
  l := [];
  for i to n do
    found := false;
    e := (p-1)/factors[i][1];
    while not found do
      x := RandomTools:-MersenneTwister:-GenerateInteger(':-range' = 2 .. p-2);
      found := evalb(x&^e mod p <> 1)
    end do;
    e := (p-1)/(factors[i][1])^(factors[i][2]);
    l := [op(l), x&^e mod p]
  end do;
  g := 1;
  for i to n do
    g := (g*l[i]) mod p
```

```
end do;
g
end proc:
```

Example 2.18 We use `findgenerator` to compute a primitive root modulo a 1024-bit prime. Let

```
> q1 := 561378849237823437209639626695358421330239881486344495906838845417463221 45\
  87113127165567687732559098437668155861260540297330848898848261719781963459449179:

  q2 := 536764790187229183471215219686142391714385392956660627307022842352548950 63\
  28707875866844215864177767975538660113907335828570679570907425483448984476864331:
```

`q1` and `q2` are (most likely) primes as can be seen by applying Maple's function `isprime` which will be discussed later; if this function returns `true` then the tested number is, with high probability, prime. Also, we check by means of the binary logarithm that both numbers have 511 bits:

```
> isprime~([q1, q2]);
  ilog2~([q1, q2]);
                                [true, true]
                                [510, 510]
```

Now, we set `p := 4*q1*q2+1` and we check that this number—which we do not print—is a 1024-bit prime:

```
> p := 4*q1*q2+1:
  isprime(p);
  ilog2(p);
                                      true
                                      1023
```

Next we compute a primitive root modulo `p`, i.e., a generator of the cyclic group $\mathbb{Z}_p^*$:

```
> findgenerator(p, [[2, 2], [q1, 1], [q2, 1]])

507158581783015838745660768924712556263357818335236570888284120213718059075471704 2\
  56887456925344543886456402017227468479254316677005127839147120046714053127092723\
  78225388310351970876146651714842638782711686064194343918066971518084609972932972\
  01647349636437715506447502018143966841099854369119418199173984027 2
```

Note that a generator of $\mathbb{Z}_p^*$ was easily found because we knew the factorization $p - 1 = 4 \cdot q1 \cdot q2$ but we would not be able to find such a generator given only p. Indeed, Maple has a function, `numtheory:-primroot`, that computes the first primitive root modulo an integer $n > 1$ such that $\mathbb{Z}_n^*$ is cyclic but this function is unable to compute a primitive root modulo the prime p above within reasonable time. The reason is that Maple does not know the factorization of $p - 1$ and so it tries to obtain it but the prime factors of $p - 1$ are too large.

Exercise 2.25 Write a Maple program that computes a prime p and the factorization of $p - 1$, given the sizes of the prime factors of $(p - 1)/2$, by pseudo-randomly choosing these factors.

Even if n is not prime, it may be the case that the group $\mathbb{Z}_n^*$ is cyclic and then, as already mentioned, a *primitive root* modulo n is a generator of $\mathbb{Z}_n^*$, i.e., an element

of order $\phi(n)$ in this group. But it may also happen that there are no primitive roots modulo n. For example:

Exercise 2.26 Prove that there are no primitive roots modulo 12 by showing that the three nonidentity elements of $\mathbb{Z}_{12}^*$ have order 2.

In fact, $n = 8$ is the smallest value for which there are no primitive roots, for it can be shown that an integer n has a primitive root if and only if $n = 2, 4, p^t, 2p^t$, where p is an odd prime. The existence of primitive roots when $n = p$ is prime will be proved in Corollary 2.6 as a consequence of a more general result about finite fields. Based on this result, we are going to show that, in fact, there are primitive roots modulo any odd prime power. We start by dealing with a squared prime, and we leave the proof of this case as an exercise:

Exercise 2.27 Show that, if p is an odd prime, then $\mathbb{Z}_{p^2}^*$ is cyclic, i.e., there exists a primitive root modulo p^2. (Hint: Let g be a primitive root modulo p. Then $g + p$ is also a primitive root modulo p and, if g is not a primitive root modulo p^2, i.e., if $g^{p-1} \equiv 1 \pmod{p^2}$, then $(g + p)^{p-1} \not\equiv 1 \pmod{p^2}$, so that $g + p$ is a primitive root modulo p^2.)

Next we show that, for any odd prime p, there exists a primitive root modulo p^t for each positive integer t.

Theorem 2.19 *Let p be an odd prime and t a positive integer. Then $\mathbb{Z}_{p^t}^*$ is a cyclic group.*

Proof By the preceding exercise we know that the result is true for $t = 2$ and that, in fact, there exists a generator g of $\mathbb{Z}_{p^2}$ such that $g \bmod p$ is also a generator of $\mathbb{Z}_p$. We claim that $g \in \mathbb{Z}_{p^t}^*$ is a generator for all $t \geq 2$. Since g is a generator of $\mathbb{Z}_{p^2}$, we know that $g^{p-1} \not\equiv 1 \pmod{p^2}$ and we will now show that $g^{p^{t-2}(p-1)} \not\equiv 1 \pmod{p^t}$. We know that this is true for $t = 2$ and, by induction, we assume that $t \geq 3$ and that

$$g^{p^{t-3}(p-1)} \not\equiv 1 \pmod{p^{t-1}}.$$

Since p does not divide g, p^{t-2} does not divide g either and so, by Euler's theorem we have that

$$g^{p^{t-3}(p-1)} \equiv g^{\phi(p^{t-2})} \equiv 1 \pmod{p^{t-2}}.$$

Thus there exists $r \in \mathbb{Z}$ such that $g^{p^{t-3}(p-1)} = 1 + rp^{t-2}$, where p does not divide r because of the induction hypothesis. Now, we raise the terms of this equality to the pth power and, using the binomial expansion for $g^{p^{t-2}(p-1)} = (1 + rp^{t-2})^p$, we obtain:

$$g^{p^{t-2}(p-1)} = 1 + prp^{t-2} + \binom{p}{2}(rp^{t-2})^2 + \cdots + (rp^{t-2})^p \equiv 1 + rp^{t-1} \pmod{p^t}.$$

Since p does not divide r we see that indeed $g^{p^{t-2}(p-1)} \not\equiv 1 \pmod{p^t}$. Now, to prove that g is a primitive root modulo p^t, let us call n its order in $\mathbb{Z}^*_{p^t}$ and observe that, by Corollary 2.1, $n|\phi(p^t) = p^{t-1}(p-1)$. On the other hand, we have that $g^n \equiv 1 \pmod{p^t}$ and hence that $g^n \equiv 1 \pmod{p}$. Since g is a primitive root modulo p (i.e., $g \bmod p \in \mathbb{Z}^*_p$ is a generator), its order in $\mathbb{Z}^*_p$ is equal to $p-1$ and divides n. Thus $(p-1)|n|p^{t-1}(p-1)$ and hence $n = p^u(p-1)$ with $u \leq t-1$. Then, if g is not a generator of $\mathbb{Z}^*_{p^t}$, its order must be a proper divisor of $|\mathbb{Z}^*_{p^t}| = p^{t-1}(p-1)$ and hence a divisor of $p^{t-2}(p-1)$. Therefore in this case we have that $u \leq t-2$ and $g^{p^{t-2}(p-1)} = (g^n)^{p^{t-2-u}} \equiv 1 \pmod{p^t}$, which is a contradiction and completes the proof. □

Exercise 2.28

(i) Modify the function `findgenerator` in order to make it able to find a primitive root modulo any integer $n \geq 2$ such that $\mathbb{Z}^*_n$ is cyclic. Use the format `proc(n,o:=numtheory:-phi(n), factors:=ifactors(o)[2])` for the input parameter declaration and make sure that the pseudo-randomly chosen elements x belong to $\mathbb{Z}^*_n$ by checking that `igcd(x, n) = 1`. The rest of the function code should be practically the same as that of `findgenerator`, just dropping the primality check and replacing `p - 1` by `o` throughout.

(ii) Use the new function to compute a primitive root modulo $2q^3$, where q is the prime $q = 3944809387454309923$. Test the result obtained by means of Maple's function `numtheory:-primroot`, by checking that if `g` is the primitive root obtained, then:
`> evalb(numtheory:-primroot(g - 1, 2 * q^3) = g);`
gives `true` as output (meaning that g is the smallest primitive root greater than $g-1$).

(iii) Define a Maple function, based on the previous one, such that the output is the smallest primitive root modulo n which is greater than a given positive integer $t < n$ (if such a primitive root exists). Compare the output of this function with that of Maple's `numtheory:-primroot`.

2.8 Finite Fields

Finite fields play an important role in cryptography and in this section we show how to construct them. We start with a small example.

2.8.1 A Field of 4 Elements

We have seen that $\mathbb{Z}_n$ is a field if and only if n is prime. This guarantees, for each prime p, the existence of a finite field of p elements (the number of elements of a

finite field is also called the *order* of the field). This result, however, leaves open the question whether there are other finite fields. Although $\mathbb{Z}_4$ is not a field, one can easily check that if we consider the set $\{0, 1, 2, 3\}$ with addition and multiplication defined by the following tables:

$\oplus$	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

$\cdot$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

then we have a field of order 4 (note that we have not used the notation $\mathbb{Z}_4$ because we reserve it for the ring of residue classes modulo 4, which, as mentioned, is not a field). Thus we see that, in addition to the fields $\mathbb{Z}_p$ with p prime (the finite *prime fields*) there are other fields. But if we attempt in a similar way to find a field on the set $\{0, 1, 2, 3, 4, 5\}$ we fail: this set cannot be given a field structure, or, in other words, there are no fields with exactly 6 elements. With a little patience one might carry out this experiment but it is really not necessary because, as the next exercise shows, there are no fields of order 6 (for more details one may consult an introductory algebra text such as [45], [48] or, more specifically, a textbook on finite fields such as [131]).

Exercise 2.29

(i) Show that if $\mathbb{F}$ is a finite field, then the least positive integer m such that $\underbrace{1+1+\cdots+1}_{m \text{ times}} = 0$ is a prime number p (called the *characteristic* of $\mathbb{F}$).

(ii) Show that the set of the elements of a finite field $\mathbb{F}$ that can be obtained by adding 1 a finite number of times is a field with the operations induced by those of $\mathbb{F}$ (i.e., a subfield). Show that this subfield is isomorphic to $\mathbb{Z}_p$, where p is the characteristic of $\mathbb{F}$, so that it is a prime field called the prime subfield of $\mathbb{F}$.

(iii) Show that $\mathbb{F}$ has a natural structure of a vector space over its prime subfield so that if the dimension of this vector space is n, then $\mathbb{F}$ is isomorphic to $\mathbb{Z}_p^n$ as vector spaces (similarly to the way each finite-dimensional real vector space is isomorphic to some $\mathbb{R}^n$). Deduce that $|\mathbb{F}| = p^n$, where p is the characteristic of $\mathbb{F}$, and hence that the order of a finite field is always a prime power.

2.8.1.1. The `BitXor` Operation

The field of order 4 we have just constructed has characteristic 2 and, as we shall soon see, addition in these fields is given by the bitwise Xor operation, which we are going to implement in a Maple function called `BitXor` and plays an important role in several cryptographic algorithms such as the one-time pad (Chap. 3) and the Advanced Encryption Standard (Chap. 4). To give a Maple implementation we start

by considering bit lists, for which the operation is simply the sum modulo 2 of the corresponding bits:

```
> BitXorList := proc()
    add(_passed[i], i = 1 .. _npassed) mod 2
  end proc:
```

Next we are going to implement the `BitXor` operation for integers in the 0..255 range (in Maple v12 and above this operation is also performed by the built-in function `Bits:-Xor`). To make the operation faster, we will use a lookup table constructed with the help of the preceding function and the tables `bitstobytetable` and `bytetobitstable` given in Appendix A.

```
> bitXortable := Array(0 .. 255, 0 .. 255, (i, j) ->
              bitstobytetable[BitXorList(bytetobitstable[i], bytetobitstable[j])]):
```

Then the `BitXor` operation with two integers in the 0..255 range as arguments is the following:

```
> BitXor := (a, b) -> bitXortable[a, b]:
```

The addition operation defined on the set {0, 1, 2, 3} by the table given above is just bitwise Xor as can easily be checked; this is the reason why we used the $\oplus$ sign to denote this operation although generically we denote the addition in a finite field by the symbol '$+$' and the multiplication by '$\cdot$'.

Example 2.19 We use the previously defined Maple function `BitXor` to check that the addition table of the 4-element field above corresponds to the bitwise Xor operation. In the following Maple function, we define the table of an operation as a matrix where the (i, j)th entry contains the result of operating the ith element with the jth element (where the elements are given in a list):

```
> tabl := (lis, operation) -> Matrix(nops(lis), (i,j) -> operation(lis[i], lis[j])):
```

Next, we compute the table corresponding to `BitXor` and we see that it agrees with the one given above:

```
> tabl([0, 1, 2, 3], BitXor);
```

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

2.8.1.2 The Field Multiplication

It is not so easy to see where the multiplication table of the 4-element field comes from, but it is a particular case of a more general construction that will allow us to construct finite fields of p^n elements, for any prime power p^n. Before giving this general construction, let us use Maple to compute the multiplication table above by means of simple polynomial operations. Recall the definition of a polynomial:

Definition 2.24 Let R be a commutative ring with identity $1 \neq 0$. A *polynomial* in one variable over R, is a formal expression

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

where n is a non-negative integer, x is the *variable* (also called an *indeterminate*) and the $a_i \in R$, $i = 1, \ldots, n$, are the *coefficients*. The set of all the polynomials in the variable x over R is denoted by $R[x]$.

We are going to identify the set $\{0, 1, 2, 3\}$ with the set $\{0, 1, x, x+1\}$ of polynomials of $\mathbb{Z}_2[x]$, with the identification done by means of the bijection that maps the latter to the former by just substituting $x = 2$:

```
> subs(x = 2, [0, 1, x, x+1]);
                                [0, 1, 2, 3]
```

Then we define a multiplication in this set of polynomials as follows:

```
> mult4 := (f,g) -> Rem(f*g, x^2+x+1, x) mod 2:
```

The preceding operation takes two polynomials in the variable x, multiplies them, and computes the remainder of dividing their product by the polynomial $x^2 + x + 1$ (see Sect. 2.8.2 for the definition of these operations), where all the polynomials are considered with coefficients in $\mathbb{Z}_2$. Now we use the function `tabl` above to compute the table of this operation in matrix form:

```
> tabl([0, 1, x, x+1], mult4);
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & x & x+1 \\ 0 & x & x+1 & 1 \\ 0 & x+1 & 1 & x \end{bmatrix}$$

If we want to see this multiplication viewed as defined on the set $\{0, 1, 2, 3\}$, we substitute $x = 2$ into the preceding matrix:

```
> subs(x = 2, %);
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 \\ 0 & 2 & 3 & 1 \\ 0 & 3 & 1 & 2 \end{bmatrix}$$

We see that the operation defined with the help of the polynomials is indeed the same as the multiplication we had previously defined on $\{0, 1, 2, 3\}$.

Exercise 2.30 Use Maple to define this multiplication explicitly on the set $\{0, 1, 2, 3\}$ and build the corresponding table using `tabl`. In order to do it, define the map $\{0, 1, 2, 3\} \to \{0, 1, x, x+1\}$ whose inverse is given by `f -> subs(x = 2, f)` (this map can be defined more generally for any positive integer: the coefficients of the polynomial corresponding to one of these integers are just its binary digits).

Then use these maps to define the multiplication as a function that assigns to a pair of elements of {0, 1, 2, 3} another element of this set (their product). Use `tabl` to compute the table of this multiplication and verify that it is the one given above.

The multiplication `mult4` defined above looks very similar to the multiplication in $\mathbb{Z}_n$, except that now we multiply polynomials and take the remainder of dividing by x^2+x+1 which plays here the role of the modulus n. The similarity goes further because the four polynomials $0, 1, x, x+1$ are the possible remainders of dividing a polynomial by x^2+x+1 just like the elements of $\mathbb{Z}_n$ are the possible remainders of integer division by n (we are assuming that all these polynomials have coefficients in $\mathbb{Z}_2$). We have seen before that the order of a finite field is always a prime power and, by generalizing this construction we are going to show that, for each prime power p^n there is a field of order p^n (which, moreover, is unique up to isomorphism).

2.8.2 The Polynomial Ring

Notations and terminology. We will adopt the standard terminology and notational conventions for polynomials. For example, if there is no ambiguity about the name of the variable, we will write f instead of $f(x)$ to denote a polynomial. The zero polynomial is the polynomial whose coefficients are all equal to 0 and is denoted simply by 0. Moreover, we follow the convention that a term of the form $a_i x^i$ with $a_i = 0$ need not be written down, so a generic nonzero polynomial $f = \sum_{i=0}^{n} a_i x^i \in R[x]$ can always be written as a sum of polynomials—called *monomials*—of the form $a_i x^i$ with $a_i \neq 0$. If $f \neq 0$, then $f = \sum_{i=0}^{n} a_i x^i$ with $a_n \neq 0$ and a_n is called the *leading coefficient* of f and n the *degree* of f, denoted $n = \deg(f)$. If $f = 0$ then, by convention, we write $\deg(f) = -\infty$ (and we also agree to the conventions that $-\infty < n$, $-\infty + -\infty = -\infty$ and $-\infty + n = -\infty$, for each integer n). a_0 is the *constant term* of f and if $f = a_0$ or, equivalently, if $\deg(f) \leq 0$, f is called a *constant polynomial*. The polynomials whose leading coefficient is 1 are called *monic polynomials*. We see that f is a monomial precisely when all the coefficients of f, except the leading coefficient, are equal to 0.

Let R be a commutative ring with identity and $f, g \in R[x]$. We can write $f = \sum_{i=0}^{n} a_i x^i$, $g = \sum_{i=0}^{n} b_i x^i$ and define their sum:

$$f + g = \sum_{i=0}^{n} (a_i + b_i) x^i .$$

The product of $f = \sum_{i=0}^{n} a_i x^i$ and $g = \sum_{i=0}^{m} b_i x^i$ is the polynomial:

$$fg = \sum_{k=0}^{n+m} c_k x^k, \text{ where } c_k = \sum_{i=0}^{k} a_i b_{k-i}, \text{ for } 0 \leq k \leq n+m.$$

We see that the sum of two polynomials is computed just by adding in R the coefficients corresponding to the monomials of the same degree. Also, if polynomials are viewed as a sum of monomials, then the product is obtained by distributivity as the sum of all the products of monomials, where two monomials are multiplied by adding their degrees and multiplying their coefficients. It is straightforward to see that $R[x]$ with the operations just defined is a ring called the *polynomial ring* in the variable x over R. Note that the zero element of $R[x]$ is precisely the 0 polynomial, i.e., the polynomial all of whose coefficients are 0. Similarly the identity of this ring is denoted by 1 and is the polynomial whose only nonzero coefficient is $a_0 = 1$.

Exercise 2.31 Show that polynomials of $R[x]$ can be defined as infinite sequences $\{a_i\}_{i\in\mathbb{N}}$ of elements of R such that all but a finite number of the a_i are zero (as usual, $\mathbb{N}$ denotes the set of natural numbers, i.e., the set of non-negative integers). Show how to define an addition and a multiplication in the set of these sequences which make it a ring isomorphic to $R[x]$.

In the following, the coefficient ring R will always be a field k and, in fact, it will usually be a finite prime field $k = \mathbb{Z}_p$. For polynomial rings over fields, a straightforward comparison of the leading coefficients shows that:

Proposition 2.10 *Let $f, g \in k[x]$, where k is a field. Then we have:*

(i) $\deg(f + g) \leq \max\{\deg(f), \deg(g)\}$.
(ii) $\deg(fg) = \deg(f) + \deg(g)$.

Proof (i) is clear. For (ii) observe first that if any of the involved polynomials is equal to 0 then the result holds by our previous convention regarding sums including the term $-\infty$. Otherwise, let a_n and b_m be the leading coefficients of f and g. Then, since $k[x]$ has no zero divisors we have that $a_n b_m \neq 0$ and this is both the coefficient of the monomial of degree $n + m$ and the leading coefficient of fg, so that fg has degree $n + m$. □

As a consequence of part (ii) of the above proposition we see that the ring $k[x]$ is an integral domain. This is one property that $k[x]$ shares with $\mathbb{Z}$ and in fact, from the point of view of divisibility, the ring $k[x]$ behaves in a way very similar to $\mathbb{Z}$. From a more general point of view whose details we will omit, both rings are *Euclidean domains* which implies the existence of division with remainder and that the Euclidean algorithm applies. We introduce some terminology entirely similar to that used for $\mathbb{Z}$. If $f, g \in k[x]$ then we say that g divides f (denoted by $g|f$) if there exists a polynomial $q \in k[x]$ such that $f = gq$. In this case, we also say that g is a divisor of f and that f is a multiple of g. Note that the units (i.e., the invertible elements) of $k[x]$ are the divisors of the polynomial 1, i.e., all nonzero constant polynomials.

Theorem 2.20 *Let $f, g \in k[x]$, $g \neq 0$. Then there exist uniquely determined polynomials $q, r \in k[x]$ such that:*

$$f = gq + r \quad \text{and} \quad \deg(r) < \deg(g).$$

Proof The case $f = 0$ is trivial, just set $q = r = 0$. Thus we will assume in the sequel that $f \neq 0$. Now, if $\deg(f) < \deg(g)$, then we set $q = 0$ and $r = f$. Therefore we may also assume that $\deg(g) \leq \deg(f)$. The proof of the existence of q and r proceeds by induction on the degree of f. If $\deg(f) = 0$ then $\deg(g) = 0$ too, so that both f and g are nonzero constant polynomials and hence units in $k[x]$, and we set $q = fg^{-1}$ and $r = 0$.

Suppose now that $\deg(f) = n > 0$, where $n \geq m = \deg(g)$, and let

$$f = \sum_{i=0}^{n} a_i x^i, \text{ and } g = \sum_{i=0}^{m} b_i x^i.$$

Then we set $f_1 = f - a_n b_m^{-1} x^{n-m} g$ and since $\deg(f_1) < \deg(f)$, we may use the induction hypothesis to conclude that there are polynomials $q_1, r \in k[x]$ such that $f_1 = q_1 g + r$, with $\deg(r) < \deg(g)$. Then, replacing this value of f_1 in the previous equation we obtain:

$$f = (a_n b_m^{-1} x^{n-m} + q_1)g + r,$$

so that $q = a_n b_m^{-1} x^{n-m} + q_1$ and r satisfy the assertion of the theorem.

To prove uniqueness, let $f = gq + r = gq' + r'$, with $\max\{\deg(r), \deg(r')\} < \deg(g)$. Then $g(q - q') = r' - r$ and so $\deg(g) + \deg(q - q') = \deg(r - r') \leq \max\{\deg(r), \deg(r')\} < \deg(g)$ (by Proposition 2.10 and the hypothesis). This inequality can only hold if $\deg(q - q') = -\infty$, i.e., we have that $q = q'$. Thus $r' - r = g(q - q') = g0 = 0$ and we also have $r' = r$. □

The preceding proof gives an algorithm to perform the division of polynomials:

Algorithm 2.6. Polynomial division.

Input: Polynomials $f, g \in k[x]$, such that $g \neq 0$.
Output: $q, r \in k[x]$ such that $f = gq + r$ and $\deg(r) < \deg(g)$.

1. `Set` $r := f$ and $q := 0$.
2. `Compute` the quotient and the remainder:
   `while` $\deg(r) \geq \deg(g)$ `do`
   $a :=$ leading coefficient of r
   $b :=$ leading coefficient of g
   $h := ab^{-1}x^{\deg(r)-\deg(g)}$
   $r := r - hg$
   $q := q + h$
   `end do`
3. `return`: q, r.

The remainder of dividing f by g is denoted by f mod g. In Maple, division of polynomials is implemented in the functions `Rem` (we already used this function when we defined the multiplication `mult4`) and `Quo`. These functions, when used

in conjunction with `mod` (or with `modp1`) implement division in the polynomial ring $\mathbb{Z}_p[x]$, where p is a prime number.

Example 2.20 Suppose that we want to compute the quotient and the remainder of the division of $x^{13}+x^{10}+x^9+x^7+x^4+x^2+1 \in \mathbb{Z}_2[x]$ by $x^8+x^4+x^3+x+1 \in \mathbb{Z}_2[x]$:

```
> f := x^13 + x^10 + x^9 + x^7 + x^4 + x^2 + 1:
  g := x^8 + x^4 + x^3 + x + 1:
```

Then we may use `Rem` which will give the remainder as output and, if provided with a fourth optional argument '`q`', will assign the value of the quotient to the variable `q`, so that:

```
> Rem(f, g, x, 'q') mod 2;
  q;
                                    x^7 + x
                                x^5 + x^2 + 1
```

This shows that, in this case, $r = x^7 + x$ and $q = x^5 + x^2 + 1$. Alternatively, we can use `Quo`, in which case the output of the function is the quotient and the remainder is optionally assigned to a variable name (`r` in our example) if it is passed as fourth argument to the function:

```
> Quo(f, g, x, 'r') mod 2;
  r;
                                x^5 + x^2 + 1
                                  x^7 + x
```

The divisibility theory of $k[x]$ closely mimics that of $\mathbb{Z}$ and also in this case the greatest common divisor of polynomials can be defined and computed by means of the Euclidean algorithm. If $f, g \in k[x]$ are polynomials, not both 0, then the *greatest common divisor* of f and g is the unique monic polynomial $d \in k[x]$ such that $d|f$, $d|g$, and if $h \in k[x]$ is such that $h|f$, $h|g$, then $h|d$. Moreover, d can be expressed in the form $d = sf + tg$ with $s, t \in k[x]$ (see, e.g., [131, Theorem 1.55]).

Remark 2.6 The requirement that the gcd be a monic polynomial is purely technical and its purpose is to make the gcd unique. Otherwise the gcd would be defined only up to units of $k[x]$, i.e., up to multiplication by nonzero elements of k.

We shall not go into the details of the Euclidean algorithm for polynomials because it is practically the same as the Euclidean algorithm for integers, just replacing integer division by polynomial division. Starting with f and g the analogous sequence of divisions is performed until reaching a zero remainder. Then the previous remainder, multiplied by the inverse in k of its leading coefficient (in order to make the polynomial monic) is the gcd of f and g. There is also a version of the extended Euclidean algorithm for $k[x]$, which computes, in addition to $d = \gcd(f, g)$, polynomials $s, t \in k[x]$ such that $d = sf + tg$. As in the integer case, when $d = 1$, f and g are said to be *relatively prime*.

Example 2.21 We use Maple to compute the greatest common divisor of the two polynomials of $\mathbb{Z}_2[x]$ we used in Example 2.20:

```
> Gcd(f, g) mod 2;
                                    1
```

We see that these polynomials are relatively prime. We use the extended Euclidean algorithm to compute $s, t \in k[x]$ such that $1 = sf + tg$:

```
> Gcdex(f, g, x, 's', 't') mod 2;
  s, t;
                                    1
   x^6 + x^5 + x^4 + x^3 + x^2 + x + 1, x^11 + x^10 + x^9 + x^6 + x^4 + x^3 + x^2 + x
```

We see that, $s = x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$, $t = x^{11} + x^{10} + x^9 + x^6 + x^4 + x^3 + x^2 + x$ in this case. We may check that they satisfy $1 = sf + tg$:

```
> Expand(s*f+t*g) mod 2;
                                    1
```

An element $\alpha \in k$ is said to be a *zero* of $f \in k[x]$ whenever $f(\alpha) = 0$ (where, as usual, $f(\alpha)$ denotes the element of k obtained by substituting $x = \alpha$ into the polynomial and carrying out the field operations, so that α is a root of the equation $f(x) = 0$ and we also say that α is a root of f). There is a simple test to determine whether an element of k is a zero of f in terms of divisibility:

Theorem 2.21 *Let $f \in k[x]$ and $\alpha \in k$. Then α is a zero of f if and only if $(x-\alpha)|f$.*

Proof We divide f by $x - \alpha$ and we have $f = (x - \alpha)q + r$. Then $f(\alpha) = r$ and hence $f(\alpha) = 0$ (i.e., α is a zero of f) if and only if $r = 0$ (i.e., $(x - \alpha)|f$). □

Example 2.22 Since 1 is a zero of $x^2 + 1 \in \mathbb{Z}_2[x]$ and also of $x^3 + 1 \in \mathbb{Z}_2[x]$, we see that $x + 1 \in \mathbb{Z}_2[x]$ divides both polynomials. Indeed, one easily checks that $x^2 + 1 = (x + 1)^2 \in \mathbb{Z}_2[x]$ and also that $x^3 + 1 = (x + 1)(x^2 + x + 1) \in \mathbb{Z}_2[x]$ (note that no minus signs appear here because $-1 = 1$ in $\mathbb{Z}_2$). In fact, 1 is a zero of the polynomial $x^n + 1 \in \mathbb{Z}_2[x]$ for all $n > 0$ and it is easily seen that $x^n + 1 = (x + 1)(x^{n-1} + x^{n-2} + \cdots + 1) \in \mathbb{Z}_2[x]$.

Corollary 2.4 *A nonzero polynomial $f \in k[x]$ has at most* $\deg(f)$ *zeros.*

Proof We use induction on $n = \deg(f)$. If $n = 0$ then f is a nonzero element of k and hence f has no zeros. Suppose now that $n > 0$. If f has no zeros then the assertion is true so we may assume that f has a zero α. By Theorem 2.21 we have that $f = (x - \alpha)q$ where $\deg(q) = n - 1$. The induction hypothesis then implies that q has at most $n - 1$ zeros from which it follows that f has at most n zeros. □

If α is a zero of $f \in k[x]$, then the maximum value of t such that $f = (x - \alpha)^t g$, with $g \in k[x]$, is called the *multiplicity* of α in f and we often say that α is a *multiple zero* (or a *multiple root*) when $t > 1$. It is not difficult to see that Corollary 2.4 is also valid if one counts multiplicities, i.e., the sum of the multiplicities of the zeros of a polynomial cannot exceed its degree.

Exercise 2.32 Consider the polynomial ring $\mathbb{Z}_2[x]$ and give examples that show that a polynomial of degree 2 can have two zeros, one zero (which then will have multiplicity 2) or no zeros.

2.8.3 The Field of p^n Elements

We are now ready to construct, given a prime power p^n, a field of p^n elements. The construction will be very similar to the one used to obtain $\mathbb{Z}_p$ from the integers but, in this case, the starting point will be $\mathbb{Z}_p[x]$ instead.

So, suppose that $f \in \mathbb{Z}_p[x]$ is a polynomial with $\deg(f) = n > 0$. We define a set $\mathbb{Z}_p[x]_f = \{g \in \mathbb{Z}_p[x] | \deg(g) < \deg(f)\}$ which is the analogue of $\mathbb{Z}_q$, for $q \in \mathbb{Z}$, $q > 1$. Indeed, while the latter set consists of the possible remainders of integer division by q, the former consists of all the possible remainders of polynomial division by f. Note that $\mathbb{Z}_p[x]_f = \{a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 | a_i \in \mathbb{Z}_p\}$, so that $|\mathbb{Z}_p[x]_f| = p^n$. We define two binary operations on $\mathbb{Z}_p[x]_f$ by $g + h = (g + h) \bmod f$, $gh = (gh) \bmod f$, i.e., the operations in $\mathbb{Z}_p[x]_f$ are the same as the corresponding operations in $\mathbb{Z}_p[x]$ followed by taking the remainder of division by f in $\mathbb{Z}_p[x]$.[8] In fact, adding two polynomials of $\mathbb{Z}_p[x]_f$ gives the same result as adding them in $\mathbb{Z}_p[x]$, so that the 'mod f' is not necessary in the case of addition but we have written the operation this way to emphasize the fact that these operations are defined exactly like the operations of $\mathbb{Z}_q$: in order to add/multiply two polynomials in $\mathbb{Z}_p[x]_f$, we just add/multiply them in $\mathbb{Z}_p[x]$ and then we reduce modulo f.

It is easily seen that $\mathbb{Z}_p[x]_f$ with these two operations is a commutative ring with identity. The proof is straightforward and $\mathbb{Z}_p[x]_f$ inherits the ring properties from $\mathbb{Z}_p[x]$ similarly to the way $\mathbb{Z}_q$ inherits them from $\mathbb{Z}$. But observe that $\mathbb{Z}_p[x]_f$ need not be a field:

Example 2.23 Let $p = 2$ and $f = x^2 + 1 \in \mathbb{Z}_2[x]$. Then $\mathbb{Z}_2[x]_f$ is not a field because the element $x + 1 \in \mathbb{Z}_2[x]_f$ is a zero divisor and hence it does not have an inverse. Indeed, we see that in $\mathbb{Z}_2[x]_f$, $(x+1)(x+1) = (x^2+1) \bmod (x^2 + 1) = 0$.

Remarks 2.4

1. The ring $\mathbb{Z}_p[x]_f$ is constructed from $\mathbb{Z}_p[x]$ just as $\mathbb{Z}_q$ is constructed from $\mathbb{Z}$. This ring can also be built in a way similar to the construction of $\mathbb{Z}/n\mathbb{Z}$ from $\mathbb{Z}$, i.e., as a quotient ring whose elements are congruence classes modulo f (also called a residue class ring). These classes are defined, using the divisibility properties of $\mathbb{Z}_p[x]$, in a similar way to congruence classes modulo a positive integer in $\mathbb{Z}$: given $g, h \in \mathbb{Z}_p[x]$, we say that $g \equiv h \pmod f$ if and only if $f | g - h$. Then, if we denote the congruence class of $g \in \mathbb{Z}_p[x]$ by $[g]$, we set $\mathbb{Z}_p[x]/(f) = \{[g] | g \in \mathbb{Z}_p[x]\} = \{g + (f) | g \in \mathbb{Z}_p[x]\}$, where $(f) = \{hf | h \in \mathbb{Z}_p[x]\}$ is the

[8] We are committing here the usual abuse of notation and using the same generic symbols for different additions and different multiplications.

ideal of $\mathbb{Z}_p[x]$ generated by f and $g+(f)=\{g+k|k\in(f)\}$. The rings $\mathbb{Z}_p[x]_f$ and $\mathbb{Z}_p[x]/(f)$ are canonically isomorphic, with isomorphism given by the map $g\mapsto[g]$ from the former to the latter. Thus both rings are essentially the same and we will usually work with the former since its elements (polynomials) require less notation.

2. Note that, although we will only need to work with the rings $\mathbb{Z}_p[x]$ in order to build the fields of order p^n, all the preceding constructions and results are valid, more generally, for any polynomial ring $k[x]$, with k a field.

We need something else in order to obtain a field from this construction. Example 2.23 shows that, as in the case of $\mathbb{Z}_q$, $\mathbb{Z}_p[x]_f$ may fail to be a field due to the existence of zero divisors. In fact, the non-existence of zero divisors is, in this case, not only necessary but also sufficient for $\mathbb{Z}_p[x]_f$ to be a field. Indeed, as the following exercise shows, any finite integral domain is a field (see also [131, Theorem 1.31]).

Exercise 2.33 Show that a finite integral domain R is a field. To prove it, consider a nonzero element $a\in R$ and show that the map from R to itself given by multiplication with a is injective. Show that this map is also surjective (and hence a bijection) and use it to prove that a is an invertible element of R. (The same argument shows, more generally, that any non-zero divisor in a finite commutative ring with 1 is invertible.)

It is clear then that, for $\mathbb{Z}_p[x]_f$ to be a field, the product in $\mathbb{Z}_p[x]$ of two polynomials of $\mathbb{Z}_p[x]_f$ cannot be equal to f. Thus, as in the construction of $\mathbb{Z}_q$, we see that f must satisfy a property similar to primality. This property is the following:

Definition 2.25 A polynomial $f\in\mathbb{Z}_p[x]$ is said to be *irreducible* if $\deg(f)>0$ and f cannot be written as a product $f=gh$, where $g,h\in\mathbb{Z}_p[x]$ and $\deg(g)$, $\deg(h)>0$.

Examples 2.4

1. Constant polynomials are not irreducible but all polynomials of degree 1 are irreducible by Proposition 2.10.
2. If $\deg(f)=2$ or 3 then f is irreducible if and only if f does not have zeros in k. Indeed, using Proposition 2.10 we see that the irreducibility of f is, in this case equivalent to not having any degree 1 factor, i.e., to the non-existence of a polynomial g such that $\deg(g)=1$ and $g|f$. But, by Theorem 2.21 this is, in turn, equivalent to f not having zeros in k.
3. The only irreducible polynomial of degree 2 in $\mathbb{Z}_2[x]$ is x^2+x+1 because neither 0 nor 1 is a zero while, on the other hand, the remaining three polynomials of degree 2 (namely, x^2, x^2+x, x^2+1) all have zeros in $\mathbb{Z}_2$. Similarly, it can easily be checked that the only irreducible polynomials of degree 3 in $\mathbb{Z}_2[x]$ are x^3+x+1, x^3+x^2+1.

Now, suppose that we want to find all the irreducible polynomials of a given degree over $\mathbb{Z}_p$. If the degree is greater than 3, then we cannot use the criterion given in Example 2.4 but we can check whether a polynomial is irreducible or not by dividing it by the irreducible polynomials of smaller degree (or we can generate

all the reducible polynomials of a given degree by multiplying all polynomials of smaller degree). This method is rather inefficient when the degree is large, and there are more powerful methods to check the irreducibility of a polynomial, but we shall not go into the details as they are not needed for our purposes. We will show instead how we can compute the irreducible polynomials (of a given degree) over a prime field using Maple.

The Maple function that tests whether a polynomial over a prime field is irreducible is `Irreduc` which, when used together with the operator `mod` in the form `Irreduc(f) mod p` (where f is the polynomial to be tested and p the prime that defines the field of coefficients) returns `true` if the polynomial $f \in \mathbb{Z}_p[x]$ is irreducible and `false` otherwise.

Example 2.24 Let us check the irreducibility of a couple of polynomials of $\mathbb{Z}_2[x]$. We have:

```
> Irreduc(x^8 + x^4 + x^3 + x + 1) mod 2;
                                  true
```

while, on the other hand:

```
> Irreduc(x^8 + x^4 + x^2 + x + 1) mod 2;
                                  false
```

This shows that the first of these polynomials of $\mathbb{Z}_2[x]$ is irreducible while the second is not. We can also obtain the irreducible factors of the latter polynomial by computing:

```
> Factor(x^8 + x^4 + x^2 + x + 1) mod 2;
                    (x^4 + x^3 + x^2 + x + 1)(x^4 + x^3 + 1)
```

This also shows that not having any zeros in the coefficient field is not a sufficient condition for a polynomial to be irreducible. It is easy to check that the polynomial $x^8 + x^4 + x^2 + x + 1 \in \mathbb{Z}_2[x]$ has no zeros in $\mathbb{Z}_2$ but, as we have just seen, this polynomial is not irreducible.

Next we give a procedure to compute all the irreducible polynomials of a given degree over a prime field. In practice, one is usually interested in the monic irreducible polynomials because any irreducible polynomial has a unique associated monic irreducible polynomial which is obtained just by multiplying the original polynomial by the inverse of its leading coefficient. This is a similar situation to the one arising when we consider the primes, i.e., the "positive irreducibles" of $\mathbb{Z}$ (the negative irreducibles are obtained from the primes just as the non-monic irreducible polynomials are obtained from the monic ones: by multiplying by a unit of the ring).

The next function takes as input a prime p which defines the coefficient field, a positive integer n which specifies the degree, and an optional parameter `monic` which is `true` by default. The output is the list of all the monic irreducible polynomials of $\mathbb{Z}_p[x]$ of degree n or, if the argument `false` is passed to `monic`, the list of all irreducible polynomials.

```
> IrreduciblePolynomialsList := proc(p, n::posint, monic := true)
  local coeffs, pols, interv;
  global x;
  x := 'x';
  if monic then
    interv := [$p^n..2*p^n - 1]
  else
    interv := [$p^n..p^(n+1)-1]
  end if;
  coeffs := map(a -> convert(a, base, p), interv);
  pols := map(l -> sort(sum(l[j]*x^(j-1), j = 1 .. n+1)), coeffs);
  #in Maple v.13 and above the preceding line may be replaced by the following:
  #pols := map(l -> PolynomialTools:-FromCoefficientList(l,x), coeffs);
  select(f -> Irreduc(f) mod p, pols)
  end proc:
```

Example 2.25 Let us compute the monic irreducible polynomials of degree 4 in $\mathbb{Z}_2[x]$ (note that when $p = 2$ all nonzero polynomials are monic):

```
> IrreduciblePolynomialsList(2, 4);
              [x^4 + x + 1, x^4 + x^3 + 1, x^4 + x^3 + x^2 + x + 1]
```

Similarly, the irreducible polynomials of degree 8 in $\mathbb{Z}_2[x]$ are:

```
> IrreduciblePolynomialsList(2, 8);
[x^8 + x^4 + x^3 + x + 1, x^8 + x^4 + x^3 + x^2 + 1, x^8 + x^5 + x^3 + x + 1,
  x^8 + x^5 + x^3 + x^2 + 1, x^8 + x^5 + x^4 + x^3 + 1,
  x^8 + x^5 + x^4 + x^3 + x^2 + x + 1, x^8 + x^6 + x^3 + x^2 + 1,
  x^8 + x^5 + x^3 + x + 1, x^8 + x^6 + x^5 + x + 1, x^8 + x^6 + x^5 + x^2 + 1,
  x^8 + x^6 + x^5 + x^3 + 1, x^8 + x^6 + x^5 + x^4 + 1,
  x^8 + x^6 + x^5 + x^4 + x^2 + x + 1, x^8 + x^6 + x^5 + x^4 + x^3 + x + 1,
  x^8 + x^7 + x^2 + x + 1, x^8 + x^7 + x^3 + x + 1, x^8 + x^7 + x^3 + x^2 + 1,
  x^8 + x^7 + x^4 + x^3 + x^2 + x + 1, x^8 + x^7 + x^5 + x + 1,
  x^8 + x^7 + x^5 + x^3 + 1, x^8 + x^7 + x^5 + x^4 + 1,
  x^8 + x^7 + x^5 + x^4 + x^3 + x^2 + 1, x^8 + x^7 + x^6 + x + 1,
  x^8 + x^7 + x^6 + x^3 + x^2 + x + 1, x^8 + x^7 + x^6 + x^4 + x^2 + x + 1,
  x^8 + x^7 + x^6 + x^4 + x^3 + x^2 + 1, x^8 + x^7 + x^6 + x^5 + x^2 + x + 1,
  x^8 + x^7 + x^6 + x^5 + x^4 + x + 1, x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1,
  x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + 1]
```

Let us now consider an example in characteristic $\neq 2$; the monic irreducible polynomials of degree 4 over $\mathbb{Z}_3[x]$ are:

```
> IrreduciblePolynomialsList(3, 4);
[x^4 + x + 2, x^4 + 2x + 2, x^4 + x^2 + 2, x^4 + x^2 + x + 1, x^4 + x^2 + 2x + 1,
  x^4 + 2x^2 + 2, x^4 + x^3 + 2, x^4 + x^3 + 2x + 1, x^4 + x^3 + x^2 + 1,
  x^4 + x^3 + x^2 + x + 1, x^4 + x^3 + x^2 + 2x + 2, x^4 + x^3 + 2x^2 + 2x + 2,
  x^4 + 2x^3 + 2, x^4 + 2x^3 + x + 1, x^4 + 2x^3 + x^2 + 1,
  x^4 + 2x^3 + x^2 + x + 2, x^4 + 2x^3 + x^2 + 2x + 1, x^4 + 2x^3 + 2x^2 + x + 2]
```

These results may be compared with the listing in [131, Table C].

In the previous example we have considered irreducible polynomials of low degree but, as we will see in Chap. 5, an irreducible polynomial of degree 128 is used in the CMAC authentication scheme. The list of irreducible polynomials of degree 128 in $\mathbb{Z}_2[x]$ is way too long to be displayed here, but we will give a procedure to find a single irreducible polynomial of a given degree. This procedure takes as input the prime p and the degree n and returns the first irreducible polynomial of $\mathbb{Z}_p[x]$ in the enumeration implicitly defined in `IrreduciblePolynomialsList`, i.e.,

the irreducible polynomial f such that the value of $f(p)$ is the lowest among all the values $g(p)$ with g irreducible of degree n. Of course, this procedure can easily be modified to find not just one but any given number of irreducible polynomials instead (provided that the specified number is less than the total number of irreducible polynomials of the given degree). The function that finds the first irreducible polynomial is the following:

```
> IrreduciblePolynomial := proc(p, n::posint)
  local a, coeffs, f, found;
  global x;
  a := p^n;
  x := 'x';
  found := false;
  while not found do
    coeffs := convert(a, base, p);
    f := sum(coeffs[j]*x^(j-1), j = 1 .. n+1);
    found := Irreduc(f) mod p;
    a := a+1
  end do;
  sort(f)
  end proc:
```

Example 2.26 Let us find the first irreducible polynomial of degree 128 in $\mathbb{Z}_2[x]$, which is the one used in the CMAC scheme:

```
> IrreduciblePolynomial(2, 128);
                          x^128 + x^7 + x^2 + x + 1
```

Of course, using Maple it is easy to find irreducible polynomials of even larger degree.

Exercise 2.34

(i) Write a Maple procedure that finds the first ten monic irreducible polynomials in $\mathbb{Z}_p[x]$ of a given degree n (or the whole list of monic irreducible polynomials if there are fewer than ten).

(ii) Write a Maple procedure that finds the "last" irreducible polynomial in $\mathbb{Z}_p[x]$ of degree n, i.e., the irreducible polynomial f of degree n for which $f(p)$ is largest.

As can be easily guessed from our previous comments, irreducible polynomials allow us to build finite fields of order p^n:

Theorem 2.22 *Let $f \in \mathbb{Z}_p[x]$. Then the ring $\mathbb{Z}_p[x]_f$ is a field if and only if f is irreducible.*

Proof The proof is the same as the one that $\mathbb{Z}_q$ is a field if and only if q is prime, except that this time we work with polynomials of $\mathbb{Z}_p[x]$ and with the (extended) Euclidean algorithm for polynomials. We have already noted that the irreducibility of f is a necessary condition for $\mathbb{Z}_p[x]_f$ to be a field because if f is not irreducible, then any factor g of f such that $0 < \deg(g) < \deg(f)$ is a zero divisor in $\mathbb{Z}_p[x]_f$. Conversely, if f is irreducible and g a nonzero polynomial in $\mathbb{Z}_p[x]_f$, then

$\gcd(f, g) = 1$ and the extended Euclidean algorithm allows us to compute polynomials $s, t \in \mathbb{Z}_p[x]$ such that $1 = sf + tg$. Then $1 \equiv tg \pmod{f}$ and hence $t \bmod f$ is the inverse of g in $\mathbb{Z}_p[x]_f$. □

The following table summarizes the parallelism between the constructions of $\mathbb{Z}_q$ and $\mathbb{Z}_2[x]_f$:

Ring	$\mathbb{Z}$	$\mathbb{Z}_p[x]$
Element	$q \in \mathbb{Z}, q \geq 2$	$f \in \mathbb{Z}_p[x], \deg(f) = n > 0$
Quotient ring	$\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$	$\mathbb{Z}_p[x]_f = \{g \in \mathbb{Z}_p[x] \mid \deg(g) < n\}$
Field	$\mathbb{Z}_q$ field iff q prime	$\mathbb{Z}_p[x]_f$ field iff f irreducible

Remarks 2.5 Theorem 2.22 shows that for each prime p and each irreducible polynomial of degree n in $\mathbb{Z}_p[x]$ there exists a field of order p^n. It can be shown that, for each prime p and each positive integer n, there is always an irreducible polynomial of degree n in $\mathbb{Z}_p[x]$, so that there is always a field of order p^n. Moreover, every finite field has order p^n for some prime p and some integer $n > 0$ as a consequence of Exercise 2.29. Note that there may be several distinct irreducible polynomials of degree n in $\mathbb{Z}_p[x]$, and each of them gives rise to a different field $\mathbb{Z}_p[x]_f$. However, all these fields are isomorphic and this allows us to speak of *the* finite field of order p^n which is denoted by $\mathbb{F}_{p^n}$ or by $GF(p^n)$ (where the initials GF stand for "Galois Field"). We emphasize, however, that in cryptographic applications, the irreducible polynomial used to construct the field is explicitly defined in order to obtain exactly the same multiplication.

Examples 2.5

1. Let $f = x \in \mathbb{Z}_p[x]$. Then f is irreducible and $\mathbb{Z}_p[x]_f$ is a field of order p. In this case $\mathbb{Z}_p[x]_f$ consists of the polynomials of degree < 1, i.e., the constant polynomials. The polynomial operations on these constant polynomials are the same as the operations of these elements when regarded as elements of $\mathbb{Z}_p$. Thus we see that, in this case, $\mathbb{Z}_p[x]_f = \mathbb{Z}_p$. As mentioned above, often the notation $\mathbb{F}_p$ is used (instead of $\mathbb{Z}_p$) to denote this field.
2. Let $f = x^2 + x + 1 \in \mathbb{Z}_2[x]$. Then the field $\mathbb{F}_4 = \mathbb{Z}_2[x]_f$ is simply the field of 4 elements constructed at the beginning of this section and the multiplication is given by the previously defined Maple function `mult4`.
3. Let us use Maple to build the multiplication table of the finite field of 9 elements. We first find an irreducible polynomial of degree 2 in $\mathbb{Z}_3[x]$:

```
> IrreduciblePolynomial(3, 2);
                                x^2 + 1
```

The multiplication in $\mathbb{Z}_3[x]_f$, where $f = x^2 + 1$, will be given by the following function:

```
> mult9 := (f, g) -> Rem(f*g, x^2+1, x) mod 3:
```

We need to build the list of elements of $\mathbb{Z}_3[x]_f$, i.e., the list of polynomials of $\mathbb{Z}_3[x]$ of degree less than 2. The next function builds the list of polynomials of degree $< n$ in $\mathbb{Z}_p[x]$:

```
> polylist := proc(p, n)
  local coeffs;
  global x;
  x := 'x';
  coeffs := map(a -> convert(a, base, p), [$0 .. p^n-1]);
  map(l -> sort(sum(l[j]*x^(j-1), j = 1 .. nops(l))), coeffs)
  end proc:
```

The elements of $\mathbb{Z}_3[x]_f$ are then the following:

```
> polylist(3, 2);
              [0, 1, 2, x, x + 1, x + 2, 2 x, 2 x + 1, 2 x + 2]
```

Now, we use the already defined procedure `tabl` to build the multiplication table of the 9-element field. As before, the elements in the preceding list correspond, in the given order, to the files and columns of the following matrix, in which the element in the (i, j) position is the result of multiplying the ith and the jth element of the list:

```
> tabl(%, mult9)
```

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & x & x+1 & x+2 & 2x & 2x+1 & 2x+2 \\
0 & 2 & 1 & 2x & 2x+2 & 2x+1 & x & x+2 & x+1 \\
0 & x & 2x & 2 & x+2 & 2x+2 & 1 & x+1 & 2x+1 \\
0 & x+1 & 2x+2 & x+2 & 2x & 1 & 2x+1 & 2 & x \\
0 & x+2 & 2x+1 & 2x+2 & 1 & x & x+1 & 2x & 2 \\
0 & 2x & x & 1 & 2x+1 & x+1 & 2 & 2x+2 & x+2 \\
0 & 2x+1 & x+2 & x+1 & 2 & 2x & 2x+2 & x & 1 \\
0 & 2x+2 & x+1 & 2x+1 & x & 2 & x+2 & 1 & 2x
\end{bmatrix}$$

There are several natural ways to represent the elements of the field $\mathbb{F}_{p^n}$, some of which are more convenient than the polynomials we have used to define the field structure. If to each polynomial of degree $< n$ we associate the list (or the string) of its coefficients, then we have a list of base-p digits that define a number in the interval $[0, p^n - 1]$ of which these digits are the p-adic expansion. Thus we see that we have natural bijections:

$$\mathbb{Z}_p[x]_f \longleftrightarrow \mathbb{Z}_p^n \longleftrightarrow \{0, 1, \ldots, p^n - 1\} = \mathbb{Z}_{p^n},$$

whose composition is given by $\sum_{i=0}^{n-1} a_i x^i \mapsto \sum_{i=0}^{n-1} a_i p^i$, i.e., it assigns to each polynomial its value for $x = p$. In this way we can deal with finite fields in a very convenient form that represents the elements of $\mathbb{F}_{p^n}$ as the integers in the interval $[0, p^n - 1]$. One should be careful, however, for, as we have seen in the case of $\mathbb{F}_4$, neither the sum nor the multiplication in this field are the same as the corresponding operations of $\mathbb{Z}_{p^n}$ when $n > 1$. In this case, $\mathbb{Z}_{p^n}$ is not a field and the bijections defined above are not (ring) isomorphisms. Indeed, the sum in $\mathbb{F}_{p^n}$ is given by the "digitwise" sum in $\mathbb{Z}_p$, i.e., to sum two elements regarded as integers, they are first expressed as sequences of digits in base p and then the corresponding digits are

added mod p. In other words, the sum consists in computing the p-adic expansion of the integers and then adding the base-p digits without carry. On the other hand, multiplying two integers in $[0, p^n - 1]$ consists simply of taking the polynomials associated to these integers and multiplying them modulo f (so that, in contrast with addition, multiplication depends on the particular choice of f). The matrix that defines the multiplication table of $\mathbb{F}_9$ that we have just seen can be put in integer form as follows (assuming that the result of computing the previous table in polynomial form is the last output produced by Maple):

```
> subs(x = 3, %);
```

$$\begin{bmatrix} 0&0&0&0&0&0&0&0&0 \\ 0&1&2&3&4&5&6&7&8 \\ 0&2&1&6&8&7&3&5&4 \\ 0&3&6&2&5&8&1&4&7 \\ 0&4&8&5&6&1&7&2&3 \\ 0&5&7&8&1&3&4&6&2 \\ 0&6&3&1&7&4&2&8&5 \\ 0&7&5&4&2&6&8&3&1 \\ 0&8&4&7&3&2&5&1&6 \end{bmatrix}$$

Looking at this matrix we see, for example, that the product of 3 by 5 is 8 in $\mathbb{F}_9$ which corresponds to the fact that the polynomial x (which corresponds to the integer 3) multiplied by the polynomial $x + 2$ (corresponding to 5) gives as a result in $\mathbb{Z}_3[x]$ the polynomial $x^2 + 2x$ and the remainder of dividing this polynomial by the irreducible $x^2 + 1$ that we used to define the field is $2x + 2$ which corresponds to the integer 8.

Exercise 2.35 Consider $\mathbb{Z}_3[x]_f$ with $f = x^3 + 2x + 1 \in \mathbb{Z}_3[x]$. Prove that f is irreducible and hence that it defines the field $\mathbb{F}_{3^3}$. Compute the product of 25 by 11 when these integers are seen as elements of this field.

2.8.4 The Field of 256 Elements

The field $\mathbb{F}_{2^8}$ is especially important for cryptography because it is the one on whose arithmetic the *Advanced Encryption Standard* is based. We are going to describe this field in detail with the help of Maple. Although we have not used it so far, Maple has a specific package for the arithmetic of finite fields, namely, the package `GF` whose calling sequence is

```
> GF(p, n, f):
```

where p is a prime number, n a positive integer and f an irreducible polynomial of degree n over $\mathbb{Z}_p$, with the last parameter being optional (if not specified, Maple itself chooses the irreducible polynomial of degree n to be used).

We are going to use this package, and the "first" irreducible polynomial of degree 8 over $\mathbb{Z}_2[x]$, $x^8 + x^4 + x^3 + x + 1$ (already mentioned in a previous example), to

define the field $\mathbb{F}_{2^8}$. This particular polynomial is especially convenient because it is the one used in the specification of the Advanced Encryption Standard (cf. Chap. 4). Thus we initialize this field as:

```
> F256 := GF(2, 8, x^8+x^4+x^3+x+1):
```

If the field is defined by the command `F := GF(p, n)`, i.e., if the irreducible polynomial is not specified, then Maple chooses an irreducible polynomial which need not be the "first" one. The irreducible polynomial used by Maple in this case is given by the command `F[extension]`.

Exercise 2.36 Define $G := GF(2, 8)$ and find the irreducible polynomial used by Maple to construct this field.

The package `GF` has several commands to convert field elements between the different formats they can take. The elements of $\mathbb{F}_{2^8}$ are binary polynomials of degree less than 8 and, as we have mentioned before, there is a natural bijective correspondence between them and the integers in the 0..255 range, and also with the set of bit strings or lists of length 8 which are the binary representations of these integers. These bit strings are commonly referred to as *bytes* so we freely use this term—as we have already been doing—to denote the integers in the 0..255 range when regarded as elements of this field. The Maple functions that convert between these integers and polynomials modulo 2 are `F256:-input` and `F256:-output` (assuming that we defined the field `F256` with the command above) so that, for example, in Maple v13 or later we have[9]:

```
> F256:-input~([$127 .. 132)]);
    [(x^6 + x^5 + x^4 + x^3 + x^2 + x + 1) mod 2, x^7 mod 2, (x^7 + 1) mod 2,
      (x^7 + x) mod 2, (x^7 + x + 1) mod 2, (x^7 + x^2) mod 2]
```

Now we can go back from binary polynomials to integers as follows:

```
> F256:-output~(%)
                        [127, 128, 129, 130, 131, 132]
```

Note that the outputs of the function `F256:-input` are polynomials modulo 2 (or, more generally, modulo p), and for doing arithmetic with these polynomials one should use the `modp1` function of Maple. For example, let

```
> g := F256:-input(7);
                                (x^2 + x + 1) mod 2
```

Then, if we want to check whether g is irreducible `Irreduc(g) mod 2` would fail, as g is already a binary polynomial, so we would use `modp1` instead:

```
> modp1(Irreduc(g), 2);
                                      true
```

Maple also has commands `F256:-ConvertIn` and `F256:-ConvertOut` to convert from modulo 2 polynomials to ordinary polynomials and vice versa. For example:

[9] In Maple versions prior to v13 the elementwise operator ~ is not available and so one would use `map` instead to make these conversion functions act on the elements of a list.

```
> F256:-ConvertOut(g);
  F256:-ConvertIn(%);
                                  x^2 + x + 1
                              (x^2 + x + 1) mod 2
```

Maple performs the `F256` arithmetic by means of the functions `F256:-'+'`, `F256:-'-'`, `F256:-'*'`, `F256:-'^'`, `F256:-inverse`, and `F256:-'/'`. Field elements must be passed to these functions as binary polynomials. For the addition we will not make use of these functions and, for efficiency, we will instead use the previously given function `BitXor`. The multiplication can then be defined as:

```
> mult256 := (a, b) -> F256:-output(F256:-'*'(F256:-input(a), F256:-input(b))):
```

For example,

```
> mult256(89, 175);
                                      198
```

We will see that this operation is heavily used in the Advanced Encryption Standard and hence, for efficiency, it is convenient to have it implemented by means of a lookup table. This can be done in the same way as in our implementation of `BitXor`.

Exercise 2.37 Construct a Maple 2-dimensional `(0..255, 0..255) Array` containing the multiplication table of `F256` and use it to define a more efficient multiplication function.

Another operation that will be used in the sequel is the computation of inverses in `F256`, which we shall also implement by means of a lookup table. For now, note that the function `(F256:-output@F256:-inverse@F256:-input)` gives the inverse of a nonzero byte (as an integer in the 1..255 range). For example, the inverse of 73 in this field is 100 because:

```
> (F256:-output@F256:-inverse@F256:-input)(73);
                                      100
```

2.8.5 The Multiplicative Group of a Finite Field

Let $\mathbb{F}_q$, with $q = p^n$, be a finite field with q elements. We are going to see that the group of units of this field, i.e., the multiplicative abelian group $\mathbb{F}_q^* = \mathbb{F}_q - \{0\}$ of all the nonzero elements of $\mathbb{F}_q$, is cyclic. First we show:

Theorem 2.23 *For every divisor d of $q - 1$ there are $\phi(d)$ elements of order d in $\mathbb{F}_q^*$.*

Proof Let $a \in \mathbb{F}_q^*$ be an element of order d, i.e., an element such that $a^d = 1$ and no lower power with positive exponent is equal to 1. Then the powers $a, a^2, \ldots, a^d = 1 \in \mathbb{F}_q$ are all distinct and are zeros of the polynomial $x^d - 1 \in \mathbb{F}_q[x]$. Since by Corollary 2.4 this polynomial has at most d zeros, we see that its zeros are all powers

of a. Therefore, all elements of order d, being zeros of $x^d - 1$, are powers of a. By Proposition 2.5, a power a^t of a has order d if and only if $\gcd(t, d) = 1$ and hence we have shown that if a has order d then the elements of order d are all the powers a^t with $\gcd(t, d) = 1$. Since there are $\phi(d)$ values of t such that $0 < t \leq d$ and $\gcd(t, d) = 1$, this means that if there is at least one element of order d, then there are exactly $\phi(d)$ such elements. To complete the proof of the theorem it remains to show that an $a \in \mathbb{F}_q^*$ of order d must exist. If we denote by $f(d)$ the number of elements of order d we know that either $f(d) = \phi(d)$ or $f(d) = 0$. But there are $q - 1$ elements in $\mathbb{F}_q^*$ and all of them have order equal to some divisor of $q - 1$, so that $\sum_{d|q-1} f(d) = q - 1$. On the other hand, by Proposition 2.8 we also have that $\sum_{d|q-1} \phi(d) = q - 1$. Thus $f(d)$ is always equal to $\phi(d)$ and never 0, which completes the proof. □

Corollary 2.5 *The multiplicative group $\mathbb{F}_q^*$ of a finite field is cyclic and has $\phi(q-1)$ generators.*

We can now see that, for every prime p, there are primitive roots modulo p:

Corollary 2.6 *If p is a prime number, then $\mathbb{Z}_p^*$ is a cyclic group of order $p-1$ which has $\phi(p-1)$ generators.*

Exercise 2.38 Show that if p is a prime number, then the only square roots of 1 modulo p are 1 and -1.

We have seen in Sect. 2.7 an algorithm to compute a generator of $\mathbb{Z}_p^*$ (for p prime) and its Maple implementation. That algorithm works in any finite cyclic group and the corresponding Maple function can easily be adapted to find a generator of the multiplicative group of a finite field. However, it is not necessary to do this because Maple already has this function implemented in the package `GF`. Using as example our previously defined 256-element field `F256`, the Maple functions to test whether a given field element is a generator and to pseudo-randomly choose a generator are `F256:-isPrimitiveElement` and `F256:-PrimitiveElement()`, respectively.

Example 2.27 Suppose we want to check whether the element corresponding to the polynomial x is a generator. We compute:

```
> F256:-isPrimitiveElement(F256:-ConvertIn(x));
                                    false
```

Thus this element is not a generator. Let's now check the element corresponding to $x + 1$:

```
> a := F256:-ConvertIn(x+1);
                                a:=(x + 1) mod 2

> F256:-isPrimitiveElement(a);
                                    true
```

We see that this is a generator of the multiplicative group of the field. To check in a more explicit way that this is indeed the case, we would apply Proposition 2.6 as follows:

```
> ifactor(255);
                                    (3) (5) (17)
> F256:-`^`(a, 85);
                        (x^7 + x^5 + x^4 + x^3 + x^2 + 1) mod 2
> F256:-`^`(a, 51);
                                 (x^3 + x^2) mod 2
> F256:-`^`(a, 15);
                              x^5 + x^4 + x^2 + 1) mod 2
```

We see that when `a` is raised to the exponents obtained by dividing the order of the group, 255, by each of its prime factors, an element different from the identity is obtained. Thus the order of `a` must be equal to the order of the multiplicative group, i.e., 255, and hence `a` is a generator. If we want to obtain a generator, we just compute:

```
> randomize():
  F256:-PrimitiveElement();
                                  (x^7 + x^4) mod 2
```

Exercise 2.39 Write a Maple function that tests whether a given element of a finite field defined by means of the `GF` package is a generator of the multiplicative group of the field and another one that finds a generator by pseudo-randomly selecting elements of the field and applying the previous function to check whether they are generators. (Hint: One may have a look at Maple's code for the corresponding functions by printing the module `GF`; this requires executing `interface (verboseproc = 2)` and then `print(GF)`.)

2.9 Quadratic Residues and Modular Square Roots

In this section we study quadratic residues and some related algorithms that have interesting cryptographic applications.

2.9.1 Quadratic Residues and the Legendre and Jacobi Symbols

Definition 2.26 Let a be an integer and $n > 1$. We say that a is a *quadratic residue* modulo n if $\gcd(a, n) = 1$ and the congruence $x^2 \equiv a \pmod{n}$ has a solution. In other words, a is a quadratic residue if and only if $a \bmod n$ is a square in $\mathbb{Z}_n^*$. If a is not a quadratic residue modulo n then we say that a is a *quadratic non-residue* modulo n.

Remark 2.7 Whether an integer a is a quadratic residue or a quadratic non-residue depends only on the congruence class of a modulo n and the quadratic residues correspond to the squares in the group $\mathbb{Z}_n^*$ (if $a \in (\mathbb{Z}_n^*)^2 = \{x^2 | x \in \mathbb{Z}_n^*\}$, we will not distinguish between saying that a is a square in $\mathbb{Z}_n^*$ or that a is a quadratic residue). For example, in $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$ we see that the squares are 1 and 4. In fact, we have that in this group $1 = 1^2 = 4^2 = 11^2 = 14^2$ and $4 = 2^2 = 7^2 = 8^2 = 13^2$ so that each of these squares has exactly four square roots.

Example 2.28 We do a slightly larger example using Maple (much larger examples are possible but it is not practical to print the results!). Taking into account that if $b^2 = a$ then also $(-b)^2 = a$ (where $-b$ is the opposite of b in $\mathbb{Z}_n$) we see that in order to compute all the squares in $\mathbb{Z}_n^*$ (and hence all the quadratic residues modulo n), we only have to compute the squares of the elements $1, 2, \ldots, \lfloor n/2 \rfloor$ (because the remaining elements of $\mathbb{Z}_n^*$ are opposites of some of these and, in fact, as seen in the preceding remark, there can even be different elements among them with the same square). Then the function that computes the quadratic residues in $\mathbb{Z}_n^*$ is the following:

```
> qr := n -> sort(ListTools:-MakeUnique(
         map(x -> x&^2 mod n, select(x -> evalb(igcd(x,n)=1), [$1..floor(n/2)])))):
```

which, when applied to $n = 187 = 11 \cdot 17$, gives:

```
> qr(187);
[1, 4, 9, 15, 16, 25, 26, 36, 38, 42, 47, 49, 53, 59, 60, 64, 67, 69, 70, 81,
  86, 89, 93, 100, 103, 104, 111, 115, 135, 137, 144, 152, 155, 157, 166, 168,
  169, 174, 179, 185]
```

We see that in $\mathbb{Z}_{187}^*$ there are exactly 40 squares:

```
> nops(%);
                                    40
```

while the order of the group $\mathbb{Z}_{187}^*$ is:

```
> numtheory:-phi(187);
                                   160
```

As in the previous remark we see that exactly one-fourth of the elements in the group are squares and this has not happened by chance. It is a consequence of the fact that, as we will see later, when n is, as in this case, a product of two distinct odd primes, each square has four square roots. For example, in $\mathbb{Z}_{187}^*$, the four square roots of 1 are 1, 67, 120, 186 (see Exercise 2.40 below).

Exercise 2.40 Write a Maple procedure that uses brute force to find all the square roots of a quadratic residue in the preceding example (and in similar examples; we will later give a more efficient algorithm to extract modular square roots). Use it to compute the four square roots of 1 and the four square roots of 115 in $\mathbb{Z}_{187}^*$.

We will denote by $\mathcal{QR}_n$ and $\mathcal{QN}_n$ the sets of quadratic residues and quadratic non-residues in $\mathbb{Z}_n^*$, respectively; note that $\mathcal{QR}_n$ is the same as the set of the squares $(\mathbb{Z}_n^*)^2$ which is, clearly, a subgroup of $\mathbb{Z}_n^*$:

Exercise 2.41 Let $n > 1$ be an integer. Prove that $\mathcal{QR}_n$ is a subgroup of $\mathbb{Z}_n^*$.

We will be especially interested in the set of squares $\mathcal{QR}_p$ of $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ for an odd prime $p > 2$. Observe that 1 is always a quadratic residue and, moreover, modulo a prime p its only square roots are ± 1. This is because they are zeros of the polynomial $x^2 - 1 \in \mathbb{Z}_p[x]$ which, as we have seen, are at most two. More generally, the same argument shows that an element of $\mathbb{Z}_p^*$ has either two square roots (if it is a quadratic residue) or none (if it is a quadratic non-residue). Next we show that $\mathcal{QR}_p$ and $\mathcal{QN}_p$ have the same number of elements:

Proposition 2.11 *Let p be an odd prime. Then, $|\mathcal{QR}_p| = |\mathcal{QN}_p| = (p-1)/2$.*

Proof If $a \in \mathbb{Z}_p^*$ is a square then $a = b^2$ for some $b \in \mathbb{Z}_p^*$ and, as we have already remarked, a has precisely two square roots that must be b and $-b$. If we consider the squaring function $sq : \mathbb{Z}_p^* \to \mathcal{QR}_p$, given by $sq(x) = x^2$, we see that each quadratic residue is mapped onto by two elements of $\mathbb{Z}_p^*$, from which the result follows. □

Remark 2.8 If $g \in \mathbb{Z}_p^*$ is a generator, then the squares in $\mathbb{Z}_p^*$ are precisely the elements that can be written in the form g^i with i even (for, in that case, $g^i = (\pm g^{i/2})^2$). In particular, we see that a generator of $\mathbb{Z}_p^*$ is always a quadratic non-residue.

A convenient notation used to identify when an integer is a quadratic residue modulo p is given by the Legendre symbol which we now define:

Definition 2.27 Let a be an integer and $p > 2$ a prime. The *Legendre symbol* $\left(\frac{a}{p}\right)$ is defined by:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p|a \\ +1 & \text{if } a \text{ is a quadratic residue mod } p \\ -1 & \text{if } a \text{ is a quadratic non-residue mod } p. \end{cases}$$

Example 2.29 We will see how the Legendre symbol can be efficiently computed; Maple does it with the function `numtheory:-legendre` which can be used to find quadratic residues and quadratic non-residues modulo primes. More generally, the function `numtheory:-quadres` can be used to search for quadratic residues even when the modulus is not prime: like the Legendre symbol this function takes the value $+1$ on quadratic residues and the value -1 on quadratic non-residues. Let us consider the prime 101 and compute the quadratic residues in $\mathbb{Z}_{101}^*$:

```
> qr101 := select(x -> evalb(numtheory:-quadres(x, 101) = 1), {$1 .. 100})
{1, 4, 5, 6, 9, 13, 14, 16, 17, 19, 20, 21, 22, 23, 24, 25, 30, 31, 33, 36,
  37, 43, 45, 47, 49, 52, 54, 56, 58, 64, 65, 68, 70, 71, 76, 77, 78, 79, 80,
  81, 82, 84, 85, 87, 88, 92, 95, 96, 97, 100}
```

We check that exactly one half of the elements of $\mathbb{Z}_{101}^*$, namely 50 elements, are quadratic residues:

```
> nops(qr101);
                                    50
```

Next, we compute a generator of $\mathbb{Z}_{101}^*$:

```
> findgenerator(101);
                                    42
```

We compute all the even-exponent powers of the generator $42 \in \mathbb{Z}_{101}^*$:

```
> evenpowers42mod101 := map(x -> 42&^x mod 101, {seq(2*i, i = 1 .. 50)});
{1, 4, 5, 6, 9, 13, 14, 16, 17, 19, 20, 21, 22, 23, 24, 25, 30, 31, 33, 36,
  37, 43, 45, 47, 49, 52, 54, 56, 58, 64, 65, 68, 70, 71, 76, 77, 78, 79, 80,
  81, 82, 84, 85, 87, 88, 92, 95, 96, 97, 100}
```

Finally, we check that these powers coincide with the quadratic residues in $\mathbb{Z}_{101}^*$:

```
> evalb(qr101 = evenpowers42mod101);
                                   true
```

The following result gives a simple way to compute the Legendre symbol:

Proposition 2.12 (Euler's criterion) *Let p be an odd prime and a an integer. Then*

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}.$$

Proof If $p|a$ then both sides of the congruence are congruent to 0 modulo p. Hence we may suppose that p does not divide a and, by Fermat's little theorem (Theorem 2.2) we have that $a^{p-1} \equiv 1 \pmod{p}$ so that $a^{(p-1)/2} \equiv \pm 1 \pmod{p}$. Let g be a generator of $\mathbb{Z}_p^*$ and $a = g^i$. Since a is a quadratic residue if and only if i is even, we see that the left side of the congruence in the statement of the proposition is equal to 1 if and only if i is even. The right side of this congruence is then equal to $g^{i(p-1)/2}$, which is 1 if and only if $(p-1)$ divides $i(p-1)/2$, i.e., if and only if i is even. Thus both sides of the congruence are ± 1 and each side is 1 if and only if i is even, which completes the proof. □

Remark 2.9 Euler's criterion provides a polynomial-time algorithm to compute the Legendre symbol and hence to decide whether an element of $\mathbb{Z}_p^*$ is a quadratic residue or not. The algorithm performs a modular exponentiation modulo p with an exponent less than p and hence its complexity is $O(\text{len}(p)^3)$. However, we shall see later that there is a faster algorithm to compute the Legendre symbol.

The next proposition collects several basic properties of the Legendre symbol:

Proposition 2.13 *Let p be an odd prime. Then the Legendre symbol satisfies:*

(i) *If $a \equiv b \pmod{p}$ then $\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$.*

(ii) $\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$.

(iii) If p does not divide b then $\left(\frac{ab^2}{p}\right) = \left(\frac{a}{p}\right)$.

(iv) $\left(\frac{1}{p}\right) = 1$ *and* $\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$.

Proof As we have already remarked, (i) is a straightforward consequence of the definition. (ii) follows from Euler's criterion since $a^{(p-1)/2} \cdot b^{(p-1)/2} \equiv (ab)^{(p-1)/2} \pmod{p}$. (iii) is an immediate consequence of (ii). For (iv), observe first that $1 = 1^2$ is always a quadratic residue and so $\left(\frac{1}{p}\right) = 1$. Finally, $\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$ by Euler's criterion. □

Remarks 2.6

1. As a consequence of part (ii) of the preceding proposition, a product of two elements of $\mathbb{Z}_p^*$ (where p is an odd prime) is a quadratic residue if and only if either both factors are quadratic residues or both are quadratic non-residues. Also, (ii) reduces the computation of the Legendre symbol of a to those of the Legendre symbols of its prime factors but this does not give an efficient method to compute the Legendre symbol because factoring a is usually difficult.
2. The fact that $\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$ is obviously equivalent to:

$$\left(\frac{-1}{p}\right) = \begin{cases} 1 & \text{if } p \equiv 1 \pmod 4 \\ -1 & \text{if } p \equiv 3 \pmod 4. \end{cases}$$

Exercise 2.42 Write a Maple function that, given a positive integer n, finds the smallest prime p such that a is a quadratic residue modulo p for all $1 \leq a \leq n$. Use it to find the smallest prime p whose least quadratic non-residue is greater than 50 and compute the value of this quadratic non-residue. Perform a similar computation for other small values of n (it has been conjectured that the least quadratic non-residue modulo p is $O(\ln p \ln\ln p)$ but no deterministic algorithm which is polynomial on the size of p is known to find a quadratic non-residue).

The most efficient method to compute the Legendre symbol $\left(\frac{a}{p}\right)$ is obtained by using a generalization called the *Jacobi symbol*:

Definition 2.28 Let a be an integer and n a positive odd integer with prime factorization $n = \prod_{i=1}^{r} p_i^{e_i}$. Then the *Jacobi symbol* $\left(\frac{a}{n}\right)$ is defined by:

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{r} \left(\frac{a}{p_i}\right)^{e_i},$$

where the symbols on the right side are Legendre symbols and we define $\left(\frac{a}{1}\right) = 1$.

Remarks 2.7

1. If n is itself prime, then the Jacobi symbol $\left(\frac{a}{n}\right)$ is just the Legendre symbol. Thus the Jacobi symbol is a generalization of the Legendre symbol and, like the latter, it clearly depends only on the congruence class of a modulo n.
2. The Jacobi symbol is multiplicative in both arguments, namely $\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right)$ and $\left(\frac{a}{mn}\right) = \left(\frac{a}{m}\right)\left(\frac{a}{n}\right)$.
3. If a is a quadratic residue modulo n, then it is clear that a is also a quadratic residue modulo p for every prime p that divides n, and hence it follows from the definition that, in this case, $\left(\frac{a}{n}\right) = 1$. But what about the converse? If n is prime we know that the converse is also true but suppose, for example that $n = pq$ is the product of two different odd primes. Then it may happen that $\left(\frac{a}{p}\right) = -1$ and $\left(\frac{a}{q}\right) = -1$, in which case $\left(\frac{a}{n}\right) = 1$ but, clearly, a is not a quadratic residue modulo n (as this would imply that it is a quadratic residue modulo p and also modulo q). A concrete example is $\left(\frac{2}{15}\right) = \left(\frac{2}{3}\right)\left(\frac{2}{5}\right) = (-1)(-1) = 1$ where, as we have seen, 2 is a quadratic non-residue modulo 15.

Exercise 2.43 Let $\mathcal{J}_n = \{a \in \mathbb{Z}_n^* | \left(\frac{a}{n}\right) = 1\}$. Prove that if n is a square, then $\mathcal{J}_n = \mathbb{Z}_n^*$ and otherwise $|\mathcal{J}_n| = \phi(n)/2$, i.e., half of the elements of $\mathbb{Z}_n^*$ are in $\mathcal{J}_n$. (Hint: If n is not a square then there exists a prime factor p of n such that the largest power of p that divides n has an odd exponent. Use the Chinese remainder theorem to find an element which is not a quadratic residue modulo p but is a quadratic residue modulo all the remaining prime divisors of n and show that multiplication by this element gives a bijective map from $\mathcal{J}_n$ to its complement in $\mathbb{Z}_n^*$.)

The Jacobi symbol can be efficiently computed if one knows the factorization of n for, in that case, it reduces to the computation of Legendre symbols which may be done, for example, by means of Euler's criterion. But there is another method that allows the computation of the Jacobi symbol without doing any factoring—except pulling out powers of 2, which is easy—and the amazing thing is that it gives an even more efficient algorithm to compute Legendre symbols. This method is based on the *Law of Quadratic Reciprocity* that was proved by Gauss for the Legendre symbol and holds equally for the Jacobi symbol. There are more than 200 known proofs of quadratic reciprocity (see [125]) and we refer to any book introducing number theory—like, for example, [77]—for some of them; among the books related to cryptography we refer to [118, 180] for proofs.

Theorem 2.24 *Let m and n be positive odd integers. Then the following conditions hold:*

(i) $\left(\frac{-1}{n}\right) = (-1)^{(n-1)/2} = \begin{cases} 1 \quad if\ n \equiv 1 \pmod 4 \\ -1\ if\ n \equiv 3 \pmod 4. \end{cases}$

(ii) $\left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8} = \begin{cases} 1 \quad if\ n \equiv \pm 1 \pmod 8 \\ -1\ if\ n \equiv \pm 3 \pmod 8. \end{cases}$

(iii) (Law of Quadratic Reciprocity)

$$\left(\frac{m}{n}\right) = (-1)^{(n-1)(m-1)/4}\left(\frac{n}{m}\right) = \begin{cases} -\left(\frac{n}{m}\right) & if\ m \equiv n \equiv 3 \pmod 4 \\ \left(\frac{n}{m}\right) & otherwise. \end{cases}$$

Exercise 2.44 Use quadratic reciprocity to characterize the odd primes p such that 5 is a quadratic residue modulo p.

Remarks 2.8

1. When $n = p$ is prime, part (i) of Theorem 2.24 tells us that whether -1 is a square modulo p (or, equivalently, $p - 1 \in \mathbb{Z}_p^*$ is a square) depends only on the congruence class of p modulo 4 and, similarly, part (ii) says that the question whether 2 is a square depends only on the congruence class of p modulo 8.
2. We mentioned above that quadratic reciprocity can be used to compute the Legendre symbol and hence to determine whether an integer a is a square modulo a prime p. This is done by using part (iii) of Theorem 2.24 to repeatedly flip the Jacobi symbol, reducing the top number modulo the bottom number after each flip (and pulling out powers of 2 by means of part (ii) of the theorem whenever the top number obtained after the reduction is even). As in the Euclidean algorithm, these numbers decrease rapidly in size which gives a very efficient algorithm. In the next example we show how this works.

Example 2.30 We determine whether 41069779796933 is a quadratic residue modulo 673275095033 (in this example both numbers are prime). The steps applied are the following:

- If both numbers are odd and the top number is greater than the bottom one, we replace the top number by the result of reducing it modulo the bottom number.
- If the top number is even, we use part (ii) of Theorem 2.24 to pull out the largest power of 2 that divides it.
- If both numbers are odd and the top number is smaller than the bottom one, we use part (iii) of Theorem 2.24 to interchange the position of both numbers.

We then have the following computation, where we only point out the use of each of the previous three steps the first time that they are used:

$$\left(\frac{41069779796933}{673275095033}\right) = \text{(reducing the top number modulo the bottom number)}$$
$$= \left(\frac{673274094953}{673275095033}\right) = \text{(by (iii) since the top number is congruent to 1 modulo 4)}$$
$$= \left(\frac{673275095033}{673274094953}\right) = \left(\frac{1000080}{673274094953}\right) = \left(\frac{2^4 \cdot 62505}{673274094953}\right)$$
$$= \text{(by (ii))}\left(\frac{62505}{673274094953}\right) = \left(\frac{673274094953}{62505}\right) = \left(\frac{49838}{62505}\right) = \left(\frac{2 \cdot 24919}{62505}\right)$$
$$= \left(\frac{24919}{62505}\right) = \left(\frac{62505}{24919}\right) = \left(\frac{12667}{24919}\right) = -\left(\frac{24919}{12667}\right) = -\left(\frac{12252}{12667}\right)$$

$$= -\left(\frac{2^2 \cdot 3063}{12667}\right) = -\left(\frac{3063}{12667}\right) = \left(\frac{12667}{3063}\right) = \left(\frac{415}{3063}\right)$$
$$= -\left(\frac{3063}{415}\right) = -\left(\frac{158}{415}\right) = -\left(\frac{2 \cdot 79}{415}\right) = -\left(\frac{79}{415}\right) = \left(\frac{415}{79}\right) = \left(\frac{20}{79}\right)$$
$$= \left(\frac{2^2 \cdot 5}{79}\right) = \left(\frac{5}{79}\right) = \left(\frac{79}{5}\right) = \left(\frac{4}{5}\right) = 1.$$

Thus we see that 41069779796933 is a quadratic residue modulo 673275095033.

The built-in Maple function `numtheory:-Jacobi` uses this algorithm to compute the Jacobi symbol $\left(\frac{m}{n}\right)$ by calling `numtheory:-Jacobi(m,n)`. In particular, this function also computes Legendre symbols and, as already mentioned, there is also `numtheory:-legendre` which is just the specialization of the preceding function when the second argument is prime. To carry out the preceding computation we can use either of these functions, for example:

```
> numtheory:-Jacobi(41069779796933, 673275095033);
                                    1
```

The algorithm applied in the preceding example is then the following. Note that the algorithm is recursive, i.e., it calls itself:

Algorithm 2.7. `Jacobi` (m, n) **(Computation of the Jacobi symbol)**.

Input: An integer m and an odd positive integer n.
Output: The Jacobi symbol $\left(\frac{m}{n}\right)$.

`if` $n = 1$ `then return` 1.
`Set` $m := m \bmod n$.
`if` $m = 0$ `then return` 0.
`if` $m = 1$ `then return` 1.
`Write` $m = 2^e m_1$ with m_1 odd.
`if` e is even `then`
    $s := 1$
`elif` $n \equiv \pm 1 \pmod 8$ `then`
    $s := 1$
`elif` $n \equiv \pm 3 \pmod 8$ `then`
    $s := -1$
`end if`.
`if` $m_1 \equiv 3 \pmod 4$ `and` $n \equiv 3 \pmod 4$ `then`
    $s := -s$
`end if`.
`return` $s \cdot$ `Jacobi`$(n, m1)$.

Remark 2.10 Except for the removal of powers of 2, the sequence of recursive calls in Algorithm 2.7 is the same as the one used by the Euclidean algorithm to compute $\gcd(m, n)$, and an analysis similar to that of the Euclidean algorithm shows that the complexity is also the same, so that the running time of this algorithm can be estimated as $O(\text{len}(m)\text{len}(n))$ or, assuming that $m < n$, $O(\text{len}(n)^2)$. Thus

we see that this algorithm computes the Legendre symbol more efficiently than Euler's criterion and hence so do Maple's functions `numtheory:-Jacobi` and `numtheory:-legendre`.

As we will see later, the quadratic residues modulo a product of two odd primes are of cryptographic interest. The following result characterizes them:

Proposition 2.14 *Let $n = pq$ be a product of two distinct primes and $x \in \mathbb{Z}_n^*$. Then x is a quadratic residue modulo n if and only if it is both a quadratic residue modulo p and a quadratic residue modulo q.*

Proof As we have already remarked, if x is a quadratic residue modulo n then it is also a quadratic residue modulo p and modulo q. Conversely, suppose that x is a quadratic residue modulo both primes, so that we have congruences $x \equiv a^2 \pmod{p}$ and $x \equiv b^2 \pmod{q}$. Then, the Chinese remainder theorem gives us a unique element $y \in \mathbb{Z}_n^*$ such that

$$\left.\begin{aligned} y &\equiv a \pmod{p} \\ y &\equiv b \pmod{q} \end{aligned}\right\} \tag{2.6}$$

Thus $x \equiv a^2 \equiv y^2 \pmod{p}$ and $x \equiv b^2 \equiv y^2 \pmod{q}$. Hence both p and q divide $x - y^2$ and, since they are different primes, so does their product n. Therefore $x \equiv y^2 \pmod{n}$ and x is indeed a quadratic residue modulo n. □

Remark 2.11 Note that the preceding proposition can also easily be deduced from Theorem 2.14. The proof thus obtained is essentially the same as the one above but it looks even more straightforward.

The following corollary explains something we noticed when computing the quadratic residues modulo 15: when n is a product of two distinct odd primes, exactly one-fourth of the elements of $\mathbb{Z}_n^*$ are quadratic residues:

Corollary 2.7 *If $n = pq$ is a product of two distinct odd primes then each quadratic residue modulo n has exactly four square roots in $\mathbb{Z}_n^*$.*

Proof Let x be a quadratic residue modulo n. Then x is both a quadratic residue modulo p and a quadratic residue modulo q, and hence it has two square roots modulo p and two square roots modulo q. Thus we get four different pairs formed by a square root of x modulo p and a square root of x modulo q. Each of these pairs defines a system of congruences that, by the Chinese remainder theorem, gives rise to a square root of x modulo n. □

2.9.2 Computing Modular Square Roots

We have seen how to determine whether the equation $x^2 \equiv a \pmod{p}$, where p is prime, has a solution, but now we are going to see how to find the solutions, i.e.,

how to compute modular square roots modulo a prime. The easiest case is when $p \equiv 3 \pmod 4$ (p is then called a *Blum prime*). For these primes we have:

Proposition 2.15 *If $p \equiv 3 \pmod 4$ is prime and a is a quadratic residue modulo p, then the square roots of a modulo p are $\pm a^{(p+1)/4} \bmod p$.*

Proof Using Euler's criterion (Proposition 2.12) and the fact that a is a quadratic residue modulo p, we have $(\pm a^{(p+1)/4})^2 \equiv a^{(p+1)/2} \equiv a \cdot a^{(p-1)/2} \equiv a \cdot \left(\frac{a}{p}\right) \equiv a \pmod p$. □

Observe that the argument used to prove the proposition does not work when $p \equiv 1 \pmod 4$ because we made use of the fact that $(p+1)/4$ is an integer, which is no longer true in this case. So we now describe an algorithm that computes square roots modulo any odd prime.

If p is an odd prime, we may write $p - 1 = 2^s t$ with t odd and consider $A = a^t \bmod p$. Then $A^{2^{s-1}} \equiv a^{(p-1)/2} \equiv 1 \pmod p$ by Euler's criterion, so we know from Proposition 2.4 that the order of A is a divisor of 2^{s-1}. Now let $h \in \mathbb{Z}_p^*$ be a quadratic non-residue and $H = h^t \bmod p$. The order of H is a power of 2 by Proposition 2.4 and, since $H^{2^{s-1}} \equiv h^{2^{s-1}t} \equiv h^{(p-1)/2} \equiv -1 \pmod p$ again by Euler's criterion, we see that the order of H is exactly 2^s. The order of its inverse $H^{-1} \in \mathbb{Z}_p^*$ is also 2^s and hence $\langle H \rangle = \langle H^{-1} \rangle$ is the unique (cyclic) subgroup of order 2^s of the cyclic group $\mathbb{Z}_p^*$, and it contains all the elements of $\mathbb{Z}_p^*$ whose order is a power of 2. In particular, $A \in \langle H^{-1} \rangle$ and hence A is a power of H^{-1} in $\mathbb{Z}_p^*$. Moreover, A is an even power of H^{-1} because otherwise it would follow from Proposition 2.5 that A has the same order as H^{-1}, namely 2^s, in contradiction with the fact that, as we have seen, the order of A divides 2^{s-1}. Thus we see that there exists an integer u such that $0 \leq u < 2^{s-1}$ and $A \equiv H^{-2u} \pmod p$. Since $A \equiv a^t \pmod p$, this is equivalent to $a^t H^{2u} \equiv 1 \pmod p$ and, multiplying this congruence by a, we obtain $a \equiv a^{(t+1)} H^{2u} \pmod p$. Now the terms on the right side of this congruence all have even exponents and we find a square root of a modulo p by halving these exponents, namely, by computing $a^{(t+1)/2} H^u \bmod p$ (see [60]).

In order to complete the description of the square roots algorithm we need to show how to find the quadratic non-residue h and how to find the integer u such that $A \equiv H^{-2u} \pmod p$. The first problem is very interesting because no deterministic polynomial-time algorithms to find a quadratic non-residue modulo p are known.[10] It is easy, however, to describe a probabilistic polynomial-time algorithm that does the job: just pick random elements in $\mathbb{Z}_n^*$ and use either Euler's criterion or Algorithm 2.7 to check whether they are quadratic non-residues. Since, as we have seen, exactly one-half of the integers in $\mathbb{Z}_p^*$ are quadratic non-residues, the expected number of trials until finding one of them is 2.

Thus, in order to complete the algorithm, it only remains to find the integer u satisfying the conditions explained above. We show how to do it in Algorithm 2.8:

[10] As in the case of primitive roots, there is a deterministic polynomial-time algorithm under the assumption that the *Extended Riemann Hypothesis* holds.

Algorithm 2.8. Computation of square roots modulo an odd prime.

Input: An odd prime p and an integer a such that $\left(\frac{a}{p}\right) = 1$.
Output: A square root of $a \bmod p$.

1. **Case** $p \equiv 3 \pmod 4$:
   $a := a \bmod p$;
   `return` $a^{(p+1)/4} \bmod p$.

2. **Case** $p \equiv 1 \pmod 4$:
   Find a random quadratic non-residue $h \in \mathbb{Z}_p^*$
   Compute $s \geq 0$ and t odd such that $p - 1 = 2^s t$
   $A := a^t \bmod p$;
   $H := h^t \bmod p$;
   $v := 0$;
   `for` i from 1 to $s - 1$ `do`
      `if` $(AH^v)^{2^{s-1-i}} \equiv -1 \pmod p$ `then`
         $v := v + 2^i$
      `end if`;
   `end do`; #Now $AH^v \equiv 1 \pmod p$, with v even
   `return` $a^{(t+1)/2} H^{v/2} \bmod p$.

This algorithm is essentially due to Tonelli, who published it already in 1891. We next prove the correctness of the algorithm and estimate its complexity:

Proposition 2.16 *Algorithm 2.8 is correct and produces as output a square root of the quadratic residue a modulo p. The expected running time of the algorithm is $O(\text{len}(p)^4)$.*

Proof We have already shown that the algorithm is correct if $p \equiv 3 \pmod 4$, when $a^{(p+1)/4} \bmod p$ is a square root of a modulo p. So suppose that $p \equiv 1 \pmod 4$ in which case we have to show that the value returned after running the `for` loop is indeed a square root of a modulo p. It is clear that $s \geq 2$ and v is initialized as $v := 0$ so the loop is started by computing $A^{2^{s-2}} \bmod p$, which is a square root in $\mathbb{Z}_p^*$ of $A^{2^{s-1}} \bmod p = 1$ and hence is equal to ± 1 (in fact, the loop can also be started at $i = 0$ but this is unnecessary because, since a is a quadratic residue, we already know that $A^{2^{s-1}} \bmod p = 1$). We show by induction on i that, after running the ith iteration of the loop which includes updating the value of v, $(AH^v)^{2^{s-1-i}} \equiv 1 \pmod p$, i.e., the value 1 is maintained for this expression after each iteration of the loop. We may start at $i = 0$, after which we have the value 1 as already remarked. So suppose that we also have the value 1 after the $(i-1)$th iteration of the loop, i.e., $(AH^v)^{2^{s-i}} \equiv 1 \pmod p$. Then in the ith iteration we compute $(AH^v)^{2^{s-i-1}} \bmod p$ which is a square root of the previous value 1 (as v has not been updated yet) and hence it is equal to ± 1. If the value is 1 then we have nothing to prove, so assume that $(AH^v)^{2^{s-i-1}} \equiv -1 \pmod p$. In this case the value of v is updated in the loop to $v := v + 2^i$ and so we have to show that $(AH^{v+2^i})^{2^{s-i-1}} \equiv 1 \pmod p$. This is indeed true because $(AH^{v+2^i})^{2^{s-i-1}} \equiv (AH^v)^{2^{s-i-1}} (H^{2^i})^{2^{s-1-i}} \equiv (-1) H^{2^{s-1}} \equiv$

$(-1)(-1) \equiv 1 \pmod{p}$ (where we used the fact that $H^{2^{s-1}} \equiv -1 \pmod{p}$, as we have seen in our previous discussion). After the last iteration of the loop, corresponding to $i = s - 1$, we have that $AH^v \equiv 1 \pmod{p}$ and hence the final value of v is the integer $2u$ in our previous discussion and $a^{(t+1)/2}H^{v/2} \bmod p$ is indeed a square root of a modulo p.

For the complexity of the algorithm observe that, given a non-square h, outside the `for` loop only a few exponentiations modulo p are computed in time $O(\text{len}(p)^3)$. The loop has $s - 1 = O(\text{len}(p))$ iterations in which, again, modular exponentiations are carried out, so the complexity is $O(\text{len}(p)\text{len}(p)^3) = O(\text{len}(p)^4)$. Finally, as we have remarked, the expected number of trials to find a non-residue is 2 and each trial uses Algorithm 2.7 which has complexity $O(\text{len}(p)^2)$ so the expected running time of this part of the algorithm is $O(\text{len}(p)^2)$. Thus the expected running time of Algorithm 2.8 is $O(\text{len}(p)^4)$. □

Remark 2.12 Observe that in the estimation of the complexity of Algorithm 2.8 we used the fact that $s = O(\text{len}(p))$. Since s is usually much smaller than $\text{len}(p)$ this bound is not very tight and, by considering average instead of worst-case complexity, one obtains the estimate $O(\text{len}(p)^3)$, which better reflects the behavior of the algorithm over many random primes.

Exercise 2.45 Prove that the case $p \equiv 3 \pmod 4$ of Algorithm 2.8 is also handled by the algorithm used when $p \equiv 1 \pmod 4$, so that it was not strictly necessary to include it as a separate case. To see this, show that if $p \equiv 3 \pmod 4$ then $s = 1$ and hence A has order 1, that is $A \equiv 1 \pmod p$. Thus $v = 0$ and the root is obtained as $a^{(t+1)/2} \equiv a^{(p+1)/4} \pmod p$.

Remark 2.13 When $p \equiv 5 \pmod 8$, Algorithm 2.8 can be modified to use $h = 2$ instead of a random non-square. Then we can prove that the algorithm computes a square root x of a as follows: set $x := a^{(p+3)/8} \bmod p$ and if $x^2 \not\equiv a \pmod p$ then $x := x2^{(p-1)/4} \bmod p$. Indeed, it is easy to see that in this case we have $s = 2$ and $t = (p-1)/4$ and, since 2 is a quadratic non-residue modulo p by Theorem 2.24, we may choose $h = 2$. Then, when entering the `for` loop with $v = 0$ and $i = 1$ we have to compute $A = a^t = a^{(p-1)/4} \bmod p$. This can only take the values 1 or -1. The first case occurs when $(a^{(p+3)/8})^2 \equiv a^{(p+3)/4} \equiv a \cdot a^{(p-1)/4} \equiv a \pmod p$, so that $a^{(p+3)/8} \bmod p$ is indeed a square root in this case. The second case makes $v := 2$ and returns the root $x \equiv a^{(t+1)/2}2^{(p-1)/4} \equiv a^{(p+3)/8}2^{(p-1)/4} \pmod p$.

Example 2.31 The algorithm for computing square roots modulo primes is used in several cryptologic algorithms and, in particular, in the quadratic sieve and number field sieve factoring algorithms that we discuss in Chap. 6. We will see as an example how the Fermat number $F_7 = 2^{2^7} + 1$ is factored with the quadratic sieve, for which purpose the square roots of F_7 modulo p are computed for a set of primes modulo which F_7 is a quadratic residue. One of these primes is:

```
> p:= 31873:
```

Let us compute the square roots of F_7 modulo p using Algorithm 2.8 with the help of Maple, i.e., let us solve the equation $x^2 \equiv 2^{2^7} + 1 \pmod{31873}$. We do it step by step. Observe first that:

```
> F_7 := 2^128+1;
  a := F_7 mod p;
                    340282366920938463463374607431768211457
                                      17919
```

and hence computing these square roots amounts to computing the square roots of 17919 in $\mathbb{Z}^*_{31873}$. Note also that

```
> p mod 4;
                                      1
```

so that we are in the "difficult" case of the algorithm and we will need to execute the `for` loop in it. Next we choose a quadratic non-residue modulo p;

```
> randomize():
   lg := 1:
   while lg = 1 do
     h := (rand(1 .. p-1))():
     lg := numtheory:-legendre(h, p)
   end do:
   h;
                                     30863
```

We compute s and t:

```
> ifactor(p-1);
                                  (2)^7 (3) (83)
> s := 7; t := 3*83;
                                       7
                                     249
```

Now we compute A and H:

```
> A := Power(a, t) mod p;
  H := Power(h, t) mod p;
                                     29298
                                     5250
```

We initialize v:

```
> v := 0:
```

We start the `for` loop with `i := 1`:

```
> i := 1: (A*H^v)^(2^(s-1-i)) mod p;
                                      1
```

The condition $(AH^v)^{2^{s-1-i}} \equiv -1 \pmod{p}$ does not hold, thus the value of v remains equal to 0. Similarly, the analogous computation for $i = 2$ and $i = 3$ gives 1 as result, so the value of v remains 0 during the first three iterations of the loop. In the fourth iteration we have:

```
> i := 4: (A*H^v)^(2^(s-1-i)) mod p;
                                     31872
```

Now we obtained $p - 1 = -1 \bmod p$ as result, so we increase the value of v accordingly:

```
> v := v + 2 ^i;
                                        16
```

Similarly, the value of v gets increased for $i = 5$ and $i = 6$:

```
> i := 5:
  (A*H^v)^(2^(s-1-i)) mod p;
                                      31872
> v := v + 2^i;
                                        48

> i := 6:
  (A*H^v)^(2^(s-1-i)) mod p;
                                      31872
> v := v + 2^i;
                                       112
```

At the end of the loop we obtain the value $v = 112$ and hence we should have that $AH^{112} \equiv 1 \pmod{p}$. We check that this is indeed the case:

```
> A*H^112 mod p;
                                         1
```

We can now find a square root of F_7 modulo p, which is:

```
> x := a^((t+1)*(1/2))*H^56 mod p;
                                      23037
```

Therefore, the other square root is:

```
> y := -x mod p;
                                      8836
```

A quick check that these are indeed the square roots of $a = 17919$ in $\mathbb{Z}^*_{31873}$:

```
> modp~([x, y]^~2, p);
                                  [17919, 17919]
```

The previous example serves to illustrate how the algorithm works step by step but Maple has this algorithm built-in in the function `numtheory:-msqrt`, which computes modular square roots whenever possible, even if the modulus is not prime. For the prime case, this function calls the function `numtheory:-_msqrtprime`, which implements Tonelli's algorithm. This function is local to the module `numtheory` but the reader interested in looking at its code can do so with the commands `kernelopts(opaquemodules=false)` followed by `interface (verboseproc=2)` and `print(numtheory:-_msqrtprime)`. Of course, in what follows we will use the function `numtheory:-msqrt` to compute modular square roots whenever necessary.

The algorithm to compute modular square roots can be extended to non-prime moduli. There is a case of special interest for cryptography which is when the modulus $n = pq$ is the product of two distinct primes p, q. We have already discussed the existence and the number of roots in this case. Based on this discussion, the following exercise is straightforward:

Exercise 2.46 Give an algorithm to solve the equation $x^2 \equiv a \pmod{n}$ when n is the product of two distinct primes and a is a quadratic residue modulo n. (Hint: Use Tonelli's algorithm and the Chinese remainder theorem.)

Chapter 3
Private-Key Encryption

In this chapter we introduce the basic ideas and concepts underlying private-key encryption (also called symmetric encryption). Particularly important is the concept of security, which for the classical ciphers studied in Chap. 1 was not precisely defined.Thus we shall start by looking at the first rigorous definition for this concept, namely *perfect secrecy* which, while providing the strongest security assurance, has important practical disadvantages that motivate the introduction of computational security concepts that are more suited for practical use. These notions are defined with the help of complexity theory and the concept of pseudo-randomness, a weakening of randomness that takes into account the fact that computational resources are limited. The security notions also make use of the concepts of one-way function and pseudo-random generator, which are of crucial importance for cryptography and are also introduced here.

3.1 Perfect Secrecy

As we have already mentioned, the Vigenère cipher was considered unbreakable after being in use for several centuries without any weaknesses having been found. But then Kasiski and Babbage discovered that this cipher was easy to break. This clearly showed that a long time passing without a cipher being broken was not a good indicator of security. But a precise definition of security did not come until almost 100 years later, when Shannon formulated it in the context of his ground-breaking work on information theory [176].

The basic idea that led Shannon to define *perfect secrecy* was that intercepting a ciphertext should not give the attacker any information about the plaintext nor any information about future encrypted messages, and that this should happen even assuming that the adversary has unbounded computational power.

To formalize the concept of perfect security we denote, as before, by $\mathcal{M}$, $\mathcal{C}$ and $\mathcal{K}$, the finite sets of plaintexts, ciphertexts and keys corresponding to the encryption

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_3, © Springer-Verlag Berlin Heidelberg 2013

scheme used, and we consider probability distributions defined on these sets and random variables M, C, K that represent the values of the message, the ciphertext and the key. The probability distribution over $\mathcal{K}$ will be simply the one defined by running a probabilistic algorithm that outputs the key and it will usually be the uniform distribution although this need not be the case. For $k \in \mathcal{K}$ we denote, as usual, by $\Pr(K = k)$ the probability that the chosen key is equal to k. Similarly, if $m \in \mathcal{M}$, $\Pr(M = m)$ is the probability that the message is equal to m, and the probability distribution on $\mathcal{M}$ may vary depending on the parties using the scheme, the language used and so on. Thus we will allow an arbitrary distribution for the messages and it is reasonable to assume that the adversary might even have some knowledge about it or, at least, that some specific messages have a much larger probability of being sent than others.

We will also assume that the probability distributions on $\mathcal{K}$ and $\mathcal{M}$ are independent, which is reasonable bearing in mind that the key is chosen before knowing what the message will be. For $c \in \mathcal{C}$ we let $\Pr(C = c)$ denote the probability that the ciphertext is c and we observe that its distribution on $\mathcal{C}$ is determined by the distributions on $\mathcal{K}$ and $\mathcal{M}$ because the probability that, for a given key k and a given plaintext m, the ciphertext is equal to c is just the probability that the decryption of c is the plaintext. Therefore, if $\mathbf{Dec}_k$ is the decryption algorithm corresponding to the key k, $\Pr(C = c) = \sum_{k \in \mathcal{K}} \Pr(K = k)\Pr(M = \mathbf{Dec}_k(c))$.

To define perfect secrecy we may assume that Eve knows the probability distribution on $\mathcal{M}$ and observes a ciphertext being sent by Alice to Bob. The idea is then that this observation should not have any effect on the knowledge that Eve has about the message sent, i.e., that the probability that a given message m was sent given the ciphertext observed by Eve is the same as the a priori probability that m would be sent. This captures the idea that Eve, on observing the ciphertext, learns nothing about the message sent even if she has unbounded computational power.

Definition 3.1 An encryption scheme (**Gen**, **Enc**, **Dec**) with message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ has *perfect secrecy* if, for every probability distribution on $\mathcal{M}$, every message $m \in \mathcal{M}$, and every ciphertext $c \in \mathcal{C}$ for which $\Pr(C = c) > 0$,

$$\Pr(M = m | C = c) = \Pr(M = m).$$

The concept of perfect secrecy is also called *unconditional security* and hence in this case we can also say that the encryption scheme is *unconditionally secure* or secure in the information-theoretic sense. Note that the condition $\Pr(C = c) > 0$ above is a technical one required for the conditional probability appearing in the definition to be defined. We will assume in the sequel that the probability distributions on $\mathcal{M}$ and $\mathcal{C}$ assign nonzero probabilities to all the elements of these sets, so that we always have $\Pr(M = m) > 0$ and $\Pr(C = c) > 0$ (this is only a matter of convenience and all the results can be adapted to the case of arbitrary distributions, at the cost of a longer wording). With this convention we note that it follows from Bayes' formula

that $\Pr(M = m|C = c)\Pr(C = c) = \Pr(C = c|M = m)\Pr(M = m)$ and hence the scheme has perfect secrecy if and only if:

$$\Pr(C = c|M = m) = \Pr(C = c).$$

Example 3.1

1. Let (**Gen**, **Enc**, **Dec**) be an encryption scheme with $\mathcal{M} = \mathcal{C} = \mathcal{K} = \{0, 1\}$ and suppose that the probability distributions on $\mathcal{M}$ and $\mathcal{K}$ are given by:

   - $\Pr(M = 0) = \frac{3}{4}$, $\Pr(M = 1) = \frac{1}{4}$.
   - $\Pr(K = 0) = \frac{2}{3}$, $\Pr(K = 1) = \frac{1}{3}$.

   Suppose further that, for each $m \in \mathcal{M}$ and each $k \in \mathcal{K}$, $\mathbf{Enc}_k(m) = m \oplus k$, where $\oplus$ denotes the Xor operation. In order to check whether this scheme has perfect secrecy, we compute the probability distribution on $\mathcal{C}$ induced by those on $\mathcal{M}$ and $\mathcal{K}$. The formula:

   $$\Pr(C = c) = \sum_{k \in \mathcal{K}} \Pr(K = k)\Pr(M = \mathbf{Dec}_k(c))$$

   gives:

   - $\Pr(C = 0) = \Pr(K = 0)\Pr(M = \mathbf{Dec}_0(0)) + \Pr(K = 1)\Pr(M = \mathbf{Dec}_1(0)) = \frac{2}{3} \cdot \frac{3}{4} + \frac{1}{3} \cdot \frac{1}{4} = \frac{7}{12}$.
   - $\Pr(C = 1) = \Pr(K = 0)\Pr(M = \mathbf{Dec}_0(1)) + \Pr(K = 1)\Pr(M = \mathbf{Dec}_1(1)) = \frac{2}{3} \cdot \frac{1}{4} + \frac{1}{3} \cdot \frac{3}{4} = \frac{5}{12}$.

   Then we can see that the scheme is not perfectly secure because the condition $\Pr(C = c|M = m) = \Pr(C = c)$ does not hold for all $m \in \mathcal{M}$ and all $c \in \mathcal{C}$:

   $$\Pr(C = 1|M = 1) = \Pr(K = 0) = \frac{2}{3} \neq \Pr(C = 1) = \frac{5}{12}.$$

   Alternatively, we can directly use the definition of perfect secrecy and check that the condition $\Pr(M = m|C = c) = \Pr(M = m)$ does not hold either. For example, let us compute $\Pr(M = 0|C = 0)$ which, by Bayes' formula, is equal to $\frac{\Pr(C=0|M=0)\Pr(M=0)}{\Pr(C=0)}$. Since $\Pr(C = 0|M = 0) = \Pr(K = 0) = \frac{2}{3}$ we obtain:

   $$\Pr(M = 0|C = 0) = \frac{\Pr(K = 0)\Pr(M = 0)}{\Pr(C = 0)} = \frac{\frac{2}{3} \cdot \frac{3}{4}}{\frac{7}{12}} = \frac{6}{7} \neq \Pr(M = 0) = \frac{3}{4}.$$

2. Let us now consider the same scheme as in 1 except for the fact that we now assume that the probability distribution on $\mathcal{K}$ is the uniform distribution. The distribution induced on $\mathcal{C}$ is now:

   - $\Pr(C = 0) = \frac{1}{2} \cdot \frac{3}{4} + \frac{1}{2} \cdot \frac{1}{4} = \frac{1}{2}$.

- $\Pr(C = 1) = \frac{1}{2} \cdot \frac{1}{4} + \frac{1}{2} \cdot \frac{3}{4} = \frac{1}{2}$.

and, from this, we obtain that now the scheme is perfectly secret because:

- $\Pr(C = 0|M = 0) = \Pr(K = 0) = \frac{1}{2} = \Pr(C = 0)$.
- $\Pr(C = 0|M = 1) = \Pr(K = 1) = \frac{1}{2} = \Pr(C = 0)$.
- $\Pr(C = 1|M = 0) = \Pr(K = 1) = \frac{1}{2} = \Pr(C = 1)$.
- $\Pr(C = 1|M = 1) = \Pr(K = 0) = \frac{1}{2} = \Pr(C = 1)$.

In any encryption scheme with message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ the inequality $|\mathcal{C}| \geq |\mathcal{M}|$ holds because the encryption functions $\mathbf{Enc}_k : \mathcal{M} \to \mathcal{C}$ are injective (otherwise unique decryption would not be possible). If, moreover, the scheme has perfect secrecy then we can show that another size limitation holds: the key space must be at least as large as the message space.

Proposition 3.1 *If* (**Gen**, **Enc**, **Dec**) *is a perfectly secret encryption scheme with message space* $\mathcal{M}$ *and key space* $\mathcal{K}$*, then* $|\mathcal{K}| \geq |\mathcal{M}|$.

Proof Let c be a ciphertext that occurs with nonzero probability. Since the scheme is perfectly secret we have that, for every $m \in \mathcal{M}$, $\Pr(C = c|M = m) = \Pr(C = c) > 0$. Thus the probability that m encrypts to c is positive and this means that there exists a key $k \in \mathcal{K}$ such that $\mathbf{Enc}_k(m) = c$. Similarly, if $m' \in \mathcal{M}$, $m' \neq m$, we have a key $k' \in \mathcal{K}$ such that $\mathbf{Enc}_{k'}(m') = c$ and this implies that $k \neq k'$ for, otherwise, $\mathbf{Enc}_k$ would not be injective. Thus we can define an injective map from $\mathcal{M}$ to $\mathcal{K}$ by assigning to m a key k such that $\mathbf{Enc}_k(m) = c$ and this proves the proposition. □

In Example 3.1.1 we have seen an encryption scheme that not only satisfies the inequality in the preceding proposition but, in fact, it also satisfies $\mathcal{M} = \mathcal{K} = \mathcal{C}$ without being perfectly secret. However, in Example 3.1.2 we have also seen that if we modify this scheme by making the probability distribution on $\mathcal{K}$ uniform, then we obtain a perfectly secret encryption scheme. This is a particular case of the following celebrated theorem of Shannon:

Theorem 3.1 (Shannon's Theorem) *Let* (**Gen**, **Enc**, **Dec**) *be an encryption scheme such that* $|\mathcal{M}| = |\mathcal{C}| = |\mathcal{K}|$*. Then the system has perfect secrecy if and only if the following conditions hold:*

(i) The probability distribution on $\mathcal{K}$ *is uniform.*
(ii) For every $m \in \mathcal{M}$ *and every* $c \in \mathcal{C}$*, there exists a unique* $k \in \mathcal{K}$ *such that* $\mathbf{Enc}_k(m) = c$.

Proof Suppose that the scheme is perfectly secret. As in the proof of Proposition 3.1 we see that, for each $c \in \mathcal{C}$ and each $m \in \mathcal{M}$ there exists at least one key $k \in \mathcal{K}$ such that $\mathbf{Enc}_k(m) = c$. If we fix m, this means that $|\{\mathbf{Enc}_k(m)\}_{k\in\mathcal{K}}| \geq |\mathcal{C}|$ and,

since the reverse inequality clearly holds, we actually have an equality. Therefore

$$|\{\mathbf{Enc}_k(m)\}_{k\in\mathcal{K}}| = |\mathcal{C}| = |\mathcal{K}|,$$

so that there is a unique key k satisfying $\mathbf{Enc}_k(m) = c$ and condition (ii) is proved.

Now, to prove (i) observe that, if we fix $c \in \mathcal{C}$ then, by (ii), for any $m \in \mathcal{M}$ there exists a unique key k such that $\mathbf{Enc}_k(m) = c$. Calling this key $k(m)$ and using Bayes' formula and the perfect secrecy of the scheme we have that:

$$\begin{aligned}\frac{\Pr(K = k(m))\Pr(M = m)}{\Pr(C = c)} &= \frac{\Pr(C = c|M = m)\Pr(M = m)}{\Pr(C = c)}\\ &= \Pr(M = m|C = c)\\ &= \Pr(M = m).\end{aligned}$$

Thus we see that $\Pr(K = k(m)) = \Pr(C = c)$, which is independent of m and shows that, for $m, m' \in \mathcal{M}$, $\Pr(K = k(m)) = \Pr(K = k(m'))$. Since all keys are of this form, we see that all of them have the same probability $\frac{1}{|\mathcal{K}|}$ and (i) is proved.

For the converse implication, assume now that both (i) and (ii) hold. For $m \in \mathcal{M}$ and $c \in \mathcal{C}$, let $k \in K$ be the unique key such that $\mathbf{Enc}_k(m) = c$. Then $\Pr(C = c|M = m) = \Pr(K = k) = \frac{1}{|\mathcal{K}|}$ no matter what the probability distribution on $\mathcal{M}$ is. Now, if $c \in \mathcal{C}$ we have that $\Pr(C = c) = \sum_{m\in\mathcal{M}} \Pr(C = c|M = m)\Pr(M = m) = \frac{1}{|\mathcal{K}|} \cdot \sum_{m\in\mathcal{M}} \Pr(M = m) = \frac{1}{|\mathcal{K}|}$. Thus we see that, for $m \in \mathcal{M}$ and $c \in \mathcal{C}$, $\Pr(C = c|M = m) = \Pr(C = c)$ which, as we have seen, is equivalent to the perfect secrecy of the system. □

Exercise 3.1 Prove that none of the ciphers considered in Chap. 1 (Caesar, substitution, Vigenère and Hill) has perfect secrecy when used to encrypt messages of arbitrary length and indicate how imposing restrictions on the plaintext space used (and not reusing keys) produces variants of these ciphers which are perfectly secret.

3.2 The One-Time Pad

We have seen that the Vigenère cipher can be easily broken with a ciphertext-only attack, if the attacker is given a sufficient amount of ciphertext. Moreover, one also sees that a longer key (or, to be more precise, a small ratio between the ciphertext length and the key length) makes the attack more difficult. In other words, using a longer key will make the unicity distance greater, and if the unicity distance is greater than the length of the available ciphertext, then the system will be very difficult to break. So the idea arises of using an *infinite* key length, a sort of limit of the (finite) Vigenère cipher. In practice, since all the encrypted messages will be finite, this amounts to using a key which has at least the same length as the message, with the

additional condition that the key is never repeated, i.e., a new key is used for each new message. But even this is not enough, for a key with regularities might make the system susceptible to statistical analysis and the security would be compromised. To avoid this, the cryptanalyst should not be able, given a portion of the key, to obtain any information about the remaining part, i.e., the sequence of characters constituting the key should be completely random. This way, one uses a new shift for each character, and this shift is unpredictable even if one knows the remaining ones. The idea of such an encryption scheme was first proposed by Vernam (an engineer working for the AT&T company) in 1917, and then Mauborgne recognized the importance of the key being random, for which the invention of this scheme can be attributed to both Vernam and Mauborgne. It is known as the *Vernam cipher* or *one-time pad*, the latter name coming from the fact that the (same portion of the) key is used just once, and a new one is used for each new message.

In the notation introduced above, the one-time pad can be defined as follows. For convenience, this cipher is usually described by using the binary alphabet but, of course, any other alphabet can be used without changing the cryptologic properties of the system.

Definition 3.2 The *one-time pad* is the following encryption scheme, where $n \geq 1$:

- $\mathcal{M} = \mathcal{C} = \mathcal{K} = \{0, 1\}^n$ for some integer n (so that these sets are all equal to the set of binary strings of length n).
- The key generation algorithm chooses keys from $\{0, 1\}^n$ according to the uniform distribution.
- If $k = (k_1, \ldots, k_n) \in \mathcal{K}$ then $\mathbf{Enc}_k((i_1, \ldots, i_n) := (i_1 + k_1, \ldots, i_n + k_n) \bmod 2$ and $\mathbf{Dec}_k = \mathbf{Enc}_k$. In other words, both the encryption and decryption algorithms act by performing the bitwise exclusive-or with the key string so that, denoting the Xor operation by $\oplus$, $\mathbf{Enc}_k(m) = k \oplus m$.

When the one-time pad was introduced, there was no notion of perfect secrecy as this concept would only be introduced by Shannon some 30 years later. However, it was already intuitively clear that the one-time pad did not allow an adversary to obtain any information about a plaintext from the observation of the corresponding ciphertext. Thus it did not come as a surprise that Shannon proved it to be perfectly secret:

Theorem 3.2 *The one-time pad is perfectly secret.*

Proof This is a direct consequence of Shannon's theorem. □

Remarks 3.1 In the one-time pad the key must be as long as the message and keys should be chosen at random, all with equal probability. These conditions must be satisfied by any perfectly secret encryption scheme as a consequence of Shannon's theorem (note that the first of them must hold if the keys and the messages have fixed length and the key space is at least as large as the message space). In fact, it can be asserted that, in some sense, the one-time pad is essentially the only perfectly secret encryption scheme (ignoring trivial differences such as the use of different alphabets

for the messages and so on). Thus one might be tempted to think that this scheme is all that a cryptographer needs or, borrowing a term from [183], that it is something like the *Holy Grail of cryptography*. But this is not the case for the one-time pad has some drawbacks. A relatively minor one is that it is not easy to generate truly random keys or, in this case, truly random sequences of bits, a process that requires access to a physical source of entropy.

There is another, more important, problem that prevents widespread use of the one-time pad: the fact that, as already mentioned, the key must be as long as the message. In symmetric cryptography, Alice and Bob need a secure channel for key exchange and, if the key has the same length as the message, this is asking too much in most situations. Indeed, if Alice and Bob have such a secure channel at their disposal to exchange the key, then why not use it to send the plaintext without encryption? Of course, there are situations in which this remark does not apply. For example, Alice and Bob might have a secure channel available today but they might not have it available tomorrow, which is when they would like to exchange the information. In this case the one-time pad can be useful and, in fact, it is said that this situation arose when the one-time pad was used by Soviet spies during the cold war. These agents were sent to other countries with their pads containing the key which would be used when needed, perhaps many years later. It is also rumored that the one-time pad is used in the *red telephone* between Washington and Moscow. However, it is clear that its use is not viable for many of the current applications of cryptography such as, for example, electronic commerce and, for this reason, its practical interest is rather limited.

Observe that, to obtain perfect secrecy in the one-time pad, we have assumed that all the messages (and also the keys and the ciphertexts) have the same length. In practice, one may work with messages of variable length $n \leq l$, where l is a suitable large integer. Each time a message is sent, a key is chosen at random among all the possible keys with the same length as the message, which amounts to using one-time pads of different lengths $n \leq l$. When several messages are sent in succession, the problem the adversary faces is to obtain information about the concatenation of these messages from the observation of the concatenation of the corresponding ciphertexts. Here the condition that a new random key be used for each new message is crucial for, otherwise, the adversary would observe a ciphertext obtained by using a key that contains a concatenation of a smaller key with itself and is, therefore, far from random.

Exercise 3.2 Prove that the one-time pad is perfectly secret by showing that $\Pr(C = c|M = m) = \Pr(C = c)$ without using Shannon's theorem. To do so, prove that:

(i) For all $c \in \mathcal{C}$ and all $m \in \mathcal{M}$, $\Pr(C = c|M = m) = \Pr(K = m \oplus c) = \frac{1}{2^n}$ (where n is, as above, the length of the messages).
(ii) For all $c \in \mathcal{C}$, $\Pr(C = c) = \frac{1}{2^n}$.

Exercise 3.3 Show that if the one-time pad is modified to make $\mathcal{K} = \{0, 1\}^n - \{0^n\}$ then the resulting encryption scheme is no longer perfectly secure. Explain what happens if we do this when $n = 1$.

Exercise 3.4 Prove that reusing keys allows a total break of the one-time pad (which should not be called that anymore!) under a chosen plaintext attack.

Exercise 3.5 The following is a dialogue extract from the movie 'Swordfish'. The movie is about a hacker, Stanley, who tries to rescue his daughter and is offered by Gabriel (a secretive counter-terrorist) the money he needs to do it. But Gabriel wants him to break a cipher in exchange:

Stanley.-*What kind of cipher?*
Gabriel.-*Vernam encryption.*
Stanley.-*A Vernam's impossible. Its key code is destroyed upon implementation. Not to mention being a true 128-bit encryption.*
Gabriel.-*Actually, we're talking 512-bit.*
Stanley.-*It's impossible.*
Gabriel.-*Tell ya what, I'll pay you ten million dollars. That should be enough to get your daughter back ... unless of course it's impossible.*
Stanley.-*Nothing's impossible.*

Question: *Comment on this scene from a cryptologic point of view.*

Exercise 3.6 The following is a quote from a cryptography text published in 2003:

> **Concluding Remarks.** If we increase the length of the keyword, then it is more difficult to break the system. Suppose we want to send a plaintext consisting of 500 letters secretly. Also we choose a paragraph of a novel, which consists of 500 letters, as our key. Then we encipher the message using this key, assuming that we could convey the key somehow to the intended recipient in a secured channel. Even if Mr. X knows the whole of the ciphertext, using whatever attack, it is not possible to break this system. Intuitively it is clear that the longer key gives more safety. This is what Shannon called **Perfect Secrecy.**

Question: *Comment on these remarks.* (You may also have a look at [64].)

Remark 3.1 One important thing that we learn from the one-time pad is the value of randomness for cryptography. Perfect secrecy is possible because the ciphertext not only looks random to an adversary but is, in fact, random, since Xor-ing non-random plaintexts with random keys produces ciphertexts as random as the keys, i.e., uniformly distributed. The system is not practical because it requires keys as long as the plaintext but in the next section we show how to relax this requirement in such a way that we can build a scheme that uses short random keys and yet is secure against real-world adversaries, whose computational power is always limited. This will be achieved with the use of pseudo-randomness, a weak form of randomness that makes mathematical objects look random to a computationally bounded adversary (see Definition 3.4 below). We will see that this concept plays a crucial role in cryptography because one of the main principles guiding the design of encryption schemes is that the encryption functions should behave like pseudo-random functions.

3.3 From Unconditional Security to Computational Security: Pseudo-Random Generators and One-Way Functions

The one-time pad is secure in the strongest possible sense but it is not a practical solution to securely exchange huge quantities of information, as the secret key must also be huge. To reach the goal of securely exchanging information using short keys, the definition of security must be relaxed. Starting with the idea that randomness of ciphertexts is what provides security we will aim for ciphertexts that, without being random, look random to a computationally bounded adversary. Hence we will weaken the security requirement in two ways: we will aim for encryption schemes that are secure only against *efficient adversaries* i.e., adversaries that run in probabilistic polynomial time and, on the other hand, we will require only that the adversary can be successful with very small probability. The first weakening is necessary because, if short keys are used, an adversary with unbounded computational resources can always mount a brute-force attack to recover the key with high probability. The second is also necessary because, even with limited computational resources, the adversary can guess a key with a nonzero (although very small) probability of finding the correct one.

The idea of modeling the adversary as a probabilistic polynomial-time algorithm (or PPT algorithm for short) is consistent with the identification of efficient computation with the kind carried out by these algorithms. Moreover, from the cryptographer's point of view, it is safer to assume that the adversary is as powerful as possible within realistic bounds and it is conceivable—although not proven—that the use of probabilistic algorithms might provide additional power. On the other hand, probabilistic algorithms are essential for cryptography and honest parties[1] must use them too to generate keys and also for encryption.

The main tools required to formalize the concept of security are pseudo-random generators and one-way functions. We will explain their basic properties and how they are used but we do not include detailed proofs of some theoretical results because they are outside the scope of this book. These proofs may be found in books dealing with the theoretical foundations of cryptography such as [86, 87] or the more introductory text [109].

3.3.1 Pseudo-Random Generators

To formalize the concept of computational security just outlined we need to make precise what is meant by "the adversary is successful with very small probability". The idea is to define the notion of *negligible success probability* by requiring that the adversary, modeled by a PPT algorithm, should be unlikely to succeed even after repeating an attack a polynomial number of times. This leads in a natural way to the following concept:

[1] The term honest is used here as a synonym of legitimate and is devoid of ethical content.

Definition 3.3 A function $f : \mathbb{N} \to \mathbb{R}^+$ is *negligible* if for every constant $c > 0$ there exists a positive integer n_0 such that $f(n) < n^{-c}$ for all $n > n_0$.

Remark 3.2 f being negligible means that $f(n)$ diminishes to zero faster than the reciprocal of any positive polynomial p, i.e., that for sufficiently large values of n, $f(n) < 1/p(n)$. In particular $f(n)$ gets arbitrarily small for sufficiently large n.

Example 3.1 The functions 2^{-n^2}, 2^{-n}, $2^{-\sqrt{n}}$, $n^{-\ln n}$ and $n^{-\ln \ln n}$ are inverses of functions with faster than polynomial growth and hence all of them are negligible but, as can readily be checked using Maple, they approach zero at widely different rates.

Exercise 3.7 Prove that f is negligible if and only if $f(n) = o(n^{-c})$ for every $c > 0$.

In the sequel we will denote by `negl` an arbitrary negligible function. Negligible functions behave well vis-a-vis polynomial functions:

Exercise 3.8 Prove that the sum of two negligible functions is negligible and, more generally, that the product of a positive polynomial by a negligible function is also negligible (here, sum and product are interpreted as pointwise operations so that, for example, $(f \cdot g)(n) = f(n) \cdot g(n)$).

As a consequence of the previous exercise, if the probability that an algorithm produces a certain output on inputs of size n is negligible, then the probability that it produces this output at least once when repeated polynomially many times is also negligible. Thus a negligible probability can be ignored for all practical purposes.

Let us now see how the outlined idea may be applied to derive a practical encryption scheme from the one-time pad. To achieve this we want to use short keys to encrypt much longer messages by means of the Xor operation. But to do that we need to somehow *expand* the key to obtain a string as long as the message that can be Xor-ed with it. The problem is that then this longer string will not be random anymore, i.e., the strings of a given length obtained by "expanding" shorter strings will not be sampled according to the uniform distribution. But, since we are assuming now that our adversary is polynomially bounded, we no longer need that these strings are random but just that they look random to the adversary. This idea is expressed by saying that the strings obey a *pseudo-random* distribution, and in order to formalize it we will consider a *distinguisher* D, i.e., a polynomial-time algorithm that outputs '0' or '1' according to whether or not the input strings look random to it. Then a probability distribution is *pseudo-random* if no polynomial-time distinguisher can detect whether it is given a string sampled according to this distribution or according to the uniform distribution. Bearing all this in mind, what we need to obtain long pseudo-random strings from short random strings is a *pseudo-random generator* which is defined as follows:

Definition 3.4 Let $\ell : \mathbb{N} \to \mathbb{N}$ be a polynomial-time computable function such that $\ell(n) > n$ for every $n \in \mathbb{N}$. Let G be a deterministic polynomial-time algorithm that,

for any input $s \in \{0,1\}^n$ (called a *seed*), outputs a string of length $\ell(n)$ (so that G can be regarded as a polynomial-time computable function $G : \{0,1\}^* \to \{0,1\}^*$ such that $G(\{0,1\}^n) \subseteq \{0,1\}^{\ell(n)}$). Then we say that G is a *pseudo-random generator*[2] (PRG, for short) of *stretch* (or *expansion factor*) ℓ if for every PPT distinguisher D there exists a negligible function $\mathtt{negl}$ such that

$$|\Pr(D(G(s)) = 1) - \Pr(D(r) = 1)| \leq \mathtt{negl}(n),$$

for every r chosen uniformly at random from $\{0,1\}^{\ell(n)}$ and every s chosen uniformly at random from $\{0,1\}^n$.

Remarks 3.2

1. Recall here that $\{0,1\}^*$ denotes the set of all (finite-length) binary strings, not to be confused with $\mathbb{Z}_2^* = \{1\}$, which is the group of units of the ring $\mathbb{Z}_2$.
2. A pseudo-random generator maps random strings of length n to pseudo-random strings of length $\ell(n) > n$ and so one might wonder how much longer $\ell(n)$ should be than n. The answer is that this is irrelevant as long as $\ell(n) > n$ for, even if the stretch is $n \to n+1$, we can obtain a PRG of stretch ℓ for any length function ℓ which is polynomial in n by just applying G recursively to its own output as many times as necessary. It can be shown (cf., [90, Theorem 3.4], [109, Theorem 6.23] or [86, Theorem 3.3.3]) that if there exists a polynomial-time distinguisher able to distinguish its output from random strings of length $\ell(n)$, then the output of G would also be distinguishable from random strings of length $n+1$.
3. The output of a pseudo-random generator cannot be random and, in fact, is far from random when the stretch is large. When G acts on inputs of length n, the number of possible inputs is at most 2^n and then this is also an upper bound for the number of strings of length $\ell(n)$ in the image of G. But there are $2^{\ell(n)}$ strings of length $\ell(n)$, where $2^{\ell(n)} > 2^n$, and so there are at least $2^{\ell(n)} - 2^n$ strings that are never generated by G although, of course, any of them can be chosen at random according to the uniform distribution. Thus the distribution generated by G cannot be the uniform distribution.
4. The preceding remarks leave open a very important question: Does a pseudo-random generator actually exist? This problem is open and a necessary condition for a positive answer is that $\mathcal{P} \neq \mathcal{NP}$ (otherwise, given a string in the output of the supposed pseudo-random generator, a short seed giving this string could be found in polynomial time, thus distinguishing this output from true randomness). However, even assuming $\mathcal{P} \neq \mathcal{NP}$, we do not know how to prove that pseudo-random generators exist (or, equivalently, that one-way functions exist; see the discussions in [109, Chap. 6] or [86, Chap. 3] as well as Sect. 3.3.3 below).

Let G be a pseudo-random generator with stretch ℓ. The following encryption scheme is an analogue of the one-time pad which uses short keys:

[2] Also called a *cryptographically secure* (or *cryptographically strong*) pseudo-random generator.

Definition 3.5 (Pseudo-random one-time pad)

- $\mathcal{M} = \mathcal{C} = \{0,1\}^{\ell(n)}$ for some positive integer n and $\mathcal{K} = \{0,1\}^n$.
- **Gen** chooses a key uniformly at random in $\{0,1\}^n$.
- If $k \in \mathcal{K}, m \in \mathcal{M}$ and $c \in \mathcal{C}$, then $\mathbf{Enc}_k(m) = G(k) \oplus m$ and $\mathbf{Dec}_k(c) = G(k) \oplus c$.

Remark 3.3 Although we have not yet formally defined the concept of computationally secure encryption, we can give an intuitive argument that shows that this scheme is secure in the sense that a PPT adversary $\mathcal{A}$, given a ciphertext, cannot guess one bit of plaintext with probability significantly (i.e., non-negligibly) greater than $1/2$. Indeed, we know that this guess is impossible when $G(k)$ is random (for, in that case, we are dealing with the one-time pad which is perfectly secret) and, since G is a pseudo-random generator by hypothesis, $\mathcal{A}$ cannot distinguish between a truly random string of length $\ell(n)$ and $G(k)$ so the guess is also impossible for $G(k)$. We shall make this a little more precise once we formally define the notion of computational security.

We are going to implement this scheme in Maple, assuming an external source of true randomness. We need a (candidate) pseudo-random generator to reduce the number of truly random bits required but these random bits are still necessary for the seed. Random bits are very difficult to obtain and a physical source is needed to generate them. Some of these sources can be: elapsed time between particle emission during radioactive decay, thermal noise from semiconductors, fluctuations in disk drive read latency times, etc. Less reliable sources are the system clock, elapsed time between keystrokes or mouse movement, content of input/output buffers, etc., but, by combining several of them, acceptable results can be obtained in practice. Operating systems have tools to generate random bits such as Linux's `/dev/random` device that uses data taken from keyboard, mouse, network, and disc activities or Windows' function `CryptGenRandom` that provides similar functionality.

There is a simpler source which, at the cost of being very inefficient, can produce bits that are random for all practical purposes: coin tossing (the physics of (fair) coin tossing is discussed in [66], whose last sentence is: "The classical assumptions of independence with probability $1/2$ are pretty solid."). Thus we will assume that the bits produced by successive coin tosses are uncorrelated (this can probably be achieved with some care) but, on the other hand, these bits might well be biased since the coin might not be fair. There is, however, a simple trick due to Von Neumann that produces unbiased bits from a biased coin (or, more generally, unbiased uncorrelated bits from a source of biased but uncorrelated bits):[3]

Von Neumann's coin flipping trick. We toss the coin an even number of times and group the obtained bits in pairs. Each '00' or '11' pair is discarded, each '10' pair replaced by a '1' bit and each '01' pair replaced by a '0' bit.

[3] There are also methods to produce uncorrelated bits from correlated bit sources but they are more complicated and we shall not study them here.

The bits produced are unbiased because if the probability that the coin lands heads up is p (where $p \leq 1$) and hence the probability that it lands tails up is $1 - p$, then the probability of obtaining a '10' pair is the same as the one of obtaining a '01' pair, namely $p(1-p)$. The only problem might be efficiency as, on average, we will have to toss the coin a number of times $\geq 4n$ to obtain n bits.

Exercise 3.9 Assuming that the probability that the coin lands heads up is p, compute the average number of coin flips that it takes to generate a bit using Von Neumann's trick.

The classical techniques to produce pseudo-random numbers useful for Monte Carlo simulations—for example, linear feedback shift registers and linear congruential generators—do not yield PRGs in the sense of Definition 3.4 and hence are not suitable for cryptographic use. The classical methods to evaluate pseudo-randomness rely on the use of statistical tests that check whether the string of bits generated has the expected number of patterns (for example, the number of times that a specific substring should appear, see [115]). But they are completely inadequate to pin down generators suitable for cryptographic use, which have to satisfy the stronger requirement that the output bit string must look random to *every* PPT algorithm. However, it is not necessary to consider all PPT algorithms because there is a much smaller subclass of algorithms that suffices to guarantee that we have a PRG. These algorithms check an obvious requirement that the PRG must fulfill to be secure: an adversary (a PPT algorithm) that sees a portion of the generator's output must be unable to efficiently predict other unseen bits. Otherwise, an adversary that sees a ciphertext from the scheme in Definition 3.5 might guess a portion of the message and hence a portion of $G(k)$ and, if it could then efficiently compute other bits of $G(k)$, it would also be able to decrypt other portions of the message. These special PPT algorithms are defined as follows:

Definition 3.6 Let $G : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time computable function with stretch $\ell : \mathbb{N} \to \mathbb{N}$. A *predictor* for G is a PPT algorithm $\mathcal{A}$ which, for every randomly chosen seed $s \in \{0,1\}^n$, with $G(s) = y_1 y_2 \ldots y_{\ell(n)}$, takes as input 1^n and a prefix $y_1 y_2 \ldots y_i$ of $G(s)$ (with $i < \ell(n)$ randomly chosen) and outputs a bit b (we think of b as a guess made by $\mathcal{A}$ trying to predict the next bit y_{i+1}). In other words, $b \leftarrow \mathcal{A}(1^n, y_1 y_2 \ldots y_i)$ (here we use the $\leftarrow$ notation to emphasize that the algorithm is probabilistic).

Remark 3.4 The input 1^n is the representation of n in *unary notation* where n is given as a string of n ones. The predictor $\mathcal{A}$ must run in time polynomial on the size of n which, if n were given in positional notation, would be the binary length $\text{len}(n)$. However, in this case, n is given in unary and so its size is n itself, not $\text{len}(n)$. Thus the unary notation means that the algorithm's running time must be a polynomial function of n and not of its binary length $\text{len}(n)$.

Using predictors, we define a special class of tests that suffice to check whether a polynomial-time computable function is a PRG:

Definition 3.7 Let $G : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time computable function with stretch $\ell : \mathbb{N} \to \mathbb{N}$. We say that G is *unpredictable* (or that G passes all next-bit tests) if for every predictor $\mathcal{A}$ there exists a negligible function $\mathtt{negl}$ such that, for every s, $G(s)$ and i as in the previous definition:

$$\Pr(\mathcal{A}(1^n, y_1 y_2 \dots y_i) = y_{i+1}) \leq 1/2 + \mathtt{negl}(n).$$

The interest of this definition is due to the following result of Yao [203] according to which *unpredictability implies pseudo-randomness* ([86, Theorem 3.3.7], [6, Theorem 9.11] or [90, Theorem 3.10]):

Theorem 3.3 *Let $G : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time computable function with stretch $\ell : \mathbb{N} \to \mathbb{N}$. Then G is a PRG if and only if G is unpredictable.*

The distinguisher in the definition of PRG (Definition 3.4) may be regarded as a statistical test trying to distinguish truly random sequences from sequences generated by G. Since the classical statistical tests are implemented as polynomial-time algorithms, the previous theorem is sometimes stated in the following way: *G passes all next-bit tests if and only if G passes all statistical tests*.

Exercise 3.10 Let G, H be pseudo-random generators with the same stretch function. Determine which of the following are also PRGs (you may have to consider whether there is some correlation between the two PRGs):

(i) $\neg G$, where $\neg$ denotes the Boolean NOT or Boolean complement.
(ii) $G \vee H$, where $\vee$ is the Boolean OR.
(iii) $G \oplus H$.

3.3.2 One-Way Functions

We have already mentioned one-way functions which, as we shall see, are perhaps the most important cryptographic primitive because they are necessary for such basic things as secure encryption and digital signatures. The concept of one-way function emerged from the development of public-key cryptography but later it was shown that their existence is essentially equivalent to the existence of nontrivial private-key cryptography: for example, the existence of one-way functions is equivalent to the existence of pseudo-random generators. However, there are no unconditional proofs of the existence of one-way functions and we do not know how to prove it even assuming $\mathcal{P} \neq \mathcal{NP}$ which is only a necessary condition. But as we shall see later, the existence of one-way functions can be proved under reasonable assumptions relative to the hardness of certain computational problems.

A one-way function is, informally speaking, a function f which is easy to compute but hard to invert. In more precise terms this means that any PPT algorithm attempting to invert the function on an element in its image, will succeed only with negligible

probability, where the probability is taken over the elements on the domain of the function and the random coins of the algorithm. This is reasonable in view of the fact that an inverse image (or preimage) of an element can always be found with negligible probability just by guessing and, on the other hand, if we allow the inverting algorithm enough power, it is always possible to invert the function just by performing a brute-force search over its domain in exponential time. To formalize this we define an experiment (a game between a challenger and an adversary $\mathcal{A}$):

Definition 3.8 The *inversion experiment* $\mathbf{Inv}_{\mathcal{A},f}(n)$ for a function $f : \{0,1\}^* \to \{0,1\}^*$, where $\mathcal{A}$ is a PPT algorithm and n a positive integer, is the following:

1. A random $x \leftarrow \{0,1\}^n$ is chosen and $y := f(x)$ is computed (by the challenger).
2. $\mathcal{A}$ is given 1^n and y as input and outputs an integer x'.
3. The output of the experiment is defined to be 1 ($\mathcal{A}$ wins the game) if $f(x') = y$, and 0 otherwise.

The formal definition of one-way function is then the following:

Definition 3.9 A function $f : \{0,1\}^* \to \{0,1\}^*$ is *one-way* if the following conditions hold:

1. (*f is easy to compute.*) There exists a PPT algorithm that on input x outputs $f(x)$.
2. (*f is hard to invert.*) For every PPT algorithm $\mathcal{A}$ there exists a negligible function `negl` such that
$$\Pr(\mathbf{Inv}_{A,f}(n) = 1) \leq \mathtt{negl}(n).$$

Remarks 3.3

1. The meaning of the 'security parameter' 1^n, given in unary in the definition above, is that $\mathcal{A}$ takes as input 1^n and $f(x)$ for some randomly chosen x such that $\text{len}(x) = n$ and has to run on polynomial time on n, independently of the length of $f(x)$. This prevents the possibility that a function might be regarded as one-way because the size of x is exponential as a function of the size of $f(x)$. For example, if $f(x) = y$, where y is the string formed by the $\lfloor \ln \ln x \rfloor$ most significant bits of x, then any inverting algorithm requires time exponential in $\text{len}(f(x))$ (because just printing its output is exponential in $\text{len}(f(x))$) but, clearly, there is an inverting algorithm polynomial in $\text{len}(x)$. Of course, in case f is a length-preserving function, i.e., $\text{len}(f(x)) = \text{len}(x)$, then the auxiliary input given by the security parameter is no longer necessary.
2. Note that, in Definition 3.8, $\mathcal{A}$ is not asked to output x but some x' such that $f(x') = f(x)$ although, of course, the only possible value is x if f is injective.
3. There are many variations on the concept of one-way function; the definition above is perhaps the most convenient and these functions are often called *strong one-way functions* (when a PPT algorithm has negligible success probability in inverting them while, for example, *weak* one-way functions are those for which

PPT algorithms fail with non-negligible probability, see e.g., [86, Chap. 2], [90, Chap. 2]).

Since the existence of one-way functions has not been unconditionally proven, we have to settle for working with *candidate one-way functions*, which are functions that can be proved one-way under the hypothesis that some well-studied computational problem is hard. The most famous of these problems is the integer factorization problem which we have already mentioned. Its presumed hardness leads to the conjecture that the function $(x, y) \mapsto x \cdot y$ is one-way if x and y are integers (or even primes) of the same length. Formally, if we want to view this as a function from $\{0,1\}^*$ to itself, we may define for a string $s \in \{0,1\}^*$, $\mathtt{mult}(s) = xy$, where x is the integer represented by the first $\lfloor n/2 \rfloor$ bits of s and y the one corresponding to the last $\lceil n/2 \rceil$ bits of s.

Remark 3.5 The necessity of putting some condition on the lengths of the factors x, y above when defining the candidate one-way function comes from the fact that factoring a number with small factors is easy. Since the product xy is even most of the time, without this restriction it would be easy to return the pair of factors $2, xy/2$.

The previous definition considers a single function but, to apply this concept to cryptographic practice, it is more convenient to deal with a family of one-way functions parameterized by a security parameter 1^n.

Definition 3.10 A family of functions $\{f_i : D_i \to R_i\}_{i \in I}$, where I is a set and D_i, R_i are finite sets, is *one-way* if the following conditions hold:

1. There exists a PPT parameter generation algorithm **Gen** which takes as input a security parameter 1^n and outputs an index i such that $\mathrm{len}(i) \geq n$.
2. There exists a PPT sampling algorithm **Samp** which, on input $i \in I$, outputs (with uniform distribution) $x \leftarrow D_i$ (except possibly with probability negligible in $\mathrm{len}(i)$).
3. There exists a deterministic polynomial-time evaluation algorithm **Eval** which, on input $i \in I$ and $x \in D_i$, outputs $f_i(x)$.
4. For every PPT algorithm $\mathcal{A}$ there exists a negligible function $\mathtt{negl}$ such that if $i \leftarrow \mathbf{Gen}(1^n)$, $x \leftarrow \mathbf{Samp}(i)$, $y := \mathbf{Eval}(i, x)$:

$$\Pr(f_i(\mathcal{A}(i, y)) = y) \leq \mathtt{negl}(n).$$

If, moreover, $D_i = R_i$ for each $i \in I$ and the f_i are bijections, then $\{f_i\}_I$ is called a family of one-way permutations.

Remark 3.6 The existence of a family of one-way functions is equivalent to the existence of a single one-way function as previously defined (see, e.g., [90, Theorem 2.13]).

Exercise 3.11 Define an *inversion experiment* for a family of functions (given by algorithms **Gen**, **Samp**, **Eval** as above) similar to the experiment in Definition 3.8 and formulate the definition of a family of one-way functions in terms of this experiment.

Example 3.2 This example is studied in detail in Chap. 6 since it has important cryptographic applications. Let $I = \{(p, g) | p \text{ prime}, \mathbb{Z}_p^* = \langle g \rangle\}$ and consider a PPT parameter generation algorithm that, on input 1^n, outputs an element $(p, g) \in I$, where $\text{len}(p) = n$ (we already know that, given p, a PPT algorithm outputting a primitive root modulo p exists and we will also see in Chap. 6 that there exists a PPT algorithm which, on input 1^n, outputs an n-bit prime p). Consider also a PPT sampling algorithm that, given $(p, g) \in I$, outputs a random element of $\mathbb{Z}_{p-1}$ or, equivalently, a random integer in $\{0, 1, \ldots, p-2\}$. We may then define, for each $(p, g) \in I$, the function

$$\texttt{dexp}_{(p,g)} : \mathbb{Z}_{p-1} \to \mathbb{Z}_p^*$$

given by $x \mapsto g^x \bmod p$. This is a family of functions indexed by I which are actually bijections because g is a generator of $\mathbb{Z}_p^*$ and so $\mathbb{Z}_p^* = \{g^0, g, g^2, \ldots, g^{p-2}\}$ (in fact, they are group isomorphisms since $g^{x+x'} = g^x g^{x'}$). Moreover, these functions are easy to compute by means of the binary exponentiation algorithm. The inverse of $\texttt{dexp}_{(p,g)}$ is the *discrete logarithm* function $\texttt{dlog}_{(p,g)} : \mathbb{Z}_p^* \to \mathbb{Z}_{p-1}$, given by $\texttt{dlog}_{(p,g)}(y) = x$ where $g^x = y \in \mathbb{Z}_p^*$. Computing the discrete logarithm is presumed to be hard relative to the parameter generation algorithm and so a PPT algorithm should only be able, given (p, g) and $g^x \bmod p$, to guess x with negligible probability. Thus this family is thought to be one-way.

A discrete logarithm function can be defined, more generally, by considering instead of $\mathbb{Z}_p^*$, any cyclic group G of order q and a generator $g \in G$. Then we have a group isomorphism

$$\texttt{dexp}_{(G,g)} : \mathbb{Z}_q \to G$$

given by $x \mapsto g^x$. The inverse isomorphism, given by $\texttt{dlog}_{(G,g)}(y) = x$ where $g^x = y \in G$ is the discrete logarithm function corresponding to the pair (G, g) (x is then called the discrete logarithm of y with respect to g, $x = \log_g y$). The following example shows that computing discrete logarithms is not always difficult:

Example 3.3 Consider the family of additive cyclic groups $\mathbb{Z}_p$ and a parameter generation PPT algorithm that, on input 1^n, outputs a pair (p, g), where p is a prime with $\text{len}(p) = n$ and g a nonzero element of $\mathbb{Z}_p$, which is also a generator of $\mathbb{Z}_p$. Then we have as before the functions $\texttt{dmul}_{(p,g)} : \mathbb{Z}_p \to \mathbb{Z}_p$, given by $x \mapsto gx \bmod p$ (these are really the "discrete exponential" functions corresponding to these groups except that, as the groups are additive now, exponentiation becomes multiplication). These functions are isomorphisms, and their inverses are the additive analogue of the discrete logarithm functions, namely, the functions $\mathbb{Z}_p \to \mathbb{Z}_p$ that to each $y \in \mathbb{Z}_p$ assign the unique $x \in \mathbb{Z}_p$ such that $gx \bmod p = y$. But finding such an x from (p, g) and y is easy because, as $g \in \mathbb{Z}_p^*$, g has a multiplicative inverse modulo p which can be obtained by using the extended Euclidean algorithm. Then we recover x by computing $x = (g^{-1} \bmod p \cdot y) \bmod p$. Hence computing discrete logarithms in the additive groups $\mathbb{Z}_p$ is easy and the family of functions $\texttt{dmul}_{(p,g)}$ is far from one-way. Note that, in order to compute discrete logarithms in this way, we use

additional information that goes beyond the group structure we are dealing with, namely, we also use the multiplication defined on $\mathbb{Z}_p^*$ and this is why this method cannot be extended to generic cyclic groups. On the other hand, we also remark that the fact that $\mathbb{Z}_p$ has prime order is not essential here and the computation of discrete logarithms is also easy in any $\mathbb{Z}_t$ even if t is not prime.

Exercise 3.12 Suppose that f is a length-preserving one-way function and g a length-preserving permutation such that $g(x)$ is efficiently computable for each x. Determine whether each of the compositions $f \circ g$, $g \circ f$, is a one-way function. What happens if the permutation g is not efficiently computable? Determine also whether the composition of two length-preserving one-way functions is one-way.

3.3.3 From One-Way Functions to Pseudo-Random Generators: The Blum–Blum–Shub PRG

We have already mentioned that PRGs can be obtained from one-way functions and this is the basic idea used in cryptography to obtain them. The fact that the existence of one-way functions implies that of PRGs—they are in fact equivalent—was proved by Håstad, Impagliazzo, Levin and Luby in the 1990s but for the PRG we want to use we only need the following weaker result which assumes the existence of a length-preserving one-way permutation (instead of a one-way function, see [6, Lemma 9.10], [90, Theorem 3.4], [86, Theorem 3.5.1], [109, Corollary 6.11]):

Theorem 3.4 *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a length-preserving one-way permutation. Then, for every polynomial-time computable function $\ell : \mathbb{N} \to \mathbb{N}$ such that $\ell(t) > t$ for all $t \in \mathbb{N}$, there exists a PRG $G : \{0,1\}^* \to \{0,1\}^*$ with stretch ℓ.*

A more explicit version of this result that enables us to actually construct a PRG can be obtained if we analyze the way in which the preceding theorem is proved. For this we need some additional notions related to one-way functions.

Definition 3.11 A *predicate* is a function:

$$B : \{0,1\}^* \to \{0,1\},$$

i.e., a function that acts on bit strings and whose output is a single bit.

Now let f be a one-way function. A *hard-core predicate* of f is a predicate that is difficult to compute given only $f(x)$ but easy to compute given x. More formally:

Definition 3.12 A predicate $B : \{0,1\}^* \to \{0,1\}$ is a *hard-core predicate* of a function $f : \{0,1\}^* \to \{0,1\}^*$ if the following conditions hold:

1. There exists a PPT algorithm $\mathcal{A}$ such that $\mathcal{A}(x) = B(x)$ for every $x \in \{0,1\}^*$.
2. For every PPT algorithm $\mathcal{G}$, there exists a negligible function `negl` such that

$$\Pr(\mathcal{G}(f(x)) = B(x)) \leq 1/2 + \mathtt{negl}(t),$$

where the probability is taken over a random uniform choice of x of length t and over the random bits of $\mathcal{G}$.

The idea behind the definition of hard-core predicate—which can be extended in a straightforward way to a family of one-way functions, see [90, Definition 2.50]—is the following. A (candidate) one-way function f does not necessarily hide everything about x, given $f(x)$. For example:

Exercise 3.13 Let $\mathtt{dexp}$ be the function $f : \mathbb{Z}_{p-1} \to \mathbb{Z}_p^*$ given by $\mathtt{dexp}(x) = g^x \mod p$, where p is a prime and g a primitive root modulo p. Use the Legendre symbol to show how the least significant bit of x can be efficiently computed from $\mathtt{dexp}(x)$.

But, if f is one-way and hence a randomly chosen x is hard to compute from $f(x)$, it seems that there should at least be one bit of x which is hard to guess from $f(x)$. So the idea of Definition 3.12 is to select specific bits in x that are hard to compute given $f(x)$.

By a theorem of Goldreich and Levin ([86, Theorem 2.5.2], [6, Theorem 9.12), a one-way function can be modified to have a hard-core predicate. In particular, this applies to one-way permutations and so Theorem 3.4 is a consequence of the following more explicit version ([86, Proposition 3.4.3]):

Theorem 3.5 *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a length-preserving one-way permutation with hard-core predicate $B : \{0,1\}^* \to \{0,1\}$. Then for every polynomial-time computable function $\ell : \mathbb{N} \to \mathbb{N}$ such that $\ell(t) > t$ for every $t \in \mathbb{N}$, the function $G : \{0,1\}^* \to \{0,1\}^*$ defined, for $x \in \{0,1\}^t$, by*

$$G(x) = B(x)B(f(x))B(f^2(x)) \ldots B(f^{\ell(t)-1}(x)),$$

where we denote the concatenation of bits by juxtaposing them, is a PRG.

The proof proceeds by showing that we can first construct a PRG of stretch $\ell(t) = t + 1$ (also called an *extender*), see for example, [90, Theorem 3.4], [109, Theorem 6.22], and then iterating this construction.

Now, in order to apply this theorem to actually build a PRG, we need a length-preserving one-way permutation with hard-core predicate. The best candidate is the modular square function defined on $\mathbb{Z}_n^*$, the group of units of $\mathbb{Z}_n$, for a composite integer n. Given $x \in \mathbb{Z}_n^*$, its square $x^2 \mod n \in \mathbb{Z}_n^*$ can be computed efficiently as it only requires an integer multiplication followed by the corresponding reduction mod n. However, as Rabin showed already in 1979, computing modular square roots is presumed hard when n is composite or, in other words, the function $x \mapsto x^2 \mod n$ is one-way assuming factoring is hard (we will give a more precise formulation of this hypothesis below). Assuming that this is indeed the case, we still have something missing in order to be able to apply Theorem 3.5 because it is necessary that

the function be a permutation, i.e., bijective. We may consider the *Blum–Williams function*:

$$\begin{array}{ccc} \mathcal{QR}_n & \stackrel{BW_n}{\longrightarrow} & \mathcal{QR}_n \\ x & \longmapsto & x^2 \bmod n. \end{array}$$

BW_n is not a permutation in general but there is a special case in which this is true.

Definition 3.13 A *Blum integer* is an integer n such that $n = pq$, where p and q are distinct primes such that $p \equiv q \equiv 3 \pmod 4$ (as we have already mentioned, a prime p such that $p \equiv 3 \pmod 4$ is also called a Blum prime).

The modular squaring map BW_n defined above has an especially good behavior when n is a Blum integer. In this case we have:

Theorem 3.6 *Let n be a Blum integer, $\mathcal{QR}_n$ the set of quadratic residues in $\mathbb{Z}_n^*$, and BW_n the modular squaring map. Then $BW_n : \mathcal{QR}_n \longrightarrow \mathcal{QR}_n$ is a permutation.*

Proof First we show that if p is a Blum prime and $x \in \mathcal{QR}_p$ then $-x \notin \mathcal{QR}_p$. Since p is a Blum prime there exists an integer k such that $p = 4k + 3$ and hence we have that

$$(-1)^{(p-1)/2} = (-1)^{2k+1} \equiv -1 \pmod p.$$

Thus, from Euler's criterion (Proposition 2.12) if follows that $-1 \notin \mathcal{QR}_p$ and, since the product of a quadratic residue by a quadratic non-residue is a quadratic non-residue because of the multiplicativity of the Legendre symbol, we see that $-x = (-1)x \notin \mathcal{QR}_p$.

Now let $n = pq$ where p and q are Blum primes. Taking into account that the set $\mathcal{QR}_n$ of squares of $\mathbb{Z}_n^*$ is finite, to prove that BW_n is a permutation it suffices to show that it is injective. Suppose then that $BW_n(x) = BW_n(y)$, i.e, that $x^2 \equiv y^2 \pmod n$. We have to show that $x \equiv y \pmod n$. Since $x, y \in \mathcal{QR}_n$ we have that also $x, y \in \mathcal{QR}_p$ and $x, y \in \mathcal{QR}_q$, so that their opposites satisfy $-x, -y \notin \mathcal{QR}_p$ and $-x, -y \notin \mathcal{QR}_q$ and, in particular, we also have that $-x, -y \notin \mathcal{QR}_n$. Our hypothesis $x^2 \equiv y^2 \pmod n$ means that $n|(x + y)(x - y)$ and, since $x \in \mathcal{QR}_n$ and $-y \notin \mathcal{QR}_n$, we have that $x \not\equiv -y \pmod n$ or, equivalently, n does not divide $x + y$. Thus either each of the two prime factors of n divides one of the terms $x + y$, $x - y$, or, alternatively, $n|(x - y)$. In the first case we may assume without loss of generality, that $p|(x + y)$, i.e., $x \equiv -y \pmod p$ but, as $x \in \mathcal{QR}_p$, this implies that $-y \in \mathcal{QR}_p$ which is a contradiction. Thus we are only left with the second alternative, i.e., $n|(x - y)$ or, in other words, $x \equiv y \pmod n$, which completes the proof. □

Remark 3.7 We have seen in Corollary 2.7 that if n is a product of two distinct odd primes, then each quadratic residue has exactly four square roots modulo n. From the preceding theorem it follows that if, furthermore, n is a Blum integer, then exactly one of these four square roots is also a quadratic residue modulo n.

We are now going to show that, if factoring is hard, then the functions BW_n (where n is assumed to be a Blum integer) define a family of one-way permutations. More generally, we can show that if factoring is hard, then the squaring functions modulo n, where n is a product of two distinct odd primes, are a family of one-way functions, so that the former result will be obtained by restricting this family to Blum integers and applying Theorem 3.6. For this, we must define more precisely the meaning of "factoring is hard". In order to do that we first define a factoring experiment:

Definition 3.14 The *factoring experiment* $\mathbf{Fact}_{\mathcal{A}}(k)$, where $\mathcal{A}$ is a PPT algorithm and k a positive integer, is the following:

1. Run a PPT algorithm that, on input 1^k outputs $n = pq$, where p and q are two random k-bit primes (except with probability negligible in k).
2. $\mathcal{A}$ is given n and outputs a positive integer r.
3. The output of the experiment is defined to be 1 if $r|n$ and $r \neq 1$, $r \neq n$ (i.e., if r is a proper factor of n) and 0 otherwise.

Then we may define the *factoring assumption*:

Definition 3.15 (Factoring assumption) For every PPT algorithm $\mathcal{A}$ there exists a negligible function $\mathtt{negl}$ such that

$$\Pr(\mathbf{Fact}_{\mathcal{A}}(k) = 1) \leq \mathtt{negl}(k).$$

Remarks 3.4

1. The factoring assumption may be defined, more generally, relative to any PPT algorithm that, on input 1^k, generates a modulus n which is a product of (not necessarily random) k-bit primes (see [109, Definition 7.45] for details).
2. The reasons why factoring is thought to be hard are studied in detail in Chap. 6, where factoring algorithms and their efficiency are analyzed.

Next we are going to see that computing square roots modulo n, where n is a product of two k-bit primes, is as hard as factoring n. In a way similar to the factoring assumption we may define what it means that "computing square roots is hard". For this we define:

Definition 3.16 The *square root computation experiment*, $\mathbf{Sqr}_{\mathcal{A}}(k)$, where $\mathcal{A}$ is a PPT algorithm and k a positive integer, is the following:

1. Run a PPT algorithm that on input 1^k outputs $n = pq$ where p and q are (except with probability negligible in k) random k-bit primes.
2. Choose a random $y \leftarrow \mathcal{QR}_n$.
3. $\mathcal{A}$ is given n and y as input and outputs $x := \mathcal{A}(n, y) \in \mathbb{Z}_n^*$.
4. The output of the experiment is defined to be 1 if $x^2 \equiv y \pmod{n}$ and 0 otherwise.

Using this experiment we may define the hardness of computing square roots:

Definition 3.17 We say that *computing square roots modulo a product of two random k-bit primes is hard* if, for every PPT algorithm $\mathcal{A}$, there exists a negligible function $\mathtt{negl}$ such that

$$\Pr(\mathbf{Sqr}_{\mathcal{A}}(k) = 1) \leq \mathtt{negl}(k).$$

We next show that computing square roots modulo n, where n is a product of two random k-bit primes, is as hard as factoring n:

Theorem 3.7 *Under the factoring assumption, computing square roots modulo a product of two random k-bit primes is hard.*

Proof Let $\mathcal{A}$ be a PPT algorithm that tries to compute a square root modulo n. To show that $\Pr(\mathbf{Sqr}_{\mathcal{A}}(k) = 1)$ is negligible as a function of k, we consider the following algorithm $\mathcal{A}_f$ that tries to factor a modulus n by using $\mathcal{A}$ as a subroutine:

Algorithm $\mathcal{A}_f$:

1. Choose a random $x \leftarrow \mathbb{Z}_n^*$ and compute $u = x^2 \bmod n$.
2. Run $\mathcal{A}$ on input (n, u) to obtain as output $y := \mathcal{A}(n, u)$.
3. Check whether $x \equiv \pm y \pmod{n}$. If this is not the case then output $\gcd(x - y, n)$; otherwise go to step 1.

Observe that $\mathcal{A}_f$ runs correctly if y is a square root of u modulo n. This is because then $x^2 \equiv y^2 \pmod{n}$ and so $(x + y)(x - y) \equiv 0 \pmod{n}$ or, equivalently, $n|(x + y)(x - y)$. But $x \not\equiv \pm y \pmod{n}$ implies that n does not divide either of these factors and hence $\gcd(x - y, n)$ must be a nontrivial factor of n. Moreover, since x was randomly chosen, x is equally likely to be each of the four square roots of u from the point of view of $\mathcal{A}$ (there are indeed four square roots by Corollary 2.7). Therefore, x is uniformly distributed among the square roots of u and, assuming that the output y of $\mathcal{A}$ is one of them, the probability that $x \not\equiv \pm y \pmod{n}$ is $1/2$. Thus we have that the probability that $\mathcal{A}_f$ factors n is just one-half the probability that $\mathcal{A}$ outputs a square root of u, that is:

$$\Pr(\mathbf{Fact}_{\mathcal{A}_f}(k) = 1) = \frac{1}{2}\Pr(\mathbf{Sqr}_{\mathcal{A}}(k) = 1).$$

Hence we obtain:

$$\Pr(\mathbf{Sqr}_{\mathcal{A}}(k) = 1) = 2\Pr(\mathbf{Fact}_{\mathcal{A}_f}(k) = 1) \leq 2\mathtt{negl}(k),$$

where $\mathtt{negl}$ is the negligible function from the factoring assumption, completing the proof. □

Remark 3.8 The converse of the preceding theorem is also true, i.e., if computing square roots modulo a product of two equal-length random primes is hard then

factoring is also hard. This is a straightforward consequence of Algorithm 2.8 and of the proof of Proposition 2.14, which in polynomial time reduces the computation of square roots modulo n to the computation of square roots modulo its prime factors.

Definition 3.18 Let $I = \{n \in \mathbb{Z}^+ | n = pq$, where p and q are distinct primes of the same length$\}$. Then the family of *Rabin functions* is the family $\{R_n : \mathbb{Z}_n^* \to \mathbb{Z}_n^*\}_{n \in I}$, where $R_n(x) := x^2 \bmod n$. Similarly, if J is the set of Blum integers, the family of *Blum–Williams permutations* is the family $\{BW_n : \mathcal{QR}_n \to \mathcal{QR}_n\}_{n \in J}$, where, again, $BW_n(x) := x^2 \bmod n$ (these are permutations as a consequence of Theorem 3.6).

These families are one-way, for we have:

Theorem 3.8 *Under the factoring assumption, the family of Rabin functions is a family of one-way functions and the family of Blum–Williams permutations is a family of one-way permutations.*

Proof To see that the Rabin functions are one-way we note that:

1. There exists a PPT algorithm that, on input k, outputs an integer n which is a product of two random k-bit primes (this algorithm can be obtained from one that outputs a random k-bit prime, see Algorithm 6.2).
2. There exists a PPT algorithm that, on input n, outputs a random $x \leftarrow \mathbb{Z}_n^*$.
3. There exists a PPT algorithm that, given $x \in \mathbb{Z}_n^*$, computes $x^2 \bmod n \in \mathbb{Z}_n^*$.
4. Finally, Theorem 3.7 implies that for every PPT algorithm $\mathcal{A}$, $\Pr(\mathcal{A}(R_n(x))^2 \equiv x \pmod n) = \Pr(\mathbf{Sqr}_{\mathcal{A}}(k) = 1) \leq \mathtt{negl}(k)$, for some negligible function $\mathtt{negl}$.

The proof that the Blum–Williams permutations are a one-way family of permutations is similar, bearing in mind that Blum integers may be generated by a PPT algorithm and that a random $x \leftarrow \mathcal{QR}_n$ may be chosen in polynomial time by just squaring a random element of $\mathbb{Z}_n^*$. □

Remark 3.9 We will see that the family of Rabin functions is also of interest for public-key cryptography. One of the reasons for this interest is the fact that this family has a property that is even stronger than the one we have just seen, namely, the Rabin functions are a family of one-way trapdoor functions. Roughly, this means that there is a "trapdoor information"—the factorization of n in this case—that allows the efficient inversion of the functions.

As mentioned before, in order to obtain a PRG from a length-preserving one-way permutation we need a hard-core predicate. For the Blum–Williams permutations this is given by the following result of Blum et al. [31]:

Theorem 3.9 *Let n be a Blum integer. Under the factoring assumption, the least significant bit is a hard-core predicate for the one-way permutation BW_n.*

This allows us to define the Blum–Blum–Shub (BBS) pseudo-random generator as follows:

Definition 3.19 Let $n = pq$, where p and q are primes such that $p \equiv q \equiv 3 \pmod 4$. Let $\ell : \mathbb{N} \to \mathbb{N}$ be a polynomial-time computable function such that $\ell(t) > t$ for every $t \in \mathbb{N}$. Then the *Blum–Blum–Shub* PRG of stretch ℓ is the function $G{:}\{0,1\}^* \to \{0,1\}^*$ defined as follows.

- Select $s \in [1, n-1]$ at random and, for $x_0 = s^2 \bmod n \in \{0,1\}^*$, with $\mathrm{len}(x_0) = t$ (i.e., $x_0 \in \{0,1\}^t$) and $i = 1, \ldots, \ell(t)$, compute:

$$\begin{array}{llll} x_1 = x_0^2 & \bmod n, & b_1 = x_1 & \bmod 2, \\ x_2 = x_1^2 & \bmod n, & b_2 = x_2 & \bmod 2, \\ \vdots & & \vdots & \\ x_i = x_{i-1}^2 & \bmod n, & b_i = x_i & \bmod 2, \\ \vdots & & \vdots & \\ x_{\ell(t)} = x_{\ell(t)-1}^2 & \bmod n, & b_{\ell(t)} = x_{\ell(t)} & \bmod 2. \end{array} \tag{3.1}$$

- Define $G(x_0) = b_1 b_2 \ldots b_{\ell(t)}$.

From the preceding results we now obtain:

Theorem 3.10 *Under the factoring assumption, Blum–Blum–Shub is a pseudo-random generator.*

Proof This is a consequence of Theorems 3.5, 3.8 and 3.9. □

Remark 3.10 All existence proofs of PRGs are, like this one, conditional. Conditional proofs of this type are usually based on *reductions* which show that a problem is hard under the assumption that some other problem (which, ideally, should be well-understood) is also hard. Here, we see that Blum–Blum–Shub is indeed a PRG if the factorization problem (for Blum integers) is hard but, of course, the latter has not been proved. However, we feel that the factorization problem, which has a longer history than PRGs, is better understood than the latter and, because of this, the reduction is useful.

Exercise 3.14 Give an informal argument showing that if pseudo-random generators exist then one-way functions exist. (Hint: If G is a PRG with stretch $2n$, show that G itself is one-way by showing that a PPT algorithm $\mathcal{A}$ able to invert G with non-negligible probability can be used as a subroutine for a distinguisher D that, given a string $s \in \{0,1\}^{2n}$ outputs 1 if $G(\mathcal{A}(s)) = s$ and 0 otherwise).

We already gave an informal proof of the fact that the pseudo-random one-time pad introduced in Definition 3.5 is "computationally secure"—in the intuitive sense mentioned in Remark 3.5, which is made more precise in Sect. 3.5.2, where the formal definitions of security are given—assuming that the underlying generator G is indeed a PRG. Now we can refine this by specifying the BBS PRG as the one to be used to generate the long keys. The informal argument relies on two reductions: the one made in Remark 3.5 shows that if we use a PRG to generate long keys, then

the pseudo-random one-time pad is computationally secure in the sense mentioned, while a second reduction is provided by Theorem 3.10 which shows that BBS is indeed a PRG under the factoring assumption. Combining both reductions, we have:

- *The pseudo-random one-time pad based on the Blum–Blum–Shub PRG is computationally secure, in the sense of Remark 3.3, under the factoring assumption.*

Remarks 3.5 BBS is the preferred PRG for cryptographic applications for the following reasons:

1. Its security reduction relies on a well-understood problem: the factorization problem.
2. BBS is relatively efficient as it requires only one modular squaring per output bit.
3. In practice, the BBS PRG can be implemented even more efficiently if one knows the prime factors p, q of n, for then the x_i in Definition 3.19 can be computed directly as $x_i = x_0^{2^i \bmod \phi(n)} \bmod n$, where $\phi(n) = (p-1)(q-1)$. In this situation, the exponents $2^i \bmod \phi(n)$ may be pre-computed and their computation requires $O(\text{len}(i))$ multiplications modulo $\phi(n)$. Thus x_i may be computed from x_0 and $\phi(n)$, without previous knowledge of x_{i-1}, in time $O(\max(\text{len}(n)^3, \text{len}(i)\text{len}(n)^2))$, while using the definition directly this computation takes time $O(i\text{len}(n)^2)$. Another advantage of this method is that if BBS is used to generate a long key (for example, for the pseudo-random one-time pad), then a ciphertext can be decrypted from any starting point without forming the bit string from the beginning.

3.3.3.1 Concluding Remarks on PRGs and One-Way Functions

We have seen in this section that a key ingredient of private-key cryptography, namely, pseudo-random generators, is obtained from the hypothesis that one-way functions exist. It can also be shown that the existence of one-way functions suffices to guarantee the existence of pseudo-random permutations, another key ingredient of private-key cryptography for, as we shall see, they are the theoretical models on which block ciphers are built. We will also see that one-way functions and, specifically, trapdoor one-way functions are also crucial for public-key cryptography so that it can be said that the concept of one-way function (and the related concept of one-way permutation) is the most important cryptographic primitive. We refer to Chap. 6 of [109] for a detailed and accessible discussion of these questions.

3.4 PRGs and Related Constructions in Maple

In this section we review, from a cryptographic perspective, Maple's algorithms for the generation of pseudo-random numbers. Since not all of them satisfy Definition 3.4, we will not call them PRGs and we will often generically refer to them as 'pseudo-random algorithms' instead.

3.4.1 The Blum–Blum–Shub PRG in Maple

The default algorithm used by Maple to generate pseudo-random numbers (the one called by the function `rand()`) is the Mersenne Twister algorithm which, although very fast and useful for Monte Carlo simulations, is not cryptographically secure and hence is not a PRG in the sense of Definition 3.4. Thus we shall not use this function, except in cases where unpredictability is not required. Among Maple's pseudo-random algorithms, only the Blum–Blum–Shub generator satisfies the conditions in Definition 3.4 and is suitable for cryptographic use. This is the PRG we are going to use in our implementations of cryptographic algorithms.

The BBS PRG is implemented in Maple as a subpackage of the `RandomTools` package. We will use, in particular, the function `NewBitGenerator` of this subpackage, whose long name is (taking into account that these packages are implemented as modules, see Maple's help) `RandomTools:-BlumBlumShub:-NewBitGenerator`. This function has a required input parameter for the seed, which is a positive integer different from 1. There are also three optional input parameters called `numbits`, `output`, and `primes` and the output of the function is a Maple procedure which is itself an implementation of BBS with the parameters specified when calling `NewBitGenerator`. `numbits` is an integer that specifies the number of bits that are computed in each call to the generator, `output` specifies whether the output is given as a sequence of bits or as an integer and, finally, `primes` takes as value either 512 (the default), 768 or 1024. In the first case, the Blum integer used as modulus by BBS is a product of two 512-bit primes, and a product of two 768-bit or two 1024-bit primes in the other two cases. These integers have 308, 462, and 616 decimal digits, respectively, and are stored in three variables local to the module `RandomTools:-BlumBlumShub` called `n512`, `n768`, and `n1024`, respectively. We can easily learn what the possible values for the modulus n are and, for example, `n1024` is the following integer:

```
> kernelopts(opaquemodules = false):
  print(RandomTools:-BlumBlumShub['n1024']);
6358895565833034702208523084557621229832212772877790391642907586084929071132845514\
  2085797721576720292498442415 28[...413 digits...]2423830887510240212368507183232 2\
  5423972751283525954972335764771832799875479564691130767232 1
```

We will use `NewBitGenerator` to output a list of pseudo-random bits obtained from a random seed. Suppose that our random seed is the following 128-bit integer:

```
> s := 242591295382282583902940438229126740883:
```

which can be written as a string or as a list[4] (with the most significant byte first) as follows:

```
> Bits:-String(s, msbfirst);
"10110110100000010110001001001101111110111110110011111101101001101000000101001000\
  11000001111111101000111000000111111101110010011"

> ListTools:-Reverse(Bits:-Split(s));
[1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0,
  0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0,
  1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0,
  1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1]
```

If we now want to generate, for example, a 512-bit string by means of BBS, using `s` as seed, we just compute:

```
> BBS := RandomTools:-BlumBlumShub:-NewBitGenerator(s, primes = 1024):
  pb := cat("", seq(BBS(), i = 1 .. 512));
"10110011011010010001010100011000100000110010011111001000011000001100110100001111\
  00000001010100110011010111100001001000110111000001110000011101101101101111011011\
  01011010110011100001010101011100111010101011011000000100001100001001010000110011\
  10000100001010110000101110111111111001100101101100110111011010110100010000000110\
  11100110101000011100011111111000111101000110000101010100111000110110110000111010\
  11011011110010001000011000111011100010111110011001011110001000001010011101110100\
  00111001111010111111100100111110"
```

If we prefer the output of BBS given as a list—which is often more convenient for carrying out further operations with it—then we would instead call (without printing the output):

```
> pb := [seq(BBS(), i = 1..512)]:
```

The security of the BBS generator is, as we have seen, conditional on the hypothesis that the factoring problem is hard and, in this specific case, on the hardness of the factorization of the three numbers `n512, n768, n1024`. When using Maple's BBS as a source of pseudo-randomness for cryptographic purposes one has to trust that this factorization is indeed hard and also that no one knows the factors. In the help page of `NewBitGenerator` it is mentioned that the factorization of these numbers was discarded and that Maple does not have it ...

On the other hand, if one does not trust this particular choice of integers, it is very easy to replace them by one's favorite Blum integers. For this it suffices to create a new procedure using Maple's `NewBitGenerator` code, in which the variable names `n512`, `n768`, and `n1024` are replaced by the numbers you choose (or by other names that point to these numbers). One must be careful, however, in case the size of these numbers is changed, for then other parameters inside this procedure should be changed too. Also, one should take care to choose numbers which make the BBS generator have a long period, as explained in [144].

[4] Here we use Maple's package `Bits` available only from version 12 onwards but, with a little further work, this can also be done with previous versions of Maple.

3.4.2 An Approximation to the One-Time Pad in Maple

In this section we present Maple implementations of both the pseudo-random one-time pad and the one-time pad and we discuss their security. We will make use of some of the Maple functions introduced in Appendix A.

The encryption scheme in Definition 3.5 is "computationally secure" assuming the existence of a PRG and a source of random bits. As already mentioned, the Blum–Blum–Shub generator satisfies the conditions in Definition 3.4 and hence we are going to use it for our implementation of the pseudo-random one-time pad.

We start with an auxiliary function called `BBSByteGen`, which uses Maple's built-in function `NewBitGenerator` (described in Sect. 3.4.1) to generate pseudo-random bytes from a random seed. `BBSByteGen` has two required input parameters called `seed` and `length`. The first is used for the seed (which will be the key used by the pseudo-random one-time pad), given as a positive integer, and the second to specify the byte length of the output. There are also two optional keyword parameters. The first is `format` which can take as values `list` (the default) or `hex`, and is used to specify whether the generated bytes will be given as a list or as a hexadecimal string. The second keyword parameter, `bbslength`, serves to specify the bit length of the primes in the modulus of Blum–Blum–Shub. Since these moduli are three Blum integers which are already built into Maple's implementation of BBS, there are only three possible values that this parameter will accept: 512, 768, or 1024, which are the lengths of the two primes used to build each of these three integers. The default value is 1024, which seems to provide a good security margin (see the remarks on the current state of factoring in Sect. 6.4.10, which suggest that a product of two 512-bit primes will be factored within a few years).

The function then calls `RandomTools:-BlumBlumShub:-NewBitGenerator` with the specified `seed` and `bbslength` values and this call produces a BBS bit generator that outputs one bit at a time in the form described in Sect. 3.3.3. Then this bit generator is used to produce the required list or string of bytes as the output of `BBSByteGen`.

As a minor technical detail let us mention that, when calling `NewBitGenerator`, the value of `seed` is increased by 1. This is due to the fact that, otherwise, `NewBitGenerator` would enter an infinite loop if `seed = 1`. Of course, the seed should be randomly chosen among numbers of reasonable size, so that it is very unlikely that the passed value would ever be equal to 1.

```
> BBSByteGen := proc(seed::posint, length::posint,
                     {format::name:=list, bbslength::{512, 768, 1024}:=1024 })
  local B, l;
  B := RandomTools:-BlumBlumShub:-NewBitGenerator(1+seed, primes=bbslength);
  l := [seq(bitstobyte([seq(B(), i = 1 .. 8)]), i = 1 .. length)];
  if format = list then
    l
  elif format = hex then
    bytestohexstring(l)
  else
    error "Unrecognized Format"
  end if
  end proc:
```

Exercise 3.15 Give an alternative Maple procedure to generate a list of pseudo-random bytes from a seed, using the option `numbits = 8` in each call to `NewBit Generator`.

Example 3.7 Suppose that we want to use the number 9234876983328430757 as seed to generate a sequence of 256 bytes, using the modulus which is a product of two 1024-bit primes. We do the following (without printing the output):

```
> BBSByteGen(9234876983328430757, 256):
```

Now, the same sequence of bytes, given as a hex string, would be:

```
> BBSByteGen(9234876983328430757, 256, format = hex);
"4a7a15beff9af7b6a28d7d5e23f55bf38725bbfc92c805ed1f69a363cbcd9491d2a0b1523e2f2a7b4\
  df49995e7acb0568ac397d7c04210a7a3174b7c8c46b7c0f5a554b6199699dd809637cd68536b5c8\
  3fd3b410ab1aa10f6e0553ae5147b9adfc044d1f62a642d588d80f1bbb1e98d1a840e343b9b0792a\
  c0580f99524a03308dbc4d52a53740efcec4935af150cd0ac0ec22960166e524ce74678e9b83636a\
  b43eab15332dbb95eaa0f8a53e97b9bd08917ae06810303f8c3429c4b2a995fd3794f63ee03a536e\
  55779c3a9af55c30f881085fa7ed563fb7536497b99662f0cac11854ede3f4405471055670f5d618\
  472be584ec16ab940fdc7e280c6d623"
```

In Definition 3.5, we described the pseudo-random one-time pad as an encryption scheme with fixed-length messages of binary length $\ell(n)$. In practice, we will allow messages of variable length and to encrypt them we will generate a list of pseudo-random bytes of the required length.

The next procedure implements the encryption/decryption function for this scheme and makes use of several functions from Appendix A, including `string posint`, which will often be used to convert seeds or keys to decimal form. The first required parameter is for the key, which can be either a positive integer (in decimal form) or an even-length hex string that can also be regarded as the base-256 expansion of a positive integer where, in order to eliminate ambiguity, each base-256 digit is represented, as already explained, by two hex digits. The second required parameter is `message`, which will take as value the plaintext when encrypting and the ciphertext when decrypting. In the first case the message is an ASCII text string but ciphertexts will always be hex strings to ensure that they are printable. The third parameter, `action`, is optional and the permitted values are `'encrypt'` and `'decrypt'`, with the former being the default.

Finally, there is another optional keyword parameter `bbslength` with the same purpose as in the function `BBSByteGen` above. The output is either the ciphertext (a hex string) or the plaintext (a text string).

```
> POTP := proc(key::{posint, string}, message::string, action::name := 'encrypt',
              {bbslength::{512, 768, 1024} := 1024})
  local k, l, m, blist;
  uses StringTools;
  if action = 'encrypt' then
    l := Length(message);
    m := convert(message, bytes)
  elif action = 'decrypt' then
    if not IsHexDigit(message) then
      error "Ciphertext must be a hex string"
    end if;
    m := hexstringtobytes(message);
```

```
    l := nops(m)
  else
    error "Unrecognized action"
  end if;
  k := stringposint(key);
  blist := BBSByteGen(k, l, ':-bbslength' = bbslength);
  m := zip(BitXor, m, blist);
  if action = 'encrypt' then
    bytestohexstring(m)
  else
    convert(m, bytes)
  end if
  end proc:
```

For example, given the following 128-bit hex key and the following message:

```
> k := "cf59d8aa3278f823bfea7e31546b6ccb":
  m := "This is a message to test the implementation of the Pseudo-Random
  One-Time Pad":
```

we may use `POTP` to encrypt the message and to recover it from the ciphertext as follows:

```
> c := POTP(k, m);
"761bece82e6ac923faa546fa84241469caf5988b1a09a4a40798a816b02709102941f95e84ae9bde7\
  41fad908331aaf81546a80feb3742c1ec2bf91322bc3c356b034d585e26d4cd32f9f2853902"

> POTP(k, c, decrypt);
"This is a message to test the implementation of the Pseudo-Random One-Time Pad"
```

The function `POTP` deals only with messages given as text strings but we can easily modify it to deal with files by using the procedures `filetobytes` and `bytestofile` defined in Appendix A. The function is actually simpler because in this case we do not have to worry about the different formats that the message may have: we will treat it always as a binary file and the resulting ciphertext/plaintext will also be written to a file. Because of this, the operations of encryption and decryption are exactly the same, so we do not have to tell the function whether we are encrypting or decrypting.

```
> POTPF := proc(key::{posint, string}, message::string, filename::string,
                filecheck::truefalse, {bbslength::{512, 768, 1024} := 1024})
  local k, l, m, blist;
  k := stringposint(key);
  m := filetobytes(message);
  l := nops(m);
  blist := BBSByteGen(k, l, ':-bbslength' = bbslength);
  bytestofile(zip(BitXor, m, blist), filename)
  end proc:
```

Example 3.5 We create a file named `testfile` in the current directory, we encrypt it with `POTPF` using the previously defined key `k` and call the ciphertext `ctestfile`. Then we decrypt it to a file named `dtestfile` and we read this file to check that it contains the original message:

```
> bytestofile(convert("This is a file to test the implementation of the
                       Pseudo-Random One-Time Pad", bytes), "testfile");
75 bytes saved to testfile

> POTPF(k, "testfile", "ctestfile");
75 bytes saved to ctestfile

> POTPF(k, "ctestfile", "dtestfile");
75 bytes saved to dtestfile

> convert(filetobytes("dtestfile"), bytes);
"This is a file to test the implementation of the Pseudo-Random One-Time Pad"
```

Exercise 3.16 Write a Maple function that integrates `POTP` and `POTPF` and permits the encryption and decryption of messages given either as text strings or as files.

Now, with a similar approach, we can give the encryption/decryption function for the one-time pad. When we discussed the one-time pad in Sect. 3.2, the length of the messages was fixed. Here we deal with variable length messages but we can fix a message length and then encrypt any message of smaller length by first padding it (for example, with '0' bytes) until it has the required length and then proceeding to encrypt it as before. This way, an adversary would not be able to know the length of the plaintext.

The one-time pad function is `OTP` below. The inputs are similar to those in `POTP` but this function is actually simpler because the key must be at least as long as the message and randomly generated. Thus the key has to be externally generated and then supplied to the function and hence the pseudo-random generator is not used.

```
> OTP := proc(key::{posint, string}, message::string, action::name := 'encrypt')
  local k, l, m, blist;
  uses StringTools;
  if action = 'encrypt' then
    l := Length(message);
    m := convert(message, bytes)
  elif action = 'decrypt' then
    if not IsHexDigit(message) then
      error "Ciphertext must be a hex string"
    end if;
    m := hexstringtobytes(message);
    l := nops(m)
  else error "Unrecognized action"
  end if;
  if type(key, string) then
    if not IsHexDigit(key) then
      error "The key must be either a hex string or an integer"
    end if;
    k := hexstringtobytes(key)
  else
    k := convert(key, base, 256)
  end if;
  if nops(k) < l then
    error "The key length is less than the message length"
  end if;
  m := zip(BitXor, m, k);
  if action = 'encrypt' then
    bytestohexstring(m)
  else
    convert(m, bytes)
  end if
  end proc:
```

Exercise 3.17 Write a Maple procedure similar to `OTP` that also includes the possibility of encrypting and decrypting binary files.

We remark that this implementation is only an approximation to the one-time pad as there is an important additional requirement that must be met: the key must be selected at random with uniform probability distribution. It is very difficult to generate a random key in software and even more so if the key, as happens in this case, is usually going to be long, but this aspect is crucial if we want the scheme to be secure. We next discuss the security aspects in more detail.

3.4.3 Practical Security Aspects

We have given an informal argument that shows that the pseudo-random one-time pad is computationally secure, but there are several aspects that must be taken into account for this to apply to the preceding construction. The first important point to observe is that the security argument assumes that the key used is truly random. This is a source of concern as it is by no means easy to generate truly random seeds to be used as keys, although we can always resort to the inefficient method of tossing a coin, using Von Neumann's trick just in case the coin is biased. But there are also other problems that must be addressed. The PRG security definition is asymptotic and, in practice, one must be careful, for the notion of "asymptotic hardness" is not meaningful when applied only to small values.

More concretely, an attacker can always mount a *brute-force attack* against BBS (or against any PRG for that matter) consisting of running through all possible seeds and computing the corresponding output strings. If the seed has length t, then there are 2^t possible seeds and, since this number grows exponentially in the size, the attack is, in principle, infeasible. But, in practice, it is perfectly feasible if t is small enough. For example, suppose that we use the function `POTP` to encrypt, say, 512-bit messages using a 32-bit key. There are 2^{32} possible keys and it is perfectly feasible—and even easy—to compute all the corresponding BBS outputs, thus distinguishing with high probability between these pseudo-random 512-bit strings and truly random strings of the same length (the former would only be a tiny subset of the latter). In this case the adversary would not only be able to obtain significant information about the plaintext but, in fact, it would be able to recover the whole plaintext with high probability if, for example, it is a text in a natural language. This is because, after the search, there would be only 2^{32} possible candidate plaintexts, one of which is the true plaintext. Since there are 2^{512} possible 512-bit messages, where an overwhelming majority would be meaningless in the language, the probability of obtaining more than two meaningful messages in the set of the 2^{32} candidate plaintexts would be very small indeed. We give an example of such an attack below.

To be secure against brute-force attacks we should be careful and choose the length of the key large enough to make these attacks infeasible. Computing 2^{64} values seems within reach today so, to be safe, a key of at least 128 bits should be used. In fact, as

we will see in Chap. 5, there are situations in which this number should be increased, because of the 'birthday attack', to at least 256. This requirement makes the problem of obtaining truly random bits even harder, since a few bits are not sufficient.

The methods offered by Maple to automatically seed its pseudo-random algorithms are not satisfactory from the cryptographic point of view because they give too few bits which, in addition, are not really random. Of course, Maple warns at appropriate places that, for cryptographic purposes, truly random seeds must be provided by the user. For this reason, automatic seeding is not available in Maple's function `NewBitGenerator`, which is the one we used in `POTP`. Maple also warns that its default pseudo-random algorithm, Mersenne Twister, is not adequate for cryptographic use, in consonance with the fact that it is not a PRG in the sense of Definition 3.4. However, the BBS-based Maple function `RandomTools:-BlumBlumShub:-NewGenerator` allows automatic seeding, with the seed being provided by Mersenne Twister, itself automatically seeded (again, Maple warns that this is not secure). One might be tempted to use a similar method to provide a seed for the Blum–Blum–Shub PRG (for example, to provide a key for POTP) but, as we show in Example 3.6 below, this would be highly insecure even disregarding the fact that Mersenne Twister does not satisfy Definition 3.4.

The Mersenne Twister algorithm can be seeded automatically by Maple in two different ways. One of these methods uses the function `randomize()` which calls the kernel function `iolib(25)` to obtain a seed. This seed is the absolute value of the current value of `iolib(25)` and is equal to `mods`$(n, 2^{32})$ where n is the number of seconds passed since January 1, 1970 (the value returned by `iolib(25)` is a 32-bit signed integer). This seeding method is far from satisfactory for several reasons. The first is that the seed is currently too small at 31 bits, (and it will never get past 2^{31}, a value that will be reached on January 19, 2038, at the same time as the so-called "2038 problem" which is related to computers storing system time in the same way as `iolib(25)`)[5] so the above outlined brute-force attack is perfectly viable against a generator seeded this way. On top of this, taking the number of seconds passed since a fixed date is anything but random and an adversary that can approximately guess the date in which the algorithm was seeded, can also accurately guess most of the bits of the seed used, making the brute-force attack even easier.

There is an alternative way to automatically seed Mersenne Twister, through the use of the function `RandomTools:-MersenneTwister:-SetState()`, which calls the function `MTKernelInterface(1)` (local to the module `RandomTools:-MersenneTwister`) which, in turn, calls the undocumented kernel function `RandNumberInterface`. This function, again, only produces a 32-bit seed, which is insufficient to prevent a brute-force attack. This problem might be alleviated by collecting several outputs of this function at different times but we shall not go into these details.

Example 3.6 **A brute-force attack against `POTP` when an inadequate key is used.** In this example, we start by performing a couple of simple experiments to show what

[5] See, e.g., http://en.wikipedia.org/wiki/Year_2038_problem for a description of this problem.

happens when automatic seeding is used and then we show how an attacker can easily break the scheme implemented in POTP even if a key of more than 300 bits is used (assuming that this key was obtained by using Mersenne Twister with automatic seeding).

Let us start by calling randomize() to initialize Mersenne Twister and then using RandomTools:-MersenneTwister:-GenerateInteger to generate a large integer to be used as a seed for BBS:

```
> s := randomize();
  RandomTools:-MersenneTwister:-GenerateInteger(':-range' = 10^99 .. 10^100);
46808685476229186764263183273582713195292903380788264618835200577632601688911928666\
  909615753148750450
```

Now, s holds the value of the seed used for Mersenne Twister and we can recover the large integer value generated by just calling:

```
> RandomTools:-MersenneTwister:-SetState(state = s);
  RandomTools:-MersenneTwister:-GenerateInteger(':-range' = 10^99 .. 10^100);
46808685476229186764263183273582713195292903380788264618835200577632601688911928666\
  909615753148750450
```

A similar thing happens if we use the alternative method to automatically seed Mersenne Twister:

```
> kernelopts(opaquemodules = false):
  s := RandomTools:-MersenneTwister:-MTKernelInterface(1):
  RandomTools:-MersenneTwister:-GenerateInteger(':-range' = 10^99 .. 10^100);
50796194028092318311692141733472791323758300163946408340532243655598461248436472766\
  134468592152981428

> RandomTools:-MersenneTwister:-SetState(state = s):
  RandomTools:-MersenneTwister:-GenerateInteger(':-range' = 10^99 .. 10^100);
50796194028092318311692141733472791323758300163946408340532243655598461248436472766\
  134468592152981428
```

What happens then if we use an integer of 100 decimal digits, obtained as above, as the POTP key? The answer is that, although this key has more than 300 bits, the security it provides is no higher than that offered by a 32-bit random key and this would also be the case even if a cryptographically secure PRG were used to generate the key from the automatic seed. Roughly speaking, this is due to the fact that the PRG is a deterministic algorithm that does not increase the randomness already present in its seed: it merely stretches the seed. In other words, the reason is that the key thus produced is not random (i.e., these keys are not chosen according to the uniform distribution among strings of their size) and, since we used a 31-bit seed for Mersenne Twister, the number of possible keys obtained this way is at most 2^{31} and so the system would be vulnerable to a brute-force attack. Let's give a specific example. Suppose that Eve observes the following ciphertext that Alice sent to Bob:

```
> c := "ffa93d77b3407b58b55b3d91065456ef1f59695f7ad7ade78dceaa4f0ade52242b75566379\
bc2d8cdd2cadf0ca194bef":
```

Eve knows the plaintext was encrypted with POTP and suspects that Alice and Bob chose their secret key by using Maple's randomize() to obtain a seed and then

applying `RandomTools:-MersenneTwister:-GenerateInteger` to obtain a 100-digit key. Eve checks the current value returned by `randomize()` (or, equivalently, by the absolute value of `iolib(25)`) and finds that:

```
> randomize();
                                1243047833
```

Now Eve knows that if the seed was chosen by `randomize()` it will be a value lower than this number. She also knows (Kerckhoffs' principle!) that the message is an English text and was encrypted with `POTP`, so she mounts the attack using the following function:

```
> POTPAttack := proc(ciphertext::string, startpoint::posint)
  local i, found, m, l;
  uses RandomTools:-MersenneTwister, StringTools;
  l := iquo(Length(ciphertext), 2);
  i := startpoint;
  found := false;
  while not found do
    SetState(state = i);
    m := POTP(GenerateInteger(':-range' = 10^99 .. 10^100), ciphertext, decrypt);
    if Length(m) = l then
      found := IsPrintable(m)
    end if;
    i := i-1
  end do;
  m
  end proc:
```

Note that the message is recognized by the fact that all its characters are printable in the sense that the function `StringTools:-Isprintable` returns `true` when applied to the string. Out of the 255 characters of Maple's 8-bit ASCII alphabet, only 95 have this property and these include all the alphanumeric characters and punctuation characters used in English text. This means that if the plaintext is not too short then it will be recognized with high probability as the other candidates will almost surely contain non-printable symbols. Before checking whether a candidate plaintext is printable, Eve also checks that the recovered message has the expected length. This is because Maple does not have a character corresponding to the 0 byte and the function `convert/bytes` used in `POTP` truncates the message when it finds a byte 0 in the byte list. Thus if this check is omitted, it could happen that a zero byte would appear near the beginning of the decrypted message and if all the previous characters were printable this could easily give a false positive.

Now Eve attacks the ciphertext she intercepted:

```
> POTPAttack(c, 1243047833);
            "We are challenging Eve to decrypt this plaintext"
```

The message was recovered very quickly because the key had been set up only about nine hours earlier. But it is important to notice that this attack would also have been successful—perhaps requiring more powerful resources—if the key had been established months or years earlier or if the system clock was not on time when `randomize()` was called, or even if the key was randomly chosen in $\{0,1\}^{31}$

because, in all these cases, the attack would only have to run over at most 2^{31} seeds. A similar remark applies in case Mersenne Twister is seeded by means of `SetState()`.

Exercise 3.18 Modify the function `POTPAttack` to make it output, in addition to the recovered plaintext, the key used to encrypt the message with `POTP`.

An encryption scheme such as the one implemented by means of `POTP` is usually called a *stream cipher*: encryption is done by generating a *stream* of pseudo-random bits by means of a PRG (BBS in our case) and then Xor-ing this stream with the plaintext to obtain the ciphertext (sometimes the term stream cipher is used to designate the PRG itself). As happens with the one-time pad, the cipher implemented in `POTP` is not secure if the key is reused, i.e., if multiple messages are encrypted with the same pseudo-random stream. If $c_1 = \texttt{POTP}(m_1, k)$ and $c_2 = \texttt{POTP}(m_2, k)$ then the result of Xor-ing c_1 and c_2 is the same as that of Xor-ing m_1 and m_2 and this may give the adversary a lot of information about m_1 and m_2 and even allow recovery of these messages (see, for example [64] where an attack of this kind is described in detail). A weakness of this type in an implementation of the stream cipher RC4—which is the most popular stream cipher and is regarded as reasonably secure if properly implemented—is described in [200].

Let us now briefly analyze the security of `OTP` by means of an example that may be helpful to better understand the differences with `POTP`. We know that if we use fixed-length messages and random keys that are never reused then the system has perfect secrecy (see Sect. 3.2). To illustrate the intuitive meaning of perfect secrecy, let's have a look at the problem an attacker faces when trying to break this scheme:

Example 3.7 Suppose that there are only three possible messages that Alice may send to Bob:

```
> m1 := "We will attack tomorrow at 11.55am":
  m2 := "We will surrender tomorrow at noon":
  m3 := "We will retire tomorrow at 11:55am":
```

Observe that all these messages have the same length (34 bytes):

```
> map(StringTools:-Length, [m1, m2, m3]);
                                [34, 34, 34]
```

Now, suppose that Alice encrypts message `m1` with the following 34-byte hex key:

```
> k1 := "b0f29e2253072462e85da1b4864f33ab073e42ab688705f3cb440e50a2c87eb8b9c0":
```

The ciphertext that Alice sends to Bob is then:

```
> c1 := OTP(k1, m1);
   "e797be553a6b48428929d5d5e52413df68532dd91ae872d3aa302e6193e64b8dd8ad"
```

Suppose that Eve is an enemy eavesdropper who observes this ciphertext. We may assume that Eve knows the three possible messages that Alice can send to

Bob and in fact, we may assume that she even knows the probability distribution of these messages. Why is it that, even with all this knowledge, the observation of the ciphertext gives Eve no additional information about which of the three messages is the one that was actually encrypted?

Let us compute the following 34-byte hex keys:

```
> k2 := OTP(c1, m2);
   "b0f29e2253072462fa5ca7a7804a77ba1a7359b6778700a1c5470e00e7c625e2b7c3"
> k3 := OTP(c1, m3);
   "b0f29e2253072462fb4ca1bc974133ab073e42ab688705f3cb440e50a2dc7eb8b9c0"
```

Now, observe what happens if we encrypt message `m2` with key `k2` and message `m3` with key `k3`:

```
> c2 := OTP(k2, m2);
   "e797be553a6b48428929d5d5e52413df68532dd91ae872d3aa302e6193e64b8dd8ad"
> c3 := OTP(k3, m3);
   "e797be553a6b48428929d5d5e52413df68532dd91ae872d3aa302e6193e64b8dd8ad"
```

The three ciphertexts are the same:

```
> evalb(c1 = c2 = c3);
                                    true
```

Because of the assumed uniform probability distribution on the key space, the three keys `k1`, `k2`, `k3` each have probability $\frac{1}{3}$ of having been chosen so the *a posteriori* probability of each message given that `c1` was sent is the same as its *a priori* probability of being sent. For example, if Eve knows that all three messages were *a priori* equally likely, it is intuitively clear that they remain equally likely once `c1` is observed: that `m1` was encrypted with `k1`, that `m2` was encrypted with `k2` and that `m3` was encrypted with `k3` are all equally likely events. Observe also that, in contrast with what happens with `POTP`, now a brute-force attack does not make sense even if the messages, and hence the keys, are very short. As we have just seen, Eve might be able to compute the keys that produce a given ciphertext from all possible messages but this will not give her any information about which of these messages was the one actually encrypted.

3.5 Private-Key Encryption Schemes and Their Security

In our previous study of classical cryptography we considered several ciphers of historical interest and studied their weaknesses. These ciphers are examples of encryption schemes, which were tentatively defined in the Introduction but whose definition we are going to make more precise now by making explicit some requirements that were left out in the preliminary definition and that are important for the study of the security properties of these schemes.

3.5.1 Private-Key Encryption Schemes

A private-key encryption scheme is basically the same concept defined in the Introduction but requiring explicitly that the three algorithms involved run in polynomial time and that, moreover, the key generation algorithm **Gen** is probabilistic and the encryption algorithm **Enc** may also be probabilistic. Of course, that **Gen** is probabilistic was already implicit in the previous definition which mentioned that keys were randomly chosen but none of the concrete encryption schemes discussed so far has a probabilistic encryption algorithm and, in fact, the realization that this condition is important for the security of the scheme came only around 1984, when [91] was published.

Recalling that 1^n denotes a positive integer in unary notation, the definition of private-key encryption scheme is then the following:

Definition 3.20 A *private-key encryption scheme*, also called a *symmetric encryption scheme*, is a 3-tuple $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ of polynomial-time algorithms that satisfy the following conditions:

- **Gen** is a probabilistic algorithm that takes as input a *security parameter* 1^n and outputs a key k such that $\text{len}(k) \geq n$. We denote this by $k \leftarrow \mathbf{Gen}(1^n)$ (where, as already remarked, the $\leftarrow$ notation is used for the output of a probabilistic algorithm as in [109], whose notational conventions regarding encryption schemes we generally follow).
- **Enc** is either a probabilistic or a deterministic algorithm that takes as input a key k and a plaintext m and outputs a ciphertext c, so that $c \leftarrow \mathbf{Enc}(k, m)$.
- **Dec** is a deterministic algorithm that takes as input a key k and a ciphertext c and outputs a plaintext m, which we denote $m := \mathbf{Dec}(k, c)$ (where the notation $:=$ is denoted to point to the output of a deterministic algorithm).
- For every key k output by **Gen** and every plaintext m, $\mathbf{Dec}(k, \mathbf{Enc}(k, m)) = m$.

If, for each key k output by $\mathbf{Gen}(1^n)$, **Enc** is only defined for messages of a given fixed length $\ell(n)$, then $\mathcal{E}$ is said to be a *fixed-length* symmetric encryption scheme.

Remarks 3.6

1. Let us first clarify the meaning of the requirement that **Gen**, **Enc** and **Dec** are polynomial-time algorithms. **Gen** takes as input an integer n and the requirement of being a PPT algorithm means that its running time is a polynomial function of n, since n is given in unary. Notice also that, as n will usually be equal to $\text{len}(k)$, this means that **Gen** is polynomial in the length of the output key. On the other hand, both **Enc** and **Dec** run in time polynomial on the total (binary) length of their inputs (the key and the plaintext or the ciphertext).
2. In the definition of private-key encryption scheme we have not made explicit where the plaintexts, ciphertexts, etc., are taken from. We will keep the same notation used before so the message space will be denoted by $\mathcal{M}$, the key space will be $\mathcal{K}$ and the ciphertext space $\mathcal{C}$. We will also often write $\mathbf{Enc}_k(m)$ and $\mathbf{Dec}_k(m)$ instead of $\mathbf{Enc}(k, m)$ and $\mathbf{Dec}(k, m)$, respectively.

3. The key will usually be a binary string of length n. Similarly, the message space will usually be $\mathcal{M} = \{0,1\}^*$, i.e., the set of all the finite-length binary strings. As already mentioned, another possible choice for $\mathcal{M}$ is $\{0,1\}^{\ell(n)}$ for a fixed length $\ell(n)$. Although other alphabets can be used, the assumption that the messages are binary strings is not restrictive in practice since every message can be efficiently coded in this form.
4. Although not included in the definition, it will usually be the case that **Gen** chooses the key uniformly at random in $\{0,1\}^n$. As suggested by the previous discussion, this is very important for the security of the scheme.

We shall study some important examples of encryption schemes that are currently used in practice. The classical ciphers we have considered so far may also be regarded as private-key encryption schemes once a probabilistic key-generation algorithm **Gen** is specified. However, all these schemes—with the exception of the one-time pad and the pseudo-random one time pad—are highly insecure. Moreover, all of them have a deterministic encryption algorithm and, as we shall see, probabilistic encryption is also a crucial requirement for the security of a non-perfectly secret encryption scheme.

3.5.2 Security Definitions for Private-Key Encryption Schemes

> Computational infeasibility, which by algorithmic standards is the enemy of progress, is actually the cryptographer's best friend.
>
> (Goldwasser, in [89]).

We will now try to make precise the informal notion of computational security that we have been using in this chapter. Recall that the idea is to base this concept on the theory of computational complexity and to obtain the notion of computational security from that of unconditional security by weakening the latter in two ways: on the one hand, the adversary no longer has unbounded computational power but is supposed to run in polynomial time instead and, on the other hand, we concede that the adversary might succeed with negligible probability. Moreover, we will assume for now that the adversary is an *eavesdropper* who observes the ciphertext corresponding to a single message and tries to recover it.

In order to arrive at our definition, it is necessary also to define when the adversary succeeds or, in other words, what it means to "break" the system. It is easy to identify *necessary* conditions for the system to be secure but the most obvious of these conditions are not usually sufficient. For example, if the adversary is able to recover the key, then it breaks the system because it is able to decrypt any messages encrypted with that key, but the converse is not true, i.e., if key recovery is infeasible we cannot conclude that the system is secure. As another example, it might happen that an adversary is able to recover the plaintext from the ciphertext without finding the key. But the infeasibility of plaintext recovery is still insufficient to provide security. It

is easy to imagine a situation where recovering some partial information about the plaintext—even a single bit in the most extreme cases—could be very valuable for the adversary. We see that there are many possible properties that lead to insecurity and this makes it difficult—or even impossible in practice—to characterize security in terms of the absence of these properties.

Ideally, one would like to regard an encryption scheme as secure if, given the ciphertext, the adversary has no information about the plaintext. But it is easy to see that this requirement would be too strong because often there is *a priori* information about the message: it might have some known format, it might be known that it is an English text or that it is taken from a very small set of possible messages, etc. In an extreme case, if the one-time pad was used, the adversary might even know that the plaintext is either 1 or 0 (or, for example, 'yes' or 'no'). Then the adversary would have a lot of information about the message but this would not violate the fact that the system is perfectly secret, which is the strongest form of security.

Bearing these remarks in mind, the concept of security we are aiming for is that an adversary (with limited computing power), on seeing a ciphertext, will not gain with non-negligible probability any significant partial information about the plaintext (in addition to the a priori information it might already have). This idea is formalized in the definition of *semantic security*, which was the first definition of encryption security to be proposed. But this definition is difficult to work with and we will use another definition based on the concept of *indistinguishability* which is much simpler and is, in fact, equivalent to semantic security (see also [87, 109] for more details).

The idea underlying the indistinguishability concept is to consider an adversary that chooses two messages of the same length. Then one of these messages is encrypted and the resulting ciphertext is given to the adversary. The encryption scheme is considered secure if the adversary cannot distinguish which of the two messages was the one actually encrypted.

In order to define indistinguishability (also called polynomial indistinguishability), we first define an experiment parameterized by a security parameter.

Definition 3.21 Let $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ be a private-key encryption scheme, $\mathcal{A}$ an adversary (i.e., a PPT algorithm), and n a positive integer. The *private-key eavesdropping indistinguishability experiment*, $\mathbf{PrivK}^{ind\text{-}eav}_{\mathcal{A},\mathcal{E}}(n)$, is the following:

1. The adversary $\mathcal{A}$ is given input 1^n and outputs two messages of the same length: m_0, m_1.
2. A key k is generated by running $\mathbf{Gen}(1^n)$ and a random bit b is chosen. The *challenge ciphertext* $c \leftarrow \mathbf{Enc}(k, m_b)$ is computed and given to $\mathcal{A}$.
3. $\mathcal{A}$ outputs a bit b'.
4. The output of the experiment is defined to be 1 if $b' = b$ and 0 otherwise. If $\mathbf{PrivK}^{ind\text{-}eav}_{\mathcal{A},\mathcal{E}}(n) = 1$ then we say that $\mathcal{A}$ *succeeded*.

Note that in the experiment above we require that the two messages m_0, m_1, have the same length. This is necessary because we assume that the encryption scheme can encrypt messages of arbitrary length and, usually, different length messages produce different length ciphertexts, so that information about plaintext length leaks allowing

the adversary to easily distinguish the plaintexts. Now, according to the previous remarks, the definition of security should formalize the fact that the probability that the adversary succeeds in the previous experiment is negligibly greater than $1/2$ (as succeeding with probability $1/2$ is easy just by outputting a random bit). The definition is then the following:

Definition 3.22 A private-key encryption scheme $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ has *indistinguishable encryptions in the presence of an eavesdropper* if, for every PPT adversary $\mathcal{A}$, there exists a negligible function $\mathtt{negl}$ such that:

$$\Pr(\mathbf{PrivK}^{ind\text{-}eav}_{\mathcal{A},\mathcal{E}}(n) = 1) \leq 1/2 + \mathtt{negl}(n),$$

where the probability is taken over the random bits used by $\mathcal{A}$, as well as the random bits used elsewhere in the experiment (i.e., when running **Gen**, when choosing a random bit b and when running **Enc** in case this algorithm is probabilistic).

Remarks 3.7

1. As mentioned above, this definition does not seem, at first sight, to formalize the idea that it should be infeasible for a polynomially bounded adversary to learn any partial information about the plaintext from the ciphertext. However, it can be shown that the formalization of this intuitive idea, namely *semantic security*, is actually equivalent to the above definition based on indistinguishability. Intuitively, this is clear if the messages to be encrypted consist of a single bit, and this basic idea can be used to show that indistinguishability implies that no single bit of a randomly chosen plaintext can be guessed with probability significantly better than $1/2$ (see [109, Claim 3.10]). This is not yet semantic security but is a step in this direction. See [109, 3.2.2] for a discussion of this and [87] for a proof of the equivalence between the two definitions.
2. The quantity $|\Pr(\mathbf{PrivK}^{ind\text{-}eav}_{\mathcal{A},\mathcal{E}}(n) = 1) - 1/2|$ is often called the *advantage* of the adversary $\mathcal{A}$. The absolute value in the definition makes sense because it ensures that the advantage is always non-negative. Then it can be said that the system has indistinguishable encryptions when the advantage of any PPT adversary is negligible as a function of the security parameter. This is clear because if $\Pr(\mathbf{PrivK}^{ind\text{-}eav}_{\mathcal{A},\mathcal{E}}(n) = 1) < 1/2$ and $\mathcal{A}$ has non-negligible advantage, then a new adversary $\mathcal{A}'$ may be constructed by reversing the output of $\mathcal{A}$, so that $\Pr(\mathbf{PrivK}^{ind\text{-}eav}_{\mathcal{A}',\mathcal{E}}(n) = 1) > 1/2$ and $\mathcal{A}'$ has non-negligible advantage too.
3. It is customary to use the notation XXX-YYY to denote a security notion, where XXX refers to the adversary goal—usually, the adversary will try to break indistinguishability—and YYY to its capabilities. With this convention, indistinguishability to eavesdroppers is often denoted IND-EAV.

Exercise 3.19 prove that the one-time pad has indistinguishable encryptions in the presence of an eavesdropper. More generally, prove that perfect secrecy for an encryption scheme $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ can be characterized in terms of an experiment $\mathbf{PrivK}^{ind\text{-}eav}_{\mathcal{A},\mathcal{E}}$ similar to the one in Definition 3.21 except for the fact that now the

adversary $\mathcal{A}$ is any algorithm (which does not require any input to output two equal-length messages, so the security parameter does not appear here). Show that $\mathcal{E}$ is perfectly secret if and only if $\Pr(\mathbf{PrivK}^{ind\text{-}eav}_{\mathcal{A},\mathcal{E}} = 1) = 1/2$.

We have given an informal argument that the fixed-length pseudo-random one-time pad presented in Definition 3.5 is computationally secure in the intuitive sense mentioned in Remark 3.3. The formal version of this security concept is IND-EAV security as defined in Definition 3.22. Thus we can now give a more precise formulation of this result:

Theorem 3.11 *The pseudo-random one-time pad of Definition 3.5 is a fixed-length private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper.*

Sketch of proof. Observe that if *POTP* denotes the fixed-length pseudo-random one-time pad then, as the output of the PRG used in *POTP* is, to a PPT algorithm $\mathcal{A}$, indistinguishable from a truly random string, from the point of view of such an $\mathcal{A}$, *POTP* behaves exactly as the one-time pad. By Exercise 3.19, the one-time pad has indistinguishable encryptions in the presence of an eavesdropper. Therefore such an $\mathcal{A}$ should be unable to succeed with non-negligible probability in the experiment $\mathbf{PrivK}^{ind\text{-}eav}_{\mathcal{A},POTP}(n)$. We refer to [109, Theorem 3.16] for a detailed proof.

Corollary 3.1 *The BBS-based pseudo-random one-time pad is IND-EAV secure under the factoring assumption.*

Proof It is a consequence of the preceding theorem and Theorem 3.10. □

The preceding discussion of security refers to the situation in which the eavesdropper is given a unique ciphertext and tries to guess to which of two messages it corresponds. But in practice, an encryption scheme will often be used for multiple encryptions (with the same key). This situation is completely different regarding security for, as we have seen, both the one-time pad and the pseudo-random one-time pad are secure only when the key is not reused. More generally, if an eavesdropper observes a repeated ciphertext, it can guess that a message has been resent and this in itself is a significant leak of information that may be—and there are historical examples of this—useful to an attacker. In some cases this could lead to a more important information leakage. A very simple scenario in which this situation occurs is the following. Suppose that a ciphertext is sent. It is not infrequent that the eavesdropper may learn a posteriori—by whatever means—information about the plaintext. Then, if the same message is later resent encrypted by means of a deterministic algorithm, the eavesdropper gains information about the new message.

We may now define the concept of security for multiple encryptions. We keep the same notation as in Definition 3.21.

Definition 3.23 The *private-key eavesdropping multiple-encryption indistinguishability experiment* $\mathbf{PrivK}^{mult\text{-}ind\text{-}eav}_{\mathcal{A},\mathcal{E}}(n)$, for a private-key encryption scheme

$\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ is the following:

1. The adversary $\mathcal{A}$ is given input 1^n and outputs, for a positive integer t, two t-tuples of messages $(m_{01}, \ldots, m_{0t})$, $(m_{11}, \ldots, m_{1t})$, where $\text{len}(m_{0i}) = \text{len}(m_{1i})$ for all $i = 1 \ldots t$.
2. A key k is generated by running $\mathbf{Gen}(1^n)$ and a random bit b is chosen. For each $i = 1 \ldots t$ the *challenge ciphertexts* $c_i \leftarrow \mathbf{Enc}(k, m_{bi})$ are computed and given to $\mathcal{A}$.
3. $\mathcal{A}$ outputs a bit b'.
4. The output of the experiment is defined to be 1 if $b' = b$ and 0 otherwise. If $\mathbf{PrivK}^{\textit{mult-ind-eav}}_{\mathcal{A},\mathcal{E}}(n) = 1$ then we say that $\mathcal{A}$ *succeeded.*

Then the encryption scheme $\mathcal{E}$ is said to have *indistinguishable multiple encryptions in the presence of an eavesdropper* if, for every PPT adversary $\mathcal{A}$, there exists a negligible function `negl` such that

$$\Pr(\mathbf{PrivK}^{\textit{mult-ind-eav}}_{\mathcal{A},\mathcal{E}}(n) = 1) \leq 1/2 + \texttt{negl}(n),$$

where the probability is taken over the random bits used by $\mathcal{A}$, as well as the random bits used elsewhere in the experiment.

Remarks 3.8

1. By the preceding remarks, *POTP* has indistinguishable encryptions but not indistinguishable multiple encryptions in the presence of an eavesdropper. But the fact that indistinguishable multiple encryption is a much stronger concept that indistinguishable single encryption may be better appreciated by noting that no encryption scheme in which **Enc** is a *deterministic function of the key and the message alone* can have indistinguishable multiple encryptions. Indeed, it suffices to consider an adversary $\mathcal{A}$ who outputs the two following ordered pairs of messages: $(m_{01} = 0^n, m_{02} = 0^n)$, $(m_{11} = 0^n, m_{12} = 1^n)$ (or any other such ordered pairs, in which the messages of the first pair are equal and the messages of the second pair are different). Then if $\mathcal{A}$ receives the ciphertexts (c_1, c_2) corresponding to the encryption of the messages of one of these pairs, $\mathcal{A}$ just checks whether $c_1 = c_2$. If the answer is affirmative $\mathcal{A}$ outputs 0, otherwise $\mathcal{A}$ outputs 1. Then, because **Enc** is deterministic, $\mathcal{A}$ succeeds with probability 1.
2. Observe that if **Enc** is a probabilistic algorithm then the argument above breaks down and $\mathcal{A}$ does not succeed with probability 1 because even if $m_{01} = m_{02}$, the corresponding ciphertexts can be (and, with high probability, should be) different.
3. Note also that the first of these remarks allows the possibility that an encryption scheme with a deterministic encryption algorithm might have multiple indistinguishable encryptions. For this it is necessary that the encryption algorithm be *stateful*, meaning that, apart from the key and the message, there is an additional input called the *state* which is initialized in some specified way and updated after each encryption operation, so that both parties using the scheme maintain the same value of the state (usually a *counter* that increases by one unit after

each encryption). In such a scheme, the same plaintext would produce a different ciphertext each time it is encrypted due to the different values of the state, so that the above argument does not apply. We remark that the encryption schemes introduced in Definition 3.20 are *stateless* (i.e., not stateful) and we will usually work with stateless schemes.

The fact that deterministic encryption does not allow indistinguishable multiple encryptions has enormous importance in practice since it means that to encrypt multiple messages, only probabilistic or stateful encryption should be used. We record this fact as follows:

Proposition 3.2 *If* $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ *is an encryption scheme such that* **Enc** *is a deterministic function of the key and the message, then* $\mathcal{E}$ *does not have multiple indistinguishable encryptions in the presence of an eavesdropper.*

3.5.3 CPA Security and CCA Security

In the previous discussion of security we have only considered a weak adversary eavesdropping on the communication between the parties using the encryption scheme. The eavesdropper is a passive attacker except for the fact that it can choose the two messages it will try to distinguish. Since it is only given one ciphertext whose corresponding plaintext it does not know, such an attack qualifies as a "ciphertext-only" attack.

We are now going to consider a much more powerful adversary $\mathcal{A}$, namely one that can mount a *chosen plaintext attack* (CPA). This adversary will be able to adaptively choose multiple messages and receive the corresponding ciphertexts (this is also called an *adaptive chosen plaintext attack* because the adversary can choose each message as a function of the earlier answers). This type of attack is formalized by allowing the adversary $\mathcal{A}$ to interact with an *encryption oracle* which acts as a "black box" that responds to $\mathcal{A}$'s queries by returning encryptions of the messages that $\mathcal{A}$ submits (always with a key k unknown to $\mathcal{A}$). This interaction continues even after $\mathcal{A}$ has received the challenge ciphertext corresponding to one of a pair of messages of the same length chosen by $\mathcal{A}$. The idea of the next definition is then that $\mathcal{A}$ should not be able to distinguish which of these two messages was encrypted even when it is given free access to the encryption oracle.

Definition 3.24 We define the *private-key CPA indistinguishability experiment* $\mathbf{PrivK}^{\textit{ind-cpa}}_{\mathcal{A},\mathcal{E}}(n)$, where $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ is a private-key encryption scheme, $\mathcal{A}$ a PPT adversary, and n any value of the security parameter, as follows:

1. A key $k \leftarrow \mathbf{Gen}(1^n)$ is generated. $\mathcal{A}$ is given input 1^n and oracle access to $\mathbf{Enc}(k, -)$. Then $\mathcal{A}$ outputs a pair of messages of the same length: m_0, m_1.
2. A random bit b is chosen and the *challenge ciphertext* $c \leftarrow \mathbf{Enc}(k, m_b)$ is computed and given to $\mathcal{A}$.

3. $\mathcal{A}$ continues having oracle access to $\mathbf{Enc}(k, -)$ and, after obtaining encryptions of several messages $\mathcal{A}$ chooses, outputs a bit b'.
4. The output of the experiment is defined to be 1 if $b' = b$ and 0 otherwise. If $\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathcal{E}}(n) = 1$ then we say that $\mathcal{A}$ *succeeded*.

Then the encryption scheme $\mathcal{E}$ is said to have *indistinguishable encryptions under a chosen plaintext attack* (or that it is IND-CPA secure or, briefly, CPA secure) if, for every PPT adversary $\mathcal{A}$, there exists a negligible function $\mathtt{negl}$ such that

$$\Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathcal{E}}(n) = 1) \leq 1/2 + \mathtt{negl}(n),$$

where the probability is taken over the random bits used by $\mathcal{A}$, as well as the random bits used elsewhere in the experiment.

The following result is a straightforward consequence of the definitions:

Proposition 3.3 *If a symmetric encryption scheme has indistinguishable encryptions under a chosen plaintext attack then it also has indistinguishable encryptions in the presence of an eavesdropper.*

Another straightforward consequence of the definitions is the following

Proposition 3.4 *Let* $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ *be a symmetric encryption scheme such that* **Enc** *is a deterministic function of the key and the message. Then* $\mathcal{E}$ *is not CPA secure.*

Proof It suffices to observe that, in experiment $\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathcal{E}}(n)$, after receiving the challenge ciphertext $c \leftarrow \mathbf{Enc}(k, m_b)$, $\mathcal{A}$ can query the oracle about the messages m_0, m_1. The oracle will respond by providing $\mathcal{A}$ with $c_0 := \mathbf{Enc}(k, m_0)$ and $c_1 := \mathbf{Enc}(k, m_1)$ and then $\mathcal{A}$ may check whether $c = c_0$ or $c = c_1$. Since the encryption function is deterministic, one of these two possibilities must hold and $\mathcal{A}$ outputs $b = i$ where $c = c_i$, succeeding with probability 1. □

Remarks 3.9

1. As happens with multiple encryptions in the presence of an eavesdropper, note that also here the encryption scheme $\mathcal{E}$ might be CPA secure without being probabilistic in case it is stateful, but this possibility is excluded in the statement of the previous proposition, as we require there that **Enc** is a deterministic function of the key and the message only.
2. In a way similar to eavesdropper indistinguishability, $|\Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathcal{E}}(n) = 1) - 1/2|$ is called the CPA advantage of the adversary $\mathcal{A}$, so that an encryption scheme is CPA secure if any PPT adversary with oracle access to encryptions has only negligible CPA advantage. In the previous proposition we have seen that if encryption is a deterministic function of the key and the message, then the CPA advantage of the adversary is equal to 1/2.
3. As in the case of eavesdropping adversaries one might define CPA security for multiple encryptions just by letting the adversary $\mathcal{A}$ output a pair of sequences

of messages to be distinguished instead of just two messages. But there is not much point in doing this because, as it can easily be seen, CPA security as above defined already implies CPA security for multiple encryptions. In particular, we also see that CPA security implies indistinguishable multiple encryptions in the presence of an eavesdropper.

4. It might seem at first sight that the notion of CPA security is too strong as it is hardly conceivable that honest parties will provide an adversary $\mathcal{A}$ with encryptions of the messages that $\mathcal{A}$ chooses. But there are many indirect ways in which $\mathcal{A}$ could obtain information about these encryptions without the voluntary cooperation of the honest parties and this information might be sufficient to make the probability of a successful attack non-negligible. For example, in [108, pp. 566–573] a historical episode from World War II that included such an attack is mentioned. In May, 1942, American cryptanalysts had information about an upcoming Japanese attack in the Pacific but there was some disagreement about what the objective might be. Some believed that Midway island was the target but the Chief of Naval Operations in Washington concluded that it was Oahu instead. The cryptanalysts learned that the target coordinates were represented by AF in Japanese ciphertexts and, to convince their top chief that this really meant Midway, they sent an unencrypted message in which Midway reported that its fresh-water distillation plant had broken down. Two days later they intercepted a Japanese ciphertext stating that AF was short of fresh water. Thus the Japanese acted as an unwilling encryption oracle that revealed the ciphertext corresponding to the plaintext 'Midway' and the fact that their encryption scheme was not CPA secure allowed the Americans to learn information which was decisive in winning the Midway battle.
5. CPA security behaves as expected in relation to other security properties. For example, CPA security implies security against CPA plaintext recovery, i.e., a PPT adversary with oracle access to encryption cannot recover the plaintext from the ciphertext with non-negligibly probability. Indeed, if $\mathcal{A}'$ is an adversary able to recover the plaintext corresponding to the challenge ciphertext in $\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A}',\mathcal{E}}(n)$, then there exists an adversary $\mathcal{A}$ with the ability to decide which of m_0, m_1 was encrypted. $\mathcal{A}$ runs $\mathcal{A}'$ as a subroutine that provides the plaintext corresponding to the challenge ciphertext and then makes its guess by comparing this plaintext to m_0 and m_1. Moreover, as we shall soon see, the most important symmetric encryption schemes in use today can be shown to be CPA secure under reasonable hypotheses. For these reasons, CPA security is often regarded as the "right" definition of security for private-key encryption schemes.

Before ending this quick review of security properties for symmetric encryption schemes, we are going to mention a more powerful type of attack: chosen ciphertext attacks (CCA). In this case, in addition to access to an encryption oracle, the adversary will also have access to a decryption oracle, with the sole exception that the oracle cannot be queried about the challenge ciphertext. The definition is the following:

Definition 3.25 We define the *private-key CCA indistinguishability experiment* $\mathbf{PrivK}^{ind\text{-}cca}_{\mathcal{A},\mathcal{E}}(n)$, where $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ is a private-key encryption scheme, $\mathcal{A}$ a PPT adversary, and n any value of the security parameter, as follows:

1. A key $k \leftarrow \mathbf{Gen}(1^n)$ is generated. $\mathcal{A}$ is given input 1^n and oracle access to both $\mathbf{Enc}(k, -)$ and $\mathbf{Dec}(k, -)$. Then $\mathcal{A}$ outputs a pair of messages of the same length: m_0, m_1.
2. A random bit b is chosen and the *challenge ciphertext* $c \leftarrow \mathbf{Enc}(k, m_b)$ is computed and given to $\mathcal{A}$.
3. $\mathcal{A}$ continues having oracle access to $\mathbf{Enc}(k, -)$ and $\mathbf{Dec}(k, -)$ but is not allowed to query the latter on the challenge ciphertext. Afterwards $\mathcal{A}$ outputs a bit b'.
4. The output of the experiment is defined to be 1 if $b' = b$ and 0 otherwise. If $\mathbf{PrivK}^{ind\text{-}cca}_{\mathcal{A},\mathcal{E}}(n) = 1$ then we say that $\mathcal{A}$ *succeeded*.

The encryption scheme $\mathcal{E}$ is said to have *indistinguishable encryptions under a chosen ciphertext attack* (or that it is IND-CCA secure, or CCA secure) if, for every PPT adversary $\mathcal{A}$, there exists a negligible function `negl` such that

$$\Pr(\mathbf{PrivK}^{ind\text{-}cca}_{\mathcal{A},\mathcal{E}}(n) = 1) \leq 1/2 + \mathtt{negl}(n),$$

where the probability is taken over the random bits used by $\mathcal{A}$, as well as the random bits used elsewhere in the experiment.

Remarks 3.10

1. Note that, in the experiment above, $\mathcal{A}$ has free access to the decryption oracle (i.e., temporary access to the decryption machine as a 'black box') except that it cannot request the decryption of the challenge ciphertext. Of course, if this exception were not included, then no encryption scheme would satisfy the definition.
2. Although this security definition is much stronger than the preceding ones there are, again, plausible scenarios where such an attack might be implemented in practice. For example, if authentication is not used, it might be possible for an adversary to impersonate one of the honest parties and receive decryptions that might help in the decryption of further messages. This situation is more likely to occur in a public-key setting where one party may communicate with multiple parties with which there may not have been previous contact. In fact, as we will see later, one such attack, described a few years ago, forced a change in a public-key encryption scheme used by web browsers.
3. CCA security is much stronger than CPA security and, in particular, it implies the *non-malleability property* which means that an adversary cannot modify a given ciphertext in order to obtain another legal ciphertext whose corresponding plaintext is related to the plaintext of the first one. The stronger requirements make CCA security harder to achieve but we will later indicate how a CCA secure scheme may be built; in view of the previous remark it is not surprising that this construction involves authentication in addition to encryption.
4. The idea to prove that CCA security implies non-malleability is the following. Suppose that the latter property does not hold, i.e., that the encryption scheme

is malleable. Then a CCA adversary, when given the challenge ciphertext c, can modify it and obtain a related one, say c' (note that we are not defining 'related', as we are only trying to give an intuitive idea of the argument). Then the adversary can query the encryption oracle about c' (because $c' \neq c$) obtaining the corresponding plaintext m'. By the malleability property, we assume that the existing relationship between c and c' corresponds to a known relationship between the corresponding plaintexts m_b and m'. Then the adversary uses the knowledge of m' and this relationship to decide which of m_0, m_1, is equal to m_b.

3.5.4 Concluding Remarks

In this chapter we have seen that complexity and randomness (together with its computational analogue, pseudo-randomness) are two fundamental concepts that play a crucial role in relation to encryption schemes, and in fact they are some of the most important tools underlying modern cryptography. The very definition of security involves these two concepts when we require that it should be too difficult for an adversary to break the system with non-negligible probability. Other relevant points related to randomness are that this concept is essential to generate keys, and also for encryption algorithms, which should be probabilistic since otherwise an encryption scheme cannot be CPA secure.

The fact that the precise definitions of security are based on computational complexity leads to security proofs that, while being only conditional so far are, nevertheless, very important because they allow the establishment of precise connections between security and the hardness of well-studied computational problems. To analyze this point of view in more detail, consider the following two quotes by well-known authors:

> ... traditionally, all cryptosystems were eventually broken. No guarantee was made as to the security of the systems, beyond "we tried and could not break it for a while."
>
> (Goldwasser in [88]),

> ... there is only *historical* evidence that factorization is an intrinsically hard problem. Generations of number theorists, a small army of computer scientists, and legions of cryptologists spent a considerable amount of energy on it, and the best they came up with are relatively poor algorithms.
>
> (Lenstra and Lenstra in [127]).

There is apparently some tension between these two remarks. Goldwasser emphasizes the importance of having a precise definition of security by pointing out the shortcomings of assuming that a cryptographic scheme is secure because no one was able to break it so far, yet the Lenstra brothers point out that the factorization problem, on which the security of the most popular public-key scheme rests, is only regarded as hard because no one was able to solve it efficiently so far.

However, both remarks are consistent with today's view of security in modern cryptography. Classical ciphers were all eventually broken and often the break happened when their security was held in the highest regard. But cryptanalysis was,

before the second half of the twentieth century, an obscure art practised by very few people who used only their ingenuity to develop themselves the "ad-hoc" methods they used. In contrast, cryptography—or cryptology, since cryptanalysis is an indispensable complement—is now largely a science and, when a problem of cryptologic relevance such as, for example, the integer factorization problem, is considered hard, it is because the scientific community (formed by mathematicians and computer scientists in this case) supports this idea. Therefore, having the security of a system precisely defined and related to a specific and well-understood problem is a big advantage. For example, we will see in Chap. 7 that the security of RSA may be related to the factorization problem and, even if we have not proved that factoring is hard and, moreover, the presumed hardness of factoring is not known to be sufficient for the security of RSA, the situation is much more satisfactory than if this relation were not known and we worked only with the RSA function.

In this chapter we have already discussed a reduction of this type, namely the two-step reduction that shows that the BBS-based pseudo-random one-time pad is IND-EAV secure under the factoring assumption. First, the problem is reduced to showing that BBS is a PRG (in the strong cryptographic sense), and then the latter problem is reduced to the factoring assumption. The reduction is tighter than in the case of RSA mentioned above, because now the factoring assumption is not just necessary but also sufficient for the IND-EAV security of the scheme. Although we do not have a guarantee that the factorization problem is hard, we are on firmer ground that if we only worked under the generic assumption that a pseudo-random generator exists. In Chap. 8 we show that CCA security (which is much stronger than IND-EAV security) can also be attained for a public-key encryption scheme assuming the hardness of a well-known computational problem.

Summing up these remarks, the security reductions that show that a scheme satisfies a precise security definition if some computational problem is hard, are—despite some limitations discussed, for example, in [120]—useful because they place the question of the security of a system on a better foundation. It would be nice to have proofs of security not relying on unproven assumptions but this is a very difficult task due to the inherent difficulty of proving lower bounds for the computational complexity of a problem. This leads to the somewhat paradoxical situation that, while there are encryption schemes that are proven to be secure in the strongest sense (the one-time pad has perfect secrecy), the existence of non-unconditionally secure encryption schemes that are secure in the weaker sense, i.e., computationally secure, has not been proved so far. Finally, we remark that in this chapter we have only given a short introduction to the most important concepts that serve to establish the foundations of cryptography and we have not given full proofs of several of the results mentioned. For the reader interested in the theoretical foundations of cryptography, an excellent introduction is [109], while [86, 87] are more advanced references that go deep on all the subjects discussed in this chapter.

Chapter 4
Block Ciphers and Modes of Operation

Most modern private-key encryption schemes are built from two ingredients: a block cipher and a mode of operation. Block ciphers are cryptographic primitives that provide encryption and decryption functions which operate on strings of a fixed size (blocks). The modes of operation act on top of block ciphers and allow the encryption of arbitrary length messages and also provide the probabilistic encryption necessary for the system to be secure. In this chapter we are going to look at these constructions and, in particular, at the most used block cipher today: the Advanced Encryption Standard.

4.1 Block Ciphers and Pseudo-Random Functions

We have already seen how block ciphers can be used as encryption schemes, such as is the case, for example, with the Hill cipher. Here we are going to study block ciphers from a more general point of view which regards them not as encryption schemes but as a kind of *cryptographic primitives* that can be used to build these schemes.

Definition 4.1 A *block cipher* is a function $F : \{0,1\}^t \times \{0,1\}^n \to \{0,1\}^n$, where, for each $k \in \{0,1\}^t$ the induced function $F_k : \{0,1\}^n \to \{0,1\}^n$ given by $F_{k(x)} = F(k,x)$ is a permutation.

Remarks 4.1

1. In the intended use of the definition above, $\{0,1\}^t$ may be thought of as the key space and the elements of $\{0,1\}^n$ (blocks) as the plaintexts and ciphertexts so that, in encryption schemes based on block ciphers, the F_k will act as the encryption functions. The parameter n is called the *block length* of F.
2. $\{0,1\}$ could be replaced by any alphabet Σ as when we implemented the Hill cipher using an alphabet consisting of English letters plus the space. As we will see below, the most important block cipher currently used, AES, can be most

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_4, © Springer-Verlag Berlin Heidelberg 2013

naturally regarded as defined over the alphabet consisting of bytes (i.e., 8-bit strings). For this reason the (binary) block length, as well as the key length, must be multiples of 8 in this case but, otherwise, this is just a trivial matter of re-scaling.

It is clear that the definition of block cipher given above is too general for cryptographic use. As an extreme example, one could define a block cipher by setting $F_k(x) = x$ for all $k \in \{0,1\}^t$ and it is obvious that an encryption scheme based on it would not be too secure. But even if we require the map $k \mapsto F_k$ to be injective, we are still far from anything resembling reasonable security. For example, if we look at the Hill cipher and its cryptanalysis, we immediately see that the reason it was so easily broken is that the maps F_k are linear. Thus they are uniquely determined by their action on a basis of the vector space formed by the blocks (if, as in the example we gave, the alphabet is not a field, the blocks do not form a vector space in the strict sense of the term but they form a free module, which also has a basis, and the linear algebra works the same). A basis is a small subset of the message space (it has only n members where the message space has 2^n or, more generally, $|\Sigma|^n$ members) and a polynomially bounded adversary can recover the key and hence completely break the system by just mounting a chosen plaintext attack (even a known plaintext attack may often be sufficient) on this subset.

In view of this example, we want to use block ciphers where the permutations F_k are far from structured, i.e., we would like that these maps would be random or, to express it a little more precisely, that the F_k corresponding to randomly chosen $k \in \{0,1\}^t$ behave like randomly chosen permutations of $\{0,1\}^n$. This could be accomplished by taking the family $\{F_k | k \in \{0,1\}^t\}$ equal to the entire symmetric group of $\{0,1\}^n$ and taking the key space large enough so that it can be bijectively mapped to this symmetric group (thus, randomly choosing a key would be equivalent to randomly choosing a permutation of $\{0,1\}^n$). Strictly speaking this would not fit the previous definition since the number of permutations of $\{0,1\}^n$ is not a power of 2 and hence does not correspond bijectively to the set $\{0,1\}^t$ for a given value of t, but this is only a minor technical matter as the key space could be, more generally, a subset of $\{0,1\}^t$.

This construction may be regarded as an attempt to obtain maximum security through randomness (as in the case of perfect secrecy) but here things are a little different, for even with these random permutations as encryption functions we would not necessarily obtain an encryption scheme with indistinguishable encryptions in the presence of an eavesdropper—not to mention the fact that an encryption scheme that directly uses these permutations to encrypt cannot be CPA secure because the encryption functions are deterministic. Indeed, we have seen how substitution ciphers are highly vulnerable to ciphertext-only attacks—by means of frequency analysis—and substitution ciphers are nothing but block ciphers with block length 1 (we used the English alphabet in our example and, since the number of symbols was less than 2^5, these symbols could have been encoded as binary strings in $\{0,1\}^5$, so this particular cipher can be regarded as a block cipher of length 5 over the binary alphabet). Moreover, note that for this attack to be successful it is not necessary to make any special assumption about the key, i.e., the permutation of the alphabet,

which may well have been chosen at random. This extreme example shows that block length must be taken into account when analyzing the security of a block cipher which is naively used as an encryption scheme. In fact, as we will see later on, even using a mode of operation which provides a CPA secure scheme, block length must be taken into account for practical security purposes and a 64-bit block length is regarded as insufficient to provide this practical security (see Remarks 4.3). For this reason, most modern encryption schemes based on block ciphers use, as a minimum, a 128-bit block length. So, returning to our ideal block cipher in which all the permutations in $\{0,1\}^n$ would appear among the F_k, let us see what this would mean for a 128-bit block length.

The order of the symmetric group $S(\{0,1\}^n)$ is $2^n!$. When $n = 128$, this number is so huge that we cannot write it down in positional notation, not even using Maple. To understand why, we will compute an approximation of the binary logarithm of this number. This is highly meaningful for our purposes because if we were to enumerate all these permutations, we could use the binary form of the corresponding numbers as keys and so this binary logarithm would give the length of the binary strings representing keys in this case. We will use Stirling's formula, $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ ([94]) which, in this case, gives the following approximation: $2^n! \approx \sqrt{2\pi 2^n}\left(\frac{2^n}{e}\right)^{2^n}$. This, in turn, gives for the binary logarithm:

$$\log_2(2^n!) \approx \log_2(\sqrt{2\pi}) + \frac{n}{2} + 2^n(n - \log_2(e)) \approx 2^n(n - 1.44).$$

Thus we see that the use of random permutations implies that the key length grows exponentially as a function of the block length. Now we use Maple to compute this value for $n = 128$:

```
> 2^128*(128-1.44);
                              4.306613635*10^40
```

This number would be the length of the binary strings used to code keys for this scheme. To better appreciate the enormous amount of storage that a single key would require, let us use large information storage units. In SI units of information, we have the following equivalences:

1 petabyte = 10^3 terabytes = 10^6 gigabytes = 10^9 megabytes = 10^{15} bytes

Then one key for the scheme would require:

```
> 4.306613635*10^40/(8*10^15);
                              5.383267044*10^24
```

petabytes of storage. For comparison, K. Kelly in an article published in 2006 in *The New York Times* estimated that, *"the entire works of humankind, from the beginning of recorded history, in all languages"* would amount to 50 petabytes of data in compressed form. So it is clear that these keys would be completely unmanageable.

Exercise 4.1 Check the binary analogues of the above-mentioned information units and compute the approximate number of *yobibytes* that would be required to store a random permutation of $\{0,1\}^{128}$.

As happens with the one-time pad, true randomness makes the system unmanageable and, indeed, even more so in this case. However, as we have also seen, complexity theory can be used to make randomness more manageable just by replacing it by pseudo-randomness. Now we are dealing with (random) functions instead of bits but we can use the same intuitions that lead to the definition of PRGs. A block cipher $F : \{0,1\}^t \times \{0,1\}^n \to \{0,1\}^n$ induces a probability distribution on the symmetric group $S(\{0,1\}^n)$ by choosing a random key $k \in \{0,1\}^t$ and the associated permutation $F_k \in S(\{0,1\}^n)$. The idea is then to call F pseudo-random if the permutation F_k corresponding to a randomly chosen $k \in \{0,1\}^t$ is indistinguishable to a PPT distinguisher D from a permutation f chosen uniformly at random from $S(\{0,1\}^n)$. The distinguisher D is a PPT algorithm and hence it cannot have access to a full description of a random permutation, which has size exponential in n. Thus D is given oracle access to the function it is trying to distinguish (which can be either F_k or f). The oracle is like a black box that will answer D's queries without giving any additional information. D can ask only a polynomial number of queries (since it is a PPT algorithm) and each query is about the value that the function takes at a point $x \in \{0,1\}^n$ that is chosen by D. Moreover, D can ask these queries adaptively, i.e., D may choose each query as a function of the earlier oracle answers. We do not need to know how the oracle works but we can imagine how an oracle could efficiently answer D's queries even in case they are about a random permutation. In this case, if queried about $x \in \{0,1\}^n$, the oracle would choose a random element $y \leftarrow \{0,1\}^n$ as its answer and would record in a table the input/output pair (x, y). In each subsequent query, the oracle would check first whether the queried input is already the first component of a table entry and, if so, it would give the same answer as in the preceding query (to ensure that it is implementing a function); otherwise, it would choose its answer at random among all elements of $\{0,1\}^n$ which do not appear as the second component of a table entry (to ensure that it is implementing a permutation). Then, after answering the query, it would record the corresponding input/output pair in the table.

The block cipher uses a fixed value of n but, in order to give a precise definition of when F is pseudo-random we have to use the asymptotic concept of negligible function and, to make it meaningful, we may let n vary by considering, more generally, a *keyed function* F, namely a function $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ (where the first input is called the key) which is length-preserving, i.e., such that $\mathrm{len}(k) = \mathrm{len}(x) = \mathrm{len}(F(k,x))$ for all $x \in \{0,1\}^*$ (thus we assume, for simplicity, that the length of the key is the same as the length of the input x and the length of the output $F(k,x)$, although this is not essential). Then, for each $k \in \{0,1\}^n$, F defines a function $F_k : \{0,1\}^n \to \{0,1\}^n$ given by $F_k(x) = F(k,x)$. If, moreover, the function F_k is a permutation of $\{0,1\}^n$ for each $k \in \{0,1\}^n$, we say that F is a *keyed permutation*. Also, in what follows we will always assume that F is an effi-

cient function in the sense that there exists a deterministic algorithm that, on input $k, x \in \{0,1\}^n$, computes $F_k(x)$ in time polynomial in 1^n.

We can now define pseudo-random functions (and pseudo-random permutations) with the help of PPT distinguishers D that output 0 or 1 according to whether or not a permutation looks random to them:

Definition 4.2 The *pseudo-random function experiment* $\mathbf{Prf}_{D,F}(n)$, where F is a length-preserving keyed function, D a PPT algorithm, and n any value of the security parameter, is the following:

1. On input 1^n, a bit $b \in \{0,1\}$ is chosen and a function $f \in (\{0,1\}^n)^{\{0,1\}^n}$ (where we use the notation Y^X for the set of functions from X to Y) is selected as follows:
   - If $b = 0$ then $f \leftarrow (\{0,1\}^n)^{\{0,1\}^n}$ (i.e., f is randomly chosen in the set of functions that map n-bit strings to n-bit strings).
   - If $b = 1$ then $k \leftarrow \{0,1\}^n$, $f := F_k$.

   D is then given input 1^n and oracle access to f.
2. D adaptively makes a polynomial number of queries to the oracle about $f(x)$, for the values $x \in \{0,1\}^n$ it chooses.
3. D outputs a bit b'.
4. The advantage of D in the experiment is defined as $Adv^{prf}_{D,F}(n) = |\Pr(b' = 1|b = 1) - \Pr(b' = 1|b = 0)|$.

The length-preserving keyed function F is said to be a *pseudo-random function* if, for any PPT algorithm D, there is a negligible function $\mathtt{negl}(n)$ such that $Adv^{prf}_{D,F}(n) \leq \mathtt{negl}(n)$. The experiment $\mathbf{Prp}_{D,F}(n)$ is defined for an efficient length-preserving keyed permutation F in an entirely similar way, with the only difference that when $b = 0$, then $f \leftarrow S(\{0,1\}^n)$ is a randomly chosen permutation. If $Adv^{prp}_{D,F}(n)$ is negligible, then F is said to be a *pseudo-random permutation*.

Remarks 4.2

1. When F is a block cipher, F may be seen as the result of letting a keyed permutation act on keys and inputs of fixed length n. In this case it does not make much sense to say that the advantage of D is negligible but, for practical purposes, we will understand this as meaning that the advantage is very small for this value of n, so that no PPT distinguisher would be able, in practice, to succeed in the above experiment. In this case we can say that the block cipher is modeled as a pseudo-random permutation or, simply, that it is a pseudo-random permutation.
2. If F is a pseudo-random permutation, then we may wonder whether distinguishing it from a random function is any easier than distinguishing it from a random permutation (or even if there are pseudo-random permutations that are not pseudo-random functions). When n is small, this is indeed the case, for an adversary able to make about $1.18 \cdot 2^{n/2}$ queries has a probability $\geq 1/2$ of finding two values $x \neq y$ such that $f(x) = f(y)$ (a collision for f) in case f is a randomly chosen function $f : \{0,1\}^n \rightarrow \{0,1\}^n$. This is a consequence of the birthday paradox (see Theorem 5.4) and means that this adversary is able to distinguish

F from a random function by just outputting 0 if such a pair is found and 1 otherwise, because if $f = F_k$ no collision can be found since F_k is a permutation. However, this argument loses significance as n grows and, asymptotically, it becomes irrelevant. Indeed, it can be shown that any pseudo-random permutation is a pseudo-random function because the probability of finding a collision using a polynomial number of queries is negligible. This is a consequence of the *PRP/PRF Switching Lemma* (see[21, Lemma 3.17] for a proof) which says that the advantage of a distinguisher trying to distinguish F from a pseudo-random function exceeds that of a distinguisher trying to distinguish it from a pseudo-random permutation by at most $q(q-1)/2^{n+1}$, where q is the number of oracle queries. Since we assume that q is a polynomial function of n, this function is negligible.

Exercise 4.2 Modify Definition 4.2 so that the experiment outputs 1 if $b' = b$ (when D succeeds) and 0 otherwise, and define the advantage of D in this case.

In modern cryptography, block ciphers are usually modeled as efficient *pseudo-random permutations* in the sense just mentioned where, for a 128-bit block length, a key length between 128 and 256 will typically be used. Block ciphers can also be modeled as efficient *strong pseudo-random permutations*, a stronger concept obtained by supposing that the distinguisher D not only has oracle access to the permutation F_k and to a random permutation f but also to their inverses F_k^{-1}, f^{-1}. This is motivated by the fact that the use of a block cipher to build an encryption scheme usually implies that honest parties must be able to compute F_k^{-1} when running the decryption algorithm. Pseudo-random permutations (or PRPs, for short) are also called *secure PRPs under CPA* while strong pseudo-random permutations are *secure PRPs under CCA* (see [90, Chap. 5]); note the analogy with the concepts of CPA security and CCA security. We will indicate how to build CPA secure encryption schemes based on block ciphers modeled as pseudo-random permutations and, of course, strong pseudo-random permutations are useful in building CCA secure schemes. We insist, however, that for reasons already explained, the naive encryption scheme obtained from a strong pseudo-random permutation F by just letting the F_k be the encryption functions is not CPA secure.

4.2 The Advanced Encryption Standard

Bearing in mind the previous considerations, modern block ciphers are built with the aim of obtaining practically efficient pseudo-random permutations that, as we will see in Theorem 4.1, serve to obtain CPA secure encryption schemes. One of the preferred constructions for this purpose is the *substitution-permutation network*, which uses a series of random-looking permutations to achieve diffusion and confusion as would be expected from a pseudo-random permutation (see, for example, [109] for the details). This is the approach used in the design of the *Advanced Encryption Standard* (AES), which is by far the most popular block cipher currently in use. However, it

is appropriate to point out explicitly that there is no known proof that this cipher is a pseudo-random permutation, not even a conditional proof based on some simpler hardness hypothesis. This is not a peculiarity of AES and the same can be said of all the block ciphers that are efficient enough to be used in practice. Thus a block cipher like AES has a somewhat different status than, say, the Blum–Blum–Shub PRG in that the security of the latter reduces to the hardness of the factorization problem, which seems simpler and hence better understood than the problem whether AES is distinguishable from a random permutation. Moreover, there is another perhaps less important but more quantitative difference: while AES was only introduced in 1998 and standardized in 2001 by the United States National Institute of Standards and Technology (NIST), the factorization problem has been studied for a longer time and in different contexts by the scientific community.

AES has been intensively studied and submitted to cryptanalytic attacks since it was introduced and, so far, the only successful practical attacks against its full version are so-called *side-channel attacks* that apply to specific implementations, which are based on physical information obtained from these implementations. We will mention below other recent attacks that show that AES deviates slightly from the ideal behavior but, for now, the hypothesis that it behaves as a pseudo-random permutation for most practical purposes is still reasonable.

4.2.1 The Data Encryption Standard

Before going into the details of AES, it is also appropriate to make a brief comment on its predecessor: the *Data Encryption Standard* (DES). The block cipher that would ultimately become DES was developed by an IBM team led by Horst Feistel in the 1970s, under the name of *Lucifer*. Then, in 1973, an unprecedented event in the history of cryptography happened: the National Bureau of Standards (NBS, the US Government office that would later become NIST) made a request for a cipher meeting strict design criteria that would serve as a Government standard for encrypting unclassified but sensitive information. After none of the initial candidates was deemed to be suitable, there was a second request in 1974 and Lucifer was submitted by the IBM team. The public request and the idea of making the selected algorithm a public standard were a departure from the historical development of cryptography in which encryption schemes were secretly developed and maintained. Thus this event can be regarded as the first implicit but clear endorsement of Kerckhoffs' principle by a Government body. In 1977 the algorithm was approved as a Federal Information Processing Standard (FIPS) under the name of the Data Encryption Standard.

In the following years DES was intensely examined and attacked but no relevant weaknesses were found and the best *practical* attack against it is still a brute-force search. DES has a block length of 64 and the key size is 56 bits so that, while searching the 2^{56}-bit key space was certainly out of reach for ordinary people in the mid-1970s, it already seemed possible that a dedicated machine could be built by a large organization—for instance, by a Government or by the NSA (the US National Security Agency)—that would be able to successfully complete the attack. Other

concerns were about the *S-boxes*, a series of 8 substitutions that form the core of the algorithm and that were suspected to have been designed or modified by the NSA—which collaborated with IBM for the final design—to hide some backdoor that could allow people in the know to break the system (also, it has been widely suggested that the key was reduced, at the NSA's request, to its 56-bit size from the 112-bit size of the original IBM proposal (see, e.g. [61] although other sources claim that the original proposal had a 64-bit key, [189]). The alleged S-box weaknesses were never confirmed but from the beginning it was clear that, even if it was reasonably secure when introduced, in view of Moore's law whose first version had already been formulated ten years earlier in [145], DES would soon be vulnerable to a brute-force attack.

Before the brute-force attack could be successfully completed, several important theoretical cryptanalytic attacks were developed. The first of them, called *differential cryptanalysis*, was developed by Biham and Shamir in the late 1980s. It requires the attacker to analyze a great quantity of plaintext-ciphertext pairs, which makes the attack hardly feasible in practice. One of the surprising—or perhaps not so surprising after all—consequences of the Biham–Shamir discovery is that the designers of DES acknowledged that they already knew differential cryptanalysis and that they designed the algorithm with 16 rounds precisely in order to make this attack infeasible in practice.

An even more powerful attack called *linear cryptanalysis* was discovered by Matsui in the early nineties. This is a known plaintext attack (so that plaintexts need not be chosen by the attacker) that requires 2^{43} plaintext-ciphertext pairs to be successful against DES. Again, this number is so large that it is hardly conceivable that the attack might be successfully implemented in practice.

Both differential and linear cryptanalysis are not specifically addressed at DES; rather, they are generic attacks against block ciphers and they should be taken into account when designing such ciphers. We refer to [114, 189, 190] for detailed descriptions of these attacks.

The intense cryptanalytic work done on DES means that it is an extremely well designed block cipher for which no practical *shortcut attacks*—i.e., practical attacks requiring significantly less time than a brute-force search—are known. But, as had been expected from the beginning, DES finally succumbed to a brute-force attack. In 1997 the RSA Data Security company launched a challenge to recover a DES key from an encrypted message and, after a collaborative Internet-based effort, the key was found just five months later (with a bit of luck since only roughly 25 % of the key space had been searched by that time). In the following years there were new versions of this challenge, culminating with the "DES Challenge III", issued in 1999. In 1998 the Electronic Frontier Foundation had built, at the cost of $250,000, a special-purpose machine called "DES Cracker", containing 1,536 chips and able to search $88 \cdot 10^9$ DES keys per second. The "DES Challenge III" was then solved by a cooperative effort where the DES Cracker worked in conjunction with a network of some 100,000 computers searching at a rate of $245 \cdot 10^9$ keys per second and finding the key in just over 22 hours. A detailed description of this attack can be found in [61].

4.2.2 Introducing AES

In 1997, the same year in which DES was definitely broken by a brute-force attack, NIST announced an initiative to develop a new encryption standard, called the Advanced Encryption Standard (AES), which would replace DES. The selection process was open and the candidates had to meet a series of requirements among which the most important were support for key lengths of 128, 192, and 256 bits and a block size of 128 bits. The evaluation criteria focused on aspects related to security, cost, and implementation characteristics and, after a selection process that took three years to be completed, in October 2000, NIST announced that the algorithm **Rijndael**, designed by Belgian cryptographers Joan Daemen and Vincent Rijmen, would become AES.

As already mentioned, the input and output of the AES algorithm are 128-bit blocks and the key length is either 128, 192 or 256 bits. AES is byte-oriented in the sense that the basic processing unit in the algorithm is a byte, i.e., a sequence of eight bits treated as a single entity. With this in mind, the blocks are just 16-byte arrays and the key is an array of 16, 24 or 32 bytes. These bytes will be identified with elements of the field $\mathbb{F}_{2^8}$ as indicated in Sect. 2.8.4 so that, in particular, the arithmetic operations in this field can be applied to them. For brevity, we will usually represent bytes in hexadecimal notation, i.e., using two hexadecimal digits for each byte.

Like almost all other block ciphers, AES is a *key-iterated block cipher* that acts in a series of *rounds*. Each round can be regarded as a simple cipher that uses a key obtained from the AES key by a process called *key expansion*. There is some heuristic evidence suggesting that this iteration process may lead to a secure (i.e., pseudo-random) cipher, although this has not been rigorously proven. Of course, the round functions have to meet some criteria such as, for example, being highly non-linear, since the composition of linear functions is again linear and hence far from pseudo-random (recall, for example, the cryptanalysis of the Hill cipher, where linearity played a crucial role). Experience shows that the security of a well-designed block cipher increases with the number of rounds and one of the criteria used to evaluate the practical security of these ciphers is to attack versions with a reduced number of rounds to see how many rounds can be successfully attacked.

In AES, each 128-bit plaintext block is initialized as a 4×4 two-dimensional byte array. Then AES transforms the plaintext into ciphertext by performing a series of operations on an intermediate result called the *state*. Thus the initial state value is just a plaintext block and the final value is the corresponding ciphertext block, given also as a 4×4 byte array. AES acts by means of the repeated application of a round to the state, where each round, in turn, consists of the application of several basic operations. For the initial state, the 16 bytes in a plaintext block are mapped columnwise to the state array, so that if the sequence of bytes corresponding to this block is denoted by:

$$in_0, in_1, \ldots, in_{15},$$

and the sequence of bytes in the corresponding ciphertext block by:

$$out_0, out_1, \ldots, out_{15},$$

then we may generically represent state flow with the initial state, an intermediate state, and the final state, which is the ciphertext block, as follows:

$$\begin{array}{|c|c|c|c|} \hline in_0 & in_4 & in_8 & in_{12} \\ \hline in_1 & in_5 & in_9 & in_{13} \\ \hline in_2 & in_6 & in_{10} & in_{14} \\ \hline in_3 & in_7 & in_{11} & in_{15} \\ \hline \end{array} \rightarrow \cdots \begin{array}{|c|c|c|c|} \hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ \hline \end{array} \rightarrow \cdots \begin{array}{|c|c|c|c|} \hline out_0 & out_4 & out_8 & out_{12} \\ \hline out_1 & out_5 & out_9 & out_{13} \\ \hline out_2 & out_6 & out_{10} & out_{14} \\ \hline out_3 & out_7 & out_{11} & out_{15} \\ \hline \end{array}$$

Similarly, the key is initialized in a two-dimensional array of bytes of size `4xNk`, where `Nk` is the result of dividing the key length by 32. For example, a 192-bit key whose sequence of bytes is

$$k_0, k_1, \ldots, k_{23},$$

is initialized as an array of the form:

$$\begin{array}{|c|c|c|c|c|c|} \hline k_0 & k_4 & k_8 & k_{12} & k_{16} & k_{20} \\ \hline k_1 & k_5 & k_9 & k_{13} & k_{17} & k_{21} \\ \hline k_2 & k_6 & k_{10} & k_{14} & k_{18} & k_{22} \\ \hline k_3 & k_7 & k_{11} & k_{15} & k_{19} & k_{23} \\ \hline \end{array}$$

The number of AES rounds `Nr`, which depends on the key size, is given by the following table:

	Nk	Nr
AES−128	4	10
AES−192	6	12
AES−256	8	14

The key is expanded by means of the function `KeyExpansion` (described below) which takes the key as input and produces an *expanded key* consisting of a sequence of *round keys* `K[0]`, `K[1]`, ..., `K[Nr]` each of which is a two-dimensional byte array of size `4x4`. For this, the initial array of size `4xNk` containing the key is expanded recursively to a two-dimensional array of bytes of size `4x4(Nr+1)`, by adding columns each of which consists of a 4-byte word. Then `K[0]` consists of the first four columns of this array, `K[1]` is given by the next four columns and so on, until `K[Nr]` which is made up of the last four columns.

There are four basic AES *operations* or *transformations* that are repeated in each round—with the exception of the last round in which one of these operations is omitted. These operations are named `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`. The first three of them take the state as input and produce as output a modified state. The last one takes as input the state and a round key `K[i]` and produces as output a modified state. The encryption process starts with an application of `AddRoundKey` to the state and `K[0]` (thus this operation is sometimes called

the 0th round) and then proceeds with `Nr-1` rounds each of which applies the four basic operations mentioned. The last round, i.e., the `Nr`th round, is different because in it the `MixColumns` operation is omitted.

We stress the fact that AES is a block cipher from which, as we shall soon see, CPA secure encryption schemes can be built assuming that AES is a pseudo-random permutation, but AES itself is not an encryption scheme. Of course, we may use a key generation algorithm to generate AES keys and then use AES functions directly to encrypt and decrypt messages—of fixed length equal to the block length—but the resulting scheme is highly insecure because the encryption algorithm is deterministic. Nevertheless, we speak of AES encryption and decryption algorithms to refer to the AES function (also called the *forward cipher function*) and its inverse. Taking into account the preceding discussion, a high-level description of the AES encryption algorithm is the following:

Algorithm 4.1. AES encryption.

Input: A plaintext block initialized as `state` and an expanded key `K[0],K[1],...,K[Nr]`.
Output: A ciphertext block output as `state`.

```
AddRoundKey(state, K[0]);          #The 0th round
for i from 1 to Nr-1 do            #Rounds 1 to Nr-1
   SubBytes(state);
   ShiftRows(state);
   MixColumns(state);
   AddRoundKey(state, K[i]);
end do;
SubBytes(state);                   #The last round begins
ShiftRows(state);
AddRoundKey(state, K[Nr]);         #The last round ends
return state.
```

Next we describe the AES operations and the function `KeyExpansion` that generates the expanded key.

4.2.2.1 SubBytes

`SubBytes` is a non-linear transformation that consists of applying to the state bytes a permutation (usually called S-box in this context). The AES S-box, in contrast with DES S-boxes, has a simple algebraic structure defined by the arithmetic of the finite field $\mathbb{F}_{2^8}$ whose elements we have identified with the bytes, so that $\mathbb{F}_{2^8}$ is also referred to as the byte space. The S-box is the product of two permutations of the byte space. The first assigns to each nonzero byte its inverse when viewed as an element of $\mathbb{F}_{2^8}$ and maps the 0 byte to itself. The second is an *affine transformation* of $\mathbb{F}_{2^8}$ when viewed as an $\mathbb{F}_2$-vector space, i.e., a linear map of $\mathbb{F}_{2^8}$ into itself—which is actually an isomorphism—followed by a translation. More concretely, `SubBytes` consists of applying to all the state bytes the function $f \circ g : \mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$, where $g : \mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$ is given by

$$g(a) = \begin{cases} a^{-1} & \text{if } a \neq 0 \\ 0 & \text{if } a = 0 \end{cases}$$

and the affine map $f : \mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$, for $a \in \mathbb{F}_{2^8}$, viewed as an 8-bit vector of $\mathbb{F}_2^8$ (where the a_i are the coefficients of the corresponding polynomial, see Sect. 2.8.4), is given by:

$$a = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} \mapsto f(a) = \begin{bmatrix} 1\,0\,0\,0\,1\,1\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,0\,0\,0\,1 \\ 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1\,1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

4.2.2.2 The `SubBytes` Operation in Maple

In AES implementations, the S-box (and hence the `SubBytes` transformation) is always given by a lookup table for efficiency. Let us see how this table can be constructed with Maple. We start by recalling the definition of $\mathbb{F}_{2^8}$:

```
> F256 := GF(2, 8, x^8+x^4+x^3+x+1):
```

The permutation of $\mathbb{F}_{2^8}$ that maps each nonzero byte to its inverse (and the 0 byte to itself) is then:

```
> invbyte := proc(byte::nonnegint)
  if byte = 0 then
    0
  else
    (F256:-output@F256:-inverse@F256:-input)(byte)
  end if
  end proc:
```

The affine map, that takes as input the 8-bit list corresponding to a byte, i.e., the result of applying the function `bytetobits` (from Appendix A) to a byte in the 0..255 range, and produces as output another byte also given as an 8-bit list, is:

```
> affmap := proc(l::list)
  local vect, M, v;
  uses LinearAlgebra:-Modular;
  vect := Mod(2,ListTools:-Reverse(l),integer);
  M := Mod(2,Matrix([seq(ListTools:-Rotate([1,0,0,0,1,1,1,1],-i),i = 0..7)]),integer);
  v := Mod(2,Vector([1,1,0,0,0,1,1,0]),integer);
  vect := AddMultiple(2,Multiply(2,M,vect),v);
  ListTools:-Reverse(convert(vect,list))
  end proc:
```

With these ingredients we can build a function `ByteSub` that implements the permutation of $\mathbb{F}_{2^8}$ underlying `SubBytes`. It consists simply of applying `invbyte` followed by `affmap`. But we cannot compose these two functions directly because

the bytes which they take as input (and those they produce as output) have different formats. In order to make these functions composable, we use the functions `bitstobyte` and `bytetobits` which are defined in Appendix A. `ByteSub` takes as input one byte given as an integer in the 0..255 range and produces as output another byte given in the same format:

```
> ByteSub := bitstobyte@affmap@bytetobits@invbyte:
```

For example, let us compute the result of applying `ByteSub` to a few bytes (in Maple versions prior to v13, the elementwise operator ~ should be replaced by a call to `map`):

```
> ByteSub~([$127 .. 132]);
                          [210, 205, 12, 19, 236, 95]
```

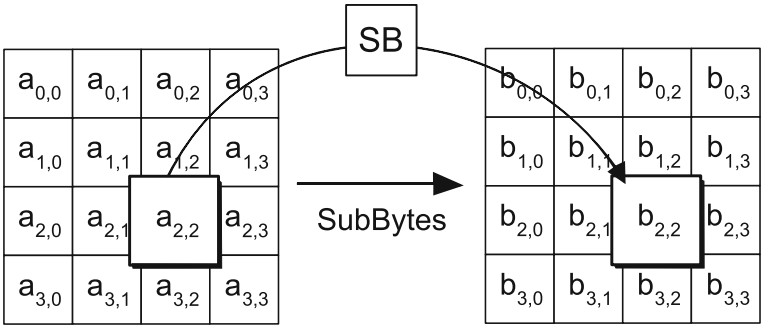
Now, the S-box can be given as a byte array such that, if b is a byte in the 0..255 range, then `ByteSub(b) = SBox[b]`:

```
> SBox := Array(0 .. 255, x-> ByteSub(x)):
```

We also record the following version of `SBox`, given as a map, which is useful, for example, when one has to apply the `SBox` to the members of a list:

```
> SB := byte ->  SBox[byte]:
```

Remark 4.1 Observe that `ByteSub` and `SBox` are simply two different ways of describing the permutation underlying the `SubBytes` transformation of AES. In our subsequent implementation we will use `SBox` as a lookup table, which produces much faster results than applying `ByteSub`. Note also that `SubBytes` is simply the result of replacing each byte `b` in the state by `SBox[b]` (or, equivalently, by `ByteSub(b)` or `SB(b)`). We do not use the `SubBytes` name for this permutation of the byte space, but rather for the operation that defines when it acts on the whole state as illustrated by the following figure:



The S-box is usually displayed as a table given by a 16×16 matrix with files and columns numbered from 0 to 15. It is also customary to use hexadecimal notation for this table so that files and columns are numbered by hexadecimal digits, from `0` to `f`. Thus this matrix has files and columns numbered by "half-bytes" and its entries are bytes. In this table, the image of the byte `b = ij`, where `i` and `j` are two hexadecimal

digits, is the byte in the `(i,j)` position. The next auxiliary function is similar to the function `bytestohexstring` in Appendix A but it acts on single bytes instead and, for readability, it does not convert them to strings but to hex symbols:

```
> bytetohex := proc (byte::(integer[0 .. 255]))
  local h;
  h := cat(``, StringTools:-LowerCase(convert(byte, hex, decimal)));
  if length(h) = 1 then
    h := cat(0, h)
  end if;
  h
  end proc:
```

The Maple function that prints the S-box matrix is then the following:

```
> PrintSBox := proc()
  local t, s;
  t := interface(rtablesize = 16);
  s := Array(SBox, order = C_order);
  print(LinearAlgebra:-Map(bytetohex, ArrayTools:-Alias(s, [16, 16])));
  interface(rtablesize = t);
  NULL
  end proc:
```

This produces the following result:

```
> PrintSBox();
```

$$\begin{bmatrix}
63 & 7c & 77 & 7b & f2 & 6b & 6f & c5 & 30 & 01 & 67 & 2b & fe & d7 & ab & 76 \\
ca & 82 & c9 & 7d & fa & 59 & 47 & f0 & ad & d4 & a2 & af & 9c & a4 & 72 & c0 \\
b7 & fd & 93 & 26 & 36 & 3f & f7 & cc & 34 & a5 & e5 & f1 & 71 & d8 & 31 & 15 \\
04 & c7 & 23 & c3 & 18 & 96 & 05 & 9a & 07 & 12 & 80 & e2 & eb & 27 & b2 & 75 \\
09 & 83 & 2c & 1a & 1b & 6e & 5a & a0 & 52 & 3b & d6 & b3 & 29 & e3 & 2f & 84 \\
53 & d1 & 00 & ed & 20 & fc & b1 & 5b & 6a & cb & be & 39 & 4a & 4c & 58 & cf \\
d0 & ef & aa & fb & 43 & 4d & 33 & 85 & 45 & f9 & 02 & 7f & 50 & 3c & 9f & a8 \\
51 & a3 & 40 & 8f & 92 & 9d & 38 & f5 & bc & b6 & da & 21 & 10 & ff & f3 & d2 \\
cd & 0c & 13 & ec & 5f & 97 & 44 & 17 & c4 & a7 & 7e & 3d & 64 & 5d & 19 & 73 \\
60 & 81 & 4f & dc & 22 & 2a & 90 & 88 & 46 & ee & b8 & 14 & de & 5e & 0b & db \\
e0 & 32 & 3a & 0a & 49 & 06 & 24 & 5c & c2 & d3 & ac & 62 & 91 & 95 & e4 & 79 \\
e7 & c8 & 37 & 6d & 8d & d5 & 4e & a9 & 6c & 56 & f4 & ea & 65 & 7a & ae & 08 \\
ba & 78 & 25 & 2e & 1c & a6 & b4 & c6 & e8 & dd & 74 & 1f & 4b & bd & 8b & 8a \\
70 & 3e & b5 & 66 & 48 & 03 & f6 & 0e & 61 & 35 & 57 & b9 & 86 & c1 & 1d & 9e \\
e1 & f8 & 98 & 11 & 69 & d9 & 8e & 94 & 9b & 1e & 87 & e9 & ce & 55 & 28 & df \\
8c & a1 & 89 & 0d & bf & e6 & 42 & 68 & 41 & 99 & 2d & 0f & b0 & 54 & bb & 16
\end{bmatrix}$$

For example, if we want to compute `ByteSub(169) = SBox[169]`, we just note that `169` is, in hexadecimal notation, `a9` and hence the image of this byte by means of the `ByteSub` permutation is the entry in file `a` and column `9` which is `d3` as can also be seen by checking that `SBox[169] = 211.`

4.2.2.3 ShiftRows

The `ShiftRows` transformation cyclically shifts the state rows to the left by offsets equal to the row index, i.e., 0, 1, 2 and 3. Thus if we denote the state bytes by $b_{i,j}$, $0 \leq i, j \leq 3$ then, using ':=' as usual to represent assignment:

$$b_{i,j} := b_{i,i+j \bmod 4}$$

(i.e., the value of $b_{i,j}$ is replaced by that of $b_{i,i+j \bmod 4}$ or, in other words, the byte $b_{i,j}$ is moved to the position $(i,\ j - i \bmod 4)$ in the state matrix.

Thus the state array is changed as follows:

$$\begin{array}{|c|c|c|c|} \hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ \hline \end{array} := \begin{array}{|c|c|c|c|} \hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ \hline b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ \hline b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \\ \hline \end{array}$$

4.2.2.4 MixColumns

`MixColumns` is an operation that acts on the state by considering it as a 4×4 matrix over $\mathbb{F}_{2^8}$ and multiplying it by another 4×4 matrix over $\mathbb{F}_{2^8}$ specified in the standard (whose bytes are written in hexadecimal below). This operation acts as follows:

$$\begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} := \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}$$

Note that when two matrices are multiplied, the result is the same as that obtained by multiplying the first matrix by each column vector of the second one and building the matrix which has these columns. Thus the name `MixColumns` comes from viewing this operation as the result of the action of the above-defined matrix on the state columns.

Exercise 4.3 Use Maple to check that the inverse of the matrix (over $\mathbb{F}_{2^8}$) used to define the `MixColumns` operation is the matrix:

$$\begin{pmatrix} \texttt{0e} & \texttt{0b} & \texttt{0d} & \texttt{09} \\ \texttt{09} & \texttt{0e} & \texttt{0b} & \texttt{0d} \\ \texttt{0d} & \texttt{09} & \texttt{0e} & \texttt{0b} \\ \texttt{0b} & \texttt{0d} & \texttt{09} & \texttt{0e} \end{pmatrix}$$

4.2.2.5 AddRoundKey

In this operation the state is modified by just doing the bitwise Xor with the round key array `K[i]` corresponding to the round. Thus, in the `i`th round, the state is modified as follows:

$$\texttt{state:=state} \oplus \texttt{K[i]}.$$

4.2.2.6 KeyExpansion

As mentioned above, `KeyExpansion` takes as input an AES key and generates a so-called "key schedule", consisting of `Nr+1` round keys `K[i]`, each of which is a 4×4 byte array which can also be regarded as a one-dimensional array consisting of 4 four-byte words (the columns of the two-dimensional array). Thus the key expansion can be seen as a process that takes as input `Nk` four-byte words (the columns of the array in which the AES key is initialized) and generates `4(Nr+1)-Nk` more words making a total of `4(Nr+1)` words which are denoted by `w[0]`, `w[1]`, ..., `w[4Nr+3]`. The first four of these words make up the array `K[0]`, the next four `K[1]`, and so on until `K[Nr]`, thus giving the round subkeys used by the `AddRoundKey` operation. The key expansion algorithm uses two other functions. One of them is called `RotWord` and performs a cyclic left shift of the bytes in a word (or, rather, an "up shift" when the word is written as a column). The other is `SubWord` and it simply applies the S-box to the bytes in a word. `KeyExpansion` also uses the "round constant word array" `Rcon` which consists of 14 four-byte words obtained by successive multiplication by the byte `02` in $\mathbb{F}_{2^8}$, starting with the word `[01,00,00,00]` (so that `Rcon[i]` $= [2^{i-1}, 0, 0, 0]$ if the bytes are represented as integers in the 0..255 range and exponentiation is taken in $\mathbb{F}_{2^8}$).

The key expansion algorithm can be given in pseudocode as follows. Note that, after building the first `Nk` words of the expanded key with the AES key, each following word `w[i]` is obtained by Xor-ing the previous word `w[i-1]` with the word `Nk` positions before, except when the position is a multiple of `Nk` or when `Nk=8`, in which cases some additional transformations are applied to `w[i-1]` prior to the Xor operation.

Algorithm 4.2. KeyExpansion.

Input: An AES key k given as a one-dimensional byte array `[k[0], k[1], ..., k[4Nk]]`.
Output: An expanded key given as an array of `Nr+1` round keys.

```
for i from 0 to Nk-1 do
   w[i] := [k[4i], k[4i+1], k[4i+2], k[4i+3]]
end do;
for i from Nk to 4Nr+3 do
   temp := w[i-1];
   if i mod Nk = 0 then
      temp := SubWord(RotWord(temp)) ⊕ Rcon[i/Nk]
   else if Nk > 6 and i mod Nk = 4
      temp := SubWord(temp)
   end if;
   w[i] := w[i-Nk] ⊕ temp
end do;
for i from 0 to Nr do
   K[i] := [w[4i], w[4i+1], w[4i+2], w[4i+3]]
end do;
return [K[0], ..., K[Nr]].
```

4.2.2.7 `KeyExpansion` in Maple

Next we implement `KeyExpansion` in Maple. We start by defining the round constants `Rcon` which we shall give as a list of 4-byte lists. Here we use the multiplication in $\mathbb{F}_{2^8}$, `mult256`, which was defined in Sect. 2.8.4.

```
> RconGen := proc()
  local i, l;
  l := [[1, 0, 0, 0]];
  for i to 13 do
    l := [op(l), map(x -> mult256(2, x), l[-1])]
  end do;
  l
  end proc:
```

The list of round constants, `Rcon`, is then:

```
> Rcon := RconGen();
[[1, 0, 0, 0], [2, 0, 0, 0], [4, 0, 0, 0], [8, 0, 0, 0], [16, 0, 0, 0],
  [32, 0, 0, 0], [64, 0, 0, 0], [128, 0, 0, 0], [27, 0, 0, 0], [54, 0, 0, 0],
  [108, 0, 0, 0], [216, 0, 0, 0], [171, 0, 0, 0], [77, 0, 0, 0]]
```

We will make use of the following function which takes as input an AES key given either as a list of integers in the 0..255 range or as a hex string and checks whether the supplied key is valid. In that case the function produces as output the list of bytes corresponding to this key and otherwise returns an error message.

```
> checkkey := proc(key::{string, list(integer[0 .. 255])})
  uses StringTools;
  if type(key, list) then
    if not member(nops(key), [16, 24, 32]) then
      error "The supplied key does not have valid length"
    end if;
    return key
  else
    if not member(Length(key), [32, 48, 64]) then
      error "The supplied key does not have valid length"
    end if;
    if not IsHexDigit(key) then
      error "The supplied key is not a hex string"
    end if;
    return hexstringtobytes(LowerCase(key))
  end if
  end proc:
```

Let us now give the Maple function implementing `KeyExpansion`, for which we shall use this same name. The function takes as input an AES key, given as a list or a hexadecimal string of 16, 24 or 32 bytes. The output is a list of 0..3–0..3 arrays (of Maple type `Array`) whose entries are bytes; each one of these arrays is a round subkey. The first 4, 6 or 8 columns of the first two arrays contain the key, which has been mapped to the arrays columnwise, starting at the top of the leftmost column. The functions `RotWord` and `SubWord` mentioned in Algorithm 4.2 are not defined here because the first is simply a version of `Rotate` from the `ListTools` package and the second consists of applying `SubBytes` to a 4-byte word, which is done here by means of the previously defined function `SB`.

```
> KeyExpansion := proc(key)
   local k, l, s, Nk, ek, ek1, i, temp;
   k := checkkey(key);
   l := nops(k);
   Nk := iquo(l, 4);
   s := l+28;
   ek := [$1 .. s];
   ek1 := ListTools:-LengthSplit(k, 4);
   for i to Nk do
     ek[i] := ek1[i]
   end do;
   for i from Nk+1 to s do
     temp := ek[i-1];
     if i-1 mod Nk = 0 then
       temp := zip(BitXor, map(SB,ListTools:-Rotate(temp, 1)), Rcon[iquo(i-1, Nk)])
     elif Nk = 8 and i-5 mod Nk = 0 then
       temp := map(SB, temp)
     end if;
     ek[i] := zip(BitXor, ek[i-Nk], temp)
   end do;
   map(x -> Array(0..3, 0..3, (i,j) -> x[j+1,i+1]), [ListTools:-LengthSplit(ek,4)]);
   end proc:
```

Example 4.1 Let us consider the 192-bit key given in Example A.2 of [73]:

```
> k192 := "8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b":
```

Next we compute the expanded key corresponding to `k192` but we do not print it:

```
> expkey := KeyExpansion(k192):
```

Let us print, for example, the last three round keys in matrix form. We use Maple's function `ArrayTools:-Alias` to put the round keys in a format suitable to be printed (as on other similar occasions, the elementwise operator ~ should be replaced by a call to `map` if a version of Maple prior to v13 is used):

```
> map(x -> bytetohex~(x), map(x -> ArrayTools:-Alias(x, [4,4]), expkey)[-3..1]);
```

$$\left[\begin{bmatrix} a7 & 94 & 82 & ad \\ e1 & 11 & 1f & 07 \\ 46 & f1 & 75 & d7 \\ 6c & df & 0a & 53 \end{bmatrix}, \begin{bmatrix} ca & 8f & 28 & bc \\ 40 & cc & 2d & 3c \\ 05 & 50 & 16 & e7 \\ 38 & 06 & 6a & b5 \end{bmatrix}, \begin{bmatrix} e9 & 44 & 8e & 01 \\ 8b & 8c & cc & 00 \\ a0 & 77 & 72 & 22 \\ 6f & 3c & 04 & 02 \end{bmatrix}\right]$$

Exercise 4.4 Compute the expanded keys corresponding to the AES keys given in Examples A.1, A.2, and A.3 in [73] and check that the results obtained correspond exactly to the ones given there.

4.2.3 AES Decryption

The four basic AES operations are invertible and so AES decryption is straightforward: it consists of running the inverse operations in reverse order, with the round

keys also taken in reverse order. There is also an equivalent algorithm for decryption in which the sequence of steps is the same as in encryption, with operations replaced by their inverses and a change in the key schedule, but we shall not go into the details, which can be seen in [62]. We briefly describe the inverse operations (obviously, `AddRoundKey`, which computes the bitwise Xor of the state bytes with the round subkey bytes, is its own inverse).

4.2.3.1 InvSubBytes

The `SubBytes` operation consists of applying to each individual byte in the state array a permutation defined by the `SBox` table we have constructed before. Thus the inverse operation, `InvSubBytes`, acts by applying the inverse permutation of the one defined by `SBox`. In Maple we can easily "invert" the `SBox` array by setting:

```
> InvSBox := Array(0 .. 255, table([seq(SBox[i] = i, i = 0 .. 255)])):
```

As in the case of the S-box, the inverse S-box may be printed as follows:

```
> PrintInvSBox := proc()
  local t, is;
  t := interface(rtablesize = 16);
  is := Array(InvSBox, order = C_order);
  print(LinearAlgebra:-Map(bytetohex, ArrayTools:-Alias(is, [16, 16])));
  interface(rtablesize = t);
  NULL
  end proc:

  PrintInvSbox();
```

$$\begin{bmatrix}
52 & 09 & 6a & d5 & 30 & 36 & a5 & 38 & bf & 40 & a3 & 9e & 81 & f3 & d7 & fb \\
7c & e3 & 39 & 82 & 9b & 2f & ff & 87 & 34 & 8e & 43 & 44 & c4 & de & e9 & cb \\
54 & 7b & 94 & 32 & a6 & c2 & 23 & 3d & ee & 4c & 95 & 0b & 42 & fa & c3 & 4e \\
08 & 2e & a1 & 66 & 28 & d9 & 24 & b2 & 76 & 5b & a2 & 49 & 6d & 8b & d1 & 25 \\
72 & f8 & f6 & 64 & 86 & 68 & 98 & 16 & d4 & a4 & 5c & cc & 5d & 65 & b6 & 92 \\
6c & 70 & 48 & 50 & fd & ed & b9 & da & 5e & 15 & 46 & 57 & a7 & 8d & 9d & 84 \\
90 & d8 & ab & 00 & 8c & bc & d3 & 0a & f7 & e4 & 58 & 05 & b8 & b3 & 45 & 06 \\
d0 & 2c & 1e & 8f & ca & 3f & 0f & 02 & c1 & af & bd & 03 & 01 & 13 & 8a & 6b \\
3a & 91 & 11 & 41 & 4f & 67 & dc & ea & 97 & f2 & cf & ce & f0 & b4 & e6 & 73 \\
96 & ac & 74 & 22 & e7 & ad & 35 & 85 & e2 & f9 & 37 & e8 & 1c & 75 & df & 6e \\
47 & f1 & 1a & 71 & 1d & 29 & c5 & 89 & 6f & b7 & 62 & 0e & aa & 18 & be & 1b \\
fc & 56 & 3e & 4b & c6 & d2 & 79 & 20 & 9a & db & c0 & fe & 78 & cd & 5a & f4 \\
1f & dd & a8 & 33 & 88 & 07 & c7 & 31 & b1 & 12 & 10 & 59 & 27 & 80 & ec & 5f \\
60 & 51 & 7f & a9 & 19 & b5 & 4a & 0d & 2d & e5 & 7a & 9f & 93 & c9 & 9c & ef \\
a0 & e0 & 3b & 4d & ae & 2a & f5 & b0 & c8 & eb & bb & 3c & 83 & 53 & 99 & 61 \\
17 & 2b & 04 & 7e & ba & 77 & d6 & 26 & e1 & 69 & 14 & 63 & 55 & 21 & 0c & 7d
\end{bmatrix}$$

4.2.3.2 InvShiftRows

The `ShiftRows` operation consists of shifting the rows of the state array by 0, 1, 2, 3 positions to the left, where each row is shifted by the same amount as its index. Thus the inverse operation just shifts rows right by the same offsets and, if we denote by $b_{i,j}$, $0 \leq i, j \leq 3$ the state bytes, their values are modified as follows:

$$b_{i,j} := b_{i,j-i \bmod 4}.$$

Thus $b_{i,j}$ goes to the $(i, i + j \bmod 4)$ position in the state array which is changed as follows:

$$\begin{array}{|c|c|c|c|} \hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ \hline \end{array} := \begin{array}{|c|c|c|c|} \hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,3} & b_{1,0} & b_{1,1} & b_{1,2} \\ \hline b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ \hline b_{3,1} & b_{3,2} & b_{3,3} & b_{3,0} \\ \hline \end{array}$$

4.2.3.3 InvMixColumns

`MixColumns` changes the state by matrix multiplication with a specified matrix over $\mathbb{F}_{2^8}$. This matrix is invertible and hence the inverse operation `InvMixColumns` acts by multiplying the state by the inverse matrix, namely

$$\begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} := \begin{pmatrix} \texttt{0e} & \texttt{0b} & \texttt{0d} & \texttt{09} \\ \texttt{09} & \texttt{0e} & \texttt{0b} & \texttt{0d} \\ \texttt{0d} & \texttt{09} & \texttt{0e} & \texttt{0b} \\ \texttt{0b} & \texttt{0d} & \texttt{09} & \texttt{0e} \end{pmatrix} \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}$$

Exercise 4.5 Show that the `SubBytes` and the `ShiftRows` operations commute. Do either of them commute with `MixColumns`?

4.2.4 Remarks on AES Design and AES Security

The first thing that should be emphasized is that, as we have already remarked, AES is a cryptographic primitive that is useful to build encryption and authentication schemes but AES itself is not such a scheme, and if naively used to encrypt and decrypt then the resulting encryption scheme is deterministic and hence is not CPA secure by Proposition 3.4. We will see later how modes of operation can be used to avoid this difficulty but, for now, we will only discuss the security of the AES primitive itself and, in particular, its resistance to key-recovery attacks.

It is generally accepted that the resistance of block ciphers against cryptanalytic attacks increases with the number of rounds but, of course, this must be balanced against the fact that, as the number of rounds increases, the cipher becomes less efficient. One of the main criteria used today in the evaluation of a block cipher's security is based on the non-existence of *shortcut attacks*, namely, key-recovery attacks faster than exhaustive search. Since there are 2^{128} possible 128-bit AES keys and the largest publicly known exhaustive key search so far has been against a 64-bit key, it is clear that the existence of a shortcut attack does not mean that the cipher is broken in practice. For example, a hypothetical attack that reduced the search to, say,

2^{112} keys, would still make the search far from feasible in practice. Nevertheless, the existence of such an attack would reveal a weakness in the cipher that might lead it to be broken sooner than expected and, as it is often said in cryptographic circles, *attacks never get worse, they only get better*. For these reasons a cipher for which shortcut attacks significantly more efficient than exhaustive search are known, is regarded as broken.

In the design of AES, the number of rounds was determined by considering the maximum number of rounds for which shortcut attacks were found [62]. For reduced versions of more than six rounds no shortcut attacks were known and the designers added four rounds more as a security margin. There are several facts that suggest that this is, in fact, a conservative approach. As mentioned in [62], two rounds of AES provide full diffusion in the sense that every state bit depends on all the state bits two rounds before, and, moreover, a change in one state bit is likely to affect half of the state bits two rounds later. The number of rounds chosen also makes it much more difficult to devise successful versions of several known attacks such as linear cryptanalysis and differential cryptanalysis. For AES versions with a key longer than 128 bits, the number of rounds was raised by one for every additional 32 bits in the key. We shall not go into the details and refer the reader to [62] instead.

A basic characteristic of the design philosophy that led to the AES algorithm is, as mentioned in [62], *simplicity*. This makes it possible to give a one-page description of AES as H.W. Lenstra, Jr. did [126] (assuming that a little algebraic background is known). This simplicity applies, in particular, to the S-box which, as we have seen, was transparently built using the algebraic structure of $\mathbb{F}_{2^8}$. On the one hand, this leaves no ground for the suspicion that the S-box might contain a secret trapdoor—as was suspected by some people in the case of the DES S-boxes—and, on the other hand, the use of the highly non-linear inversion function of $\mathbb{F}_{2^8}$, complemented with the affine map we have described, aims at making the system resistant to linear and differential cryptanalysis. According to present knowledge it seems that AES is indeed resistant to these attacks.

The remaining AES operations are also very simple. `ShiftRows` has the purpose of making the algorithm resistant to certain specific attacks and `MixColumns` produces diffusion among the state bytes, as the change of one byte produces a change in the four output bytes in the same column of the state. The key schedule also uses the S-box to produce a "non-linear mixture" of the key bits with the purpose, among other things, of obtaining resistance against related-key attacks (defined below) and attacks in which the key is partially known by the cryptanalyst.

The algebraic simplicity of AES's design led some critics to manifest their skepticism or their doubts regarding the security of the algorithm but so far these doubts have not been vindicated by successful attacks exploiting the algebraic structure of the cipher. In 2002, an 'algebraic attack' called the *XSL attack*, based on the description of AES by means of a system of quadratic equations, was announced by N. Courtois and J. Pieprzyk in [56], purporting to lead to a shortcut attack on the 256-bit version of AES. This attack, whose complexity analysis is heuristic, was received with considerable skepticism by the cryptographic community, and several cryptographers expressed their conviction that the attack as originally presented is

not effective. A clear presentation of the algebraic aspects of AES and of the tools that could eventually be used to mount a successful algebraic attack is given in [49].

We have already mentioned that there is no proof that AES is a pseudo-random permutation and the same can be said of all other efficient block ciphers used in practice. In fact, there are some recent attacks that show that AES deviates from the ideal behavior. One of them is the *related-key attack* (RKA) presented in [25], which allows a key-recovery for AES-256 in $2^{99.5}$ steps. In the RKA model, the block cipher can be queried not only under the target key but also under other keys derived from it in ways chosen by the adversary, which makes these attacks virtually impossible to implement in practice and hence not too significant. However, still more recently, a new attack that does not require related keys and is inspired by the meet-in-the -middle attacks against hash functions, was presented in [32]. This is a key-recovery attack against AES-128 and requires only $2^{126.1}$ steps. Thus this attack recovers the key about four times faster that a pure brute-force attack, and a similar advantage factor holds for the similar attacks against AES-192 and AES-256. These are the first shortcut attacks on the full AES and, while they have no practical consequences because of their high complexity, they should certainly be watched.

In conclusion, the only successful practical attacks against AES so far are side-channel attacks against specific implementations. Thus, AES is for now considered secure and, in fact, it was approved to protect classified information by the US Government in 2003 and included in NSA's Suite B Cryptography ([149]).

4.3 Modes of Operation

Modes of operation are algorithms that allow the encryption of arbitrary-length messages using a block cipher and, moreover, can provide the non-deterministic encryption that is necessary to achieve CPA security. Originally, a couple of such modes were specified for DES and, more recently, NIST specified five confidentiality modes to be used with symmetric block ciphers and, in particular, with AES (see [68]). In this section we review these five modes and we show how some of them can be used to build CPA secure encryption schemes. In particular, we prove that this is the case for a version of counter mode when used in combination with a block cipher which is a pseudo-random permutation.

4.3.1 Confidentiality Modes

Next, we describe the five confidentiality modes specified in [68].

4.3.1.1 Electronic Codebook Mode

Electronic codebook mode (ECB) is the most straightforward mode as it essentially consists of directly applying the forward cipher function to encrypt messages. An arbitrary plaintext is first padded to make its length a multiple of the block length. There are several padding algorithms and perhaps the most common is the so-called 10^i *padding* which consists of appending a 1 bit followed by as many zero bits as necessary to make the message length a multiple of the block length n, taking care to add a full block in case the message length is already a multiple of n (this is necessary to make the padding function injective and hence to make the message uniquely recoverable). Then the message m consists of a certain number of blocks, i.e., $m = m_1 m_2 \dots m_l$ (where, for simplicity, we denote the concatenation of blocks by juxtaposing them). If we denote by F_k the block cipher encryption function (i.e., the forward cipher function) corresponding to the key k and D_k the decryption function (i.e., the inverse $D_k = F_k^{-1}$), ECB is the encryption scheme (**Gen**, **Enc**, **Dec**) where **Gen** is a probabilistic key-generating algorithm and **Enc** and **Dec** are defined by:

- $\mathbf{Enc}(k, m) = c_1 c_2 \dots c_l$, where $m = m_1 m_2 \dots m_l$ and $c_i = F_k(m_i)$, for $i = 1, \dots, l$.
- $\mathbf{Dec}(k, c) = m_1 m_2 \dots m_l$, where $c = c_1 c_2 \dots c_l$ and $m_i = D_k(c_i)$, for $i = 1, \dots, l$.

Thus each block is encrypted independently of the remaining ones by just applying the block cipher encryption function, and decryption is similar.

The encryption scheme thus defined is deterministic and hence it cannot be CPA secure. But there is worse since, in fact, ECB mode does not have indistinguishable encryptions in the presence of an eavesdropper. This is because if the same block appears twice in the plaintext, the corresponding ciphertext blocks are also equal and hence it is easy to distinguish a message with repeated blocks from a message with no repeated blocks. This allows an attacker to recognize plaintext patterns, which facilitates the use of statistical attacks. The weakness of this mode is even more evident in the presence of an active adversary. This adversary could reorder ciphertext blocks and even insert old ciphertext blocks into the ciphertext without the manipulation being detected by the honest parties. For all these reasons, ECB mode is *extremely insecure* and should not be used.

4.3.1.2 Cipher Block Chaining mode

Cipher Block Chaining mode (CBC) uses "chaining" (a sort of feedback) to make the encryption of a plaintext block m_i dependent not only on m_i and the key k, but also on the preceding blocks $m_1, \dots, m_{i-1}$ as well as on an *initialization vector* (IV), which is randomly chosen with length equal to the block length n. As in ECB mode, the message is padded prior to encryption to complete an integral number of blocks:

Algorithm 4.3. CBC encryption.

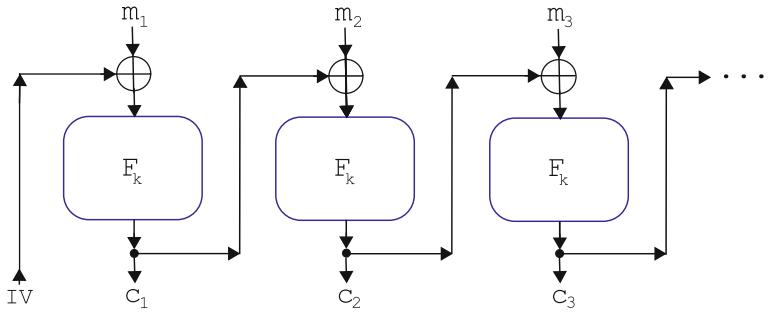
Input: An l-block message $m = m_1 \dots m_l$, and a block cipher key k.
Output: The ciphertext $\mathbf{Enc}(k, m) = c_1 \dots c_l$.

`Initialization`:

$IV \leftarrow \{0,1\}^n$, chosen at random.

`Encryption`:

$c_0 := IV$;
`for` i `from` 1 `to` l `do`
$c_i := F_k(m_i \oplus c_{i-1})$
`end do`;
`return` $c_1 \dots c_l$.



CBC Encryption

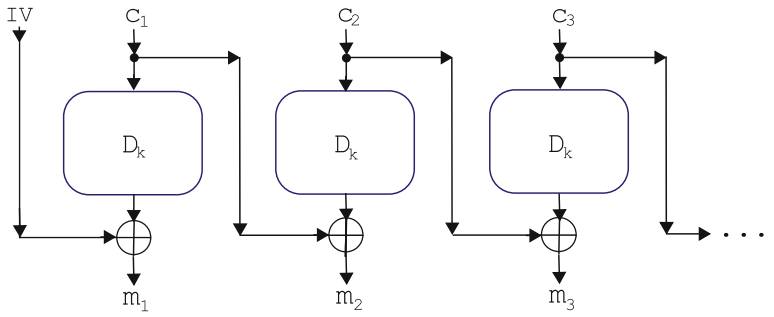
Decryption is similar:

Algorithm 4.4. CBC decryption.

Input: An IV, a ciphertext $c = c_1 \dots c_l$, and a block cipher key k.
Output: The plaintext $\mathbf{Dec}(k, c) = m_1 \dots m_l$.

`Decryption`:

$c_0 := IV$;
`for` i `from` 1 `to` l `do`
$m_i := D_k(c_i) \oplus c_{i-1}$
`end do`;
`return` $m_1 \dots m_l$.



CBC Decryption

CBC mode removes the disadvantages inherent to ECB mode: since the IV is randomly chosen and each ciphertext block depends on the preceding ones (and also on the IV), it is unlikely that a statistical attack might be successful or that the ciphertext can be manipulated without being detected. In fact, if the underlying block cipher is a pseudo-random permutation, then CBC is CPA secure (we refer for this to [90, Theorem 6.19]). However, the so-called 'counter version' of CBC mode, in which the random IV is replaced by a *counter* that is initially randomly chosen and is increased by one in each subsequent encryption, is not CPA secure (see Exercise 4.6 below).

The fact that if a plaintext block is changed then not only the corresponding ciphertext block but also the following ones are changed, makes this mode suitable for authentication purposes, as we will see in Chap. 5.

CBC mode is generally used as the default mode in many implementations but it also has some inconveniences. One of them is that encryption is carried out sequentially and hence it cannot be parallelized; note, however, that decryption is parallelizable. Another minor drawback is that the IV must be transmitted along with the ciphertext (so it can be regarded as part of it) and this means that there is "message expansion" because the ciphertext has one more block than the plaintext. Another problem is that transmission errors are propagated for, if ciphertext block c_i suffers such an error, then the plaintext blocks m_i and m_{i+1} can be incorrectly decrypted, although the remaining blocks decrypt correctly if there are no more transmission errors. Of course, these errors can be corrected by using suitable error-correcting codes for transmission.

Exercise 4.6 Show that if the IV in CBC mode is randomly chosen for the first encryption and then increased by 1 for each subsequent encryption, then the resulting scheme is not CPA secure.

4.3.1.3 Cipher Feedback Mode

Cipher feedback mode (CFB) is a mode that can be used to turn a block cipher into a stream cipher by using the block cipher to generate a stream of pseudo-random bits that are then Xor-ed to the plaintext to obtain the ciphertext bits. This mode

requires an integer parameter s such that $1 \leq s \leq n$, where n is the block length of the cipher. The plaintext length must be a multiple of s so padding may be necessary except when $s = 1$. The value of s is often used in the name of the mode, so that one speaks of, e.g., the 1-bit CFB mode, the 8-bit CFB mode, or the 128-bit CFB mode. Moreover, as in CBC mode, a random IV is used (actually, for both modes, the IV is only required in [68] to be *unpredictable* and choosing it at random suffices). We denote by LSB_t the t least significant (or rightmost) bits of a bit string, by MSB_t the t most significant (leftmost) bits, and we use $||$ for the concatenation of bit strings. The encryption and decryption algorithms are given in Algorithms 4.5 and 4.6.

Algorithm 4.5. CFB encryption.

Input: A message with l s-bit blocks, $m = m_1 \dots m_l$, and a block cipher key k.
Output: The ciphertext $\mathbf{Enc}(k, m) = c_1 \dots c_l$.

Initialization:

$IV \leftarrow \{0,1\}^n$, chosen at random.

Encryption:

$x_1 := IV$;
for i from 1 to l do
$c_i := m_i \oplus MSB_s(F_k(x_i))$;
$x_{i+1} := LSB_{n-s}(x_i)||c_i$
end do;
return $c_1 \dots c_l$.

Algorithm 4.6. CFB decryption.

Input: An IV, a ciphertext $c = c_1 \dots c_l$, and a block cipher key k.
Output: The plaintext $\mathbf{Dec}(k, c) = m_1 \dots m_l$.

Decryption:

$x_1 := IV$;
for i from 1 to l do
$m_i := c_i \oplus MSB_s(F_k(x_i))$;
$x_{i+1} := LSB_{n-s}(x_i)||c_i$
end do;
return $m_1 \dots m_l$.

CFB mode is CPA secure when used with a block cipher that is a pseudo-random permutation. It is easy to implement since it does not use the block cipher decryption function and 1-bit CFB does not require padding. But it has also some drawbacks such as the fact that in order to encrypt s bits a full n-bit block must be encrypted. Moreover, CFB mode propagates errors since any erroneously transmitted ciphertext block affects the decryption of that block and also of the next $\lceil n/s \rceil$ blocks. In addition, each block in the key stream depends also on some bits of the previous ciphertext block so that it is not possible to precompute the stream before encryption.

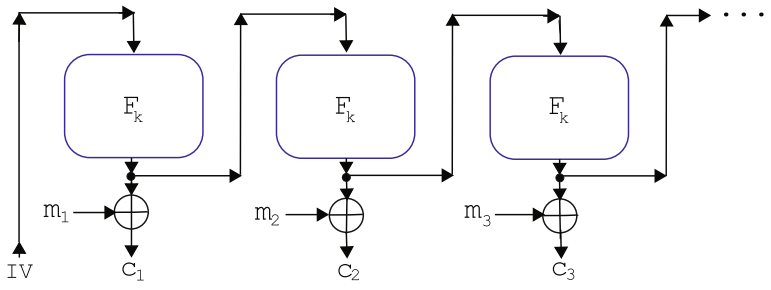
4.3.1.4 Output Feedback Mode

Output feedback mode (OFB) is similar to CFB in that it uses a block cipher to generate a pseudo-random stream that is Xor-ed with the message. The starting point for this stream is a random IV of length n (the block length of the cipher) and the subsequent stream members are obtained by successively applying the forward cipher function so that, in contrast to CFB, the key stream is generated independently from the ciphertext blocks. In OFB mode the IV need not be unpredictable and hence it need not be random but it must be a *nonce* (in the cryptographic sense), i.e., it must be unique to each execution of the encryption operation with the same key (we discuss this aspect below). However, choosing it at random is a good way to ensure that the IV is, with high probability, not repeated across encryptions with the same key. The encryption algorithm can be described as follows:

Algorithm 4.7. OFB encryption.

Input: An l-block message $m = m_1 \dots m_l$, and a block cipher key k.
Output: The ciphertext $\mathbf{Enc}(k, m) = c_1 \dots c_l$.

```
Initialization:
    IV ← {0, 1}^n, chosen at random.
Key Stream Computation:
    r_0 := IV;
    for i from 1 to l do
        r_i := F_k(r_{i−1})
    end do;
Encryption:
    for i from 1 to l do
        c_i := m_i ⊕ r_i
    end do;
    return c_1 ... c_l.
```



OFB Encryption

The decryption algorithm is practically the same, except that now the roles of plaintext and ciphertext are interchanged and the IV is given as part of the ciphertext.

Observe that, in this mode, no padding is necessary. If the length of the message is not a multiple of the block length, the last block may be a partial block of length

Algorithm 4.8. OFB decryption.

Input: An IV, a ciphertext $c = c_1 \dots c_l$, and a block cipher key k.
Output: The plaintext $\mathbf{Dec}(k, c) = m_1 \dots m_l$.

```
Key Stream Computation:
    r_0 := IV;
    for i from 1 to l do
        r_i := F_k(r_{i-1})
    end do;
Decryption:
    for i from 1 to l do
        m_i := c_i ⊕ r_i
    end do;
    return m_1 ... m_l.
```

$t < n$ which, in this case, is Xor-ed with the partial block formed by the first t bits of r_l. We also remark that there is a variant of OFB mode in which, if n is the block size of the underlying block cipher, a shorter block length $t < n$ is specified (often $t = 1$ or $t = 8$) and encryption and decryption proceed as indicated above except that, for $i > 0$, the ith block in the keystream consists of the t first bits of r_i.

As already mentioned, in OFB mode the IV must be a nonce, i.e., different for each encryption. If the same IV is used for another encryption with the same key then the same stream $\{r_i\}$ would be obtained. This would allow an adversary to obtain a lot of information about plaintexts by simply Xor-ing the ith ciphertext blocks of both encryptions, because in this case $c_i \oplus c'_i = (m_i \oplus r_i) \oplus (m'_i \oplus r_i) = m_i \oplus m'_i$. Thus the adversary would know the result of Xor-ing two plaintext blocks and the attack described in [64] would be applicable. Moreover, an adversary who knows a plaintext block, say m_i, can easily find r_i by computing $r_i = m_i \oplus c_i$ and, if the same IV is used again, the adversary would know r_i and would be able to decrypt the ith ciphertext block of the new message.

One disadvantage of CBC and CFB modes is, as we have seen, error propagation. Another is that these modes are not really adequate for applications in which real time encryption/decryption is required because applying F_k is usually too costly for real time. OFB mode addresses these disadvantages because the key stream is generated independently from the ciphertext blocks. On the one hand, this implies that there is no error propagation, since an error in the transmission of a ciphertext block affects only that particular block, a characteristic that makes this mode suitable for transmissions through very noisy channels (such as, for example, transmissions from satellites). On the other hand, the key stream may be precomputed prior to ciphertext transmission and only the Xor operation—which is very fast—must be carried out during the encryption/decryption process, making the mode suitable for real time applications.

OFB mode has good security properties as it can also be shown to be CPA secure if the IV is randomly chosen and the underlying block cipher is a pseudo-random

permutation. In this mode the decryption function of the block cipher is not used and this allows the block cipher to be replaced by a one-way function in some applications.

OFB mode is highly vulnerable to an attack by an active adversary who knows a plaintext block m_i corresponding to the ciphertext block c_i. The adversary may then choose another plaintext block x and replace the ciphertext block c_i by $c_i' = x \oplus m_i \oplus c_i$. Then the honest user will decrypt c_i' by computing $c_i' \oplus r_i = x \oplus m_i \oplus c_i \oplus r_i = x \oplus m_i \oplus m_i = x$. The best way to prevent this type of attack is by using some authentication scheme, as we will see in the next chapter; such a scheme would alert the honest user that the ciphertext has been tampered with. In fact, as a general rule to defend against active attacks, encryption should always be accompanied by authentication.

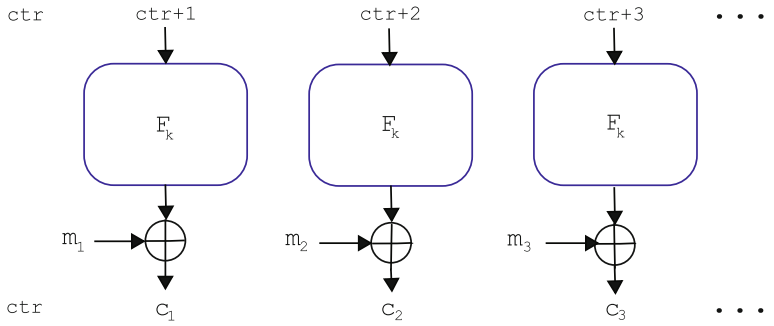
4.3.1.5 Counter Mode

Counter mode (CTR) is similar to OFB in that a stream of input blocks is generated and these blocks are Xor-ed with the message blocks to produce the ciphertext. The stream is generated by considering a sequence of *counters*, which are all distinct, and applying the forward cipher function to these counters. The counters are obtained from an initial counter (called also an IV since it plays a similar role to the IV of other modes) by regarding them as (the binary expansions of) integers and generating each new counter by adding 1 mod 2^n, where n is the block length of the underlying block cipher, to the preceding one (hence the "counter" name). There are different versions of CTR depending on how the initial counter is generated. One possibility is to choose it at random in $\{0, 1\}^n$ and another is to use a non-random nonce but, in any case, all the counter values must always be distinct across all the messages encrypted with a given key. The CTR encryption algorithm is the following:

Algorithm 4.9. CTR encryption.

Input: An l-block message $m = m_1 \dots m_l$, and a block cipher key k.
Output: The ciphertext $\mathbf{Enc}(k, m) = c_1 \dots c_l$.

```
Initialization:
   ctr ← {0, 1}^n, chosen at random.
Key Stream Computation:
   r_0 := ctr;
   for i from 1 to l do
      r_i := F_k(ctr + i)
   end do;
Encryption:
   for i from 1 to l do
      c_i := m_i ⊕ r_i
   end do;
   return c_1 ... c_l.
```



CTR Encryption

The decryption algorithm is very similar:

Algorithm 4.10. CTR decryption.

Input: An initial counter ctr, a ciphertext $c = c_1 \dots c_l$, and a block cipher key k.
Output: The plaintext $\mathbf{Dec}(k, c) = m_1 \dots m_l$.

Key Stream Computation:

$r_0 := ctr$;
for i from 1 to l do
$r_i := F_k(ctr + i)$
end do;

Decryption:

for i from 1 to l do
$m_i := c_i \oplus r_i$
end do;
return $m_1 \dots m_l$.

Note that an adversary who observes a counter being repeatedly used in encryptions with the same key can obtain the result of Xor-ing two plaintext blocks (corresponding to these repeated counters) by just Xor-ing the corresponding ciphertext blocks and, as we have already mentioned, this may give a lot of information about these plaintexts.

CTR mode has several advantages that make it the mode of choice in many applications and it looks likely that it will surpass CBC as the most commonly used mode. For example, both encryption and decryption are parallelizable and the keystream blocks can be generated from the initial counter before starting the transmission. Moreover, it is very simple, it does not use the cipher decryption function and does not require padding as the last block can be a partial block similarly to OFB mode. Also, it allows random access during decryption, i.e., the ith block of ciphertext can be decrypted independently of the remaining ones. Even more importantly, CTR has very good security properties and, as shown in Sect. 4.3.2, if the underlying block cipher is a pseudo-random function, the randomized version of CTR described above is CPA secure.

Exercise 4.7 Suppose that, after encryption with a mode of operation, an IV and a ciphertext were transmitted and, in the process, one bit was flipped. Determine how much plaintext one can expect to recover correctly in all the possible cases obtained by assuming that the flipped bit was either (1) the first bit of the IV or (2) the first bit of the first ciphertext block, and the mode used was one of the following: (a) CBC, (b) OFB, (c) CTR.

4.3.1.6 Final Remarks on Confidentiality Modes

Observe that the modes that use an IV, namely, all modes except ECB, require knowledge of the IV value for decryption. We have regarded the IV (or the initial counter in the case of CTR mode) as an input of the decryption algorithm different from the ciphertext but, of course, in practice the IV may be considered as part of the ciphertext—for example, as the first block—and then the ciphertext input will consist only of the ciphertext and the key as in the general definition of encryption scheme.

We have mentioned that CBC, CFB, OFB and CTR modes (with a randomly chosen IV) give rise to CPA secure encryption schemes when the underlying block cipher is assumed to be a pseudo-random permutation. In the next section we prove this for CTR mode and a similar argument can be used for the other modes. ECB mode, in contrast, does not even give an IND-EAV secure scheme if no restrictions are imposed on the message space.

Exercise 4.8 Prove that if the message space consists of one-block messages and the underlying block cipher is a pseudo-random permutation, then ECB mode defines an IND-EAV secure scheme. (Hint: Given an adversary $\mathcal{A}$ attacking the IND-EAV security of the scheme, use it—as in the proof of Theorem 4.1 below—to construct a distinguisher for the distinguishing experiment in Definition 4.2.)

4.3.2 A CPA Secure Encryption Scheme

We next prove that CTR mode, when used with a randomly chosen initial counter and a key generation algorithm that chooses keys uniformly at random, defines a CPA secure scheme assuming that the underlying block cipher F is a pseudo-random function. This is an important result and its proof illustrates well the kind of reductions that are made to obtain CPA security from a pseudo-randomness hypothesis:

Theorem 4.1 *If the underlying block cipher F is a pseudo-random function then CTR mode with a random initial counter is CPA secure.*

Proof For simplicity, we will also denote by CTR the encryption scheme obtained from the block cipher F by using CTR mode with a random initial counter. The idea of the proof is to consider the ideal encryption scheme iCTR obtained by using a

truly random function $f : \{0,1\}^n \to \{0,1\}^n$ instead of the forward cipher function F_k and to prove that this scheme is CPA secure. The scheme obtained this way is not efficient because there is no efficient way to describe the random function. The function might be represented by a table storing the n-bit value of $f(x)$ for each $x \in \{0,1\}^n$, requiring a total of $n \cdot 2^n$ bits of storage and this description has exponential length in n. However, this is irrelevant for our purpose and the analysis of the idealized scheme lets us isolate the possible weaknesses of CTR mode itself, since we are assuming an ideal block cipher. Thus, proving that this idealized scheme is CPA secure shows that CTR mode is good from a security point of view and it only remains to show, by exploiting the pseudo-randomness of F, that CPA security is preserved when we replace f by F. This is a consequence of the fact that if an adversary $\mathcal{A}$ attacking the CTR scheme (with underlying block cipher F) has non-negligible advantage then it can be used as a subroutine by a distinguisher that will be able to distinguish F from a random f also with non-negligible advantage. The distinguisher D runs $\mathcal{A}$ using its own queries to the oracle to answer $\mathcal{A}$'s queries—so that $\mathcal{A}$ acts exactly as in experiment $\mathbf{PrivK}^{ind-cpa}_{\mathcal{A},\,\mathrm{CTR}}(n)$—and observes whether $\mathcal{A}$ succeeds or not. In the first case, D guesses that its oracle is a pseudo-random function and, otherwise, that it is a random function.

So let us prove that the idealized scheme iCTR is CPA secure. The key generation algorithm for this scheme takes as input a security parameter 1^n and outputs a key, which is now just a random function $f \leftarrow (\{0,1\}^n)^{\{0,1\}^n}$, while the encryption algorithm uses f instead of F_k. Let $\mathtt{ctr}_0$ be the initial random value of the counter used to encrypt the challenge ciphertext in the $\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n)$ experiment and let $\mathtt{ctr}_i$ be the initial value used when the encryption oracle answers the ith query of $\mathcal{A}$. We assume that $\mathcal{A}$ makes a polynomial number of queries to the oracle and also that the messages m_0, m_1, that $\mathcal{A}$ outputs at the start of the experiment have a polynomial number of blocks (or, equivalently, polynomial length), where the term polynomial is relative to the security parameter 1^n used by the key generation algorithm (here n is the block length of F and, for simplicity, we may also assume that n is the key length of F). Thus we may consider a polynomial upper bound $q = q(n)$ on the number of oracle queries and also on the number of blocks in each encryption, and hence we may suppose that the random function f is applied to

$$\mathtt{ctr}_i + 1, \mathtt{ctr}_i + 2, \ldots, \mathtt{ctr}_i + l_i,$$

where $i = 0$ corresponds to the counter sequence used to obtain the challenge ciphertext and the remaining values of i for which $1 \leq i$ (and that are also $\leq q$) to the counter sequence used in the successive queries of $\mathcal{A}$ to the oracle. If, for each $1 \leq j \leq l_0$, the jth block of the challenge ciphertext was obtained by using a value of the counter equal to $\mathtt{ctr}_0 + j$, which is different from $\mathtt{ctr}_i + j'$, for any i in the query sequence and any $1 \leq j' \leq l_i$, then the values $r_t = f(\mathtt{ctr}_0 + t)$, for $t = 1, \ldots, l_0$ are, in the view of $\mathcal{A}$, randomly and uniformly distributed in $\{0,1\}^n$ since f was not applied to any of these values during $\mathcal{A}$'s oracle queries. Thus these values yield a one-time pad when they are Xor-ed with the message blocks and hence

the probability that $\mathcal{A}$ succeeds in the $\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n)$ experiment is exactly $1/2$ in this case.

Suppose now that, on the contrary, there exist indices i, $j \leq l_0$ and $j' \leq l_i$ such that $\mathtt{ctr}_0 + j = \mathtt{ctr}_i + j'$. Then the preceding argument does not apply because the adversary might deduce the value $f(\mathtt{ctr}_0 + j)$ from its ith oracle query and Xor it with the corresponding block of the challenge ciphertext to determine which of the two messages m_0, m_1 was encrypted. We do not need to know the exact probability of success of $\mathcal{A}$ in case this happens, and we may assume that it is very high (even equal to 1). We can show that this case happens with negligible probability and hence the advantage of $\mathcal{A}$ in experiment $\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n)$ will be negligible. Let us give an upper bound on the probability that a repeated value of the counter occurs. For this we may assume that i varies from 1 to q and that $l_i = q$ for all $i = 0, \ldots, q$. Then we want to compute the probability that, for some $i, j \in \{1, \ldots q\}$, $\mathtt{ctr}_0 + j \in [\mathtt{ctr}_i + 1, \mathtt{ctr}_i + q]$. This is equivalent to $\mathtt{ctr}_0 \in [\mathtt{ctr}_i - q + 1, \mathtt{ctr}_i + q - 1]$ for some $i \in \{1, \ldots q\}$. Each of these intervals has length $2q - 1$ and there are q of them so, assuming that these intervals are disjoint—which maximizes the probability of a repeated counter value—we have that the total number of values of $\mathtt{ctr}_0$ that produce a repeated counter value is $\leq 2q^2$. Since all the $\mathtt{ctr}_i$ are randomly chosen, the probability that $\mathtt{ctr}_0$ is equal to one of these values is $\leq 2q^2/2^n = q^2/2^{n-1}$. Therefore, the probability that a repeated counter occurs is bounded by $q^2/2^{n-1}$. If we denote by $\mathtt{Rep}$ the event that a repeated counter value occurs and by $\mathtt{NoRep}$ its complement, then by Proposition 2.1 we have that:

$$\begin{aligned}\Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n) = 1) = {} & \Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n) = 1 | \mathtt{Rep}) \cdot \Pr(\mathtt{Rep}) \\ & + \Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n) = 1 | \mathtt{NoRep}) \cdot \Pr(\mathtt{NoRep}).\end{aligned}$$

We have just proved that $\Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n) = 1 | \mathtt{NoRep}) = 1/2$ and $\Pr(\mathtt{Rep}) \leq q^2/2^{n-1}$. Combining these bounds with the obvious bounds $\Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n) = 1 | \mathtt{Rep}) \leq 1$ and $\Pr(\mathtt{NoRep}) \leq 1$ and substituting these values in the previous equality, we obtain the bound:

$$\Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n) = 1) \leq \frac{1}{2} + \frac{q(n)^2}{2^{n-1}}.$$

Since $q(n)$ is polynomial in n, $q^2/2^{n-1}$ is a negligible function and hence the idealized scheme iCTR is CPA secure according to Definition 3.24.

Next we turn to the counter scheme CTR and we use the fact that iCTR is CPA secure to prove that CTR is CPA secure too. Let $\mathcal{A}$ be an adversary attacking the CPA security of CTR. We construct a distinguisher D that, using $\mathcal{A}$ as a subroutine, tries to determine whether a given function f is equal to some F_k for a randomly chosen key or is randomly chosen in $(\{0,1\}^n)^{\{0,1\}^n}$ (see Definition 4.2) and we compute

the distinguisher advantage $Adv^{prf}_{D,F}(n)$ as a function of the advantages of $\mathcal{A}$ in the IND-CPA experiments for iCTR and CTR. D runs as follows:

- A bit b_1 is chosen and D is given oracle access to a function $f : \{0,1\}^n \to \{0,1\}^n$ which is either $f := F_k$ for $k \leftarrow \{0,1\}^n$ if $b_1 = 1$ or a random function otherwise.
- D runs $\mathcal{A}$ on input 1^n and when $\mathcal{A}$ outputs messages $m_0, m_1 \in \{0,1\}^n$, D chooses a random bit $b \leftarrow \{0,1\}^n$ and returns to $\mathcal{A}$ the challenge ciphertext c computed as follows:
  - $\mathtt{ctr}_0 \leftarrow \{0,1\}^n$.
  - If the blocks of m_b are $m_{b1}, \ldots, m_{bl}$, D queries the oracle and obtains the values $r_i := f(\mathtt{ctr}_0 + i)$, for $i = 1, \ldots, l$.
  - c is the ciphertext whose blocks are $c_i := r_i \oplus m_{bi}$.
- $\mathcal{A}$'s oracle queries are answered by D by returning a ciphertext computed as in the preceding item.
- When $\mathcal{A}$ outputs a bit b', D outputs a bit b'_1 defined as $b'_1 = 1$ if $b' = b$ and $b'_1 = 0$ otherwise.

Let us now analyze D's advantage in the distinguishing experiment. If D is given oracle access to a random function from $\{0,1\}^n$, then the view of $\mathcal{A}$ when run as indicated above is identical to the view of $\mathcal{A}$ in the experiment $\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n)$ since a random function is used to 'encrypt' the messages. On the other hand, if D is given oracle access to some F_k, for a randomly chosen k, then the view of $\mathcal{A}$ is the same as in experiment $\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{CTR}}(n)$ because when answering $\mathcal{A}$'s oracle queries, D computes the answer in the same way as when encrypting the message with CTR and key k.

Now let $\varepsilon(n) = \Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{CTR}}(n) = 1) - \frac{1}{2}$ and $\nu(n) = \Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n) = 1) - \frac{1}{2}$. Then $\varepsilon(n)$ and $\nu(n)$ are the advantages of $\mathcal{A}$ in the IND-CPA experiments corresponding to CTR and iCTR, respectively, and we have just seen that $\nu(n) \leq \frac{q(n)^2}{2^{n-1}}$. Since the latter function is negligible, we have that $\nu(n)$ is negligible too and our goal is to show that $\varepsilon(n)$ is also negligible. Since $b'_1 = 1$ precisely when $\mathcal{A}$ succeeds, we have that $\Pr(b'_1 = 1 | b_1 = 1) = \Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{CTR}}(n) = 1) = \frac{1}{2} + \varepsilon(n)$ and $\Pr(b'_1 = 1 | b_1 = 0) = \Pr(\mathbf{PrivK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{iCTR}}(n) = 1) = \frac{1}{2} + \nu(n)$. The advantage of D in the distinguishing experiment (see Definition 4.2) is equal to the absolute value of the difference between these two values, namely:

$$Adv^{prf}_{D,F}(n) = |\Pr(b'_1 = 1 | b_1 = 1) - \Pr(b'_1 = 1 | b_1 = 0)| = |\varepsilon(n) - \nu(n)|$$

This advantage is negligible because we are assuming that F is a pseudo-random function and, since $\nu(n)$ is also negligible, we see that so is $\varepsilon(n)$. Therefore, CTR is CPA secure. □

Remarks 4.3

1. The preceding theorem completes the picture of the reasons why block ciphers and modes of operation have such a relevant role in modern cryptography. The main idea may be summarized as follows: block ciphers plus modes of operation is the best way known so far to obtain private-key encryption schemes that are both very efficient in practice and likely secure. We do not have security assurance because we do not have a proof that a block cipher is a pseudo-random permutation but this shows that, leaving aside attacks against implementations (such as side-channel attacks which are, of course, very important in practice), the natural way to attack such a scheme is to attack the pseudo-randomness of the block cipher.
2. From a practical point of view, the proof of the preceding theorem gives us some relevant information about how the expected level of security of the scheme varies as a function of the block size n. The advantage $\varepsilon(n)$ of the adversary is close to $\nu(n)$ (since the difference between them is a negligible function) and $\nu(n)$ is bounded by the function $q(n)^2/2^{n-1}$. When a 128-bit block size is used, as in the case of AES and, say, 2^{32} queries about 2^{32}-block messages are made (so that the adversary advantage is maximized), the value of the latter function is $1/2^{63}$, which is close to 10^{-19}. The adversary advantage is then close to this number and hence we may conclude that the AES-based CTR encryption scheme is secure against such an adversary (provided that the AES function is pseudo-random). However, suppose that we use a block cipher with a 64-bit block length (such as, for example, DES). Then the value of the function $q(n)^2/2^{n-1}$ is equal to 2 for an adversary making 2^{32} queries and hence the bound on the advantage of such an adversary becomes meaningless. Each message queried would be $64 \cdot 2^{32} = 2^{38}$ bits long, which is just above 34 GB and hence such an attack is not unimaginable taking into account today's storage capabilities. This suggests that using a block cipher with a 64-bit block length or less may not be secure in practice.

Exercise 4.9 Give an informal argument showing that, if the underlying block cipher is a pseudo-random permutation, then OFB mode gives a CPA secure encryption scheme if and only if the IV is never repeated under the same key except possibly with negligible probability. (Hint: Show that if the IV is never repeated, then the probability of a repeated value in the keystreams arising from adversary queries is negligible and argue as in the proof of Theorem 4.1.)

4.3.2.1 The Modes of Operation Do Not Yield CCA Secure Schemes

We have not seen so far any example of a CCA secure scheme and so we might wonder whether the CPA secure CTR scheme is also CCA secure. However, it is easy to show that this is not the case. If $(\mathtt{ctr}, c)$ is the challenge ciphertext corresponding to the message m (where `ctr` denotes the initial counter value), then a CCA adversary may construct a new ciphertext by just flipping one bit—say, the ith bit—of c and query the decryption oracle about this new ciphertext. Since this ciphertext is different from the challenge ciphertext, the oracle will answer the query and return a message m'

which, as is easily seen, is the same as m with the ith bit flipped. This allows the adversary to distinguish which of the two messages submitted was encrypted and to succeed in the $\mathbf{PrivK}^{ind\text{-}cca}$ experiment with probability 1 (since the adversary may submit two messages that differ in at least one bit distinct from the ith bit). In fact, this attack is even stronger because it allows the adversary to recover any encrypted message m with just one query, by flipping the ith bit of the message m'.

Exercise 4.10 Show that CBC and OFB modes do not define CCA secure encryption schemes even assuming that the underlying block cipher is a pseudo-random function.

4.4 AES in Maple

In Sect. 4.2.2 we have given Maple implementations of the S-box and the Key Expansion algorithm. We are now going to give a full implementation of AES in Maple as a preliminary step to implementing the modes of operation. A more detailed Maple version, which also includes the possibility of performing statistical tests on the diffusion properties of AES, as well as all the modes of operation, can be found in [93].

4.4.1 AES Operations in Maple

4.4.1.1 `SubBytes` and its Inverse

We start with the `SubBytes` operation and its inverse. We could map the above-defined `SB` function to the entries of the state array but, to increase speed, we will instead replace the state entries by looking directly at the `SBox` and `InvSBox` tables. The input of these functions is the state array, given as a two-dimensional Maple `Array` of type 0..3, 0..3 and the output is the modified version of the state.

```
> SubBytes := proc(state)
   local i, j;
   for i from 0 to 3 do
     for j from 0 to 3 do
       state[i, j] := SBox[state[i, j]]
     end do
   end do;
   state
   end proc:

> InvSubBytes := proc(state)
   local i, j;
   for i from 0 to 3 do
     for j from 0 to 3 do
       state[i, j] := InvSBox[state[i, j]]
     end do
   end do;
   state
   end proc:
```

4.4.1.2 `ShiftRows` and its Inverse

The `ShiftRows` operation consists of shifting the rows of the state array by 0, 1, 2, 3 positions to the left, i.e., each row is shifted by the amount given by its index. The inverse operation just shifts rows right by the same offsets. These operations are not implemented "in place" as we need to use a copy of the state array inside the corresponding function.

```
> ShiftRows := proc(state)
   local st, i, j;
   st := Array(state);
   for i to 3 do
     for j from 0 to 3 do
       st[i, j] := state[i, (i+j mod 4)]
     end do
   end do;
   st
   end proc:

> InvShiftRows := proc(state)
   local st, i, j;
   st := Array(state);
   for i to 3 do
     for j from 0 to 3 do
       st[i, j] := state[i, (j-i mod 4)]
     end do
   end do;
   st
   end proc:
```

4.4.1.3 `MixColumns` and its Inverse

The `MixColumns` operation makes heavy use of multiplication in $\mathbb{F}_{2^8}$ and hence, for efficiency, it is convenient to implement this multiplication by means of a lookup table. Here we use the function `mult256` that was defined in Sect. 2.8.4:

```
> multtable := Array(0 .. 255, 0 .. 255, (i, j) -> mult256(i,j)):
```

The next function implements the `MixColumns` operation. The input is the state array and the output the modified version of the state obtained by carrying out the matrix multiplication `M·state`, where `M` is the matrix with coefficients in $\mathbb{F}_{2^8}$ given by `M = Matrix([seq(ListTools:-Rotate([2,3,1,1],-i), i=0..3)])`.

For efficiency, we implement this matrix multiplication directly by using the function `multtable` and the addition table of $\mathbb{F}_{2^8}$, `bitXortable` (given in Sect. 2.8):

```
> MixColumns := proc(state)
  local st, i;
  st := Array(0..3, 0..3);
  for i from 0 to 3 do
    st[0,i]:=bitXortable[multtable[state[0,i],2], bitXortable[multtable[state[1,i],3],
                bitXortable[state[2,i], state[3,i]]]]
  end do;
  for i from 0 to 3 do
    st[1,i]:=bitXortable[state[0,i], bitXortable[multtable[state[1,i],2],
                bitXortable[multtable[state[2, i], 3], state[3, i]]]]
```

```
  end do;
  for i from 0 to 3 do
    st[2,i]:=bitXortable[state[0,i], bitXortable[state[1,i],
                bitXortable[multtable[state[2,i],2], multtable[state[3,i],3]]]]
  end do;
  for i from 0 to 3 do
    st[3,i]:=bitXortable[multtable[state[0,i],3], bitXortable[state[1,i],
                bitXortable[state[2,i], multtable[state[3,i],2]]]]
  end do;
  st
  end proc:
```

The inverse operation is similarly defined, multiplying by the inverse matrix over $\mathbb{F}_{2^8}$ of the one used in the function `MixColumns`. This inverse is given by `Matrix([seq(ListTools:-Rotate([14,11,13,9],-i),i=0..3)])`. As before, we define this multiplication directly using the operation tables of $\mathbb{F}_{2^8}$.

```
> InvMixColumns := proc(state)
  local st, i;
  st := Array(0..3, 0..3);
  for i from 0 to 3 do
    st[0,i]:=
    bitXortable[multtable[state[0,i],14],bitXortable[multtable[state[1,i],11],
    bitXortable[multtable[state[2,i],13], multtable[state[3,i],9]]]]
  end do;
  for i from 0 to 3 do
    st[1,i]:=
    bitXortable[multtable[state[0,i],9],bitXortable[multtable[state[1,i],14],
              bitXortable[multtable[state[2,i],11], multtable[state[3,i],13]]]]
  end do;
  for i from 0 to 3 do
    st[2,i]:=
    bitXortable[multtable[state[0,i],13],bitXortable[multtable[state[1,i],9],
              bitXortable[multtable[state[2,i],14], multtable[state[3,i],11]]]]
  end do;
  for i from 0 to 3 do
    st[3,i]:=
    bitXortable[multtable[state[0,i],11],bitXortable[multtable[state[1,i],13],
              bitXortable[multtable[state[2,i],9], multtable[state[3,i],14]]]]
  end do;
  st
  end proc:
```

4.4.1.4 AddRoundKey

The `AddRoundKey` operation is the result of Xor-ing the bytes in the state matrix with those in the corresponding round subkey of the expanded key (i.e., the new state matrix is obtained by adding over $\mathbb{F}_{2^8}$ the round subkey matrix to the old state matrix). We do not define the inverse operation because `AddRoundKey` is its own inverse.

```
> AddRoundKey := proc(roundk, state)
  local i, j;
  for i from 0 to 3 do
    for j from 0 to 3 do
      state[i, j] := bitXortable[state[i, j], roundk[i, j]]
    end do
  end do;
  state
  end proc:
```

4.4.2 AES Encryption and Decryption

We now give Maple implementations of AES forward and inverse cipher functions.

4.4.2.1 AES Rounds

First we define an AES round. The input is a round subkey (given as a 0..3, 0..3 array) and a state array, and the output is a modified state.

```
> Round := (roundk,state) ->
           AddRoundKey(roundk,MixColumns(ShiftRows(SubBytes(state)))):
```

The inverse round, which is used for decryption, is given by the next function:

```
> InvRound := (roundk,state) ->
              InvSubBytes(InvShiftRows(InvMixColumns(AddRoundKey(roundk,state)))):
```

4.4.2.2 The Forward Cipher Function

The next function implements the forward cipher function of AES. The input is an expanded key in the format produced by `KeyExpansion` and a plaintext block given as a state array. The output is the state array with the corresponding ciphertext block.

```
> AESEncrypt := proc(expandedkey, state)
  local st, Nr, i;
  st := Array(state);
  Nr := nops(expandedkey)-1;
  st := AddRoundKey(expandedkey[1], st); #The 0th round
  for i from 2 to Nr do
    st := Round(expandedkey[i], st) #Rounds 1 to Nr-1
  end do;
  AddRoundKey(expandedkey[Nr+1], ShiftRows(SubBytes(st))) #The last round
  end proc:
```

4.4.2.3 The Inverse Cipher Function

The inverse cipher function takes as input an expanded key and a state array (a ciphertext block) and outputs the state with the corresponding plaintext block.

```
> AESDecrypt := proc(expandedkey, state)
  local Nr, st, i;
  st := Array(state);
  Nr := nops(expandedkey)-1;
  st := InvSubBytes(InvShiftRows(AddRoundKey(expandedkey[Nr+1], st)));
  for i from 0 to Nr-2 do
    st := InvRound(expandedkey[Nr-i], st)
  end do;
  AddRoundKey(expandedkey[1], st)
  end proc:
```

4.4.2.4 Encryption and Decryption of Lists of Bytes

We are now going to give a Maple function to encrypt or decrypt a list of bytes such as the one that can be obtained by reading a file from disk or by converting a text message to the list of corresponding 8-bit ASCII codes. Since an AES block consists of 16 bytes, the length of the list must be a multiple of 16 and we will see later how messages can be padded to produce an integral number of blocks. The input is a key (given either as a list of bytes or as a hexadecimal string), a list of bytes (as integers in the 0..255 range), and an optional name that can be either 'encrypt' or 'decrypt' with the former as default. The output is the list of bytes corresponding to the encrypted or decrypted input blocks. We warn the reader that 'encrypt' and 'decrypt' here are just shorthand for the application of the forward cipher function and the inverse cipher function, respectively. As we have seen when discussing the modes of operation, these functions are seldom applied directly to plaintexts in order to obtain the corresponding ciphertexts (and, if they are, the resulting scheme is not secure, as happens with ECB mode).

```
> AES := proc(key, bytelist::list, action::name:=encrypt)
  local k, ek, states, len, i;
  k := checkkey(key);
  ek := KeyExpansion(k);
  states := [ListTools:-LengthSplit(bytelist, 16)];
  len := nops(states)-1;
  states := Array(0..len, map(x -> Array(0..3, 0..3, (i,j) -> x[i+4*j+1]), states));
  if action = encrypt then
    for i from 0 to len do
      states[i]:=AESEncrypt(ek, states[i])
    end do;
  elif action = decrypt then
    for i from 0 to len do
      states[i]:=AESDecrypt(ek, states[i])
    end do;
  else
    error "encrypt or decrypt expected"
  end if;
  states := map(x -> ArrayTools:-Alias(x, [16]), states);
  [seq(seq(states[i][j], j = 1 .. 16), i = 0 .. len)]
  end proc:
```

Example 4.2 Let us consider the 256-bit key and the 128-bit plaintext given in [73, Example C.3]. The result of encrypting and decrypting the plaintext is:

```
> k := "000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f":
  m := "00112233445566778899aabbccddeeff":

> c := bytestohexstring(AES(k, hexstringtobytes(m)));
                    "8ea2b7ca516745bfeafc49904b496089"

> bytestohexstring(AES(k, hexstringtobytes(c), decrypt));
                    "00112233445566778899aabbccddeeff"
```

4.4.2.5 The Known Answer Tests

Next, in order to test the AES implementation, we introduce the "Known Answer Tests" of [12] and we begin with the variable text test which consists of encrypting a sequence of plaintexts using a fixed key. Since the plaintext and the key are predetermined, the input of this test is simply the key length (in order to choose among three possible keys of different lengths) but, as the output is too long to be printed here, we introduce an additional input parameter `len` that allows specification of the number of ciphertext blocks to be printed. The function that implements this test is the following:

```
> VarTxtTest := proc(keylength::{128, 192, 256}, len := keylength)
  local i, t;
  for i to len do
    t := ListTools:-Reverse(convert([0$(128-i), 1$i], base, 2, 256));
    printf("%s|%s\n", bytestohexstring(t),
           bytestohexstring(AES([0$(iquo(keylength,8))],t)))
  end do;
  printf("\n")
  end proc:
```

Example 4.3 We print the first eight ciphertext blocks obtained with a 256-bit key. The output can be compared to that appearing in [12]:

```
> VarTxtTest(256, 8);
80000000000000000000000000000000|ddc6bf790c15760d8d9aeb6f9a75fd4e
c0000000000000000000000000000000|0a6bdc6d4c1e6280301fd8e97ddbe601
e0000000000000000000000000000000|9b80eefb7ebe2d2b16247aa0efc72f5d
f0000000000000000000000000000000|7f2c5ece07a98d8bee13c51177395ff7
f8000000000000000000000000000000|7818d800dcf6f4be1e0e94f403d1e4c2
fc000000000000000000000000000000|e74cd1c92f0919c35a0324123d6177d3
fe000000000000000000000000000000|8092a4dcf2da7e77e93bdd371dfed82e
ff000000000000000000000000000000|49af6b372135acef10132e548f217b17
```

Next, we introduce the variable key test, which is very similar to the preceding one, except that now the plaintext is fixed and the key varies.

```
> VarKeyTest := proc(keylength::{128, 192, 256}, len := keylength)
   local i, k;
   for i to len do
     k := ListTools:-Reverse(convert([0$(keylength-i), 1$i], base, 2, 256));
     printf("%s|%s\n", bytestohexstring(k), bytestohexstring(AES(k, [0$16])))
   end do;
   printf("\n")
   end proc:
```

Example 4.4 The output corresponding to the first 8 keys in the 192-bit `VarKeyTest` is the following (compare with [12]):

```
> VarKeyTest(192, 8);
800000000000000000000000000000000000000000000000|de885dc87f5a92594082d02cc1e1b42c
c00000000000000000000000000000000000000000000000|132b074e80f2a597bf5febd8ea5da55e
e00000000000000000000000000000000000000000000000|6eccedf8de592c22fb81347b79f2db1f
f00000000000000000000000000000000000000000000000|180b09f267c45145db2f826c2582d35c
f80000000000000000000000000000000000000000000000|edd807ef7652d7eb0e13c8b5e15b3bc0
fc00000000000000000000000000000000000000000000000|9978bcf8dd8fd72241223ad24b31b8a4
fe00000000000000000000000000000000000000000000000|5310f654343e8f27e12c83a48d24ff81
ff00000000000000000000000000000000000000000000000|833f71258d53036b02952c76c744f5a1
```

Exercise 4.11 Implement the GFSbox and KeySbox tests of [12] and compare the results obtained to the known answers given there.

4.5 Some Modes of Operation in Maple

Here we give Maple implementations of two of the modes discussed in the previous section: OFB and CTR. Other modes can be similarly implemented but we will leave this task as an exercise.

4.5.1 OFB and CTR in Maple

OFB and CTR modes are similar in that encryption proceeds by Xor-ing the plaintext blocks with the blocks of a key stream and hence we will use a unique function to implement both.

4.5.1.1 A Couple of Preliminary Functions

In addition to the previous functions `checkkey`, `KeyExpansion` and `AESEncrypt`, which were used in the AES implementation, we will need another couple of functions that we shall define before giving the modes function. The first is just a variant—which is more convenient for our purposes—of the `BitXor` operation given in Sect. 2.8. This variant, called `BitXorAL` calls the previous function `bitXortable` to compute the bitwise Xor of a 0..3, 0..3 array `a` with a list `l` of no more than 16 terms, where the array is passed as the first argument and the list as the second. If `t` is the length of `l`, then the output is the list of length `t` obtained by Xor-ing the elements of `l` with the first `t` elements of `a` with respect to `Fortran_order` (in which the entries of the array are accessed in 'column-major' order, i.e., columnwise, see Maple's help for details). Maple's built-in function `ArrayTools:-Alias` is used here to view the 2-dimensional Array as a 1-dimensional Array with this order:

```
> BitXorAL := proc(a::Array, l::list)
  local li, i, v;
  li := l;
  v := ArrayTools:-Alias(a, [16]);
  for i to nops(l) do
    li[i] := bitXortable[v[i], l[i]]
  end do;
  li
  end proc:
```

To implement CTR mode we need a function to generate the counters which, in turn, requires two ingredients. One of them is an algorithm to choose the initial counter block corresponding to each encryption; this initial counter will be the IV and we will later give functions to generate it. The other ingredient is a method to generate the subsequent counters in such a way that their values are never repeated across

all encryptions done with the same key. To achieve this we use an "incrementing function" which is called after encrypting each block in order to obtain the counter for the next block. We will use the so-called 32-bit incrementing function which adds 1 to the counter modulo 2^{32}, where the counter is viewed as a 16-digit number in base 256, in big-endian order, i.e., with the most significant byte first. Therefore, the 32-bit incrementing function only modifies the last four bytes of the counter—that initially are all zero—and cycles among all the possible values of these 4 bytes. We could instead add modulo 2^{128}—as indicated in our previous description of CTR—so that the counter values would cycle among 2^{128} possible ones, but this is not really necessary. The 32-bit version provides for 2^{32} different counter values each of which is used to encrypt one 2^4-byte block. This means that the plaintext can have up to 2^{36} bytes (64 gibibytes, close to 69 gigabytes) without any counter value being repeated during encryption. This is more than enough since Maple's list size limit imposes a maximum plaintext size of $2^{26} - 4$ bytes, i.e., about 67 MB. It is possible to modify the implementation to allow for larger plaintexts but, given other realistic space-time constraints, there would be little point in doing so. Bearing these considerations in mind, the incrementing function is the following:

```
> inc32 := proc(count)
  local i;
  i := 3;
  while count[i, 3] = 255 do
    count[i, 3] := 0;
    i := i-1 mod 4
  end do;
  count[i, 3] := count[i, 3] + 1
  end proc:
```

4.5.1.2 The Low Level Encryption/Decryption Function

The function that implements OFB and CTR modes (for both encryption and decryption) is given next. It has four input parameters which are, in order: the AES key (either a list of bytes or a hex string), the plaintext or ciphertext given as a list of bytes, the name of the mode (either OFB or CTR) and, finally, the IV given either as a list of bytes or a hex string. The message is a list of bytes because this function will be called from other higher-level functions—which will be given later—to encrypt and decrypt messages in other formats. The first three parameters are required and the last, i.e., the one corresponding to the IV, may be omitted. If four arguments are provided, then the function encrypts, otherwise it decrypts (note that the IV is not necessary for decryption as it is included in the ciphertext).

The key should be randomly chosen and the IV may also be randomly chosen, although we will later give an alternative way to generate it.

```
> AESModes := proc(key, bytelist::list, mode::name, iv::list)
  local k, ek, inv, states, len, i, kst, count;
  uses ListTools;
  k := checkkey(key);
  ek := KeyExpansion(k);
  if nargs = 4 then
    inv := checkkey(iv);
```

```
  states := [inv, LengthSplit(bytelist, 16)]
else
  states := [LengthSplit(bytelist, 16)]
end if;
len := nops(states)-1;
states := Array(0 .. len, states);
if mode = OFB then
  kst := Array(0 .. 3, 0 .. 3, (i, j) -> states[0][i+4*j+1]);
  for i to len do
    kst := AESEncrypt(ek, kst);
    states[i] := BitXorAL(kst, states[i])
  end do
elif mode = CTR then
  count := Array(0 .. 3, 0 .. 3, (i, j) -> states[0][i+4*j+1]);
  for i to len do
    states[i] := BitXorAL(AESEncrypt(ek, count), states[i]);
    inc32(count)
  end do
else
  error "Unrecognized mode; either OFB or CTR should be used"
end if;
if nargs = 4 then
  [seq(op(states[i]), i = 0 .. len)]
else
  [seq(op(states[i]), i = 1 .. len)]
end if;
end proc:
```

Example 4.5 Let us do an encryption and decryption in CTR mode, using the preceding function. We take the 0 vector as plaintext block and also as key and as IV, i.e., these three inputs equal to the list consisting of 16 zeros. Then we obtain:

```
> AESModes([0$16], [0$16], CTR, [0$16]);
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 102, 233, 75, 212, 239, 138,
  44, 59, 136, 76, 250, 89, 202, 52, 43, 46]
```

The plaintext is a unique 128-bit block but the ciphertext consists of two blocks: the first one is just the IV (or the initial counter in this case) and the second is the ciphertext proper, i.e., the block that results from encrypting the plaintext block. Now, we decrypt:

```
> AESModes([0$16], %, CTR);
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The next function, `ModesTest`, serves to test the preceding implementation of OFB and CTR modes. In accordance with the usage in [68], where some test vectors are given, all inputs will be hex strings, except the one corresponding to the parameter `mode` which will be a name (OFB or CTR). As in the previous function, the presence or absence of the last argument (the IV) determines whether the function encrypts or decrypts.

```
> ModesTest := proc(key::string, message::string, mode::name, iv::string)
  local messagelist, t, i;
  messagelist := hexstringtobytes(StringTools:-LowerCase(message));
  t := AESModes(key, messagelist, args[3 .. nargs]);
  if nargs = 4 then
    t := map(bytestohexstring, [ListTools:-LengthSplit(t[17 .. -1], 16)]);
    printf("%s\n", "Ciphertext blocks:")
  else
```

```
    t := map(bytestohexstring, [ListTools:-LengthSplit(t, 16)]);
    printf("%s\n", "Plaintext blocks:")
  end if;
  for i to nops(t) do
    printf("%s\n", t[i])
  end do;
  printf("\n")
  end proc:
```

Example 4.6 We compute the ciphertext blocks corresponding to Example F.4.3 (OFB-AES192.Encrypt) in [68]. The key, IV and plaintext in this example are:

```
> k192 := "8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b":
  IV := "000102030405060708090a0b0c0d0e0f":
  pl := "6bc1bee22e409f96e93d7e117393172aae2d8a571e03ac9c9eb76fac45af8e5130c81c46a\
        35ce411e5fbc1191a0a52eff69f2445df4f9b17ad2b417be66c3710":
```

The corresponding ciphertext blocks, which may be compared to those in [68], are then the following:

```
> ModesTest(k192, pl, OFB, IV);
Ciphertext blocks:
cdc80d6fddf18cab34c25909c99a4174
fcc28b8d4c63837c09e81700c1100401
8d9a9aeac0f6596f559c6d4daf59a5f2
6d9f200857ca6c3e9cac524bd9acc92a
```

Now, the decryption test with the IV plus the ciphertext obtained above recovers the plaintext blocks of `pl`:

```
> ModesTest(k192, "000102030405060708090a0b0c0d0e0fcdc80d6fddf18cab34c25909c99a417\
  4fcc28b8d4c63837c09e81700c11004018d9a9aeac0f6596f559c6d4daf59a5f26d9f200857ca6c3\
  e9cac524bd9acc92a", OFB);
Plaintext blocks:
6bc1bee22e409f96e93d7e117393172a
ae2d8a571e03ac9c9eb76fac45af8e51
30c81c46a35ce411e5fbc1191a0a52ef
f69f2445df4f9b17ad2b417be66c3710
```

Example 4.7 We compute the ciphertext blocks corresponding to Example F.5.5 (CTR-AES256.Encrypt) in [68]. The plaintext is the same as in the preceding example; the key and the initial counter are given below, and the ciphertext blocks are computed:

```
> k256 := "603deb1015ca71be2b73aef0857d77811f352c073b6108d72d9810a30914dff4":
   initcounter := "f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff":
   ModesTest(k256, pl, CTR, initcounter);
Ciphertext blocks:
601ec313775789a5b7a7f504bbf3d228
f443e3ca4d62b59aca84e990cacaf5c5
2b0930daa23de94ce87017ba2d84988d
dfc9c58db67aada613c2dd08457941a6
```

The decryption test corresponding to the ciphertext just obtained produces the plaintext blocks of Example F.5.5 (CTR-AES256.Decrypt) as follows:

```
> ModesTest(k256, "f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff601ec313775789a5b7a7f504bbf3d22\
  8f443e3ca4d62b59aca84e990cacaf5c52b0930daa23de94ce87017ba2d84988ddfc9c58db67aada\
  613c2dd08457941a6", CTR);
Plaintext blocks:
6bc1bee22e409f96e93d7e117393172a
ae2d8a571e03ac9c9eb76fac45af8e51
30c81c46a35ce411e5fbc1191a0a52ef
f69f2445df4f9b17ad2b417be66c3710
```

4.5.2 Encryption and Decryption with OFB and CTR

In the previous sections we built encryption and decryption functions for OFB and CTR modes, using AES as block cipher. These functions only accept messages given as lists of bytes and now we are going to define higher level functions that can encrypt and decrypt real messages using these modes of operation. Both OFB and CTR use an IV. In the case of OFB, the IV must be a nonce, i.e., it must be unique for each execution of the mode under a given key. For CTR, the IV (called, in this case, the initial counter) must be chosen so that all counters are different from each other across all of the messages that are encrypted under a given key (in both cases, except possibly with negligible probability). In the case of CTR we have seen in the proof of Theorem 4.1 that non-repetition of counter values ensures that an adversary has negligible advantage in an IND-CPA attack against this mode, assuming that the underlying block cipher is a pseudo-random permutation (similarly, CPA security for OFB with a nonce as IV can be proved as in the first part of the proof of Theorem 4.1, because then the keystream generated behaves like a one-time pad). In the same proof we have also seen that, if the initial counter is randomly chosen, then the probability of a repeated counter value is negligible and, as a consequence, the adversary's advantage in an IND-CPA attack is also negligible. Thus, for both modes, choosing a random IV is sufficient to ensure that the required conditions are satisfied except possibly with negligible probability. However, since generating random IVs requires a physical source of entropy, we are going to give Maple functions that also offer the possibility to generate these initialization vectors in a different way that is sufficient to ensure that, in this restricted environment, the non-repetition conditions mentioned above are satisfied.

Other than choosing it at random, a common technique to obtain a nonce is based on using a timestamp, i.e., a sequence of bytes obtained from the date/time at which the nonce is generated. Since Maple only retrieves the current date/time up to the second, in order to obtain a nonce we will add values obtained from one of Maple's pseudo-random algorithms. The only purpose of these values is to prevent a repetition of the IV value if several encryptions are done within the same second (as the nonce will be generated at encryption time) and so we do not need to worry about the seed being non-random in this case.

Taking into account these remarks, the next function generates a nonce of length n (i.e., a list of n bytes) as follows. Using `StringTools:-FormatTime` with the appropriate format string and then using `sscanf`, a list of six bytes is obtained, corresponding to the last two digits of the year, the month of the year, the day of the month, the hour (in 24-hour clock format), the minute and the second. This sequence of six bytes is then completed with $n - 6$ pseudo-randomly generated bytes. Note that it would be a mistake to initialize the generator each time that `nonce()` is invoked because, since the seed is taken from the system clock, this would allow the possibility of obtaining two equal values in succession. On the other hand, a repeated value could possibly be obtained if several computers are simultaneously used (or

used when their clocks show the same time) to perform encryptions with the same key, but we will assume that this is not the case.

```
> nonce := proc(n::posint)
  local l, r;
  l := sscanf(StringTools:-FormatTime(%y%m%d%k%M%S), %2d%2d%2d%2d%2d%2d);
  r := RandomTools:-Generate(list(nonnegint(range = 255), n-6));
  [op(l), op(r)]
  end proc:
```

The previous function will be optionally used not only for OFB mode but also to set the initial value of the counter in CTR mode. The procedure we are going to follow for this is similar to one suggested in [68]. At the start of each CTR encryption, the initial counter will be obtained by calling `nonce(12)` and completing it with four zeros on the right to make a 16-byte block. The function that selects the IV for OFB and CTR modes is then the following:

```
> selectiv := proc()
  [op(nonce(12)), 0$4]
  end proc:
```

An example of several IVs for OFB or CTR modes is then the following:

```
> for i to 6 do
    selectiv()
  end do;
       [10, 7, 28, 16, 46, 24, 92, 246, 238, 121, 44, 223, 0, 0, 0, 0]
        [10, 7, 28, 16, 46, 24, 5, 225, 186, 43, 99, 37, 0, 0, 0, 0]
       [10, 7, 28, 16, 46, 24, 196, 26, 95, 16, 231, 228, 0, 0, 0, 0]
       [10, 7, 28, 16, 46, 24, 89, 250, 161, 17, 179, 55, 0, 0, 0, 0]
        [10, 7, 28, 16, 46, 24, 170, 82, 24, 89, 92, 59, 0, 0, 0, 0]
       [10, 7, 28, 16, 46, 24, 220, 141, 49, 122, 174, 7, 0, 0, 0, 0]
```

Note that, as these IVs were all generated in the same second, the timestamp parts, i.e., the first six bytes are all the same, but the pseudo-random parts (the next six bytes) take care of making the IVs distinct. Observe also that if actual encryptions are performed and the IV is generated just when performing these encryptions, the timestamp part of each of these IVs would be different if no two encryptions are performed in the same second.

We have already given the incrementing function that will generate the subsequent counter values but we also have to bear in mind that, since the counter is initialized by using the nonce each time the CTR encryption function is called, one has to ensure not only that the value returned by the nonce function never repeats—as already discussed—and that the counter value never repeats during one encryption, but also that no repeated counter values arise during different encryptions with the same key. We leave this as an exercise for the reader.

Exercise 4.12 Show that, in practice (taking into account the maximum size of a Maple list), one will never obtain a repeated counter value during CTR encryptions if the initial counter is selected with `selectiv` and increased with `inc32`.

4.5.2.1 Encrypting and Decrypting Files

Next, we give functions to encrypt and decrypt binary files using AES-based OFB or CTR modes. We use the functions `bytestofile`, `filetobytes` and `checkkey`, given in Appendix A, as well as the previously defined functions `AESModes` and `selectiv`. The input for the encryption function below is the AES key (as a hex string or a list of bytes), the name of the file to be encrypted or decrypted (as a string delimited by double quotes), the name of the file where the result of the encryption/decryption operation is to be written, the mode of operation (a name equal to `OFB` or `CTR` with the latter as default) and three optional keyword parameters. The first of them, `iv` is for the IV and, by default, it selects the IV by calling `selectiv()`, but can also be fed with an externally generated random IV. The parameter `filecheck` is used for file checking (in order to prevent inadvertently overwriting an existing file) and `logoutput` specifies (with the same format as in `bytestofile` and with default value `terminal`) where the output log is to be written. By default, the function checks whether a file of the same name exists in the working directory but this check can be overridden by specifying `filecheck = false`.

```
> AESFileEncrypt := proc(key, filename::string, cfilename::string, mode::name:=CTR,
          {iv:=selectiv(), filecheck::truefalse:=true, logoutput::name:=terminal})
  local k, f;
  k := checkkey(key);
  f := filetobytes(filename);
  f := AESModes(k, f, mode, iv);
  bytestofile(f, cfilename, filecheck, logoutput)
  end proc:
```

The file decryption function is similar, except that now the `iv` parameter is missing because the IV is included in the ciphertext:

```
> AESFileDecrypt := proc(key, cfilename::string, filename::string, mode::name:=CTR,
                         {filecheck::truefalse:= true, logoutput::name:=terminal})
  local k, f;
  k := checkkey(key);
  f := filetobytes(cfilename);
  f := AESModes(k, f, mode);
  bytestofile(f, filename, filecheck, logoutput)
  end proc:
```

Example 4.8 Let us check the behavior of these functions. We generate a list of bytes, write it to a file named `"testfile"` in the current directory and encrypt it in CTR mode—with default values and using the previously defined key `k192`—writing the ciphertext to a file named `"ctestfile"`:

```
> k192 := "8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b":
  testlist := RandomTools:-Generate(list(nonnegint(range = 255), 16381));
  testfile := bytestofile(testlist, "testfile");
16381 bytes saved to testfile

> AESFileEncrypt(k192, "testfile", "ctestfile");
16397 bytes saved to ctestfile
```

Next, we decrypt the previously encrypted file, we read it from disk to a list and we compare this list with the one originally used to generate the file that was encrypted

obtaining that, as expected, the lists are identical. This means that the decrypted file is identical to the original one.

```
> AESFileDecrypt(k192, "ctestfile", "dtestfile");
  evalb(filetobytes("dtestfile") = testlist);
16381 bytes saved to dtestfile

                                    true
```

Exercise 4.13 In a similar way to the preceding example, generate a test file whose size in bytes is not a multiple of the block size, encrypt it in OFB mode, decrypt it, and check that the decrypted file coincides with the original one.

4.5.2.2 Encrypting and Decrypting Text Strings

We are now going to provide encryption and decryption functions for text strings. As on other occasions, we will use the ASCII character set (ISO Latin 1) as alphabet for plaintexts (of course, only printable characters will usually appear in plaintexts) but the ciphertext alphabet will be made out of the lower-case hexadecimal characters since using the ASCII character set would produce non-printable ciphertexts and, moreover, the null character—the character with 0 ASCII code, which could appear in a ciphertext with the same chance as any other character—would have to be treated as a separate case (the 0 ASCII code is assigned to the empty string and not to a specific character and, when a list of integers in the 0..255 range is converted to the corresponding string, if the list contains a zero value, the string is truncated at that point). Bearing this in mind, the function that encrypts text strings is the following:

```
> AESTextEncrypt := proc(key, plaintext::string, mode::name:=CTR, {iv:=selectiv()})
  local k, t;
  k := checkkey(key);
  t := convert(plaintext, bytes);
  bytestohexstring(AESModes(k, t, mode, iv))
  end proc:
```

The decryption function is similar:

```
> AESTextDecrypt := proc(key::{list, string}, ciphertext::string, mode::name:=CTR)
  local k, t;
  k := checkkey(key);
  t := AESModes(k, hexstringtobytes(ciphertext), mode);
  convert(t, bytes)
  end proc:
```

Example 4.9 Consider the following text string:

```
> frc := "Friends, Romans, countrymen, lend me your ears; I come to bury Caesar,
  not to praise him.":
```

We are going to encrypt this string in OFB mode with the following (already used) key and IV:

```
> k256 := "603deb1015ca71be2b73aef0857d77811f352c073b6108d72d9810a30914dff4";
  IV := "2ed2fee62859e07f10e5569310e633a6"
```

The resulting ciphertext is the following (observe that the first 32 hexadecimal digits correspond to the IV):

```
> AESTextEncrypt(k256, frc, OFB, iv = IV);
"2ed2fee62859e07f10e5569310e633a6f8d94058d061e11640682a23074b37e9e78d0ec34896303b9\
  3338dd26a23ba0ac9803e2a52b25b90b80641458e66b1d5690da6cb442fd2d014ece255f11c63ac4\
  92c70b009a2e51c14e4a3340e5f7ccc77d53811bd9f4b5c90"
```

The result of decrypting the ciphertext just obtained is:

```
> AESTextDecrypt(k256, %, OFB);
"Friends, Romans, countrymen, lend me your ears; I come to bury Caesar, not
   to praise him."
```

The same plaintext, encrypted and decrypted in CTR mode, with an automatically generated initial counter (which reveals the time at which encryption was performed) gives:

```
> AESTextEncrypt(k256, frc);
"0a071c1333370eecaa39a6bf00000000292c632ba2449033da12d1e02c08359a2894bd4bb8dacff9f\
  6e8969b6a2220de5d066f959dd1a4b1d3ea79c15d09d9c2c292af03918247cf4daaf71e28a3fac95\
  98e0066eb1fec9c72d20e7d10060c1100a8c3c34937d27eb7"

> AESTextDecrypt(k256, %);
"Friends, Romans, countrymen, lend me your ears; I come to bury Caesar, not to
   praise him."
```

Exercise 4.14 Use the ciphertext obtained from encrypting the plaintext `frc` with key `k256` and initialization vector `IV` in Example 4.9 to investigate the questions asked in Exercise 4.7. Perform the encryptions in OFB and CTR modes, flip either the first bit of the resulting ciphertext string (this would amount to replacing the first '2' in the string, i.e., the first hexadecimal digit of the IV by an 'a') or the first bit in the 33rd hexadecimal digit in the ciphertext string (i.e., the first bit of the first ciphertext block, not including the IV). Then decrypt the changed ciphertext with the same key and check how much correct plaintext is recovered.

Exercise 4.15 As a preliminary step for the implementation of ECB, CBC and CFB modes, write Maple functions to pad messages (using the 10^i padding algorithm) and to remove padding. Since these functions will work at the byte level and the binary expansion of 128 is 1 followed by seven zeros, the padding function should make the byte length of the message a multiple of 16 (corresponding to a bit length that is a multiple of 128). This can be accomplished by adding to the message, regarded as a list of bytes, a byte 128 to mark the beginning of the padding and then as many 0 bytes as required (and a full block of this form in case the byte length of the message is already a multiple of 16). The unpadding function should then remove all the trailing zeros and the byte 128 immediately preceding them.

Exercise 4.16 In a way similar to the preceding implementation of OFB and CTR modes, write implementations of ECB, CBC and CFB modes.

Chapter 5
Message Authentication

In the preceding chapters we studied encryption schemes whose goal is to ensure confidentiality and prevent an adversary from learning information about messages sent over a communication channel. This chapter is devoted to the study of schemes that pursue an entirely different goal, namely, to ensure authenticity.

5.1 Confidentiality Versus Authenticity

Since ancient times, cryptography has been closely associated with providing confidentiality and so it is often implicitly or explicitly identified with the use of encryption and decryption. But, important as confidentiality is, modern cryptography goes far beyond this objective and *message authentication* and *message integrity* are perhaps even more important goals. Indeed, when one receives a message from another party through an open channel, one often may not care whether privacy has been maintained but certainly one wants to know for sure who the originator of the message is and whether or not the message has been altered during transmission. An obvious example where message integrity is essential while confidentiality is not required is in the transmission of stock quotes.

The goal of message authentication is to address these concerns by providing assurance to the receiver that the message is *authentic*, i.e., that it was really sent by the legitimate sender, and that the message *integrity* was not compromised or, in other words, that the message was not modified en route by an adversary.

In this chapter we will look at techniques that provide message integrity in the private-key setting, i.e., we will assume that the honest parties share some secret key. These techniques will try to guarantee that any modification of the message will be detected by the honest parties.

Before going into the details of how authentication may be achieved, it is convenient to emphasize that encryption does not provide it and hence the techniques seen so far are not adequate for this purpose. It is a common source of misunderstanding to

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_5, © Springer-Verlag Berlin Heidelberg 2013

think that when a message is encrypted it is not possible for the adversary to modify it in a meaningful way and hence that encryption suffices to guarantee integrity. But this idea is far off the mark as can be seen by considering, for example, the one-time pad encryption scheme, which has the strongest security property of all, namely, perfect secrecy. In this case, an adversary who knows that some quantity is being transmitted may alter the message in a meaningful way as the following example shows:

Example 5.1 Let us consider the implementation of one-time pad encryption given by the Maple function `OTP` defined in Sect. 3.4.2. Suppose that Alice wishes to send to Bob (Bob might be a bank in this example) the following message:

```
> m := "Please transfer from my account to Eve's account the amount of $1000":
```

Alice sends the message to Bob encrypted with `OTP` using the following key:

```
> k := "d1e5342179e59719d99fc6384c8288a743ab02b002b9564b96fdafc9c45f49ce6cde50924d\
       e9c66c933a30b898eb707a4d29a7a005b23173409ee099db51f4328ca9f839"
```

Thus Alice sends to Bob the following ciphertext:

```
> c := OTP(k, m);
c := "818951400a80b76dabfea84b2ae7fa8725d96ddd22d42f6bf79ecca6b1313dee18b170d73b8c\
      e11fb35b53dbf79e1e0e6d5dcfc525d35c1c35f094b9b437d416bd99c809"
```

Eve intercepts the ciphertext and she would like the amount to be transferred to her account to be higher, more like $9000. She suspects that the last four characters correspond to the amount to be transferred and hence that the fourth-to-last ciphertext character probably comes from the "1" in $1000. Thus she computes the bitwise Xor of bytes 49 and 57 (which are the ASCII codes of "1" and "9") and the fourth-to-last byte in the ciphertext, `bd` (which is 189 in decimal). This gives:

```
> bytestohexstring(BitXor(BitXor(49, 57), 189));
                                    "b5"
```

Thus Eve knows that she wants to replace `"bd"` by `"b5"`. She does it and Bob receives the ciphertext:

```
> c1 := StringTools:-Substitute(c, "bd", "b5");
c1 := "818951400a80b76dabfea84b2ae7fa8725d96ddd22d42f6bf79ecca6b1313dee18b170d73b8\
       ce11fb35b53dbf79e1e0e6d5dcfc525d35c1c35f094b9b437d416b599c809"
```

Now, Bob decrypts and voilà!:

```
> OTP(k, c1, decrypt);
          "Please transfer from my account to Eve's account the amount of $9000"
```

Thus Eve, without knowing the secret key, has altered the ciphertext to her advantage. This was a consequence of the fact that Eve guessed a plaintext character but, even without this knowledge, an adversary that knows that the last four characters encrypt a number can also change the ciphertext in a meaningful way. For example, if the adversary flips the fifth bit in the fourth-to-last ciphertext byte (since the decimal digits have ASCII codes ranging between 48 and 57, changing one of the four most significant bits would replace the digit by a non-digit character), `bd` is, again, changed to `b5` and the dollar amount is changed to $9000. Similarly, if the sixth

bit of this byte is flipped, `bd` is changed to `b9` and the dollar amount is changed to $5000. When doing these changes, the adversary does not know whether the number will be increased or decreased but the example clearly shows that a highly secure encryption scheme like `OTP` does not provide authenticity or integrity.

Observe also that, even without any knowledge about the message, a small random change to the ciphertext in the previous example would produce a plaintext different from the one originally sent and hence would violate the authenticity and integrity of the message. Expanding on this idea, we mention another basic reason why all the encryption schemes seen so far fail to provide authenticity. It is the fact that they allow *spoofing attacks* in which an adversary may masquerade as a legitimate user just by sending an arbitrary ciphertext and exploiting the fact that the ciphertext in question will give, when decrypted with the secret key, a valid plaintext. The plaintext may, and very likely will, be meaningless but even so this kind of attack is disturbing and should be prevented.

The conclusion is that authentication requires different techniques than encryption and, because of the problems mentioned, it is generally admitted that the best practice is, in most situations, to use encryption and authentication together and not encryption alone.

5.2 Message Authentication Codes

We are going to introduce a class of cryptographic schemes called *Message Authentication Codes* (or MACs, for short) that will prevent an adversary from impersonating a legitimate user and from modifying a message without the legitimate users noticing it. MACs are used in the private-key setting and, similarly to symmetric encryption schemes, require parties to share a secret key.

5.2.1 Defining MACs

Before giving the formal definition, we informally explain the idea of how a MAC is used. Two users (often called the sender and the receiver) share a secret key k and want to communicate in an authenticated way. Concretely, the sender wants to send a message m to the receiver and for this she does the following. She uses a *tag generation algorithm*, that we shall denote by **Mac**, to compute a *MAC tag* t, using m and k as inputs. Then, she sends the pair (m, t) to the receiver. After receiving (m, t), the receiver *verifies* that t is a valid tag for m by running a *verification algorithm* **Ver** that takes as input m, k, and t and shows whether t is a valid tag for m.

Definition 5.1 A *message authentication code* is a 3-tuple $\mathcal{I} = (\mathbf{Gen}, \mathbf{Mac}, \mathbf{Ver})$ of polynomial-time algorithms that satisfy the following conditions:

- **Gen** is a probabilistic algorithm that takes as input a *security parameter* 1^n and outputs a key k such that $\text{len}(k) \geq n$, which we denote, as usual, by $k \leftarrow \mathbf{Gen}(1^n)$.
- The tag generation algorithm **Mac** is either a probabilistic or a deterministic algorithm that takes as input a key k and a message m and outputs a tag t. We write this as $t \leftarrow \mathbf{Mac}(k, m)$.
- The verification algorithm **Ver** is a deterministic algorithm that takes as input a key k, a message m, and a tag t and outputs a bit $b \in \{0, 1\}$. This output $b := \mathbf{Ver}(k, m, t)$ is interpreted as meaning *valid* when $b = 1$ and *invalid* when $b = 0$.

We require that, for any key k output by **Gen** and any message $m \in \{0, 1\}^*$, the following correctness condition holds: $\mathbf{Ver}(k, m, \mathbf{Mac}(k, m)) = 1$. We sometimes write $\mathbf{Mac}_k(m)$ instead of $\mathbf{Mac}(k, m)$ and $\mathbf{Ver}_k(m, t)$ instead of $\mathbf{Ver}(k, m, t)$.

If, for each key k output by $\mathbf{Gen}(1^n)$, $\mathbf{Mac}(k, -)$ is only defined for messages of a given fixed length $\ell(n)$, and $\mathbf{Ver}(k, m, t) = 0$ for every message m of length $\neq \ell(n)$, then $\mathcal{I}$ is said to be a *fixed-length* MAC of length $\ell(n)$.

5.2.2 Security for MACs

We have already seen that authentication security is not provided by encryption security and so it is necessary to define the concept of security for a MAC. The intuitive idea is very simple: an adversary should not be able to produce messages that will look authentic to the receiver; only the sender should be able to do that. Bearing in mind how the MAC algorithms are used, this means that no polynomial-time adversary should be able to generate a pair (m, t) such that $\mathbf{Ver}(k, m, t) = 1$, where m is a message that did not originate with the sender.

The reader should be warned against misinterpreting the above requirement. For example, it is clear that if an adversary $\mathcal{A}$ is able to recover the secret key k, then $\mathcal{A}$ is able to break this intuitive notion of security because it could then compute $\mathbf{Mac}(k, m)$ for any m it chooses. However, as in the case of encryption schemes, security against key recovery is far from fulfilling the requirement just stated because there are many conceivable ways in which an adversary could *forge* a valid pair (m, t) without knowing the key. The definition of MAC security should capture the idea that these forgeries should not be feasible. Note also that, as already mentioned, a forgery in which m looks random should also count as a security break because the concept of a meaningful message is application-dependent and the definition should be as strong as possible to include a wide range of situations such as, for example, when m is a randomly chosen key.

There is another kind of attack that allows the adversary to forge a valid pair (m, t) easily: the adversary can use such a pair if it was previously sent by the sender, and such a pair will be accepted by the verification algorithm. This is called a *replay*

attack and the question arises whether such an attack should be regarded as a break of the system. It is clear that these attacks are dangerous and should not be allowed in real life; for example, a message such as the one in the last section ordering the transfer of an amount of money to Eve's account could be replayed by Eve with the effect of increasing the amount of money that she would receive. But it is also clear that a MAC by itself does not have the capability to protect against such attacks since the definition does not include any notion of state in the verification algorithm. This problem can be solved by making the verification algorithm stateful or by letting a higher-level application deal with it. Bearing this in mind, we shall not consider a replay attack as a break of the system and the security definition we are going to give will not include protection against such attacks.

The security definition for MACs will be based on the protection against forgeries under an *adaptive chosen message attack* in which the adversary not only obtains valid pairs (m, t) but is allowed to choose the messages m in such pairs. Allowing the adversary to mount this kind of attack may seem too generous but there are situations in which such attacks are realistic and, as in the case of encryption, if anything it is best to err on the side of caution and allow the adversary more power than it might have in practice.

As in the case of security for encryption schemes, we start by defining an experiment for a MAC $\mathcal{I} = (\mathbf{Gen}, \mathbf{Mac}, \mathbf{Ver})$, an adversary (i.e., PPT algorithm) $\mathcal{A}$, and a positive integer n for the security parameter:

Definition 5.2 The *authentication unforgeability experiment* under an adaptive chosen message attack $\mathbf{Mac}^{uf\text{-}cma}_{\mathcal{A},\mathcal{I}}(n)$ is the following:

1. A key k is generated by running $\mathbf{Gen}(1^n)$.
2. The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\mathbf{Mac}(k, -)$. $\mathcal{A}$ asks a set of queries $\mathcal{Q}$ (where each query is a message) to the oracle and outputs a message/tag pair (m, t).
3. The output of the experiment is defined to be 1 if and only if the following two conditions hold:
   a. $\mathbf{Ver}(k, m, t) = 1$
   b. $m \notin \mathcal{Q}$.

Now, the definition of security formalizes the fact that the adversary cannot succeed in the previous experiment with non-negligible probability:

Definition 5.3 A message authentication code $\mathcal{I} = (\mathbf{Gen}, \mathbf{Mac}, \mathbf{Ver})$ is *existentially unforgeable under an adaptive chosen-message attack* (UF-CMA secure or just CMA secure, for short) if, for every PPT adversary $\mathcal{A}$, there exists a negligible function $\mathtt{negl}$ such that:

$$\Pr(\mathbf{Mac}^{uf\text{-}cma}_{\mathcal{A},\mathcal{I}}(n) = 1) \leq \mathtt{negl}(n).$$

Remark 5.1 The advantage of $\mathcal{A}$ in the CMA authentication unforgeability experiment may be defined as the quantity $Adv^{uf\text{-}cma}_{\mathcal{A},\mathcal{I}}(n) = \Pr(\mathbf{Mac}^{uf\text{-}cma}_{\mathcal{A},\mathcal{I}}(n) = 1)$ and

hence an equivalent formulation of the security definition is that the advantage of any PPT adversary is negligible.

Example 5.2 Let us reconsider the situation of Example 5.1, where Alice wishes to send to Bob the message:

```
> m := "Please transfer from my account to Eve's account the amount of $1000"
```

but suppose that now, instead of encrypting the message, Alice sends a message/tag pair (m, t) where t has been computed with a UF-CMA secure MAC. Eve now wants to replace m by:

```
> m' := "Please transfer from my account to Eve's account the amount of $9000"
```

However, since an authentication MAC is being used, Eve has to compute a forgery (m', t') and, by Definition 5.3, she can only do that with negligible probability. The fact that m' is closely related to m does not help Eve in this case.

Exercise 5.1

(i) Let F be a block cipher of length n and consider the MAC $\mathcal{I}' = (\mathbf{Gen}, \mathbf{Mac}, \mathbf{Ver})$, where **Gen** chooses k uniformly at random in $\{0,1\}^n$ and, if $m = m_1||m_2||\ldots||m_\ell$, where each m_i is an n-bit block, then $\mathbf{Mac}(k, m) = F_k(m_1) \oplus F_k(m_2) \oplus \ldots \oplus F_k(m_\ell)$ (while **Ver** is the algorithm that computes the tag of a message and compares it to the given one). Show that this MAC is not CMA secure and that there are adversaries that, even without making any oracle queries, have advantage 1. (Hint: Remember that an adversary is successful if it can forge a pair (m, t) satisfying $\mathbf{Ver}(k, m, t) = 1$, where m is *any message of its choice* whose tag has not been previously computed by the honest parties.)

(ii) Consider the following modification of the preceding MAC. If $m = m_1||\ldots||m_\ell$, where $m_i \in \{0,1\}^{n/2}$, then $\mathbf{Mac}(k, m) = F_k(\langle 1\rangle||m_1) \oplus \ldots \oplus F_k(\langle \ell\rangle||m_\ell)$, where $\langle i\rangle$ is a binary $(n/2)$-bit encoding of the integer i. Show that there is an adversary which makes three oracle queries and has advantage 1.

5.3 Constructing MACs

We have defined MACs and we have also given a definition of security for MACs. In this section we show how to construct MACs that meet the requirements of these definitions.

5.3.1 MACs from Pseudo-Random Functions

A natural approach to construct a MAC is just to take a (keyed) pseudo-random function F and to consider the tag obtained by directly applying this function to the message m. We will consider messages of fixed length n and the MAC $\mathcal{I}_F = (\mathbf{Gen}, \mathbf{Mac}, \mathbf{Ver})$ given by the following algorithms:

- **Gen**: $k \leftarrow \{0,1\}^n$ (where k is chosen uniformly at random on input 1^n).
- **Mac**: $\mathbf{Mac}(k,m) := F_k(m)$.
- **Ver**: $\mathbf{Ver}(k,m,t) := \begin{cases} 1 & \text{if } F_k(m) = t \\ 0 & \text{otherwise.} \end{cases}$

In contrast with the case of encryption schemes, where an encryption algorithm that is a deterministic function of the key and the message makes the scheme automatically insecure, we have that $\mathcal{I}_F$ has a deterministic **Mac** algorithm and is, nevertheless, UF-CMA secure:

Theorem 5.1 *If F is a pseudo-random function, then $\mathcal{I}_F$ is a UF-CMA secure MAC.*

Proof The intuitive idea behind the proof is that forging a tag on a new message requires the adversary to guess the value of the pseudo-random function when applied to that message. If the function were purely random then all the possible tag values would be equally likely and, since we are applying the function to messages of length n and F is length-preserving, there are 2^n of these values, so the probability of guessing the correct one would be 2^{-n} and hence negligible (since it decreases exponentially). If F is pseudo-random instead of random, then a PPT algorithm can distinguish it from a random function only with negligible probability and, because of this, the adversary will have negligible advantage in the CMA authentication unforgeability experiment. Thus the proof will follow the pattern established in the proof of Theorem 4.1, namely, we will first show that the ideal MAC scheme based on a random function is UF-CMA secure and then we will deduce from this fact that $\mathcal{I}_F$ is also UF-CMA secure.

Let us call, for simplicity, $\mathcal{I} = \mathcal{I}_F$ and let $i\mathcal{I}$ be the ideal MAC that is similar to $\mathcal{I}$ but uses a truly random function $f : \{0,1\}^n \to \{0,1\}^n$ instead of F. As in the proof of Theorem 4.1, this idealized scheme is not efficient and hence is not a MAC in the strict sense but, for the purposes of this proof, this fact is irrelevant. The adversary $\mathcal{A}$ can query the oracle about the value of f on several messages and then it has to guess the value $f(m)$ for a different message m. Since f is a random function with 2^n possible values, the probability that $\mathcal{A}$ guesses correctly or, in other words, the advantage of $\mathcal{A}$ in the authentication unforgeability experiment is $Adv^{\textit{uf-cma}}_{\mathcal{A},i\mathcal{I}}(n) = 2^{-n}$.

Next, in order to bound the advantage $Adv^{\textit{uf-cma}}_{\mathcal{A},\mathcal{I}}(n)$ we construct a distinguisher D that, using $\mathcal{A}$ as a subroutine, tries to determine whether a given function f is equal to some F_k for a randomly chosen key or is randomly chosen in $(\{0,1\}^n)^{\{0,1\}^n}$ (see Definition 4.2) and we use the distinguisher advantage $Adv^{\textit{prf}}_{D,F}(n)$ to obtain the required bound. D runs as follows:

- On input 1^n, D is given oracle access to a function $f : \{0,1\}^n \to \{0,1\}^n$ which is either $f := F_k$ for $k \leftarrow \{0,1\}^n$ or a random function.
- D runs $\mathcal{A}$ on input 1^n and when $\mathcal{A}$ queries the oracle in the $\mathbf{Mac}^{\textit{uf-cma}}_{\mathcal{A},\mathcal{I}}(n)$ experiment on a message m_i, D queries its own oracle f and returns the oracle's answer, $t_i = f(m_i)$ to $\mathcal{A}$.

- $\mathcal{A}$ finally outputs a message/tag pair (m, t). Then D queries its oracle on m and obtains as response $f(m) = t'$. If $t = t'$ and $\mathcal{A}$ did not query its oracle on m, then D outputs 1, otherwise it outputs 0.

Let us now analyze D's advantage in the distinguishing experiment. If D is given oracle access to a random function from $\{0, 1\}^n$, then the view of $\mathcal{A}$ when run as indicated above is identical to the view of $\mathcal{A}$ in experiment $\mathbf{Mac}^{uf\text{-}cma}_{\mathcal{A},i\mathcal{I}}(n)$ since a random function is used to compute the tags. On the other hand, if D is given oracle access to some F_k, for a randomly chosen k, then the view of $\mathcal{A}$ is the same as in experiment $\mathbf{Mac}^{uf\text{-}cma}_{\mathcal{A},\mathcal{I}}(n)$ because when answering $\mathcal{A}$'s oracle queries, D computes the answer by using F_k. In both cases, D outputs 1 exactly when $\mathcal{A}$ succeeds in the corresponding authentication unforgeability experiment. Therefore we have, bearing in mind Definition 4.2 and also the fact that F is a pseudo-random function, that there exists a negligible function $\mathtt{negl}(n)$ such that

$$Adv^{prf}_{D,F}(n) = |Adv^{uf\text{-}cma}_{\mathcal{A},\mathcal{I}}(n) - Adv^{uf\text{-}cma}_{\mathcal{A},i\mathcal{I}}(n)| = |Adv^{uf\text{-}cma}_{\mathcal{A},\mathcal{I}}(n) - 2^{-n}| \leq \mathtt{negl}(n).$$

This clearly implies that $Adv^{uf\text{-}cma}_{\mathcal{A},\mathcal{I}}(n) \leq 2^{-n} + \mathtt{negl}(n)$, so that the adversary's advantage is negligible and the proof is complete. □

Remark 5.2 This construction can be implemented in practice by using a block cipher F but it has the serious drawback that then the message length will be equal to the block length, which is too short for practical use. However, there is the possibility of extending this construction to variable-length messages in such a way that the MAC obtained remains UF-CMA secure as proved, for example in [109, Theorem 4.6]. But this construction is very inefficient and there are more efficient ways of obtaining secure MACs as we will see in the next sections.

Exercise 5.2 Based on the AES implementation given in Chap. 4, write a Maple implementation of the MAC $\mathcal{I}_F$ defined above, when F is the AES forward cipher function.

5.3.2 CBC-MAC

One basic idea that permits the construction of a block cipher-based MAC which is, at the same time, both efficient and secure, is to use as **Mac** algorithm CBC mode on top of a block cipher F. The idea is simply to encrypt the message (which, after padding, will consist of an integer number of blocks) in CBC mode with the 0 vector as IV, and then to take the last ciphertext block as the message tag. In this case, **Gen** will simply choose the key uniformly at random and **Ver** is the canonical algorithm that given a key k, a message m, and a tag t, computes $\mathbf{Mac}(k, m)$ and checks whether its value is equal to t, outputting 1 in this case and 0 otherwise. Thus we only need to define **Mac**, which is the following algorithm:

Algorithm 5.1. CBC-Mac.

Input: An l-block message $m = m_1 \dots m_l$, and a block cipher key k.
Output: The tag t_l.
`Tag computation`:
$t_0 := 0^n$;
`for` i `from` 1 `to` l `do`
$t_i := F_k(m_i \oplus t_{i-1})$
`end do`;
`return` t_l.

When CBC mode is used for encryption, it is crucial that the IV must be unpredictable and this is the reason why it is chosen at random. However, in the authentication version of CBC the IV is not random and this is for a good reason as the following exercise shows:

Exercise 5.3 Prove that the variant of CBC-MAC which chooses t_0 uniformly at random in $\{0, 1\}^n$ and outputs $t_0||t_l$ as tag is not CMA secure.

Remark 5.3 The CBC-MAC construction is CMA secure as long as it is applied to messages of fixed length $l \cdot n$ as was proved in [15], see also [90, Theorem 9.5] and [109, Theorem 4.10]. However, CBC-MAC is not CMA secure when applied to messages of variable length. Indeed, suppose that an adversary obtains the valid pairs (m_1, t_1) and $(m_1||m_2, t_2)$. Then the adversary outputs the pair $(m_2 \oplus t_1, t_2)$ and has advantage 1 in the authentication unforgeability experiment because $m_2 \oplus t_1 \neq m_1, m_1||m_2$ and, by the above algorithm, $\mathbf{Mac}_k(m_2 \oplus t_1) = F_k(m_2 \oplus t_1 \oplus t_0) = F_k(m_2 \oplus t_1) = t_2$. In this case we see that only two oracle queries are sufficient for a security break.

There are several known ways to obtain versions of CBC-MAC that are CMA secure for variable-length messages, cf. [109, 4.5] for an outline. For example, one method described in [15] consists simply of *prepending* the message m to be authenticated with an n-bit encoding of the message length $\text{len}(m)$. However, the method consisting of *appending* the length of the message as the last n-bit block is not secure as the following exercise shows:

Exercise 5.4 Show that the variant of CBC-MAC that appends to the message a block encoding the message length before computing the tag, is not CMA secure when messages of varying length are used. (Hint: Consider three arbitrary n-bit blocks x, y, z, such that $x \neq y$. Query the oracle about x, y, and $x||\langle 1\rangle||z$ to obtain the corresponding tags. Then consider the message $y||\langle 1\rangle||\mathbf{Mac}_k(x) \oplus \mathbf{Mac}_k(y) \oplus z$ and check that the tag corresponding to this message is the same as the tag for the message $x||\langle 1\rangle||z$.)

Exercise 5.5 Prove that the version of CBC-MAC that uses the 10^i padding algorithm for messages whose length is not a multiple of the block length n but does not add an extra block whenever the length of the message is already a multiple of n, is not CMA secure.

We are going to present one variant of CBC-MAC that has been specified by NIST under the name of CMAC (see [69]) and has a security reduction as shown in [105]. We also give an implementation of CMAC in Maple using AES as block cipher.

5.3.3 CMAC and Its Maple Implementation

The core of the CMAC algorithm is a variation of CBC-MAC that Black and Rogaway proposed under the name XCBC and that was later improved under the name OMAC by Iwata and Kurosawa. CMAC is the NIST specification of OMAC1, one of the variants of OMAC. In XCBC, the padding algorithm (the 10^i padding) is modified by not adding an extra block whenever the bit length of the message is already a nonzero multiple of 128. In addition to the block cipher key k (an AES key in our case), two additional keys $k1$, $k2$ are used. If no padding was added, the algorithm is the same as CBC-MAC except for the fact that, before encrypting the last block with k, this block is Xor-ed with $k1$. If, on the other hand, padding was added, then a similar operation is performed, Xor-ing the last block with $k2$ before encryption. CMAC proceeds in a similar way, except for the fact that keys $k1$ and $k2$ are not independent from k but are derived from it. This is done as follows. Consider the field $\mathbb{F}_{2^{128}}$ defined by the irreducible polynomial $x^{128} + x^7 + x^2 + x + 1$ of $\mathbb{Z}_2[x]$ (this is the "first" irreducible polynomial of degree 128 over $\mathbb{Z}_2$ as we saw in Example 2.26). The elements of this field can be represented, as we have seen, by the integers in the $0..2^{128}$–1 range and hence by the 128-bit (or 16-byte) vectors (where we assume that the most significant bit—or byte—is the leftmost one). Let $L = F_k(0^{128})$ be the result of encrypting the 0-block with AES using k as key. L belongs to $\mathbb{F}_{2^{128}}$ and the CMAC subkeys $k1$ and $k2$ are obtained by taking $k1 = 2 \cdot L$ and $k2 = 4 \cdot L$, where '$\cdot$' denotes multiplication in $\mathbb{F}_{2^{128}}$. Then the algorithm proceeds in the same way as XCBC explained above.

As seen in Exercise 5.5, CBC-MAC is not CMA secure if 10^i padding is used but messages whose length is a multiple of the block length are left unpadded, for then a forgery may be trivially constructed by finding two different messages with the same tag. This is precisely the padding method used in CMAC but now there is the additional operation of Xor-ing the last block with either k_1 or k_2 and this is what makes it hard to produce forgeries where two different messages have the same tag. CMAC may thus be regarded as CBC-MAC with a modified padding algorithm that is key-dependent, if we regard Xor-ing the last block with one of the subkeys as part of the padding. Since the subkeys are different, an adversary will be able to find two messages which have the same tag only with negligible probability.

We start by giving an auxiliary function to make a cyclic left shift on the list of bits corresponding to a 16-byte list:

```
> rotatebitsleft := proc(bytes)
  uses ListTools;
  map(bitstobyte, [LengthSplit(Rotate(Flatten(map(bytetobits, bytes)), 1), 8)])
  end proc:
```

The next function implements multiplication by 2 (the element corresponding to the polynomial x) in $\mathbb{F}_{2^{128}}$. For this we could also use the `GF` package but, since we will not multiply by other elements of the field and, moreover, multiplication by 2 is very simple (as it essentially reduces to a left shift), we shall implement it directly by means of the next function. Note that the 134 that appears in the body of the function comes from the fact that this is the value that the polynomial $x^7 + x^2 + x$, (obtained by discarding the highest and lowest degree monomials in the polynomial defining $\mathbb{F}_{2^{128}}$) takes for $x = 2$.

```
> MultBy2 := proc(bytes::list)
  local f, r;
  f := bytes[1];
  r := rotatebitsleft(bytes);
  if f < 128 then
    r
  else
    zip(BitXor, r, [0$15, 134])
  end if
  end proc:
```

Exercise 5.6 Justify the claim that the previous function implements multiplication by 2 in the field $\mathbb{F}_{2^{128}}$ (defined by the irreducible polynomial $x^{128} + x^7 + x^2 + x + 1 \in \mathbb{Z}_2[x]$).

The next function generates the subkeys $k1$, $k2$, used by CMAC, from an AES key.

```
> CmacSubkeys := proc(key::list)
  local l, m;
  l := AES(key, [0$16]);
  l := MultBy2(l);
  m := MultBy2(l);
  [l, m]
  end proc:
```

Example 5.3 Let us consider Example D.1 in [69]. We compute the subkeys corresponding to the AES key defined therein:

```
> bytestohexstring~(CmacSubkeys(hexstringtobytes(
                              "2b7e151628aed2a6abf7158809cf4f3c")));
  ["fbeed618357133667c85e08f7236a8de", "f7ddac306ae266ccf90bc11ee46d513b"]
```

The next function implements CBC-MAC for lists of bytes. The inputs are the AES key, the message and the IV, and the output is the MAC, all of them given as lists of bytes.

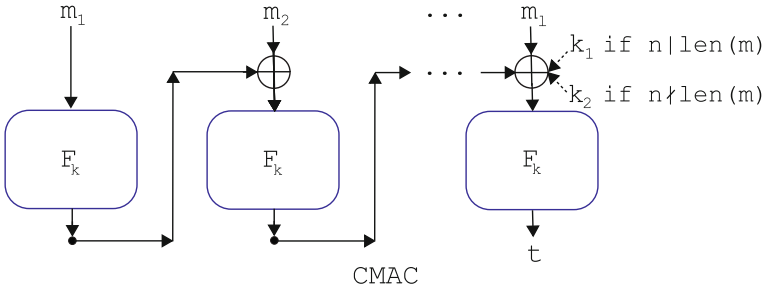
```
> CBCMAC := proc(key, bytelist, iv)
  local k, ek, states, len, i, kst, count;
  ek := KeyExpansion(key);
  states := [iv, ListTools:-LengthSplit(bytelist, 16)];
  len := nops(states)-1;
  states := Array(0..len, map(x -> Array(0..3, 0..3, (i,j) -> x[i+4*j+1]), states));
  for i to len do
    states[i] := AESEncrypt(ek, AddRoundKey(states[i-1], states[i]))
  end do;
  states := map(x -> ArrayTools:-Alias(x, [16]), states);
  [seq(states[len][j], j = 1 .. 16)]
  end proc:
```

We now give the CMAC function that computes the CMAC tag associated with a message. This function makes use of several functions defined in Appendix A, including the function messagetobytes. The input arguments are the key given either as a list of bytes or as a hex string, the message (with the same format as in the preceding function), and a name indicating the type of message. The output is the CMAC tag given as a hexadecimal string.

```
> CMAC := proc(key::{list, string}, message::string, messagetype::name := file)
  local k, m, l, s, b;
  k := checkkey(key);
  m := messagetobytes(message, messagetype);
  l := nops(m);
  s := CmacSubkeys(k);
  if l mod 16 <> 0 or l = 0 then
    m := [op(m), 128, 0$(15-(l mod 16))];
    b := zip(BitXor, m[-16 .. -1], s[2]);
    m := [op(m[1 .. -17]), op(b)]
  else
    b := zip(BitXor, m[-16 .. -1], s[1]);
    m := [op(m[1 .. -17]), op(b)]
  end if;
  m := CBCMAC(k, m, [0$16]);
  bytestohexstring(m)
  end proc:
```

Note that the message is padded, if necessary, inside the function. The padding method consists of adding a bit 1 to mark the beginning of the padding and then completing it with as many bits 0 as needed to make the length a multiple of 128. Since we work at the byte level and the binary expansion of 128 is 1 followed by seven zeros, we use a byte 128 to mark the beginning of the padding and then bytes 0 until completing the desired length.

The CMAC algorithm may be represented as follows, for an l-block message $m = m_1||m_2||\ldots||m_l$, where each block has length n (in the NIST standard and in our implementation, $n = 128$ since the block cipher used is AES). The subkey k_1 is used when the message is not padded, i.e., when $n|\text{len}(m)$, and k_2 is used when n does not divide $\text{len}(m)$, in which case the 10^i padding is used.



Example 5.4 In this example we use some of the test strings in [69, Appendix D]; the results may be compared to the tags given there. The hexadecimal strings to be used as messages and keys are the following:

```
> m128 := "6bc1bee22e409f96e93d7e117393172a":
  m320 := "6bc1bee22e409f96e93d7e117393172aae2d8a571e03ac9c9eb76fac45af8e5130c81c4\
           6a35ce411":
```

```
m512 := "6bc1bee22e409f96e93d7e117393172aae2d8a571e03ac9c9eb76fac45af8e5130c81c4\
         6a35ce411e5fbc1191a0a52eff69f2445df4f9b17ad2b417be66c3710":
k128 := "2b7e151628aed2a6abf7158809cf4f3c";
k192 := "8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b";
k256 := "603deb1015ca71be2b73aef0857d77811f352c073b6108d72d9810a30914dff4"
```

The resulting tags are the following; note that in some cases the empty string is being authenticated:

```
> CMAC(k128, "", hex);
  CMAC(k128, m320, hex);
  CMAC(k192, "", hex);
  CMAC(k192, m128, hex);
  CMAC(k256, "", hex);
  CMAC(k256, m512, hex);
                    "bb1d6929e95937287fa37d129b756746"
                    "dfa66747de9ae63030ca32611497c827"
                    "d17ddf46adaacde531cac483de7a9367"
                    "9e99a7bf31e710900662f65e617c5184"
                    "028962f61b7bf89efc6b551f4667d983"
                    "e1992190549f6ed5696a2c056c315410"
```

Exercise 5.7 Convert the hexadecimal string `m512` to a list of bytes and write this list to a file named `"m512"` in the current directory. Then compute the CMAC tag of this file using the key `k256` and check that it is equal to the tag `CMAC(k256,m512, hex)` computed above.

Apart from the key generation algorithm, for which we refer to our earlier discussions on the topic, there is still one ingredient missing in the implementation of CMAC, namely, the verification algorithm. It is given in the next function whose input is the (received) message, the received tag and the key, and whose output is `VALID` (if indeed the tag corresponds to the message) or `INVALID` otherwise.

```
> CMACVer :=
  proc(key::{list, string}, message::string, tag::string, messagetype::name:=file)
  if not (StringTools:-IsHexDigit(tag) and StringTools:-Length(tag) = 32) then
    error "Incorrectly formatted tag"
  end if;
  if evalb(CMAC(key, message, messagetype) = StringTools:-LowerCase(tag)) then
    VALID
  else
    INVALID
  end if
  end proc:
```

Example 5.5 Let us verify one of the previously computed tags:

```
> CMACVer(k192, m128, "9e99a7bf31e710900662f65e617c5184", hex);
                                    VALID
```

Exercise 5.8 Let k be an AES key and k_1, k_2, the corresponding CMAC subkeys. Let x be a byte string of length i, where $0 \leq i \leq 15$ and $m = x||128||0^{15-i}$. Prove that the messages x and $m \oplus k_1 \oplus k_2$ have the same CMAC tag under the key k.

Exercise 5.8 shows that it is easy to find two different messages with the same CMAC tag, but this does not compromise the security of the scheme because of one important detail: it is necessary to know the key or, at least, the subkeys k_1, k_2. Thus, even if we consider the operation of Xor-ing the last block with either k_1 or k_2 as part

of the padding algorithm, padding is not injective in this case. However, because of CMAC's security reduction, it looks as if it were injective to an adversary, which is unable to compute such a 'collision' except with negligible probability. Therefore, the situation is very different from the one considered in Exercise 5.5, where such collisions were easy to find without knowledge of the key.

Example 5.6 The method outlined in Exercise 5.8 lets us easily find, with the help of Maple, different messages with the same CMAC tag assuming, of course, that the key is known. For example, consider the key `k128` defined in Example 5.4. Then the empty string and the string `"8c337a285f9355aa858e2191965bf9e5"` have the same CMAC tag:

```
> CMAC(k128, "", hex);
                    "bb1d6929e95937287fa37d129b756746"

> CMAC(k128, "8c337a285f9355aa858e2191965bf9e5", hex);
                    "bb1d6929e95937287fa37d129b756746"
```

Similarly, the hexadecimal string consisting of 30 consecutive zeros and the string `"0c337a285f9355aa858e2191965bf965"` have the same CMAC tag:

```
> CMAC(k128, "00000000000000000000000000000000", hex);
                    "ee26a41b40b8904aa6b8fcaa41b7a8a3"

> CMAC(k128, "0c337a285f9355aa858e2191965bf965", hex);
                    "ee26a41b40b8904aa6b8fcaa41b7a8a3"
```

Exercise 5.9 Find a 16-byte string with the same CMAC tag as the string `"0123456789abcdef"` under the key `k192` of Example 5.4.

5.4 CCA Security and Authenticated Encryption

In Sect. 3.5.2 we defined CCA security and mentioned that we would use authentication in order to obtain this kind of security. In this section we will see that CPA secure encryption schemes together with MACs allow the construction of CCA secure symmetric encryption schemes and of schemes that guarantee both confidentiality and authenticity.

5.4.1 A CCA Secure Encryption Scheme

The intuitive idea behind the construction of a CCA secure encryption scheme is the following. The sender and the receiver agree on a CPA secure private-key encryption scheme $\mathcal{E}$ and a UF-CMA secure MAC $\mathcal{I}$ and share two secret keys, one for each of these schemes. In fact, we will assume that the MAC has a stronger security property called *strong unforgeability* (or SUF-CMA) which, in addition to being UF-CMA, requires that if (m, t) is an authenticated message, the adversary $\mathcal{A}$ is not able (as

always, except perhaps with negligible probability) to find a different valid tag $t' \neq t$ on m. This requirement is not too restrictive because for it to hold it suffices that the MAC has *unique tags* in the sense that, for every key k and every message m, there is a unique tag t such that the pair (m, t) is valid. This property holds whenever **Mac** is a deterministic algorithm, which is often the case and, of course, both the basic CBC-MAC and CMAC have unique tags.

Using $\mathcal{E}$ and $\mathcal{I}$, we define an encryption scheme $\mathcal{E}_{\mathcal{I}} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$, as follows:

- **Gen**: The key generation algorithms of $\mathcal{E}$ and $\mathcal{I}$ are run on input 1^n, producing keys k_1, k_2, respectively. The output is the ordered pair (k_1, k_2).
- **Enc**: On input a key (k_1, k_2) and a message m, m is encrypted with the encryption algorithm of $\mathcal{E}$ and key k_1 producing a ciphertext c, and then the MAC tag t of c is computed using key k_2. The output is the ordered pair (c, t).
- **Dec**: On input a key pair (k_1, k_2) and a ciphertext (c, t), the verification algorithm of $\mathcal{I}$ is used with key k_2 to verify the message/tag pair (c, t). If verification fails then the decryption algorithm fails, otherwise **Dec** outputs the result of decrypting c with $\mathcal{E}$ using key k_1.

The security result is then the following:

Theorem 5.2 *If $\mathcal{E}$ is a CPA secure private-key encryption scheme and $\mathcal{I}$ is a SUF-CMA secure MAC, then $\mathcal{E}_{\mathcal{I}}$ is a CCA secure private-key encryption scheme.*

Sketch of proof We explain the basic idea—which is very simple—underlying the proof of the theorem and refer, e.g., to [109, Theorem 4.20] for a detailed proof as well as for further background on this result and its significance. Let us call the ciphertext (c, t) *valid* if the verification algorithm of the MAC $\mathcal{I}$ accepts t as a valid tag for c. Since $\mathcal{I}$ is UF-CMA secure, the adversary has negligible probability of generating any valid ciphertext (i.e., a forgery in experiment $\mathbf{Mac}^{uf\text{-}cma}_{\mathcal{A},\mathcal{I}}(n)$) except if the ciphertext was generated by the honest users or was provided to the adversary by the $\mathcal{E}_I$-encryption oracle. In more detail, in experiment $\mathbf{PrivK}^{ind-cca}_{\mathcal{A},\mathcal{E}_{\mathcal{I}}}(n)$, $\mathcal{A}$ has access to both an encryption and a decryption oracle. Some queries to the latter may be about ciphertexts obtained from querying the former, but these queries do not give $\mathcal{A}$ new information as it already knew the plaintext from some previous encryption query. Thus the only potentially interesting decryption queries are those about ciphertexts that $\mathcal{A}$ has not obtained from valid encryptions. But, as indicated above, in order to construct such a ciphertext $\mathcal{A}$ has to output a forgery in experiment $\mathbf{Mac}^{uf\text{-}cma}_{\mathcal{A},\mathcal{I}}(n)$ which it can only do with negligible probability since $\mathcal{I}$ is UF-CMA secure. Thus $\mathcal{A}$ will not gain information from these queries either and this amounts to saying that the decryption oracle only gives useful information with negligible probability and hence is essentially useless. Therefore, to prove that $\mathcal{E}_I$ is CCA secure it is sufficient to show that it is CPA secure and it is easy to see that this property follows from the fact that $\mathcal{E}$ is CPA secure by hypothesis.

The previous discussion did not take into account that $\mathcal{A}$ might possibly construct a valid ciphertext which is not a forgery in experiment $\mathbf{Mac}^{uf\text{-}cma}_{\mathcal{A},\mathcal{I}}(n)$ if, when it

receives the challenge ciphertext (c, t), $\mathcal{A}$ is able to construct a valid ciphertext of the form (c, t') with $t \neq t'$: this would not be a MAC forgery because c is the same in both cases. But the $\mathcal{E}_{\mathcal{I}}$-ciphertexts (c, t) and (c, t') are *different* and hence $\mathcal{A}$ could query the decryption oracle on (c, t'), obtaining the corresponding plaintext, which is the same as the one for (c, t). This is prevented by requiring that $\mathcal{I}$ should be SUF-CMA secure, which makes it hard for $\mathcal{A}$ to construct the ciphertext (c, t').

5.4.2 Obtaining Authenticated Encryption

We have studied schemes that provide privacy and schemes that provide authenticity but so far we have not considered any scheme that simultaneously provides both. In contrast to what might appear at first sight, the combination of confidentiality and integrity is not automatically guaranteed by the use of an encryption scheme and a MAC, even assuming that both are secure.

We next review the common approaches towards the goal of *authenticated encryption*, which use different ways to combine an encryption scheme and a MAC.

5.4.2.1 Encrypt-and-Mac

The first idea that comes to mind, given an encryption scheme and a MAC, both assumed secure, is just to apply the two schemes independently to the message m, i.e.—using the previous notations and letting k_1 be an encryption key and k_2 a MAC key—to compute (and send to the receiver) the pair (c, t), where $c = \mathbf{Enc}_{k_1}(m)$ and $t = \mathbf{Mac}_{k_2}(m)$. The receiver then computes $m = \mathbf{Dec}_{k_1}(c)$ and $\mathbf{Ver}_{k_2}(m, t)$. If the value 1 is obtained, then the receiver outputs m, otherwise it outputs $\perp$, meaning that authentication has failed. This method is, in general, insecure for a very simple reason: a MAC is not designed to hide the contents of the authenticated message and hence it is possible that the tag t leaks significant information as the following easy example shows:

Example 5.7 Let (**Gen**, **Mac**, **Ver**) be a UF-CMA secure MAC. Then it is straightforward to see that the MAC obtained by replacing the tag generation algorithm **Mac** by the following one:

$$\mathbf{Mac}'_k(m) := m||\mathbf{Mac}_k(m)$$

is also UF-CMA secure. Suppose now that we combine this MAC with a CPA secure—or even CCA secure—scheme as indicated above. Then the receiver obtains a pair of the form $(c, m||t)$ (where t is the tag corresponding to m in the original MAC) which, if observed by an adversary, reveals the entire message m. Thus this scheme is not even IND-EAV secure.

One might think that this example is not relevant because of the artificial nature of the **Mac** algorithm used. But this is not a worthwhile objection because a secure construction should be secure in *all* its variants. Moreover, it is easy to see that this construction is not secure either if an ordinary MAC with unique tags—such as, for example, CMAC—is used. Indeed, we may consider an adversary that queries the oracle for authenticated encryption of several messages—which might be chosen at random—until finding two messages, m_0, m_1 say, with different MAC tags t_0, t_1, respectively (note that, with high probability, any pair or random messages will have this property). Then the eavesdropper indistinguishability experiment is performed for the pair m_0, m_1, so that the adversary receives a challenge ciphertext (c_b, t_b) resulting from applying the scheme to m_b for a random bit b. Now, if the adversary had only c_b at its disposal, it would be unable to output the correct bit because we are assuming the encryption scheme is CPA secure but, in this case, the adversary also sees t_b which must be equal to exactly one of the tags t_0, t_1. If $t_b = t_0$ then the adversary outputs 0 and if $t_b = t_1$ then it outputs 1. In each case the adversary guess is correct with probability 1 and so the scheme does not have indistinguishable encryptions.

5.4.2.2 Mac-then-Encrypt

The previous approach fails because MACs do not protect confidentiality and a natural way to remedy this deficiency is to first authenticate and then encrypt. This is accomplished as follows: given a message m, the receiver is sent c where $c = \mathbf{Enc}_{k_1}(m||t)$ with $t = \mathbf{Mac}_{k_2}(m)$. In this case the authentication tag is concatenated to the plaintext and sent encrypted inside the ciphertext. As in the previous case, the receiver computes $m||t = \mathbf{Dec}_{k_1}(c)$ and $\mathbf{Ver}_{k_2}(m, t)$ and outputs m if verification is successful and $\perp$ otherwise. It is clear that now the previous attack on the Encrypt-and-Mac approach will not work because the MAC tag is encrypted and so it cannot leak information about the message. But this scheme is also insecure as the following easy example from [106] shows:

Example 5.8 Suppose that we have a CPA secure encryption scheme and a UF-CMA secure MAC and that we replace the encryption algorithm **Enc** of the encryption scheme by $\mathbf{Enc}'$ defined by

$$\mathbf{Enc}'_k(m) := \mathbf{Enc}_k(m)||1,$$

i.e., a 1 bit is appended to the ciphertext obtained by applying **Enc**. Of course, the corresponding decryption algorithm $\mathbf{Dec}'$ proceeds by simply stripping out the last ciphertext bit and then applying the original decryption algorithm **Dec**.

Now suppose that we use the original MAC and the modified encryption scheme in the previously described manner consisting of authentication followed by encryption. Although we have not yet formally defined the security concept for authenticated encryption (the formal definition is given below), it is easy to figure out what is going

on and to see that the resulting scheme does not guarantee authenticity by showing that an adversary can forge a valid encrypted message as follows:

- Query the oracle about an arbitrary message m and obtain the corresponding ciphertext $c = \mathbf{Enc}'_{k_1}(m||t) = \mathbf{Enc}_{k_1}(m||t)||1$.
- Output the ciphertext c' obtained by replacing the last bit of c by a 0 bit.

We see that then c' is a forgery, because $\mathbf{Dec}'_{k_1}(c') = \mathbf{Dec}_{k_1}(c) = m||t$ and $\mathbf{Ver}_{k_2}(m,t) = 1$ since the pair (m,t) generated by the oracle is valid.

This attack is not very natural but is sufficient to prove that the Mac-then-Encrypt approach is not generically secure although it may be secure in specific implementations. A more detailed discussion as well as further examples can be found in [123]. This approach is used in the SSL protocol whose security is discussed in this reference, where it is shown that the current practical implementations of the protocol are safe.

5.4.2.3 Encrypt-then-Mac

This method inverts the order in which the MAC and the encryption scheme are used. To wit, the message m is encrypted and then the MAC tag is computed for the ciphertext obtained, so that this is precisely the CCA secure encryption scheme we have considered. Thus the sender transmits the pair (c,t) where $c \leftarrow \mathbf{Enc}_{k_1}(m)$ and $t \leftarrow \mathbf{Mac}_{k_2}(c)$. The receiver then computes $\mathbf{Ver}_{k_2}(c,t)$ and, if the ciphertext is valid, it outputs $m = \mathbf{Dec}_{k_1}(c)$.

If the keys k_1 and k_2 are independent, the encryption scheme is CPA secure, and the MAC is SUF-CMA secure, then the Encrypt-then-Mac combination can be proven to be secure in the sense defined below. The idea is that an adversary cannot forge a valid ciphertext because the MAC is secure and confidentiality is guaranteed by the already mentioned fact that the system is actually CCA secure. Thus this approach gives the best we could hope for: both CCA security and existential unforgeability under adaptive chosen message attacks.

5.4.2.4 Security for Authenticated Encryption

We now make precise what it means that an encryption scheme provides secure authenticated encryption. As mentioned, the scheme should provide both confidentiality and integrity and we are going to formalize this. We will consider a private-key encryption scheme $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ such that the decryption algorithm may output the symbol $\perp$ indicating that the ciphertext is rejected, and we define the following integrity experiment:

Definition 5.4 The *ciphertext integrity experiment* $\mathbf{Auth}^{\textit{int-ctxt}}_{\mathcal{A},\mathcal{E}}(n)$ is the following:

1. A key k is generated by running $\mathbf{Gen}(1^n)$.

2. The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\mathbf{Enc}_k$. $\mathcal{A}$ asks a set of queries $\mathcal{Q} = \{m_1, m_2, \ldots m_q\}$ to the encryption oracle and receives ciphertexts $c_i = \mathbf{Enc}_k(m_i)$, which make up the set $\mathcal{QC} := \{c_1, \ldots, c_q\}$. $\mathcal{A}$ eventually outputs a candidate ciphertext c.
3. The output of the experiment is defined to be 1 if and only if the following two conditions hold:

   a. $\mathbf{Dec}_k(c) \neq \perp$.
   b. $c \notin \mathcal{QC}$.

   Otherwise, the output of the experiment is 0.

Now, the definition of ciphertext integrity formalizes the fact that the adversary cannot succeed in the previous experiment with non-negligible probability:

Definition 5.5 A private-key encryption scheme $\mathcal{E}$ achieves *ciphertext integrity* (INT-CTXT, for short) if, for every PPT adversary $\mathcal{A}$, there exists a negligible function `negl` such that

$$\Pr(\mathbf{Auth}^{\textit{int-ctxt}}_{\mathcal{A},\mathcal{E}}(n) = 1) \leq \mathtt{negl}(n).$$

Remarks 5.1

1. As usual, we may call $\mathcal{A}$'s advantage in the experiment $Adv^{\textit{int-ctxt}}_{\mathcal{A},\mathcal{E}}(n) = \Pr(\mathbf{Auth}^{\textit{int-ctxt}}_{\mathcal{A},\mathcal{E}}(n) = 1)$. The scheme provides ciphertext integrity whenever the advantage of any PPT adversary $\mathcal{A}$ is negligible.
2. A related concept is *plaintext integrity* (INT-PTXT) which requires that in the experiment similar to the above one, an adversary can only produce a ciphertext c such that $\mathbf{Dec}_k(c) \notin \mathcal{Q}$ with negligible probability. Plaintext integrity is weaker than ciphertext integrity because there is the possibility that for a scheme $\mathcal{E}$ with plaintext integrity, the adversary might still produce a ciphertext c such that $\mathbf{Enc}_k(m_i) = c$ for some $i = 1 \ldots q$, but $c \notin \mathcal{QC}$.

Ciphertext integrity provides the required notion of message integrity in the context of an encryption scheme and is the analogue in this setting of UF-CMA for MACs. Thus we will define authenticated encryption as follows:

Definition 5.6 A private-key encryption scheme $\mathcal{E}$ provides *authenticated encryption* if $\mathcal{E}$ is CPA secure and achieves ciphertext integrity.

Remarks 5.2

1. To motivate this definition we can give an informal argument that shows that if $\mathcal{E}$ provides authenticated encryption, then it provides message integrity in the sense that an adversary $\mathcal{A}$ cannot produce an encrypted message that was not previously sent by an honest user. If $\mathcal{A}$ intercepts a ciphertext c sent by an honest user and is able to construct a ciphertext c' such that $\mathbf{Dec}_k(c') \neq \perp$ and $\mathbf{Dec}_k(c') \neq \mathbf{Dec}_k(c)$, then $\mathcal{A}$ has non-negligible advantage in Experiment

$\mathbf{Auth}^{int-ctxt}_{\mathcal{A},\mathcal{E}}(n)$, contradicting the definition of authenticated encryption. In fact, the definition of authenticated encryption is strong enough to imply CCA security and the reason is the same as in Theorem 5.2, namely the fact that an adversary is unable to create—except with negligible probability—a valid ciphertext that was not obtained from a query to the encryption oracle. Being able to build such a ciphertext would violate ciphertext integrity and hence the adversary only obtains useful information from encryption queries, which are not sufficient for it to succeed because the scheme is assumed to be CPA secure. The fact that authenticated encryption implies CCA security is proved in [16, Theorem 3.2] to which we refer for a thorough discussion of all these properties as well as of the approaches to authenticated encryption we have mentioned.

2. Let us go back to the approaches to authenticated encryption mentioned above and, in particular, to the Encrypt-then-Mac construction. It is proved in [16, Theorem 4.4] that, when applied to a CPA secure encryption scheme and a SUF-CMA secure MAC, it yields a scheme achieving authenticated encryption (in particular, this has Theorem 5.2 as a corollary and the proof is similar to the one for that theorem). Encrypt-and-Mac is clearly weaker and it should not be used. Mac-then-Encrypt, however, while not being secure in general due to the interaction between the encryption scheme and the MAC, is secure in some important particular cases (such as when being used with randomized counter mode) and, in these cases, it may be preferable to Encrypt-then-Mac. The latter is generically—as claimed before—the best of the three mentioned approaches to authenticated encryption and hence is the one that is usually recommended for practical use. Two of the most important Internet protocols, namely, SSL/TLS and IPSec use, respectively, the Mac-then-Encrypt and the Encrypt-then-Mac approaches.

5.5 MACs Based on Universal Hashing

One important cryptographic primitive, especially for authentication-related applications, is provided by hash functions. The generic term *hash function* is applied to functions that take as input an arbitrary bit string and map it to a fixed-length string—called a *message digest* or, simply, a *digest*—which is usually shorter, so that the hash function is said to *compress* the strings. Hash functions play a relevant role in database applications but for use in cryptography they have to satisfy some additional properties that we will explain in a forthcoming section. For now we will restrict our attention to a special class of hash functions generically called *universal hash functions*. The idea is the following: a universal hash function is an efficiently computable family of functions, indexed by a parameter called the key, with the property that the probability over all keys that two different elements in the domain have the same image (then we speak of a *collision*) is small. Thus we will consider a parameterized family of functions $h = \{h_k : A \to B\}_{k \in K}$. Such a family may also be regarded as a single entity and then we often speak of a *keyed hash function*,

which may be identified with the single function

$$h : K \times A \to B$$

given by $h(k, x) = h_k(x)$. Thus we may define:

Definition 5.7 Let $h = \{h_k : A \to B\}_{k \in K}$ be a keyed hash function and ε a positive real number. Then h is *ε-almost universal* if, for any $x \neq y$ in A, $\Pr(h_k(x) = h_k(y)) \leq \varepsilon$, where k is randomly chosen in K.

Remarks 5.3

1. We will see later that in cryptography, fixed hash functions (which do not depend on a key) are often used. However, to study them from the complexity-theoretic point of view we have to look at keyed hash functions.
2. Observe that, if $\mathcal{I} = (\mathbf{Gen}, \mathbf{Mac}, \mathbf{Ver})$ is a MAC with deterministic tag generation algorithm, then the family $\{\mathbf{Mac}_k(-)\}_{k \in K}$ is a keyed hash function with which the MAC is often identified.

The idea behind the use of universal hash functions for message authentication is to apply the hash function to the message and then to process the output obtained by means of a pseudo-random function. Almost ε-universality was generalized by Krawczyk [122] as follows (we may assume here, without loss of generality, that the hash function maps binary strings to binary strings):

Definition 5.8 Let $h = \{h_k\}_K$ be a keyed hash function and $\varepsilon > 0$. We say that h is *ε-almost Xor universal* if, for any $x \neq y$ and any a, we have that $\Pr(h_k(x) \oplus h_k(y) = a) \leq \varepsilon$ for k randomly chosen.

An ε-almost Xor universal hash function h may be used to construct a MAC (called UMAC) by defining the **Mac** algorithm as follows. Consider a pseudo-random function $\{f_k\}_K$ and, for a message m, define:

$$\mathbf{Mac}_k(m) = h_k(m) \oplus f_k(r),$$

where r is a nonce that must not be repeated over all tags computed with the key k. Then Krawczyk proved that this MAC is secure in the sense that the probability of success of a single forgery attempt in a chosen-message attack is no greater that ε.

5.5.1 GCM

We are going to exhibit an ε-almost Xor universal hash function and to build a Maple implementation of a MAC based on this function following the ideas explained above. Actually, the construction is a little more complicated because the MAC is, in fact, a scheme for authenticated encryption called GCM (an acronym for *Galois Counter*

Mode) that follows the Encrypt-then-Mac paradigm. When used for authentication only, GCM becomes a MAC called GMAC. GCM/GMAC has been specified by NIST [70] and, if used following this specification, it has a security reduction, given in [138], to the assumption that the underlying block cipher is a pseudo-random permutation.

One of the main reasons to construct a MAC based on a universal hash function is efficiency: universal hash functions are usually very fast. We are going to implement GCM and GMAC in Maple and we will see that the implementation of GMAC is much faster than, for example, that of CMAC. We do not include a too-detailed account of GCM and its security properties; we will be satisfied instead with giving only a broad description of the scheme as we implement it, and we refer for all the details to [70] and [138].

As mentioned, GCM is an authenticated encryption mode that provides both confidentiality and authenticity. Moreover, the authentication assurance can be provided for both the confidential (i.e., encrypted) data and for additional data that is not encrypted. If the GCM input is restricted to data that is not to be encrypted, then the resulting specialization is a MAC called GMAC. The two main GCM functions are called authenticated encryption and authenticated decryption. The first encrypts the confidential data and computes the MAC tag of both the confidential data and the additional non-confidential data. The authenticated decryption function decrypts the confidential data contingent on the verification of the tag; if verification fails it outputs `FAIL` and the ciphertext is not decrypted (this corresponds to outputting $\perp$ in our previous description of authenticated encryption).

5.5.1.1 GCM Primitives: GHASH and GCTR

The main ingredients of GCM are a universal hash function called GHASH and a variant of CTR mode which is used for encryption, thus we will start by defining these two primitives. GHASH is a so-called "polynomial hash function" because it acts on the message (regarded as a sequence of blocks) by regarding each 128-bit block as an element of the field $\mathbb{F}_{2^{128}}$ and evaluating the polynomial of $\mathbb{F}_{2^{128}}[x]$ whose coefficients are given by these blocks at the value $x = s$, where s is a 128-bit "hash subkey" derived from the GCM key k. Specifically, if s is the 128-bit hash subkey and $b = b_1||b_2||\ldots||b_t$, where the b_i are 128-bit blocks viewed as elements of $\mathbb{F}_{2^{128}}$, then the output of GHASH is the 128-bit block:

$$\sum_{i=1}^{t} b_i \cdot s^{t-i+1},$$

where we denote by '$\cdot$' the multiplication in $\mathbb{F}_{2^{128}}$. Thus GHASH is the hash function $\{g_s\}_{s\in\{0,1\}^{128}}$ given by $g_s(b) = \sum_{i=1}^{t} b_i \cdot s^{t-i+1}$ for $b \in \{0,1\}^{128t}$.

But for a few technical details, it is essentially shown in [138] that, keeping the notation above, GHASH is ε-almost Xor universal for $\varepsilon = t/2^{128}$. The GMAC

construction is similar to UMAC, for which the forgery success probability with this hash function is no greater than $t/2^{128}$, and the authors use the security reduction for the latter mode to derive a security reduction for GMAC. In the same reference a security reduction for GCM encryption is also given.

In order to implement GHASH in Maple, we will define the field $\mathbb{F}_{2^{128}}$ by means of the package `GF`; the irreducible polynomial used is the same as the one used for CMAC, namely $x^{128} + x^7 + x^2 + x + 1$. A technical detail that must be taken into account is that in NIST's implementation of GCM, the 128-bit (16-byte) blocks are identified with the elements of $\mathbb{F}_{2^{128}}$ by using the little-endian convention, i.e, the block $[a_0, \ldots, a_{127}]$ corresponds to the polynomial $a_0 + a_1 x + \ldots + a_{127} x^{127}$ (while, for example, the analogous correspondence used in CMAC uses the big-endian convention). We start by initializing the field in Maple and by giving a function for conversion of 128-bit blocks to elements of the field:

```
> G := GF(2, 128, x^128+x^7+x^2+x+1);
                     G := Z[2][x]`/`<x^128+x^7+x^2+x+1>

> blocktopoly := bl -> modp1((':-ConvertIn')(bl, x), 2):
```

We now give the `GHASH` function. The inputs are the hash subkey given as a 128-bit list `s` and a bit list `l` whose length is a multiple of 128, i.e., equal to `128t`. The output is a 128-bit block given as a list.

```
> GHASH := proc(s::list, l::list)
  local L, t, Y, p, q, i, u;
  L := [ListTools:-LengthSplit(l, 128)];
  t := nops(L);
  Y := [0$128];
  p := blocktopoly(s);
  q := blocktopoly(Y);
  for i to t do
    q := G:-`*`(G:-`+`(q, blocktopoly(L[i])), p)
  end do;
  u := modp1((':-ConvertOut')(q), 2);
  [op(u), 0$(128-nops(u))]
  end proc:
```

Another primitive of GCM is the function `GCTR`, a variation of CTR mode which is used for encryption. The inputs are an AES expanded key `ek`, an initial counter block `ICB` and a message `X`, all of them given as lists of bytes. The incrementing function used for the counter is the previously defined function `inc32`, as specified in [70]. The output is the ciphertext given as a list of bytes.

```
> GCTR := proc(ek::list, ICB::list, X::list)
  local st, n, CB, i;
  if X = [] then return [] end if;
  st := [ListTools:-LengthSplit(X, 16)];
  n := nops(st);
  st := Array(1 .. n, st);
  CB := Array(0 .. 3, 0 .. 3, (i, j) -> ICB[i+4*j+1]);
  for i to n do
    st[i] := BitXorAL(AESEncrypt(ek, CB), st[i]);
    inc32(CB)
  end do;
  [seq(op(st[i]), i = 1 .. n)]
  end proc:
```

5.5.1.2 GCM Authenticated Encryption and Decryption Functions

The preceding functions give us the primitives we need to build the authenticated encryption and authenticated decryption functions of GCM. As usual, we will first define these functions for lists of bytes and then we will build on top of them the higher-level functions that deal with files and strings. We need a function to convert an integer to its binary form. This is accomplished by the function `integer2bits` below, which is entirely similar to `bytetobits` except that is not table-based and hence is slower (this does not matter much as the function is only applied twice in each authentication-encryption). The input parameters are `n` for the decimal integer to be converted and `numbits`, which specifies the minimum number of bits so that if the integer has fewer bits than this number, its binary expansion is completed with leading zeros to make a list of length `numbits` (the function is used in the functions `GCMAE` and `GCMAD` below to convert an integer to its 64-bit binary form). The output of `integer2bits` is the list of bits in the binary representation of the integer, given in big-endian order. In Maple version 12 and above, the built-in function `Bits:-Split` may be used instead, but note that this function outputs the bits in little-endian order.

```
> integer2bits := proc(n, numbits)
  local l;
  l := ListTools:-Reverse(convert(n, base, 2));
  [0$(numbits-nops(l)), op(l)]
  end proc:
```

We next give the authenticated encryption function for lists of bytes, `GCMAE`, which requires four input lists: an AES key `k`, an IV (which is essentially a nonce), the plaintext `P` and the additional authenticated data (AAD, herein called `A`). Moreover, another input will be the byte length `t` of the tag to be produced by the function. The expanded key is required several times here and this is why the key expansion is invoked in this function and not in the lower-level one `GCTR`. There is an initial counter which depends on the IV. In [70] it is recommended that implementations restrict support to the length of 96 bits (12 bytes) for IVs, in order to promote interoperability, efficiency and simplicity of design. Following this recommendation we will adopt 12 as the byte length of the IV and, because of this, the initial counter is just `J = IV||0^31||1`. This initial counter is used twice, the first time increased by 1 (so that, in this case, it is `IV ||0^30||1||0` in bit string format) to encrypt the plaintext `P` by means of `GCTR`, and the other to compute the tag using again `GCTR` and the output of the `GHASH` function previously computed. The output of `GCMAE` is a list consisting of two lists of bytes, the first corresponding to the ciphertext and the second to the tag.

```
> GCMAE := proc(k::list, IV::list, P::list, A::list, t::posint)
  local ek, H, C, c, a, u, v, S, T;
  uses ListTools;
  ek := KeyExpansion(k);
  H := AESEncrypt(ek, Array(0 .. 3, 0 .. 3));
  C := GCTR(ek, [op(IV), 0$3, 2], P);
  c := 8*nops(C);
  a := 8*nops(A);
```

```
u := 128*ceil(c/128)-c;
v := 128*ceil(a/128)-a;
S := GHASH(Flatten(map(bytetobits, convert(ArrayTools:-Alias(H, [16]), list))),
     [op(Flatten(map(bytetobits, A))), 0$v, op(Flatten(map(bytetobits, C))), 0$u,
      op(integer2bits(a,64)), op(integer2bits(c,64))]);
T := GCTR(ek, [op(IV), 0$3, 1], map(bitstobyte, [LengthSplit(S, 8)]))[1 .. t];
[C, T]
end proc:
```

Example 5.9 Let us now test the preceding function by using some of the examples in [138].

1. Test Case 7:

```
> map(bytestohexstring, GCMAE([0$24], [0$12], [], [], 16));
  map(bytestohexstring, GCMAE([0$24], [0$12], [0$16], [], 16));
                  ["", "cd33b28ac773f74ba00ed1f312572435"]
  ["98e7247c07f0fe411c267e4384b0f600", "2ff58d80033927ab8ef4d4587514f0fb"]
```

2. Test Case 16, with a 256-bit key:

```
> giv := hexstringtobytes("cafebabefacedbaddecaf888");
  gk := hexstringtobytes("feffe9928665731c6d6a8f9467308308feffe9928665731c6d6a8f94\
        67308308");
  ga := hexstringtobytes("feedfacedeadbeeffeedfacedeadbeefabaddad2");
  gp := hexstringtobytes("d9313225f88406e5a55909c5aff5269a86a7a9531534f7da2e4c303d\
        8a318a721c3c0c95956809532fcf0e2449a6b525b16aedf5aa0de657ba637b39");
  c := map(bytestohexstring, GCMAE(gk, giv, gp, ga, 16));

  ["522dc1f099567d07f47f37a32a84427d643a8cdcbfe5c0c97598a2bd2555d1aa8cb08e48590dbb\
  3da7b08b1056828838c5f61e6393ba7a0abcc9f662", "76fc6ece0f4e1768cddf8853bb2d551b"]
```

Next we give the authenticated decryption function, which is very similar to `GCMAE`. The input parameters are almost the same, with the ciphertext replacing the plaintext and with an additional list of bytes `T` which corresponds to the tag. The output is either the plaintext if authentication was successful or FAIL otherwise.

```
> GCMAD := proc(k::list, IV::list, C::list, A::list, T::list, t::posint)
  local ek, H, P, c, a, u, v, S;
  uses ListTools;
  if nops(T) < t then
    return FAIL
  end if;
  ek := KeyExpansion(k);
  H := AESEncrypt(ek, Array(0 .. 3, 0 .. 3));
  P := GCTR(ek, [op(IV), 0$3, 2], C);
  c := 8*nops(C);
  a := 8*nops(A);
  u := 128*ceil(c/128)-c;
  v := 128*ceil(a/128)-a;
  S := GHASH(Flatten(map(bytetobits, convert(ArrayTools:-Alias(H, [16]), list))),
       [op(Flatten(map(bytetobits, A))), 0$v, op(Flatten(map(bytetobits, C))), 0$u,
        op(integer2bits(a,64)), op(integer2bits(c,64))]);
  if T = GCTR(ek, [op(IV),0$3,1], map(bitstobyte, [LengthSplit(S,8)]))[1 .. t] then
    P
  else
    FAIL
  end if
  end proc:
```

Example 5.10 We use the authenticated decryption function to validate the result of Test Case 16 in the previous example; we assume that the ciphertext value `c` is still in memory.

```
> GCMAD(gk, giv, hexstringtobytes(c[1]), ga, hexstringtobytes(c[2]), 16);
  evalb(% = gp);
                                    true
```

Below we give functions that implement GMAC, i.e., the MAC part of GCM/GMAC for both text strings and files but we leave as an exercise to write the corresponding Maple functions for authenticated encryption and authenticated decryption of real messages using GCM, in a similar way as we did with earlier modes.

Exercise 5.10 Write GCM Maple functions for authenticated decryption and authenticated encryption that take as input either text strings or binary files and produce as output the concatenation of the IV plus the ciphertext plus the tag (in the case of authenticated encryption) and either the plaintext or `FAIL` (in the case of authenticated decryption).

5.5.2 GMAC

As mentioned above, GMAC is the MAC obtained when GCM is used for authentication only or, in other words, when GCM is specialized to an empty plaintext. Next we give the GMAC functions in Maple for computing the MAC tag (`GMAC`) and for verification (`GMACVer`). The inputs for `GMAC` are the key, the IV, the message (all of them strings) and, optionally, the message type (`text`, `hex` or `file`, with the last being the default) and the tag length given by a keyword parameter `t` (with 16 as default). The output is the MAC tag given as a hex string. We mention here that in [138] and also in [70], it is explicitly mentioned that, for security reasons, the tag length must be fixed for each key.

```
> GMAC := proc(key::{list, string}, iv::string, message::string,
              messagetype::name := file, {t::posint := 16})
  local k, IV, m;
  uses StringTools;
  k := checkkey(key);
  if not (IsHexDigit(iv) and Length(iv) = 24) then
    error "Incorrectly formatted IV"
  end if;
  if not member(t, [$12 .. 16]) then
    error "Incorrect tag length"
  end if;
  IV := hexstringtobytes(LowerCase(iv));
  m := messagetobytes(message, messagetype);
  bytestohexstring(GCMAE(k, IV, [], m, t)[2])
  end proc:
```

The verification function `GMACVer` is given next. The input parameters are the same as those of `GMAC` with the addition of the tag (as a hex string) and without the

tag length. The output is either VALID if the tag corresponds to the message under the given key or INVALID otherwise.

```
> GMACVer := proc(key::{list, string}, iv::string, message::string, tag::string,
                  messagetype::name := file)
  local k, t;
  uses StringTools;
  k := checkkey(key);
  if not (IsHexDigit(iv) and Length(iv) = 24) then
    error "Incorrectly formatted IV"
  end if;
  if not (IsHexDigit(tag) and member(Length(tag), [seq(2*i, i = 12 .. 16)])) then
    error "Incorrectly formatted tag"
  end if;
  t := iquo(Length(tag), 2);
  if evalb(GMAC(k,LowerCase(iv),message,messagetype,':-t'=t) = LowerCase(tag)) then
    VALID
  else
    INVALID
  end if
  end proc:
```

Examples 5.1

1. Test Case 1 in [138]. We compute the tag corresponding to the empty string with the 0 string as IV and key, and then we verify the authenticity of the message:

```
> t0 := GMAC("00000000000000000000000000000000","000000000000000000000000","",hex);
                    "58e2fccefa7e3061367f1d57a4e7455a"

> GMACVer("00000000000000000000000000000000","000000000000000000000000","",t0,hex);
                                  VALID
```

2. We generate a test file in the current directory, named "cmacfile" as follows:

```
> cmactxt := "This is the content of a file used to test GMAC":
  bytestofile(convert(cmactxt, bytes), "cmacfile", false);

47 bytes saved to cmacfile
```

   We compute the GMAC tag (with "000102030405060708090a0b0c0d0e0f" as key and 0 as IV) for this file and then we verify that the tag is valid, which means that the file has not been modified in between:

```
> t1:=GMAC("000102030405060708090a0b0c0d0e0f","000000000000000000000000","cmacfile");
                    "46daef78cd2295c531bf8b8a8271442b"

> GMACVer("000102030405060708090a0b0c0d0e0f","000000000000000000000000","cmacfile",
          t1);
                                  VALID
```

3. Finally, we compute the GMAC tag corresponding to the text string used to define the file cmacfile and we observe that, as expected, the tag is the same as the one for the file:

```
> GMAC("000102030405060708090a0b0c0d0e0f","000000000000000000000000",cmactxt,text);
                    "46daef78cd2295c531bf8b8a8271442b"
```

Exercise 5.11 Write a Maple program that makes a time comparison between CMAC and GMAC when applied to a file. Use it to compare the time taken by the two algorithms on a 1 MB file (GMAC should be about 4.5 times faster than CMAC).

5.6 Collision Resistant Hash Functions

We have already mentioned hash functions and the fact that a generic hash function is one that maps strings of (almost) arbitrary length to short fixed-length strings but, in this section, we are going to deal with hash functions specially designed for cryptographic purposes. Imagine that we want to assign to each message a short fingerprint or message digest that guarantees the message integrity so that if the message is changed by someone or something then the corresponding fingerprint is different and we detect the change simply by comparing it with the original one. These message digests will be called *hash values* or simply *hashes*. In principle, for this setup to work we need that the function

$$\text{message} \mapsto \text{hash value}$$

be injective because otherwise, any other message with the same image would be deemed authentic by this procedure. Now, if we use for the hashes the same alphabet, say $\{0, 1\}$, as for the messages, then the injectivity of the function requires the use of hashes that are, on average, as long as the messages themselves. This defeats the purpose of using these hashes because we want to deal with messages of arbitrary length—or, at least, with very long messages—and we want the hashes to be short and easily manageable; if they have the same length as the messages they are not useful as one would rather check the integrity of the message by direct inspection. So, if we want to use short hashes we have to dispense with injectivity. But then, as already mentioned, all preimages of a given hash value would be authenticated by it, which is clearly undesirable and even more so in a cryptographic context (more about that later). So it seems that we are in a deadlock ... or are we?

The solution is to consider functions $h : \{0, 1\}^* \to \{0, 1\}^n$ that, while far from being injective,[1] look like that to any PPT adversary. The idea is then that it should be computationally infeasible to show, in an effective way, that these functions are not injective or, in other words, to find a collision, i.e., two different messages with the same hash value.

This idea poses another problem. Here, in contrast with the preceding section where we defined universal hash functions, we are dealing with a single, unkeyed function. The idea that finding a collision should be hard is difficult to formalize for a single function because for the given h collisions always exist, and so there is an algorithm that outputs m and m' such that $h(m) = h(m')$ (it is another matter to find these values in practice). In contrast, if keyed hash functions $\{h_k\}$ are considered, this idea may easily be formalized in complexity-theoretic terms: we should simply require that any probabilistic polynomial-time algorithm (or adversary) can find a collision for h_k, where k is randomly chosen, only with negligible probability. Note that in this case there is no such collision-finding algorithm as before; now the

[1] Since $\{0, 1\}^*$ is an infinite set, there is an infinite number of messages which are mapped to the same hash value and, even if in practice we restrict the domain of the function to some $\{0, 1\}^m$ with $m > n$, the function cannot be injective by the pigeonhole principle.

algorithm would have to work for all functions h_k in order to include the one that is chosen at random. Since in what follows we are only going to deal with unkeyed hash functions, we will not go into the details of the formal definition for keyed functions and we refer instead to [109, 4.6.1].

Definition 5.9 A *collision resistant hash function* is a function $h : \{0, 1\}^* \to \{0, 1\}^n$ which is efficiently computable and such that it is hard (in the practical sense) to find a *collision* for h, i.e., a pair $x, x' \in \{0, 1\}^*$ such that $x \neq x'$ and $h(x) = h(x')$.

Remark 5.4 We stress again that the definition of collision resistance just given is informal and hence imprecise but is very useful in practice. It just means that no efficient algorithm for computing collisions is known to exist. For example, if the best collision-finding algorithm known for a given function is estimated to take one million years with the available computing resources, then we clearly would regard the function as collision resistant.

In addition to collision resistance, there are other properties that are often considered for hash functions for cryptographic use. The most important of these properties are the following:

- *Second preimage resistance*: This property means that, given $x \in \{0, 1\}^*$, it is hard to find an $x' \neq x$ such that $h(x) = h(x')$. This property is also called *weak collision resistance* since any h which is collision resistant is obviously second preimage resistant.
- *Preimage resistance*: This means that, for a randomly chosen $x \in \{0, 1\}^*$, it is hard to find, given $y = h(x)$, an $x' \in \{0, 1\}^*$ such that $h(x') = y$. This essentially says that h is a *one-way function*.

Remarks 5.4

1. The relation between preimage resistance and collision resistance has some subtle points. It can be shown that a collision resistant function need not be preimage resistant (see Exercise 5.12 below) but, intuitively, it seems clear that any "genuine" collision resistant or even, second preimage resistant function must also be preimage resistant. The reason is that if the latter property does not hold, then given $x \in \{0, 1\}^*$ we compute $y = h(x)$ and find a preimage x' such that $h(x') = y = h(x)$, so that if $x \neq x'$ we have found a second preimage. If h is a "genuine" hash function for which any hash value has a large number of preimages, then $x \neq x'$ indeed holds with high probability. Hence, for such "genuine" hash functions, we have that collision resistance implies second preimage resistance and this, in turn, implies that the function is one-way. We refer to [90, 8.4] where this point is discussed in detail for a very general class of keyed hash functions.
2. The converses of the implications mentioned in the item above do not hold. As shown below, finding a second preimage (or, in other words, showing that h is not second preimage resistant by finding a collision with a given x in the domain of h) requires a much larger computational effort than merely finding

a collision. Thus a hash function like SHA-1 is currently regarded as second preimage resistant but not as collision resistant (see the discussion in Sect. 5.7). Of course, these terms are applied here in the practical sense and SHA-1 could well be found not to be second preimage resistant in the future. On the other hand, the function $h(x) = x^2 \bmod n$, where n is the product of two large primes, is thought to be one-way by Theorem 3.8 but it is clearly not second preimage resistant because, for a given x, we have that $h(x) = h(-x)$ and $x \neq -x$.

3. The function that assigns to each binary string its parity bit, i.e., the Xor of all the bits in the string, is a hash function with only two possible hash values and, obviously, does not have any of the resistance properties mentioned. This construction can be generalized to an n-bit valued hash function by regarding bit strings as the binary expansions of positive integers and assigning to $x \in \{0, 1\}^*$ the bit string of length n (with leading zeros added if necessary) corresponding to the integer $x \bmod 2^n$. Now, the number of possible hash values is 2^n, which can be large but, again, this hash function does not have any of the required properties for cryptographic use. Collisions are trivially found just by adding leading zeros and, even if we limit ourselves to computing hashes of strings without leading zeros, we only have to ensure that the values of the integers associated with the strings differ by a multiple of 2^n to quickly find collisions. Moreover, preimages are also easily computed. The construction of collision resistant hash functions is nontrivial as we will later see.

Exercise 5.12 Prove that a collision resistant hash function need not be preimage resistant by using the following example [140, 9.20]. Let g be a collision resistant hash function with an n-bit output and define an $(n+1)$-bit hash function as follows:

$$h(x) = \begin{cases} 1||x & \text{if } \mathrm{len}(x) = n, \\ 0||g(x) & \text{otherwise.} \end{cases}$$

5.6.1 A Couple of Applications

We have mentioned that hash functions can be used to ensure message integrity and we will see that these functions have many important cryptographic applications. Before studying them in more detail, we now present, in a generic way, a couple of relevant applications that give an idea of the importance of the resistance properties we have mentioned. A function that is neither collision resistant nor preimage resistant is not useful as a hash function for cryptographic use, so we consider a couple of situations were one of these properties is required.

Protecting passwords and PINs. One simple application is the protection of passwords or PIN numbers. In operating systems, for example, user passwords are not stored and the hash values of these passwords are stored instead. Then, to guarantee access, the system computes the hash of the supplied password and compares it with the stored value. This way, even privileged users that have access to the system are

not able in principle—although the security provided by this method is often weak in practice due to the use of a weak hash function—to recover the password that would allow them to impersonate the legitimate user. A similar method is used to protect PIN numbers in ATM machines. It is clear that, for this setup to be effective, the hash function used should be at least preimage resistant but collision resistance is not required.

Authentication and digital signatures. The main cryptographic application of hash functions is in authentication. We have already seen that the tag generation algorithm of a MAC is really a keyed hash function and also how universal hash functions can be used for building MACs. But the unkeyed hash functions that we are now considering are also very important for authentication and we will see in Sect. 5.6.5 how they can be used to build MACs. Another reason for their cryptographic interest is that they allow the use of cryptographic primitives that accept only short inputs in cryptographic schemes that accept much longer inputs. For example, if we have a MAC $\mathcal{I}$ that serves to authenticate short messages, we can extend its domain by first applying a hash function h to the message to be authenticated and then applying $\mathcal{I}$'s **Mac** algorithm, obtaining a new MAC $\mathcal{I}'$ that computes the tag of m as $\mathbf{Mac}'(k, m) := \mathbf{Mac}(k, h(m))$; the verification algorithm must be modified accordingly by first applying the hash function to the message. If h is collision resistant and $\mathcal{I}$ is UF-CMA secure, then the new MAC $\mathcal{I}'$ can be shown to be UF-CMA secure too, the intuitive reason being that an adversary that comes up with a forgery (m, t) for $\mathcal{I}'$ either has to find a collision $h(m) = h(m_i)$ for some of its queries m_i (thus breaking the collision resistance of h) or, if $h(m) \neq h(m_i)$ for each query, then the pair $(h(m), t)$ gives a valid MAC forgery for $\mathcal{I}$.

As we will see in Chap. 9, a similar method is very important in public-key cryptography, were it is used to build digital signature schemes from public-key encryption schemes. In both cases, what is really authenticated (or signed) is not the message itself but the hash of the message so that, when signing a particular message, all messages with the same hash value are also signed. From a practical point of view it might seem that it is sufficient that h is second preimage resistant because this will prevent an adversary from producing a forgery for a given message it chooses. However, this is really not enough because there are many concrete scenarios in which the stronger property of collision resistance is needed. For example, a party willing to sign a certain message (which could be a contract), might prepare in advance two versions of the message with entirely different meanings but with the same hash value. In Example 5.12 we show a concrete way in which this can be accomplished.

5.6.2 The Merkle–Damgård Construction

We have not yet seen any examples of collision resistant hash functions but in this section we are going to give a construction that is widely used for the purpose of building such functions. It should be emphasized, however, that there are no known examples of hash functions whose collision resistance can be proved without

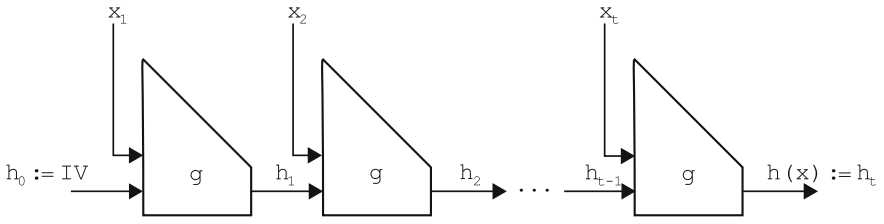
additional assumptions. This property can be conditionally proved for some hash functions under certain hypothesis such as the factoring assumption but these hash functions turn out to be rather inefficient and here we will look at efficient functions that are collision resistant in the above-mentioned informal sense and widely used in practice.

The method we are going to describe is called the *Merkle–Damgård construction* and was independently discovered by these authors in the late 1980s. The idea is to construct a hash function $h : \{0, 1\}^* \to \{0, 1\}^n$ from a fixed-length hash function $g : \{0, 1\}^{n+r} \to \{0, 1\}^n$ (with $r > 0$), usually called a *compression function*. This construction is important because it preserves collision resistance, i.e., if g is collision resistant then so is h. This facilitates the design of practical collision resistant hash functions because compression functions are easier to handle.

There are different variants of the Merkle–Damgård construction according to technical details such as the padding method employed and so on. We will focus on the method applied to construct the SHA-1 hash function and the SHA-2 family of hash functions that will be discussed below. Note that, in practice, the size of the messages to which the hash function will be applied is limited and will typically be well below $2^{n/2}$ bits. Also, the compression function $g : \{0, 1\}^{n+r} \to \{0, 1\}^n$ may, for convenience, be seen as a function of two inputs $g : \{0, 1\}^n \times \{0, 1\}^r \to \{0, 1\}^n$.

Algorithm 5.2. Merkle–Damgård construction.

1. Let $g : \{0, 1\}^n \times \{0, 1\}^r \to \{0, 1\}^n$ be a compression function and IV an n-bit initialization vector.
2. The message x is padded with a 1 bit followed by a variable number of 0 bits and, finally, the encoding of the message length in bits, so that the total length of the padded message is the smallest possible multiple of r.
3. The padded message is parsed into a sequence of r-bit blocks $x_1, x_2, \ldots, x_t$.
4. A hash value is initialized by setting $h_0 := IV$.
5. For i from 1 to t, $h_i := g(h_{i-1}, x_i)$ is computed.
6. Set $h(x) := h_t$.



Merkle-Damgård Construction

The following theorem, due to Merkle and Damgård, proves that this construction preserves collision resistance:

Theorem 5.3 *If, in the Merkle–Damgård construction, g is a collision resistant compression function, then h is a collision resistant hash function.*

Proof The idea is to prove that a collision for h gives two different intermediate values $(h_{i-1}, x_i) \neq (h'_{i-1}, x'_i)$ which are a collision for g. Indeed, suppose that $x \neq x'$ are two strings such that $h(x) = h(x')$ and let $x_1, \ldots, x_t$ and $x'_1, \ldots, x'_{t'}$ be the sequences of blocks corresponding to the padded messages. Similarly, let h_i and h'_i be as in Algorithm 5.2 so that, by hypothesis, we have that $h_t = h'_{t'}$ and hence $g(h_{t-1}, x_t) = g(h'_{t'-1}, x'_{t'})$. Then, either this is a collision for g or both inputs are equal, in which case $x_t = x'_{t'}$ which, in particular means that $\text{len}(x) = \text{len}(x')$ and hence that $t = t'$. Thus we also have that $h_{t-1} = h'_{t-1}$ and hence $g(h_{t-2}, x_{t-1}) = g(h'_{t-2}, x'_{t-1})$. Then, again, either this is a collision for g or $h_{t-2} = h'_{t-2}$. Since $x \neq x'$ there must exist an index i, $1 \leq i \leq t$, such that $x_i \neq x'_i$ and the pair (h_{i-1}, x_i), (h'_{i-1}, x'_i) is a collision for g. □

5.6.3 SHA-256

We are going to apply the Merkle–Damgård construction to obtain SHA-256 (where the initials SHA stand for *Secure Hash Algorithm*), a hash function with a 256-bit output which is, for now, collision resistant in the practical sense that no efficient collision-finding algorithm is known. This function, together with SHA-224, SHA-384 and SHA-512, forms the SHA-2 family that, like the hash function SHA-1, is specified by NIST in [74] in the framework of the *Secure Hash Standard* (SHS). The hash functions in the SHA-2 family are some of the best available today, after other widely used hash functions like MD5 and SHA-1 itself have been shown to be insecure (we will discuss the security of hash functions later).

Next we describe the SHA-256 function. We start by defining the compression function that is used to generate it and we shall denote this function by C. We need a series of auxiliary functions in order to define C. These functions act on 32-bit words, represented here by the letters x, y, z, and produce as output a new 32-bit word. The auxiliary functions are the following:

- $Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$,
- $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$,
- $\Sigma_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$,
- $\Sigma_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$,
- $\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$,
- $\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$,

where $\wedge$ represents the bitwise And, $\neg$ the bitwise complement, $\oplus$ the bitwise Xor, $ROTR^n$ the cyclic right shift by n positions, and SHR^n the right shift by n positions. In Algorithm 5.3 we give the compression function, where the symbol '+' represents addition modulo 2^{32}.

Algorithm 5.3. Compression function of SHA-256.

Input: An initial 256-bit hash value H and a 512-bit message block M.
Output: A new 256-bit hash value.
`Parse` $H = H_0||H_1||\ldots||H_7$ and $M = M_0||M_1||\ldots||M_{15}$ into 32-bit words. `Initialize` 64 constant 32-bit words $K_0^{\{256\}}, \ldots, K_{63}^{\{256\}}$ representing the first 32 bits of the fractional parts of the cube roots of the first sixty-four prime numbers.
`Prepare the message schedule` consisting of sixty-four 32-bit words, as follows:

$$W_t = \begin{cases} M_t & 0 \le t \le 15, \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_1^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63. \end{cases}$$

`Initialize` eight variables with the initial hash value:
$a := H_0; b := H_1; c := H_2; d := H_3; e := H_4; f := H_5; g := H_6; h := H_7$.
`for` t `from` 0 `to` 63 `do`
$T_1 := h + \Sigma_1^{\{256\}}(e) + Ch(e, f, g) + K_t^{\{256\}} + W_t$
$T_2 := \Sigma_0^{\{256\}}(a) + Maj(a, b, c)$
$h := g$
$g := f$
$f := e$
$e := d + T_1$
$d := c$
$c := b$
$b := a$
$a := T_1 + T_2$
`end do;`
`return` $C(H, M) := a + H_0||b + H_1||c + H_2||d + H_3||e + H_4||f + H_5||g + H_6||h + H_7$.

Now, SHA-256 is obtained by applying the Merkle–Damgård construction to the compression function with the following preprocessing:

- **Padding**. If the message has a length of l bits then append a 1 bit to the end of the message, followed by j zero bits, where j is the smallest, non-negative solution to the equation $l + 1 + j \equiv 448 \pmod{512}$. Then append the 64-bit block that is equal to the binary representation of l.
- **Set the initial hash value**. The IV is the 256-bit vector formed by the eight 32-bit words obtained by taking the first 32 bits of the fractional parts of the square roots of the first eight prime numbers.

5.6.4 SHA-256 in Maple

In this section we build a Maple implementation of SHA-256 following the previous discussion and the specification in [74]. We will use some conversion functions from Appendix A, in particular `messagetobytes` and some functions on which it depends. We start by giving some more conversion functions that will be useful. The first of them is the inverse of the function `integer2bits`, which converts a

list of bits to the decimal integer it represents (in Maple version 12 and above this may be done with the built-in function `Bits:-Join`):

```
> bits2integer := proc(lis::list)
  local l;
  l := nops(lis);
  add(lis[j]*2^(l-j), j = 1 .. l)
  end proc:
```

The next function will be used to convert a list of 32 bits to a hexadecimal string with eight hex digits (it can be easily adapted for other sizes but here we only use it as mentioned). Note that if the number represented by the list uses fewer than eight hex digits, leading zeros are added to the string to make its length equal to eight.

```
> bits2hex := proc(l::list)
  local h;
  h := StringTools:-LowerCase(convert(bits2integer(l), hex));
  cat("0"$(8-length(h)), h)
  end proc:
```

Now we are ready to construct the 64 SHA-256 constants. These are 32-bit words which are computed as the first 32 bits of the fractional part of the cube roots of the first 64 primes. We will use a precision of 16 decimal digits in this computation to prevent round-off errors. The next function computes the first 32 bits of the fractional part of x^q, where x and q are real numbers:

```
> frac32bin := (x, q) -> integer2bits(floor(evalf(2^32*frac(x^q), 16)), 32):
```

Now, the list `K` of SHA-256 constants is the following (we do not print it as the binary form of these constants is not very readable):

```
> K := map(x -> frac32bin(x, 1/3), map(ithprime,[$1..64])):
```

We print these constants in hexadecimal, as they appear in [74]:

```
> map(bits2hex, K);
 ["428a2f98", "71374491", "b5c0fbcf", "e9b5dba5", "3956c25b", "59f111f1",
   "923f82a4", "ab1c5ed5", "d807aa98", "12835b01", "243185be", "550c7dc3",
   "72be5d74", "80deb1fe", "9bdc06a7", "c19bf174", "e49b69c1", "efbe4786",
   "0fc19dc6", "240ca1cc", "2de92c6f", "4a7484aa", "5cb0a9dc", "76f988da",
   "983e5152", "a831c66d", "b00327c8", "bf597fc7", "c6e00bf3", "d5a79147",
   "06ca6351", "14292967", "27b70a85", "2e1b2138", "4d2c6dfc", "53380d13",
   "650a7354", "766a0abb", "81c2c92e", "92722c85", "a2bfe8a1", "a81a664b",
   "c24b8b70", "c76c51a3", "d192e819", "d6990624", "f40e3585", "106aa070",
   "19a4c116", "1e376c08", "2748774c", "34b0bcb5", "391c0cb3", "4ed8aa4a",
   "5b9cca4f", "682e6ff3", "748f82ee", "78a5636f", "84c87814", "8cc70208",
   "90befffa", "a4506ceb", "bef9a3f7", "c67178f2"]
```

Next we define some of the operations used in SHA-256 and, as a preliminary step, we start with the bitwise operations $\wedge$, $\oplus$ and $\neg$ which will be defined here for bit lists (and used only in the case of 32-bit lists) and called `bitAnd`, `bitXor` and `bitNot`, respectively. $\oplus$ has frequently been used already (usually for bytes) but here we define it again for bit lists (thus `bitXor` is different from the previously defined function `BitXor`):

```
> bitAnd := proc(l,m)
  [seq(l[i]*m[i], i=1..nops(l))];
  end proc:
```

```
> bitXor := proc()
  add(args[i], i = 1 .. nargs) mod 2
  end proc:

> bitNot := proc(l)
  [seq(1-l[i], i = 1 .. nops(l))]
  end proc:
```

The next function implements the sum of 32-bit words modulo 2^{32}. The function is capable of adding more than two 32-bit lists, as will be done on some occasions:

```
> Summod2e32 := proc()
  local l;
  l := map(bits2integer, [args]);
  integer2bits(add(i, i = l) mod 2^32, 32)
  end proc:
```

We next define the $ROTR^n$ and SHR^n operations on words:

```
> Rotr := (l, n) -> ListTools:-Rotate(l, -n):
> Shr := (l, n) -> [0$n, op(Rotr(l, n)[1+n .. nops(l)])]:
```

Now we give the six SHA-256 'logical functions' of [74, 4.1.2]. We use names similar to those in [74] (where some of these names are given by Greek letters which here we represent by the letters' names).

```
> Ch := (x,y,z) -> bitXor(bitAnd(x,y), bitAnd(bitNot(x),z)):
> Maj := (x,y,z) -> bitXor(bitAnd(x,y),bitAnd(x,z), bitAnd(y,z)):
> Sigma0 := x -> bitXor(Rotr(x,2), Rotr(x,13), Rotr(x,22)):
> Sigma1 := x -> bitXor(Rotr(x,6), Rotr(x,11), Rotr(x,25)):
> sigma0 := x -> bitXor(Rotr(x,7), Rotr(x,18), Shr(x,3)):
> sigma1 := x -> bitXor(Rotr(x,17), Rotr(x,19), Shr(x,10)):
```

We are now ready to deal with the SHA-256 preprocessing. Padding is carried out by the next function. The input is the message given as a list of bits and the output is the padded list.

```
> shapad := proc (message::list)
  local l, j, bl;
  l := nops(message);
  bl := integer2bits(l, 64);
  j := 447-l mod 512;
  [op(message), 1, 0$j, op(bl)]
  end proc:
```

The padded message must be partitioned into 512-bit blocks and each of these blocks, in turn, into sixteen 32-bit words. This is accomplished by the next couple of functions:

```
> partition := (lis,long) ->
               [seq(lis[1+(i-1)*long..i*long],i=1..iquo(nops(lis),long))]:
> preprocess := lis -> map(x -> partition(x, 32), partition(lis, 512)):
```

Next we define the initial hash value `Hinit` used by the compression function of SHA-256. It consists of eight 32-bit words which are obtained as the first 32 bits of the fractional part of the square roots of the first eight primes. We use the previously defined function `frac32bin` to compute these fractional parts:

```
> Hinit := map(x -> frac32bin(x, 1/2), map(ithprime, [$1 .. 8])):
```

We display these words in hex to allow easy comparison with those in [74]:

```
> map(bits2hex, Hinit);

  ["6a09e667", "bb67ae85", "3c6ef372", "a54ff53a", "510e527f", "9b05688c",
    "1f83d9ab", "5be0cd19"]
```

Preprocessing is finished and we are now ready to go into the implementation of the SHA-256 compression function. The first thing to do is to develop the "message schedule", which will be called in each iteration of the compression function, i.e., for each message block. The message schedule consists of sixty-four 32-bit words (lists in our case) which are produced by the following function. Note that the remember option[2] is important here as we are using recursion (the function `W` calls itself).

```
> W := proc(block, i::nonnegint)
  option remember;
  if i <= 15 then
    block[1+i]
  elif i <= 63 then
    Summod2e32(sigma1(W(block,i-2)),W(block,i-7),sigma0(W(block,i-15)),W(block,i-16))
  end if
  end proc:
```

The next function, `sha256`, is an implementation of SHA-256 that acts on a message given as a list of bytes and produces a hash value in the same format. We separate this function from the main one that will act on messages given either as files or strings because the byte list format is more convenient for later use. This function closely follows the specification—and the notation—in [74].

```
> sha256 := proc(message)
  local m, N, ms, i, t, a, b, c, d, e, f, g, h, H, T1, T2;
  m := ListTools:-Flatten(map(bytetobits, message));
  m := preprocess(shapad(m));
  N := nops(m);
  H := Hinit;
  for i to N do
    ms := [seq(W(m[i], t), t = 0 .. 63)];
    a := H[1]; b := H[2]; c := H[3]; d := H[4];
    e := H[5]; f := H[6]; g := H[7]; h := H[8];
    for t from 0 to 63 do
      T1 := Summod2e32(h, Sigma1(e), Ch(e, f, g), K[t+1], ms[t+1]);
      T2 := Summod2e32(Sigma0(a), Maj(a, b, c));
      h := g; g := f; f := e; e := Summod2e32(d, T1);
      d := c; c := b; b := a; a := Summod2e32(T1, T2)
    end do;
    H := zip(Summod2e32, [a, b, c, d, e, f, g, h], H)
  end do;
  map(bits2integer, map(x -> ListTools:-LengthSplit(x, 8), H))
  end proc:
```

Next we give `SHA256`, which computes the SHA-256 hash of a message given as a file, a hex string or a text string. The input parameter `messagetype` must be specified to `file`, `hex` or `text`, with the last being the default. The output is a 256-bit hex string.

[2] This option lets Maple record in a table the result of each iteration of the procedure, so that it can be retrieved for successive iterations without having to be recomputed; see Maple's help for details.

```
> SHA256 := proc(message, messagetype::name := text)
  local m;
  m := messagetobytes(message, messagetype);
  bytestohexstring(sha256(m))
  end proc:
```

Examples 5.2

1. We compute the example 'One Block Message Sample' in [148]:

```
> SHA256("abc");
     "ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad"
```

2. The example labeled 'Two Block Message Sample' in [148] is the following:

```
> SHA256("abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq");
     "248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1"
```

3. An example involving both a text string and a file is the following. We first create the file in the current directory:

```
> sha256txt:="This string is written to the file sha256file in order to test SHA-256":

> bytestofile(convert(sha256txt, bytes), "sha256file");
70 bytes saved to sha256file
```

   Next, we compute the hashes for the string `sha256txt` and the file "`sha256file`"; of course, since the file only contains this string, these values will be equal:

```
> SHA256(sha256txt);
     "89c20c495a6b5ae462b08dd4e4791be989731d0402ca7773796995fe8ee9098b"

> SHA256("sha256file", file);
     "89c20c495a6b5ae462b08dd4e4791be989731d0402ca7773796995fe8ee9098b"
```

Exercise 5.13 In a similar way to the preceding implementation of SHA-256, give Maple implementations of the remaining functions in the SHA-2 family, namely, SHA-224, SHA-384 and SHA-512, following [74].

5.6.5 MACs with Hash Functions: HMAC

The idea of using unkeyed cryptographic hash functions to construct a MAC is quite recent and one of the motivations is that some popular hash functions such as SHA-1 and SHA-2 are noticeably faster than block ciphers in software and hence MACs based on them are also significantly faster than, say, CBC-MAC and its variants. One of the problems that arises when trying to apply this idea is that unkeyed hash functions do not have secret keys, which is a necessary ingredient of a MAC. Several methods of combining a hash function with a secret key to obtain a MAC have been proposed and the one which is, by far, the most popular is called HMAC (for *hash-based MAC*). HMAC was introduced by Bellare, Canetti and Krawczyk in 1996 [13] and has recently been standardized by several bodies, in particular as the Internet standard RFC 2104 [124], as an ISO standard (ISO/IEC 9797-2), and also by NIST [76]. Apart from being very simple and efficient, one of the advantages of HMAC is that it has a reductionist security proof under reasonable assumptions about the strength of the underlying hash function.

5.6.5.1 HMAC

We are going to describe the HMAC algorithm with a view to implementing it in Maple, mentioning its security properties by pointing to the appropriate references but without going into the details. We assume that we are given a cryptographic hash function H that is obtained by means of the Merkle–Damgård construction from a compression function that takes inputs of B bytes and produces a hash value of L bytes (for example, if H is SHA-256, then $B = 64$ and $L = 32$). Optionally, we may use a parameter t such that $4 \leq t \leq L$ which serves to truncate HMAC's output to t bytes. We denote by length(K) the byte length of the key K.

Algorithm 5.4. HMAC.

Input: A key K and a message *text*.
Output: A 256-bit hash value.
`Key Processing`:
    `if` length$(K) = B$ `then`
        $K_0 := K$
    `elif` length$(K) > B$ `then`
        $K_0 := H(K)||00...00$ ($B - L$ zeros are appended to create a B-byte string)
    `else`
        $K_0 := K||00...00$ ($B -$ length(K) zeros are appended to create a B-byte string).
`Compute MAC tag`:
    ipad := the byte `x36` repeated B times (*inner pad*);
    opad := the byte `x5c` repeated B times (*outer pad*);
    `return` $H((K_0 \oplus opad)||H((K_0 \oplus ipad)||text))$.

For security reasons it is recommended that len$(K) \geq L$ but a longer key does not add significantly to the security if the key is random, so that often len$(K) = L$. As mentioned, the output is often truncated, for example to 128 bits if a hash function with a 256-bit output is used. One interesting characteristic of HMAC is that it uses the hash function H as a "black box", which makes it very easy to implement and also makes it easy to replace the hash function by another one if necessary.

As already mentioned, HMAC has a security reduction to very reasonable assumptions which, as shown in [13], boil down to some mild constraints on the underlying hash function. Indeed, it is shown there that an adversary able to forge the HMAC function is also able to find collisions in the hash function even when the IV (namely, the IV in the Merkle–Damgård construction from which H is obtained) is random and secret which, in principle, is harder than finding collisions in the hash function as it would require interaction with the legitimate user of the function in order to generate the input/output pairs. The property of hash functions being violated here is weaker than collision resistance but it seems that some popular hash functions like MD5 and SHA-1 which have recently been shown not to be collision resistant (see the discussion in Sect. 5.7) can also be attacked in this way. However, this possibility (like the standard collision attacks against MD5 and SHA-1 which make these hash functions unsuitable for many cryptographic uses) seems not to have led to any suc-

cessful attacks against the versions of HMAC based on these functions. Thus HMAC still seems secure in practice, and even more so if a collision resistant hash function is used.

5.6.5.2 HMAC in Maple

Next we develop a Maple implementation of HMAC, specifically HMAC-SHA-256, based on the hash function previously implemented as `SHA256`. We will follow Algorithm 5.4 but we are going to simplify the key processing step somewhat. Since SHA-256 uses 512-bit (64-byte) input blocks, and key sizes 512 bits or less seem perfectly adequate from a security point of view, we are going to consider only keys up to this size. In fact, we will only allow 16, 24 or 32-byte keys (i.e., AES keys, with the largest size being the preferred one for security reasons) since this seems sufficient and, this way, we may use the Maple function `checkkey` (from Appendix A) for key processing. In addition to `checkkey` we will also use `messagetobytes` and `bytestohexstring`, also from Appendix A, as well as the previous functions `BitXor` and, of course, `sha256`.

The function `HMAC` is given below. It takes three inputs: the key given either as a list of bytes or as a hex string, the message given as a string, and a name which can be either `file, text` or `hex`, with the last as default. In case `file` is specified, the second argument should be the name of a file in the current directory or given with the full path if the file is located elsewhere. The output is the MAC tag given in hexadecimal.

```
> HMAC := proc(key, message, messagetype::name := hex)
  local m, k, ipad, opad;
  m := messagetobytes(message, messagetype);
  k := checkkey(key);
  k := [op(k), `$`(0, 64-nops(k))];
  ipad := [`$`(54, 64)];
  opad := [`$`(92, 64)];
  m := sha256([op(zip(BitXor, k, ipad)), op(m)]);
  bytestohexstring(sha256([op(zip(BitXor, k, opad)), op(m)]))
  end proc:
```

Examples 5.3 We compute a couple of examples taken from [110]:

```
> HMAC("0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b",
       "Hi There", text);

     "198a607eb44bfbc69903a0f1cf2bbdc5ba0aa3f3d9ae3c1c7a3b1696a0b68cf7"

> HMAC("0102030405060708090a0b0c0d0e0f10111213141516171819 1a1b1c1d1e1f20",
       "cdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcd\
        cdcdcdcdcdcdcdcdcdcdcdcd");

     "372efcf9b40b35c2115b1346903d2ef42fced46f0846e7257bb156d3d7b30d3f"
```

As mentioned, the fact that the hash function in HMAC acts as a black box makes it easy to build variants of the preceding implementation, which we leave as an exercise.

Exercise 5.14 Give Maple implementations of HMAC-SHA-224, HMAC-SHA-384 and HMAC-SHA-512 by using the corresponding hash functions as indicated in the names.

5.7 The Birthday Attack on Hash Functions

As we have seen, a hash function suitable for cryptographic use should be collision resistant or, in some special cases which we have not discussed in detail, second preimage resistant or preimage resistant at least. We are going to describe a generic collision attack that gives a lower bound for the length of the hash values of a cryptographic hash function.

5.7.1 The Birthday Paradox

In order to understand the collision attack we are going to present, let us first consider a brute-force attack against preimage resistance or second preimage resistance. If the hash function considered, h, produces hashes of length n, then there are 2^n possible hash values. Suppose then that, given a hash value y, the preimage resistance is attacked by picking random strings x and computing $h(x)$ until we find a solution, i.e., a string which hashes to y. Then the expected number of trials is 2^n by Proposition 2.2, and this is also the number of trials expected in a second preimage attack where, given a string x, one tries to find a different string x' such that the pair (x, x') is a collision for h.

It could seem at first sight that this is also the average cost of finding a collision of h and this is indeed the case if an initial string is selected and then further strings are picked at random until finding one with the same hash as the initial string. But this is far from being the best strategy to search for a collision and the reason is the so-called *birthday paradox* which arises as a response to the following question: How many people must be in a room for the probability that at least two of them share the same birthday to be $> 1/2$?

The answer, assuming that birthdays are independent and uniformly distributed among 365 days, is that 23 persons suffice. This is called a 'paradox' because the number 23 is surprisingly low at first sight, especially if one compares it, for example, with the minimum number of persons so that the probability that at least one of them shares their birthday with a *fixed* person is $> 1/2$. If we view the *birthday map*, i.e., the function that assigns to each person his/her birthday as a sort of toy hash function with only 365 hash values, the above question is like asking about the number of hash values that should be computed to find a collision with probability $> 1/2$, while the problem referring to a fixed person asks for the probability of finding a collision with something which is fixed in advance, i.e., it is similar to finding a second preimage of the birthday of the specified person. The same reasoning that applies to this toy example applies also to hash functions in general and makes it *much easier* to find a collision than to find, say, a preimage. Let us quantify this claim.

Theorem 5.4 *Suppose that r integers between 1 and n are chosen at random with uniform probability distribution. If $p(n,r)$ denotes the probability that at least two of them are the same, we have that $p(n,r) \geq 1 - e^{-r(r-1)/2n}$. If n is large, then $p(n,r) \geq 1/2$ when $r \approx 1.18\sqrt{n}$.*

Proof Let E_i be the event that, after having chosen i numbers, with $i \leq r$, all of them are different. Then the probability that the $(i+1)$th number chosen is different from all the previous ones is $(n-i)/n$ and hence we see that $\Pr(E_{i+1}|E_i) = (n-i)/n = 1 - \frac{i}{n}$. Thus, if we set $q(n,r) = 1 - p(n,r)$, we have, bearing in mind the fact that $\Pr(E_{i-1}|E_i) = 1$, $\Pr(E_1) = 1$, and Bayes' formula, that

$$q(n,r) = \Pr(E_r) = \Pr(E_r|E_{r-1}) \cdot \Pr(E_{r-1}) = \cdots = \prod_{i=1}^{r-1} \Pr(E_{i+1}|E_i) = \prod_{i=1}^{r-1} (1 - \frac{i}{n}).$$

Now, from Taylor's series for the natural logarithms $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots = \sum_{i=1}^{\infty} \frac{(-1)^{(i+1)} x^i}{i}$, we have that $1 - x \leq e^{-x}$. Thus $1 - \frac{i}{n} \leq e^{-i/n}$ and hence

$$q(n,r) = \prod_{i=1}^{r-1} (1 - \frac{i}{n}) \leq \prod_{i=1}^{r-1} e^{-i/n} = e^{-\sum_{i=1}^{r-1} \frac{i}{n}} = e^{-r(r-1)/2n},$$

from which it follows that

$$p(n,r) \geq 1 - e^{-r(r-1)/2n}.$$

If we set $1 - e^{-r(r-1)/2n} = 1/2$, we obtain $2 = e^{r(r-1)/2n}$ and hence $\ln 2 = r(r-1)/2n$. When r is large we can approximate $r \approx r - 1$ and from the preceding equality we obtain $r^2 \approx 2n \ln 2$ and hence

$$r \approx \sqrt{2n \ln 2} \approx 1.18\sqrt{n},$$

so that $p(n,r) \geq 1/2$ for this value of r. □

Example 5.11 Let us apply the preceding result to the birthday problem with the help of Maple. If birthdays are uniformly distributed and we pick r people at random, then their birthdays behave like the integers in Theorem 5.4 with $n = 365$. Thus we have:

```
> evalf(1.18*sqrt(365));
                              22.54386834
```

which suggests that 23 people are indeed enough for the probability of two of them sharing the same birthday to be $\geq 1/2$. In this case the involved numbers are relatively small and we can compute the probability as a function of the number of people, without using the preceding approximation, as follows:

```
> p := (n, r) -> 1-(mul(n-i, i = 0 .. r-1))/n^r:
  ep := (n, r) -> 1-exp(-(1/2)*r*(r-1)/n):
```

```
> for r in {20, 23, 30, 40, 50, 60} do
    print(r, p(365., r), ep(365., r))
  end do;

                    20, 0.4114383837, 0.4058051274
                    23, 0.5072972342, 0.5000017522
                    30, 0.7063162428, 0.6963200177
                    40, 0.8912318099, 0.8819900460
                    50, 0.9703735796, 0.9651312541
                    60, 0.9941226609, 0.9921662587
```

In each of the preceding rows, the first number represents r, the second is the real probability of a collision and the third the estimate of this probability given by the preceding theorem. We see that the probability for $r = 23$ is indeed greater than (and close to) $1/2$ as predicted by the theorem, so that $n = 365$ and $r = 23$ are large enough for the estimate to be accurate. Note that the estimate given by the theorem is very close already for these relatively small numbers and also that the probability of a collision grows quickly with r and is already near 90 % for 40 people and over 99 % for 60 people.

There are many versions of the birthday paradox and another one, with cryptographic interest in relation to collisions of hash functions, arises when we consider two sets of strings and ask about the probability of a collision between a member of the first set and a member of the second set. We will restrict ourselves to the case in which both sets have the same size, which is enough for our purposes.

Theorem 5.5 *Suppose that two sets of r integers between 1 and n are chosen at random with uniform probability distribution, where $r < \sqrt{n}$. If n is large, the probability that the two sets overlap is $\geq 1 - e^{-r^2/2n}$. Moreover, this probability is close to $1/2$ when $r \approx 0.83\sqrt{n}$.*

Proof Since $r < \sqrt{n}$, it is likely that the r integers of each set are all different and we assume that this is the case, but the argument will work even if there are a few repeated elements in the sets.[3] We observe that the probability that one given element of the first set is different from each element in the second set is $(1 - \frac{1}{n})^r$ and hence the probability q that the two sets are disjoint is $(1 - \frac{1}{n})^{r^2}$. Using the approximation $1 - x \approx e^{-x}$ for $x = 1/n$ we obtain that the probability p that the two sets overlap is

$$p = 1 - q \approx 1 - e^{-r^2/n}.$$

Now, equating this value to $1/2$ we obtain $\ln 2 = r^2/n$ which, in turn, gives:

$$r = \sqrt{n \ln 2} \approx 0.83\sqrt{n}.$$

□

[3] Note that, strictly speaking, a set cannot have repeated elements and we are really dealing here with *multisets*, which are similar to sets except for the fact that they can contain multiple instances of each member. But we shall commit abuse of language and speak of sets anyway.

Exercise 5.15 In the following questions, we assume that people are randomly chosen and birthdays uniformly distributed among the possible ones.

(i) Find the minimum number of people in a room such that the probability that at least one of them shares a birthday with another person entering the room is $\geq 1/2$.
(ii) Find the minimum value of n such that if a group of $2n$ people is separated into two groups of n people each, the probability that one person of the first group shares a birthday with one person of the second group is $\geq 1/2$.

5.7.2 The Birthday Attack

The birthday paradox gives rise to a brute-force collision attack against hash functions which is much more efficient than fixing a string and trying to find another one with the same hash value. The idea is simply to compute as many hash values as possible for strings chosen at random, store these strings together with their hash values, and then compare these values trying to find a collision. When the number of hash values computed is slightly larger than $\sqrt{2^n}$, where n is the bit length of the hash function, then the probability of a collision among two of these strings will be $\geq 1/2$, so that finding a collision requires time $O(2^{n/2})$. Why is this so?

The reason is simple: if we pick strings at random and assume that their hash values are uniformly distributed among all the possible values, then computing these hashes is like choosing at random integers between 1 and 2^n, so that Theorem 5.4 applies.

To better appreciate the possible impact of this *birthday attack*, let us consider a widely used hash function: MD5. MD5 was introduced by R. Rivest in 1991; it is obtained from a compression function by the Merkle–Damgård construction and has an output length of 128 bits. Thus a brute-force attack against the second preimage resistance is out of question, as 2^{128} hash operations are far from feasible today. However, a birthday attack can find a collision by just computing slightly more than 2^{64} hash operations and is regarded as feasible (although costly) by taking as reference some distributed computations of similar size that have already been done. In consequence, MD5 is regarded as broken since a few years ago and its usage is strongly discouraged. In fact, stronger attack techniques against MD5 have been discovered, starting with the work of Xiaoyun Wang and colleagues who found the first collisions for MD5 in 2004 [196] and leading to attacks that are able to find MD5 collisions in just a few seconds on a personal computer. Furthermore, it was shown in [188] how to create a rogue *Certification Authority certificate* trusted and regarded as valid by all common web browsers on the date it was announced (a demonstration is available at [185]). We will describe these certificates when studying public-key cryptography and public-key infrastructures but, for now, let us just mention that this certificate, constructed by exploiting MD5 weaknesses, would allow its authors to

issue SSL certificates (the ones used for secure web browsing) to any website they like, including rogue websites claiming to be legitimate ones.

It is important to notice that, even if these stronger attacks were not known, a generic birthday attack is sufficient today to break the collision resistance of MD5 and of any hash function with a 128-bit output. This imposes a lower bound on the output length of any cryptographic hash function and 160 bits is a bare minimum that is already starting to look insufficient. The most widely used hash function today is SHA-1, introduced in 1995 and currently specified in [74]. SHA-1 is also obtained from the Merkle–Damgård iterative construction and has an output length of 160 bits, which means that a generic birthday attack would require just over 2^{80} hash operations. This seems achievable in the near future and, moreover, a collision attack requiring only 2^{69} operations—improved later to require just 2^{63} operations —was announced already in 2005, in [195]. This attack should be certainly regarded as feasible and other improved attacks have also been announced. Thus, although no actual collisions have been found so far, it is only a matter of time and SHA-1 is also regarded as broken. It is clear that a 160-bit output is no longer adequate for future use and so the hash functions that are currently being recommended are those in the SHA-2 family, such as SHA-256 implemented above. These functions have not shown noticeable weaknesses so far and are secure against generic birthday attacks because their minimum output length is 224 bits for SHA-224 which would put the cost of such an attack at over 2^{112} operations, which is currently infeasible. Even so, NIST has set up an open contest to design the new SHA-3 standard whose winner Keccak, was announced in October 2012.

Exercise 5.16 Suppose that h is a hash function with a 128-bit output. Use Maple's function `ep` defined in Example 5.11 to compute numerical estimates of the probability that a collision for h will be found in case 2^{65} or 2^{66} hash values are computed. Obtain similar estimates in case h has a 256-bit output and the number of hash values computed is one of the following: 2^{66}, 2^{129} or 2^{130}.

The preceding discussion of birthday attacks should not be viewed as implying that a hash function which is not vulnerable to a birthday attack is collision resistant. On the contrary, resistance to birthday attacks is only a necessary condition that is far from sufficient for collision resistance. For example, we may consider a hash function that assigns to each string its first 256 bits. This function is immune to a birthday attack for the same reason that SHA-256 is: the attack would require the computation of over 2^{128} hash operations. However, finding collisions for this function is entirely trivial.

As we have mentioned, the birthday attack is not efficient against (second) preimage resistance and so it might seem that its applicability is very limited because in applications such as authentication and digital signatures an adversary would not be interested in a collision with a random message but, rather, in a collision with a specific message, namely, the one that is being authenticated or signed. This is, however, a naive misconception as we are going to show with an example that shows how a birthday attack can be used to obtain a collision between 'meaningful' messages which are far from random.

Example 5.12 **The birthday attack in Maple.**
Eve is going to purchase a property from Alice and they agreed that the price that Eve will pay is $950000, i.e., nine hundred fifty thousand dollars. The contract is going to be digitally signed (see Chap. 9) and the 40-bit hash function obtained by truncating the MD5 output to the first 40 bits will be used. This is a very weak function but we will assume for the sake of this example—in order to make it easily computable with Maple—that Alice and Eve live in a world with low computational resources in which computing 2^{26} hash operations is (barely) feasible but computing 2^{40} such operations is not. The digital signature algorithm used follows the 'hash-then-sign' paradigm, which means that what is really signed is the hash value of the contract (see Sect. 9.3 for details). Thus, if Eve is able to find another contract with the same hash as the one they agreed and with a different content, she could later claim that this other contract is the one that Alice signed. She only has $150000 available and this is the amount she would like to pay for the property so she starts to think how she can forge another contract with this amount and the same hash as the legitimate one. She thinks about the following possibility: make a "bad contract" with the smaller amount in it and make a lot of variants of this contract by changing nonessential things so that the content remains the same, hoping that one of these variants will collide with the legitimate contract.

But Eve soon realizes that this approach will lead her nowhere because this way she has to carry out 2^{40} hash operations on average. This follows from the theory already discussed but Eve can also arrive at this conclusion with the help of Maple as follows. Let p be the probability that a random message produces a collision with the legitimate contract. Then $p = 2^{-40}$ and so, if she keeps on generating bad contracts and we assume they behave as if they were random, then the probability that the ith trial succeeds and all the previous trials fail is $(1-p)^{(i-1)}p$. Thus, taking into account the definition of expectation in Definition 2.5, the expected value of the number of messages she would have to try until finding a collision is:

```
> sum(i*(1-p)^(i-1)*p, i = 1 .. infinity);

                                  1/p
```

This is equal to 2^{40} so it is not feasible for Eve to carry out this computation on her personal computer.

Then Eve recalls the birthday attack but the problem is that she wants to forge the legitimate contract and not a random one. But she soon realizes that if she makes a large number of variants of both the legitimate contract and the bad one, she has a chance of finding a collision. So she proceeds as follows.

The hash function that will be used in the signature is:

```
> Hash40 := str -> StringTools:-SubString(StringTools:-Hash(str), 1 .. 10):
```

The statement (contract) that both parties agreed to sign is (this is only an extract but everything works the same):

```
> m1 := "By signing this statement Eve agrees to pay Alice the amount of $950000
(nine hundred fifty thousand dollars) for the house mentioned below"
```

But the message Eve would like Alice to sign is a little different. It is as follows:

```
> m2 := "By signing this statement Eve agrees to pay Alice the amount of $150000
(one hundred fifty thousand dollars) for the house mentioned below"
```

Eve wants to find a variant of `m2` with the same hash value as a variant of `m1`, so that these variants have the same meaning as `m2` and `m1`, respectively. Later she will claim that they signed the variant of the fraudulent message `m2`, so that she only has to pay Alice $150000. To build the variants of the messages Eve can just identify several places in which the wording of the message can be changed without changing the meaning. If there are t positions in which a change can be made, then by making/not making the possible change she will obtain 2^t variants of the message, all with the same meaning. In order to automatize the construction of these variants, Eve has the following idea. The involved messages have a number of spaces (ASCII-32 characters) and it is possible to replace some of them by another character that looks the same when printed or displayed on a computer screen. That character is ASCII-160 (or the character with hex code 0xA0 in the ISO/IEC 8859 standard) which is a "no-break space", i.e., a space used to prevent line breaks at its position. So Eve is going to do the following. She is going to build the required variants of `m1` and `m2` by just replacing some of the ASCII-32 spaces by ASCII-160 spaces. All the variants of each one of the two messages `m1` and `m2` look the same and keep the meaning of the original message but actually they are different and they will have different hash values. Since `m1` and `m2` have 22 spaces each, this method allows Eve to take $t = 22$ and to build up to 2^{22} variants of each of these two messages.

Each message variant will be associated with a number between 0 and $2^t - 1$ (we will call this number the "message number") and will have an ASCII-32 space replaced by an ASCII-160 one at the ith space position if the corresponding bit in the binary expansion of this number (starting from the least significant one) is a "1" while the space will not be changed if the corresponding bit is a "0". Note that the ASCII-160 spaces in a message variant could make it look slightly odd when printed as they could force line breaks to occur at unexpected places. On average one expects half of the spaces in the first t space positions to be replaced by non breaking spaces so this would not be a problem most of the time. However, if the message to be signed is sufficiently long, Eve can prevent the undesired side effects by selecting the space positions where ASCII-32 spaces can be replaced by ASCII-160 ones in such a way that no two consecutive spaces in the message can be replaced by ASCII-160 spaces. An even more effective way to prevent uneven line breaking would be to select the replaceable space positions so that, when the message is printed in a more or less standard size, the replaceable space positions do not appear near the final positions of each line.

The function to replace a character (the one in the nth position) in a string by an ASCII-160 space is the following:

```
> replacespace := proc(str, n)
  StringTools:-Delete(StringTools:-Insert(str, n, StringTools:-Char(160)), n .. n)
  end proc:
```

The next function builds the message variants. The input is a text string, a number n such that $0 \leq n < 2^t - 1$, where t is less than or equal to the number of spaces in the text string, and a list containing the positions of the spaces in the string (in a moment, we will see how Eve can build this list). The output is the message variant in which the spaces corresponding to "1" bits in n are replaced by ASCII-160 spaces as explained above.

```
> variant := proc(n, message, spaceslist)
  local bin, v, l, i;
  bin := convert(n, base, 2);
  v := message;
  l := spaceslist;
  for i to nops(bin) do
    if bin[i] = 1 then
      v := replacespace(v, l[i])
    end if
  end do;
  v
  end proc:
```

For example, the variant of `m1` corresponding to the number 1553 is:

```
> v1553 := variant(1553, m1, [StringTools:-SearchAll(" ", m1)]);

"By signing this statement Eve agrees to pay Alice the amount of $950000
  (nine hundred fifty thousand dollars) for the house mentioned below"
```

Of course, the message looks the same as the original one but four of the spaces are non-breakable now; we show below which these spaces are.

Observe that the list of space positions is determined by the text string and hence it would not be necessary to pass it to the `variant` procedure as a separate argument. But Eve is doing it this way because, when she mounts the birthday attack, she will have to build many variants of the same message and it would be wasteful to compute the spaces list when computing each of these variants. On the other hand, observe also that another way to see that `v1553` and `m1` are really different is just by computing their hash values and putting them side by side:

```
> Hash40(m1), Hash40(v1553);

                          "473c560db9", "17cc0d8ead"
```

We can check the positions in `v1553` where the ASCII-160 spaces have replaced the ASCII-32 ones. They correspond to the four "1" bits in the binary expansion of 1553 and are the following:

```
> StringTools:-SearchAll(StringTools:-Char(160), v1553);

                                  3, 30, 54, 61
```

Now, Eve is ready to mount her attack. She knows Theorem 5.5 according to which if she builds $1.66 \cdot 2^{19}$ variants of each of the two contracts she already has a chance approximately equal to $1/2$ of finding a collision. So she intends to consider a few more variants and her first idea, aiming to reduce searching to a minimum, is to build an array indexed by all the possible hash values, i.e., the numbers from 0 to $2^{40} - 1$, and then to compute, say, 2^{21} hash values of variants of `m1` and store the

corresponding message number (which allows the retrieval of the message) in the entry whose index is the obtained hash value (storing database entries this way is one important—noncryptographic—application of hash functions). Then she would start computing the hash values of variants of `m2` and, for each of them, she would check whether the array entry corresponding to that index is filled and, if this is the case, she would obtain the required collision without having to search the array. The problem is that this method requires the use of a one-dimensional array of size 2^{40} and Maple cannot handle arrays so large (and much less in Eve's world). This is an extreme example of a "time-memory trade-off" in which using a very large array—which means using a lot of memory—reduces searching, and hence computing time, to a minimum.

Exercise 5.17 Use Maple to implement the attack just mentioned when the hash function used is MD5 truncated to its first 24 bits. Use a 2^{24}-sized Maple `Array` to store 2^{14} message numbers, corresponding to variants of `m1`, in the array entries whose indices are the corresponding hash values and then search for a collision with one of the variants of `m2`.

Eve regrets that she does not have enough memory to carry out this attack against `Hash40` but, things being as they are, she thinks of another way to do it. The problem with the preceding method is that using hash values as indices requires a 2^{40}-sized `Array` but what if she instead uses the message numbers as indices? Eve soon realizes that, as she only is going to use 2^{21} values for the variants of `m1`, she can store them in a one-dimensional `Array` of size 2^{21} indexed this way. Then she can compute the hashes of variants of `m2` until finding a collision. The problem now is that, in exchange for the much lower memory requirements, this method requires searching the array for each computed hash value of a variant of `m2`. Eve knows that searching an unordered array (or list) is very inefficient (the naive algorithm that scans the array elements one by one has linear complexity $O(n)$, where n is the array length) and so this method would require close to 2^{40} comparisons. But, finally, Eve realizes that there is a much better way that still allows a form of time-memory trade-off. She is going to build a table indexed by the hash values of the 2^{21} variants of `m1` whose entries are the corresponding message numbers. Afterwards, she will sort the list of indices of this table (i.e., the list of hash values of variants of `m1`) using Maple's built-in implementation of merge sort (with complexity $O(n \ln n)$), which will require on average less than 2^{26} comparisons and will be feasible with her computing resources. Finally, she will generate the hashes of variants of `m2` and, using the binary search algorithm, search for each of them on the sorted list of table indices. When she finds one of these values in the indices list she has found the required collision. The binary search algorithm has complexity $O(\ln n)$, which is much faster than the linear complexity of the naive algorithm mentioned above (see [116] for details on sorting and searching algorithms).

This attack is implemented in Maple's function `FindCollision` below. The inputs are two text strings, the name of the hash function used, and the output length (in bits) of this function. The output is a list containing two text strings which are vari-

ants of the two input messages and form a collision for the hash function considered or, if no collision is found, a printed message informing that the search failed.

```
> FindCollision := proc(a::string, b::string, h::name, n::(And(posint, even)))
  local asp, bsp, s, j, t, ht, l;
  asp := [StringTools:-SearchAll(" ", a)];
  bsp := [StringTools:-SearchAll(" ", b)];
  t := (1/2)*n+1;
  if nops(asp) < t then
    error "too few spaces in '%1'", a
  end if;
  if nops(bsp) < t then
    error "too few spaces in '%1'", b
  end if;
  s := 0; j := 0;
  ht := table(map(x -> convert(h(variant(x,a,asp)), decimal, hex) = x, [$0..2^t-1]));
  printf("%d hash values stored\n\n", 2^t);
  l := sort(ListTools:-Flatten([indices(ht)]));
  printf("hash values sorted, search starts ...\n\n");
  while s = 0 and j < 2^t do
    s := ListTools:-BinarySearch(l, convert(h(variant(j, b, bsp)), decimal, hex));
    j := j+1;
    if j mod 100000 = 0 then
      printf("%d hashes searched\n", j)
    end if;
  end do;
  printf("\n");
  if s = 0 then
    print("Search failed")
  else
    [variant(ht[l[s]], a, asp), variant(j-1, b, bsp)]
  end if;
  end proc:
```

Observe that, since nonconsecutive hash values are to be used as indices in the search function, a Maple `Array` cannot be used and a table is built instead. Now, Eve uses this function to find a Hash40-collision between a variant of `m1` and a variant of `m2` (here we use the tilde '~'—the LaTeX symbol for a non-breakable space—to mark the ASCII-160 spaces for clarity, but Maple prints them as ordinary spaces and, of course, so will Eve when presenting the contract to Alice):

```
> c := FindCollision(m1, m2, Hash40, 40);

2097152 hash values stored

hash values sorted, search starts ...

100000 hashes searched
200000 hashes searched
300000 hashes searched
400000 hashes searched
500000 hashes searched

c := ["By signing~this statement~Eve~agrees to~pay~Alice the~amount of $950000~
   (nine hundred~fifty thousand~dollars) for the house~mentioned below",

      "By signing this~statement~Eve agrees~to pay~Alice~the amount~of $150000
   (one~hundred~fifty~thousand dollars) for the~house mentioned below"]
```

Eve checks that the two messages indeed form a collision:

```
> Hash40(c[1]), Hash40(c[2]);
                        "3f61863f3f", "3f61863f3f"
```

Now Eve presents Alice contract `c[1]` to be signed, with the idea of later claiming that Alice signed `c[2]` instead, as the same signature would be valid for both versions. But Alice, just before signing the message, insists that `c[1]` is missing a comma between the words "statement" and "Eve", and makes this change to the contract. Then Alice and Eve sign the following:

```
> d := StringTools:-Insert(c[1], 25, ",");

  "By signing~this statement,~Eve~agrees to~pay~Alice the~amount of $950000~
    (nine hundred~fifty thousand~dollars) for the house~mentioned below"
```

But now the previous computation that Eve carried out in order to find a collision between a variant of `m1` and one of `m2` is useless for her because

```
> Hash40(d);
                                "363bf9ea35"
```

The hash value of the new message is different from the one obtained in the collision. So, if Eve now wants to forge the message that bears Alice's signature, she would have to break the second preimage resistance of `Hash40` and find a collision between `d` and a variant of `m2`. Using a generic brute-force attack, this would require about 2^{40} hash computations and, as already indicated, is not feasible for her ... Eve does not know the reasons that led Alice to make this change at the last moment but she is beginning to suspect that perhaps Alice is not so ignorant about cryptography matters as she appeared to be ... might she not be the Alice whom all cryptography books talk about?

Remarks 5.5

1. When executing `FindCollision(m1,m2,Hash40,40)`, some versions of Maple produce an "object too large" error. If this happens, the experiment may still be carried out by reducing the number of collision bits and calling, for example, `FindCollision(m1,m2,Hash36,36)`, where `Hash36` is defined similarly to `Hash40` but takes only the first 36 bits of the MD5 hash. This will quickly find a collision and should work in all (recent) versions.
2. Observe that the birthday attack is parallelizable. If Eve had more computers available to parallelize the search then she could store the hash value of the variants of the first message in several tables of manageable size and, after generating the hashes of the variants of the second message, search for them in these tables.
3. Note also that in this attack the hash function is used as a black box and the attack would work the same for any other hash function with the complexity depending only on the output length of the function.
4. The method used by Eve in her attack above is a time-memory trade-off that reduces computing time by using a lot of memory. There are other ways of implementing the birthday attack that decrease the amount of storage required at the cost of increasing running time. For example, Eve could have adopted the following approach. For the case of a hash function h with an n-bit output and two messages $m1$ and $m2$, she could construct a function F from $\{0, 1\}^n$ to

itself as follows. Let $x = b||t \in \{0,1\}^n$, where b is the leading bit of x and t an $(n-1)$-bit string. Then define:

$$F(x) = \begin{cases} h(\texttt{variant}(t, m1)) \text{ if } b = 0, \\ h(\texttt{variant}(t, m2)) \text{ if } b = 1. \end{cases}$$

where `variant` is like the Maple function of the same name defined above which computes the variant of the message corresponding to a given number and we assume that the output of the hash function is given as a binary string. Then Floyd's cycle-finding algorithm can be used in the way described, for example, in [106, 7.5] to look for a collision of h. Half of the time, on average, the collision found will not be useful because it will be either a collision between two variants of $m1$ or two variants of $m2$, but the remaining cases will give a collision between a variant of $m1$ and one of $m2$ and this can be done with very low memory requirements at the cost of more computing time.

5. The attack in Example 5.12 was carried out in a restricted computing environment but we can easily extrapolate from it to the real world. Since computing over 2^{64} hash operations is regarded as feasible, this attack shows that a hash function with a 128-bit output does not offer security anymore and this is why such hash functions are regarded as 'broken'. Even a 160-bit hash function like SHA-1 would probably be within reach of a massive birthday attack but, as we have mentioned, there are improved attacks for this function that require about 2^{63} hash operations, thus SHA-1 is also regarded as broken. The currently recommended hash function is SHA-256 and this is the one we use in our implementations.

Chapter 6
Algorithmic Number Theory for Cryptography and Cryptanalysis: Primality, Factoring and Discrete Logarithms

In the preceding chapters we have introduced the most important aspects of private-key cryptography and we have noticed that prime numbers underlie many of the constructions and algorithms discussed. Also, computational number-theoretic problems which are presumed to be hard made their appearance and we mentioned, in particular, the integer factorization problem, which was briefly discussed in Sect. 3.3.3, and the discrete logarithm problem (Sect. 3.3.2). In the coming chapters we will study public-key cryptography and we will see that all these aspects play a relevant role in this setting. In fact, number theory and, in particular, presumedly hard number-theoretic problems such as the ones just mentioned, are of central importance for public-key cryptography because the security of most public-key schemes relies on the hardness of some of these problems. The study of the known algorithms to solve these hard problems can thus be seen as a form of cryptanalysis and, as such, is an indispensable complement to cryptography and a prerequisite for the practical evaluation of the security of public-key schemes in order to establish, for example, the key sizes that should be used. Therefore, before going on to discuss public-key cryptographic schemes, we are going to dedicate this chapter to the aspects of algorithmic number theory which are most relevant to these schemes.

6.1 Large Primes and How to Find Them

We have seen that prime numbers play an important role in some cryptographic constructions like the Blum–Blum–Shub generator. We will see later on that prime numbers are also crucial for many other schemes and protocols used in public-key cryptography, the usual requirements for these primes being that they should be large (such as, for example, the 1024-bit primes we have used in some of our examples) and that they should be randomly chosen among the primes of suitable size. How to do this is far from obvious and a key aspect which will make this task possible is that, in some sense that will be made precise later on, primes are 'abundant', which makes them easy to find but difficult to guess.

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_6, © Springer-Verlag Berlin Heidelberg 2013

6.1.1 Searching for Large Random Primes

The natural way of finding very large random primes is to choose, at random, integers of the required size and submit them to a *primality test*, an algorithm that takes an integer as input and produces a yes/no output that tells us whether the integer is prime. The idea then would be to repeat this process until a prime is found but this poses the following two problems:

1. Are there primality tests efficient enough to quickly determine whether an integer of, say, 1024 bits, is prime? More precisely, are there polynomial-time primality tests that are also efficient in practice?
2. Assuming that the answer to the preceding question is *yes*, how many times will it be necessary to apply the primality test to random integers, on average, to find a prime? Observe that if this number is polynomial in the size of the prime and the answer to the first question is also positive, then we obtain a polynomial-time algorithm to find a random prime.

It is often assumed that a positive answer to the first question is enough to solve the problem of finding large primes but this is only a necessary condition and if the problem finally turns out to be easy it is also because the expected number of primality tests to be performed is *low* (and indeed polynomial in the size of the prime) which is an indirect way of saying that the primes are, in some sense, abundant among the integers.

To better appreciate the importance of having a satisfactory answer to the second question above, let us compare this problem to the similar one of finding a random perfect square (an integer which is the square of another one) of 1024 bits. Of course, this can easily be accomplished by just taking a suitable 512-bit integer and squaring it. However, as the primes appear to be distributed pseudo-randomly among the integers, we do not have a similar method for computing a prime from a random integer with only one arithmetical operation so the meaningful experiment would be the following one: assume that the primes are, on average, as dense as the squares among the natural numbers or, equivalently, that our task were to find a 1024-bit square by taking random integers of this size and submitting them to a 'squareness test' that would check whether they are really squares. In the next example we carry out this experiment with Maple.

Example 6.1 We build a Maple function to search for squares that proceeds by first generating k-bit pseudo-random integers and then submitting them to Maple's squareness test `issqr` that checks whether they are squares. As on other occasions we insist that generating pseudo-random numbers is not a trivial matter for we need a source of true random bits. For this demonstration, we are going to use Maple's default pseudo-random algorithm with automatic seeding provided by the `randomize()` function; this is sufficient for our purposes here but, as we have repeatedly warned, it is not a secure method for other cryptographic uses for several reasons: the generator is not unpredictable and the automatic seeds are too small and far from random.

The square-searching algorithm is implemented in the function `Search` below, which actually serves a dual purpose: to find either squares (using Maple's `issqr` as indicated above) or primes (using Maple's `isprime` in this case). The first input parameter of the function is for a name which can be either `squares` or `primes` and serves to tell the function what to search for. The remaining input parameters `length` and `numtrials` are for two positive integers, the first of which specifies the bit length of the integer to be searched for and the second, the number of trials to be performed. If `length` is specified as k and `numtrials` as r then the function makes r attempts to find a k-bit square (or prime) and prints on screen all the squares (or primes) it finds, if any, finishing with a `Search Finished` statement printed at the end.

```
> Search := proc(object::name, length::posint, numtrials::posint)
  local test, i, n;
  if object = 'squares' then
    test := 'issqr'
  elif object = 'primes' then
    test := 'isprime'
  else
    error "unknown object"
  end if;
  randomize();
  for i to numtrials do
    n := RandomTools:-Generate(integer(range = 2^(length-1) .. 2^length-1))
    if test(n) then
      print(n)
    end if
  end do;
  print('Finished Search')
  end proc:
```

Now, we carry out the test and search for a 1024-bit square among, say, 100000 pseudo-random integers:

```
> Search(squares, 1024, 100000);
                              Finished Search
```

The procedure found no squares and most likely won't find them no matter how many trials we perform because *it is not feasible to find a square by this method*. To see that the problem does not lie with the procedure itself, let us make an attempt to find 16-bit squares by testing just 1000 pseudo-random integers:

```
> Search(squares, 16, 1000);
                                    51984
                                    41209
                                    33856
                                    33489
                              Finished Search
```

Now four squares were found and this was due to the fact that squares are much denser among 16-bit integers than among 1024-bit integers. The reason why the search for a 1024-bit square was hopeless is that the probability that a random 1024-bit integer turns out to be a square is:

```
> evalf((2^512-ceil(sqrt(2^1023)))/2^1023);

                           4.368994848 10^-155
```

One can try to understand how small this probability is by bearing in mind the current estimates that the number of atoms in the observable universe is close to 10^{80}...

As the preceding experiment shows, in order to be able to find large primes for cryptographic use, it is necessary that they are much denser than squares. Fortunately, it turns out that this is indeed the case and to appreciate it, let us have a look at the distribution of primes.

6.1.2 The Distribution of Prime Numbers

The first important thing about the distribution of the primes was already known by Euclid (it is Proposition 20 in Book IX of the *Elements*):

Theorem 6.1 *The set of primes is infinite.*

Proof Assume, on the contrary, that the set of primes is finite, say $\{p_1, \ldots, p_k\}$. Then $n = 1 + \prod_{i=1}^{k} p_i$ has, by Theorem 2.2 a prime factor p. Since $n \equiv 0 \pmod{p}$ but $n \equiv 1 \pmod{p_i}$ for all $i = 1, \ldots, k$, we see that p does not belong to the set $\{p_1, \ldots, p_k\}$, which is a contradiction and completes the proof. □

The fact that the set of primes is infinite does not mean that the primes are so abundant as to make them easy to find. The squares also form an infinite set and we have seen that they are not easy to find by choosing numbers at random and testing whether they are squares. An indirect way to see that squares are much less abundant than primes is to compute the sum of the series defined by their inverses, something that was already known to Euler and that Maple also knows:

```
> sum(1/x^2, x = 1 .. infinity);
```

$$\frac{1}{6}\pi^2$$

Now, what happens if we do the same with the inverses of the primes? The series $\sum_{p \text{ prime}} \frac{1}{p}$ is divergent as Euler showed (in a non-rigorous way) already in 1737. Using Maple's function `ithprime` we can compute the sum of the inverses of the first million primes as follows:

```
> evalf(add(1/ithprime(i), i = 1 .. 1000000));
                                3.068219048
```

It is estimated that the sum of the inverses of all the primes computed so far is about 4 and is not going to increase significantly in the near future since the series diverges very slowly. Anyway, this shows that primes are much more abundant than squares and fosters hopes that it will be easy to find large primes. But this does not give a very precise idea about *how* the primes are distributed. Looking at primes in short intervals it is easy to appreciate that their distribution is very irregular; for example, given a prime it is very difficult to predict where the next prime will be: it can be just two positions away or it can be much further away. However, if we look

at primes in large intervals we soon see a pattern emerge. The first mathematician to do that was Gauss, who computed all the primes up to 3,000,000. We are going to do a similar thing with larger intervals using Maple. Note that to choose a prime of, say, 1024 bits, the relevant question related to problem 2 above is: what is the probability that a random 1024-bit integer is prime? Following the steps of Gauss and with this question in mind, we count the number of primes in a series of large intervals. For this we use Maple's function `isprime` which is a probabilistic primality test (discussed in more detail below) that always gives the correct answer (`true`) when the number submitted to the test is prime and has a very low probability (which might even be zero although some heuristic arguments suggest that this is not the case) of outputting `true` when the number is composite (so that, if the output of the function is `false`, we have theoretical certainty that the number is composite). A quick computation gives:

```
> map(nops, map(x -> select(isprime, x),
            [seq([`$`((2^64)^i-2^15..(2^64)^i)], `in`(i, [1, 2, 4, 8, 16, 32]))]));
                  [720, 374, 178, 95, 45, 25]
```

This shows that if we consider intervals of length 2^{15} centered at integers of 64, 128, 256, ..., 2048 bits, each time that the size of the numbers doubles, the density of primes in the interval is reduced by approximately half. Moreover, the relative frequencies of primes in these intervals are close to the inverse of the logarithm of the center of the interval as can easily be checked. Experimental data similar to these (but with smaller numbers) led Gauss in 1792–1793 to conjecture that the probability of a large random integer close to n being prime is about $\frac{1}{\ln n}$. Another way of interpreting this is to look at the function $\pi(x)$ that counts the number of primes that are less than or equal to a given positive integer x. The preceding conjecture suggests the estimate:

$$\pi(x) = \sum_{2 \le n \le x} \frac{1}{\ln n}.$$

If we count the total number of integers from 2 to x, which is approximately equal to x and we multiply it by the smallest of these probabilities which is $\frac{1}{\ln x}$, this leads to the conjecture, made by Gauss and Legendre at the beginning of the nineteenth century, that for x large, $\pi(x)$ is approximately equal to $\frac{x}{\ln x}$. It is intuitively obvious (at least for those with a basic understanding of integration) that a more precise approximation to $\pi(x)$ should be obtained by considering the 'logarithmic integral'

$$\mathrm{li}(x) = \int_2^x \frac{dt}{\ln t},$$

of which the preceding sum is a 'Riemann sum'. This was also conjectured by Gauss and his conjecture may be formulated as follows. Recall that two functions with positive real values, f and g, are asymptotic (denoted $f(x) \sim g(x)$) whenever $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$. The conjecture of Gauss (and Legendre) about the distribution of primes was eventually proven in 1896 by Hadamard and de la Vallée Poussin as the following theorem (see [77] for a proof):

Theorem 6.2 (The Prime Number Theorem, PNT) $\pi(x)$ *is asymptotic to* $\frac{x}{\ln x}$ *(and also to* $\mathrm{li}(x)$*).*

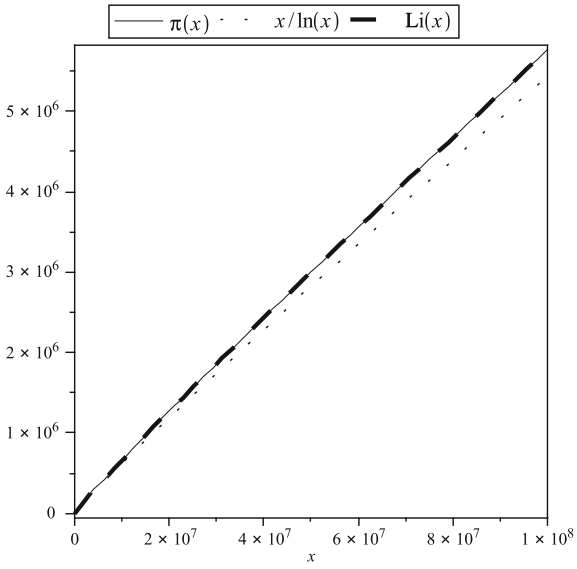
Remark 6.1 The probabilistic interpretation of the PNT is the following: a random integer, drawn uniformly from the interval $[1, x]$, will be prime with probability asymptotic to $1/\ln x$. As a consequence, the expected number of draws required to obtain a random prime from this interval is asymptotic to $\ln x$.

Exercise 6.1 Use Maple to compute the exact value of the ratio between the number of primes in the interval $[10^8–10^7, 10^8]$ and the number of primes in $[10^{12}–10^7, 10^{12}]$ and compare these values to the values predicted by the PNT in the two versions given above. (Hint: Do not use `numtheory:-pi` which will be too slow but write a procedure that uses `isprime` to count the number of primes in an interval instead.)

The following plot shows the amazing smoothness of $\pi(x)$ when viewed from a distance and also how close the approximations by $\frac{x}{\ln x}$ and $\mathrm{li}(x)$ are. Here we use Maple's function `Li` which represents the 'Cauchy principal value' of the integral $\int_0^x \frac{dt}{\ln t}$.[1]

```
> plot([numtheory:-pi(floor(x)), x/ln(x), Li(x)], x = 2..10^8, linestyle = [1, 7, 6],
  color = [black, black, black], thickness = [0, 1, 3], font = [times, roman, 8],
  axes = boxed, legend = [Pi(x), 'x/ln(x)', Li(x)], legendstyle = [location = top])
```

This produces the following plot:



The magnitude of the error in the approximation of $\pi(x)$ by $\mathrm{Li}(x)$ in the range plotted above is well below the square root of x. This fact is related to one of the

[1] This is the so-called 'American' version of the logarithmic integral, while the 'European' version $\mathrm{li}(x) = \int_2^x \frac{dt}{\ln t}$ is related to it by the formula $\mathrm{li}(x) = \mathrm{Li}(x) - \mathrm{Li}(2) = \mathrm{Li}(x) - 1.045163780$ (the notations Li and li are usually interchanged but we use Li this way to conform to Maple's usage).

Millennium Prize Problems [50], namely, the Riemann Hypothesis which, although usually stated in terms of the ζ function (see [77]), is equivalent to:

$$\pi(x) = \mathrm{Li}(x) + O(x^{\frac{1}{2}+\varepsilon})$$

for $\varepsilon > 0$, and also to $\pi(x) = \mathrm{Li}(x) + O(x^{\frac{1}{2}} \ln x)$ (note that $\mathrm{Li}(x)$ and $\mathrm{li}(x)$ are interchangeable in these asymptotic estimates).

Returning to the problem of finding large primes, the PNT now tells us what the probability is that a random integer of a given size turns out to be prime. The PNT entails that if this integer is close to x (where x is large), then this probability is approximately equal to $\mathrm{Li}(x)/x$. For example, if we want to choose a 1024-bit prime at random, we may test odd integers only (further reductions are possible but we shall not bother about this) and we know that, approximately, one in each

```
> evalf(2^1023/(2*(Li(2^1024)-Li(2^1023))));
                                354.7379025
```

of these odd integers (say, one in 355) will be prime. This makes it perfectly possible in practice to find such a prime provided that we have an efficient primality test at our disposal. In the next example we carry out an experiment similar to the one in Example 6.1 but using primes instead of squares. For now we will use Maple's probabilistic primality test `isprime` but we shall give an explicit alternative later when discussing primality testing.

Example 6.2 From the previous discussion it follows that if we search for a prime among, say, 2000 1024-bit integers, the expected number of primes found will be:

```
> 1000/(354.74);
                                2.818966003
```

To carry out this search we will use the previously defined function `Search`, specifying that primes are to be searched for. The search method is thus the same as the one we used for squares, only with `isprime` replacing `issqr`, but the results are very different:

```
> Search(primes, 1024, 2000);
1559778344764876718654490785046718694117996300552855083961309795186209922144988460\
  0743850817987043405186161597169491568493524743026603283153516829276464143248193 2\
  5620499725006835764894291184576533117048663651919167131075798448120590485689127 8\
  5491445393463982285807900238721649693476188960843975995549325273967
1221470048835205478008094528531016517664038704322845366170232503846629861702513191\
  6906243082806229798184187478723104225179345630202135572700096493262243119620861 5\
  1236376442779389980289296855225764851351726854102307416745429608917750103788674 9\
  6802796430879354856259943925892203310400050217589414130412524323163
```

The preceding discussion implies that the expected number of odd integers that should be tested in order to get a 1024-bit prime is approximately 355 (this is just the expected value of the random variable that takes as its value the number of independent tests which, as we have seen in Proposition 2.2, is $1/p$ if p is the probability that a random odd integer of this size passes the test). More generally, from a complexity-theoretic point of view, the fact that the expected number of

random (odd) integers to be tested in order to find a random prime near n is $O(\ln n)$ means that if we have a deterministic primality test of complexity $O(f(\ln n))$ (where f is some function), then the algorithm to find the random prime (by picking integers of the required size at random and submitting them to the primality test until a prime is found) would be a Las Vegas algorithm with expected running time equal to $O(f(\ln n)\ln n)$. We shall return to this question after we discuss primality testing and we will see that, although a polynomial-time deterministic primality test exists, a much more efficient algorithm for finding random primes exists if we also allow the primality test to be probabilistic.

Exercise 6.2 An alternative method which is often proposed to search for large primes consists of choosing at random an integer of the appropriate size and then looking for the next prime (for which Maple has the built-in function `nextprime`).

(i) Explain why this method does not choose primes uniformly at random.

(ii) Suppose that we use this method to choose a 1027-bit prime. Show that the prime

```
p = 829941989940592799681292290602624775771923191122802210343305520413604881789386\
    668418471359893967687633734762986527049578799912769146684379554452260666406153\
    850673539530720664117262802108324083361812055783159794496017908673406145849102\
    955238323521611423148927423872014310829119220858348387997402386170556695883
```

(from http://users.cybercity.dk/~dsl522332/math/primegaps/gaps20.htm) has a probability of being chosen by this method that is 8475 times higher than the probability of the prime

```
q = 829941989940592799681292290602624775771923191122802210343305520413604881789386\
    668418471359893967687633734762986527049578799912769146684379554452260666406153\
    850673539530720664117262802108324083361812055783159794496017908673406145849102\
    955238323521611423148927423872014310829119220858348387997402386170557282853
```

being chosen. Use Maple to compute both probabilities.

(iii) Write a Maple program to choose primes by this method and use it to obtain a large sample of 1024-bit primes. Compare the average length of the gaps[2] preceding the primes in this sample with the average length of the gaps corresponding to a sample of the same size in which the 1024-bit primes are chosen by taking random 1024-bit integers and testing them with `isprime`. Give an explanation for the difference between these values.

6.1.2.1 Extensions of the PNT and the Riemann Hypothesis

Euclid's theorem on the infinitude of the primes and the PNT have generalizations to the primes in a congruence class modulo an integer n or, equivalently in an arithmetic progression (this is just another name for a residue class) with common difference n. The natural question is: which residue classes a mod n contain primes and how

[2] The term *prime gap* refers to the difference between two consecutive primes.

many? It is obvious that if $\gcd(a, n) > 1$ then a common prime factor of a and n divides every term of the residue class, which can only contain at most one prime. But if this trivial case is excluded, there is a classical theorem of Dirichlet showing that the number of primes in the residue class is infinite (see, e.g., [77, Theorem 3.3.1]):

Theorem 6.3 (Dirichlet) *If a, n are integers such that $n > 0$ and $\gcd(a, n) = 1$, then the arithmetic progression $\{a, a+n, a+2n, \ldots\}$ contains infinitely many primes.*

Since the distribution of primes in large intervals is so regular, one can expect not only that there are infinitely many primes in each of these arithmetic progressions but also that the primes are uniformly distributed among them. Let us denote by $\pi(x; n, a)$ the number of primes $\leq x$ that are congruent to a modulo n (where $\gcd(a, n) = 1$). The question is then to determine, given x and n, how alike are the numbers $\pi(x; n, a)$ for all the $a < n$ such that $\gcd(a, n) = 1$. In the following exercise we check this for a couple of values of n when $x = 10^7$.

Exercise 6.3

(i) Write a Maple procedure to compute $\pi(x; n, a)$ when $\gcd(a, n)$=1.
(ii) Use the above procedure to compute the number of primes less than 10^7 that end in 1, 3, 7, 9 (in the decimal numeration system).
(iii) Compute the number of primes $< 10^7$ congruent to $1, 2, \ldots, 10 \pmod{11}$.

The answer to the preceding exercise shows that, in the $[2, 10^7]$ interval, the primes have an astoundingly uniform distribution across the residue classes. For example, the number of primes ending in 1, 3, 7 and 9 are, respectively, 166104, 166230, 166211, and 166032. This is strongly reminiscent of the PNT and, indeed, there is a version of the PNT for arithmetic progressions that was conjectured by Dirichlet and Legendre and proved by de la Vallée Poussin:

Theorem 6.4 (PNT for arithmetic progressions) *For each n, a such that $n > 1$ and $\gcd(a, n) = 1$, $\pi(x; n, a)$ is asymptotic to both $\frac{x}{\phi(n) \ln x}$ and $\frac{\mathrm{Li}(x)}{\phi(n)}$, where $\phi(n)$ denotes Euler's ϕ-function.*

Observe that, since the number of $a < n$ such that $\gcd(a, n) = 1$ is precisely $\phi(n)$, this theorem merely says that, asymptotically, the primes are uniformly distributed among all the nontrivial residue classes modulo n. Not only that, but it is conjectured that the error term of this approximation is small (of the same order as in the PNT), which constitutes the *Extended Riemann Hypothesis*: if $\gcd(a, n) = 1$ then

$$\pi(x; n, a) = \frac{\mathrm{Li}(x)}{\phi(n)} + O(x^{\frac{1}{2}+\varepsilon}).$$

Remark 6.2 The PNT for arithmetic progressions has the following probabilistic interpretation. If we draw a random integer $\leq x$ from the congruence class $a \pmod{n}$, where $\gcd(a, n) = 1$, then this integer will be prime with probability asymptotic to

$\frac{x/(\phi(n)\ln x)}{x/n} = \frac{n}{\phi(n)\ln x}$. As a consequence, the expected number of draws of integers in this residue class required to obtain a random prime $\leq x$ is asymptotic to $\frac{\phi(n)}{n}\ln x$.

6.2 Primality Testing

From the previous discussion we know that large primes are relatively abundant so that the only thing that remains for us to be able to find them is to have efficient primality tests to distinguish primes from composites. We have already seen that the sieve of Eratosthenes and trial division can be used for primality testing but these algorithms are of exponential complexity and so they are not able to test primes of cryptographic size (although they are useful as auxiliary components of other algorithms). So far we have used Maple's function `isprime` in our examples dealing with large primes but we have not discussed how it works. In this section we are going to look at the most common primality tests.

6.2.1 The Fermat Test and Pseudoprimes

Since the search-based primality tests considered so far are not able to deal with primes of the size required for cryptographic use, a plausible idea is to look for some simple formula that characterizes the primes among the integers and use it as a primality test. But simplicity is no guarantee of efficiency and a remarkable example is provided by Wilson's theorem and its converse, cf. [180, Theorem 2.22], according to which, if n is an integer greater than 1, then n is prime if and only if $(n-1)! \equiv -1 \pmod{n}$. This looks simple enough but has a crucial drawback: computing the factorial has exponential complexity and is impractical for the sizes required for cryptographic applications (actually, Wilson's theorem as a primality test is even less efficient than trial division, see Exercise 6.5 below).

Exercise 6.4

(i) Prove that if p is prime then $(p-1)! \equiv -1 \pmod{p}$ (Wilson's theorem). (Hint: The only elements of $\mathbb{Z}_p^*$ which are their own inverses are 1 and $p-1$. By collecting all the remaining elements of $\mathbb{Z}_p^*$ in pairs formed by an element and its inverse and multiplying them, show that $(p-2)! \equiv 1 \pmod{p}$.)

(ii) Prove the converse of Wilson's theorem by showing that if the congruence $(n-1)! \equiv -1 \pmod{n}$ holds, then any proper divisor of n must divide both $(n-1)!$ and $(n-1)!+1$, which is impossible.

Exercise 6.5

(i) Prove that computing $(n-1)! \bmod n$ by the naive method, with each multiplication followed by reduction modulo n, has complexity $O(n \ln^2 n)$.

(ii) Write a Maple procedure that uses Wilson's theorem to test primality and perform an experiment with a sample of numbers of different sizes to check the running time against the complexity estimated above.

Another natural candidate for a primality test is Fermat's little theorem (Theorem 2.2), according to which if n is prime and $b > 1$ is such that $\gcd(b, n) = 1$, then $b^{n-1} \equiv 1 \pmod{n}$ (if n satisfies this congruence we also say that n *passes the Fermat test to base* b). We know that the computation of $b^{n-1} \bmod n$ is fast by the binary method and so it seems that this formula can be used as an efficient primality test except for a little detail: satisfying this congruence is a necessary condition for n to be prime but is not sufficient!

Indeed, look at what happens if we take $b = 2$ and $n = 341$:

```
> Power(2,340) mod 341;
                                    1
```

Thus 341 satisfies Fermat's congruence with $b = 2$ but $341 = 11 \cdot 31$ is composite. The compositeness of 341 also follows from the fact that if we take $b = 3$ then:

```
> Power(3, 340) mod 341;
                                   56
```

Thus 341 passes the Fermat test to base 2 but does not pass it to base 3. This suggests the following definition:

Definition 6.1 Let n be a composite number and $b > 1$ an integer such that $\gcd(b, n) = 1$. Then n is a *pseudoprime* to the base b (or a *Fermat pseudoprime* to the base b) if $b^{n-1} \equiv 1 \pmod{n}$. For brevity we will simply say that n is a b-pseudoprime.

The existence of b-pseudoprimes seems to make the Fermat test useless as a primality test but this is really not so if one is satisfied with obtaining a result which is correct if it says that the number is composite but only probably correct if it says that the number is prime. In fact, a large odd integer that passes the Fermat test to base 2 (also known as the Chinese test because Chinese mathematicians discovered it long before Fermat) is very probably prime. The reason is that the number of 2-pseudoprimes is small in comparison to the number of primes and, moreover, the ratio of 2-pseudoprimes to primes diminishes sharply as n gets large. We are going to do a few experiments with Maple to get an intuitive feeling about this.

The Fermat test to base 2 is the following:

```
> Fermat2Test := n -> evalb(Power(2, n-1) mod n = 1):
```

The function that finds whether a given odd integer is a 2-pseudoprime is the following:

```
> Fermat2Pseudoprime := n -> not isprime(n) and Fermat2Test(n):
```

We can see, for example, that the 2-pseudoprimes less than 10000 are the following:

```
> select(Fermat2Pseudoprime, [seq(2*i-1, i = 2 .. 5000)]);
[341, 561, 645, 1105, 1387, 1729, 1905, 2047, 2465, 2701, 2821, 3277, 4033,
  4369, 4371, 4681, 5461, 6601, 7957, 8321, 8481, 8911]
```

We see that there are only 22 2-pseudoprimes below 10000, while the number of primes in this range is:

```
> numtheory:-pi(10000);
                                    1229
```

In order to make more meaningful comparisons between the number of primes and the number of pseudoprimes, we may use the following counting function, which takes as input an integer range and the name of a Maple Boolean-valued procedure, and returns the number of odd integers in the range for which the procedure returns `true`.

```
> count := proc(range::(integer .. integer), func::name)
  local s, e, count, i;
  s := op(range)[1];
  if type(s, even) then
    s := s+1
  end if;
  e := op(range)[2];
  if type(e, even) then
    e := e-1
  end if;
  count := 0;
  for i from s by 2 to e do
    if func(i) then
      count := count+1
    end if
  end do;
  count
  end proc:
```

For example, we may compare the number of 2-pseudoprimes to the number of primes in the interval $[10^8, 10^8 + 10^7]$ as follows:

```
> map(x-> count(10^8 .. 10^8+10^7, x), [`Fermat2Pseudoprime`, `isprime`]);
                                [94, 541854]
```

The first of these integers is the number of 2-pseudoprimes and the second is the number of primes. There is already a remarkable difference between the two numbers—there are almost 6000 primes in the interval for each 2-pseudoprime—but we are going to do a more complete experiment to appreciate how the number of 2-pseudoprimes in a fixed-length interval varies when we double the size of the involved numbers. For this computation it will be convenient to modify the above function `count` in order to avoid applying `isprime` twice to each odd number (one when computing the number of primes and the other when computing the number of 2-pseudoprimes). So we will build a function called `TestError` that computes, for a given interval, the number of odd integers that pass a test given by a Maple function `func` (in the case we are interested, `Fermat2Test`) and also the number of primes in the interval, as well as the probability that an integer chosen at random in the interval that passes the test given by `func` is composite (i.e., the probability of error of the test `func` in the given interval).

We use `isprime` in order to count the primes but this function is not guaranteed to always give the correct result: in theory, it might happen that some composite could be declared prime by `isprime` although no such example has been found so far. Even if this were to happen, the deviation would be so small that its effect would be negligible but, anyway, it is known that the output of `isprime` is always correct for numbers below 10^{15} such as those we will use in our experiment.

The function `TestError` takes as input an integer range, `range`, and a function name, `func`, for a Boolean-valued Maple function acting on integers. The output is a list of three elements, the first of which is the number of odd integers in the given interval that pass the test `func`. The second element of this list is the number of primes in the interval and, finally, the last element is equal to Pr(n is composite|`func(n) = true`) whenever `func` is a function that returns `true` when applied to primes and n is randomly chosen among the odd integers in the interval defined by `range`.

```
> TestError := proc(range::(integer .. integer), func::name)
  local s, e, count1, count2, i;
  s := op(range)[1];
  if type(s, even) then
    s := s+1
  end if;
  e := op(range)[2];
  if type(e, even) then
    e := e-1
  end if;
  count1 := 0;
  count2 := 0;
  for i from s by 2 to e do
    if isprime(i) then
      count1 := count1+1
    elif func(i) then
      count2 := count2+1
    end if
  end do;
  [count2, count1, evalf(count2/(count1+count2))]
  end proc:
```

Next we use this function to compute the probability of error of the Fermat test to base 2 when the integer is randomly chosen in any of two intervals of length 2^{28}, the first consisting of 32-bit integers and the second consisting of 48-bit integers.

```
> TestError(2^31 .. 2^31+2^28, Fermat2Test);
  TestError(2^47 .. 2^47+2^28, Fermat2Test);
                      [382, 12457225, 0.00003066399510]
                              [0, 8237902, 0.]
```

In these lists, the first element is the number of 2-pseudoprimes in the interval, the second is the number of primes, and the third is the probability that a randomly chosen integer in the interval that passes the Fermat base-2 test is composite. We see that, as was to be expected because of the PNT, the number of primes in the 48-bit interval is approximately two-thirds of the number of primes in the 32-bit interval. In contrast, the number of 2-pseudoprimes decreases much faster and, consequently, the probability of error of the Fermat test to base 2 decreases dramatically as the size of the numbers increases (and these numbers are still much too small for cryptographic use). Indeed, while there are over 8 million primes among the first 2^{28} 48-bit numbers,

there are no 2-pseudoprimes among them so that the probability of error in this interval is zero. Of course, the error probability will be non-null if one chooses much larger intervals (as it can be shown that the number of 2-pseudoprimes is infinite) but this suggests that a very fast method to find a large probable prime is to pick at random an odd integer of the required size and submit it to this test.

The theoretical result that explains the outcomes of these experiments is an old theorem of Erdös, cf. [60, Theorem 3.4.2], that asserts that if $b \geq 2$ the number of b-pseudoprimes that are $\leq x$ is $o(\pi(x))$ as $x \to \infty$, i.e., pseudoprimes become increasingly rare compared with primes as x grows.

Exercise 6.6 Write a Maple function that, on input b and n, implements the base-b Fermat test for n. Use this function to carry out experiments similar to the preceding ones for different bases b such as, for example, $b = 3$ or $b = 5$ and also for a given set of bases such as, for example $\{2, 3, 5, 7\}$.

6.2.1.1 Carmichael Numbers

We have seen that, although the composite number 341 passes the Fermat test to base 2, it already fails the test to base 3 so the idea arises of considering the Fermat test for all the possible bases: if an integer n passes the test to all the bases, is it necessarily prime? (note that, as we work modulo n it is sufficient to consider the bases $b \in \mathbb{Z}_n^*$ such that $1 < b < n - 1$).

The answer to the preceding question is no: even considering all the possible bases we do not get a primality test from Fermat's congruence or, in other words, there are composite integers n that are b-pseudoprimes for all the possible bases $b \in \mathbb{Z}_n^*$. The most straightforward way to see this is to make a search for these integers using Maple. The following function uses brute force to test whether an odd integer n has this property:

```
> Ftest := proc(n::odd)
  local t, i;
  if isprime(n) then
    return false
  end if;
  t := 1;
  for i from 2 to n-2 while t = 1 do
    if gcd(i, n) = 1 then
      t := Power(i, n-1) mod n
    end if
  end do;
  evalb(t = 1)
  end proc:
```

Now, we search among the odd integers up to 10001 (as we will see in a moment, the numbers we are looking for are necessarily odd):

```
> select(Ftest, [seq(2*i+1, i = 1 .. 5000)]);
                  [561, 1105, 1729, 2465, 2821, 6601, 8911]
```

We have quickly found seven integers less than 10000 that pass all the Fermat tests while being composite. These numbers have a name:

Definition 6.2 A composite integer n that is b-pseudoprime for all $b \in \mathbb{Z}_n^*$ is called a *Carmichael number*.

Carmichael numbers are an obstacle that prevents use of the Fermat test to prove that a prime number is indeed prime. As we have seen, 561 is the smallest Carmichael number but there are many others (in fact, their number is infinite, see reference below). Carmichael numbers are easily characterized by the following theorem. This result is called *Korselt's criterion* and, amazingly, it was discovered before any Carmichael number was known; it seems that Korselt believed that such numbers did not exist but, in 1910, R.D. Carmichael discovered 15 of them. Recall that an integer is called square-free if it is not divisible by the square of any prime (or, for that matter, by the square of any integer).

Theorem 6.5 (Korselt's criterion) *A composite integer $n>2$ is a Carmichael number if and only if it is square-free and $(p-1)|(n-1)$ for each prime divisor p of n. Moreover, the number of distinct prime factors of a Carmichael number is at least* 3.

Proof Assuming that n is a Carmichael number, we first show that n is square-free. Suppose, on the contrary, that p is a prime such that $p^2|n$. Let g be a primitive root modulo p^2, which exists by Exercise 2.27. The order of g in $\mathbb{Z}_{p^2}^*$ is equal to the order of this group which is $\phi(p^2) = p(p-1)$. If n has no prime divisors other than p, let $b = g \bmod p^2$ and, otherwise, let n' be the product of all the prime divisors of n other than p, and let b a solution (which exists by the Chinese remainder theorem) of the system of congruences

$$\left.\begin{array}{l} x \equiv g \pmod{p^2} \\ x \equiv 1 \pmod{n'} \end{array}\right\} \tag{6.1}$$

Then b is a primitive root modulo p^2 (since $b \equiv g \pmod{p^2}$) and, moreover, satisfies $\gcd(b, n) = 1$ because it is not divisible by p nor by any other prime divisor of n in case they exist (since $b \equiv 1 \pmod{n'}$). We show that n is not b-pseudoprime. Indeed, if $b^{n-1} \equiv 1 \pmod{n}$ then, since $p^2|n$, we also have that $b^{n-1} \equiv 1 \pmod{p^2}$. In this case, the order of b in $\mathbb{Z}_{p^2}^*$, which is equal to $p(p-1)$, divides $n-1$ by Proposition 2.4. But $p|n$ implies $n - 1 \equiv -1 \pmod{p}$ and hence $n-1$ cannot be divisible by $p(p-1)$, a contradiction. Thus we have shown that any Carmichael number is square-free.

We next prove that, for each prime p dividing n, $(p-1)|(n-1)$. Let g be a primitive root modulo p. Since n is square-free, using the Chinese remainder theorem we can find an integer b such that $b \equiv g \pmod{p}$ and $b \equiv 1 \pmod{n/p}$. Then $\gcd(b, n) = 1$ and $b^{n-1} \equiv g^{n-1} \pmod{p}$. On the other hand, since n is a Carmichael number we have that $b^{n-1} \equiv 1 \pmod{p}$ and hence $g^{n-1} \equiv 1 \pmod{p}$ too. Since g has order $p-1$ in $\mathbb{Z}_p^*$, it follows from Proposition 2.4 that $(p-1)|(n-1)$.

For the converse, suppose that n is square-free and $p-1$ divides $n-1$ for all prime divisors of n. Let b be an integer such that $\gcd(b, n) = 1$. Then, for every prime divisor p of n we have that $b^{p-1} \equiv 1 \pmod{p}$ and, since $(p-1)|(n-1)$,

b^{n-1} is a power of b^{p-1} so that we also have $b^{n-1} \equiv 1 \pmod{p}$. Since each of these primes p divides $b^{n-1} - 1$, their product, namely n, also divides it, so that $b^{n-1} \equiv 1 \pmod{n}$ and n is b-pseudoprime.

Finally, we show that if n is a Carmichael number, it is a product of at least three distinct primes. We only have to show that $n = pq$, with p, q, primes, is impossible. If $n = pq$ with $p < q$, then we know that $n - 1 \equiv 0 \pmod{q - 1}$. But $n - 1 = p(q - 1) + p - 1 \equiv p - 1 \pmod{q - 1}$, which is not congruent to 0 modulo $q - 1$, completing the proof. □

Remark 6.3 Note that a Carmichael number n is always odd. Otherwise, $n-1$ would be odd and hence it could not be divisible by even numbers of the form $p - 1$ with p prime.

Exercise 6.7 Show that n is a Carmichael number if and only if n is composite and $b^n \equiv b \pmod{n}$ for every integer b. (Hint: If n is a Carmichael number, then for every prime divisor p of n, $b^n \equiv b \pmod{p}$ which, when p does not divide b, follows from Fermat's little theorem and the fact that $(p - 1)|(n - 1)$.)

Using the preceding theorem we may write a much more efficient test (although still inefficient because it requires the prime factorization of n; factorization algorithms will be discussed later on) to determine whether an odd integer is a Carmichael number:

```
> CarmichaelTest := proc(n::odd)
  local f, p, c, i;
  f := ifactors(n)[2];
  p := map(x -> x[1], f);
  if nops(p) <= 2 or ormap(x-> evalb(x[2] <> 1), f) then
    return false
  end if
  c := 0;
  i := 1;
  while c = 0 and i <= nops(p) do
    c := (n-1) mod (p[i]-1);
    i := i+1
  end do;
  evalb(c = 0)
  end proc:
```

With this test we can find, for example, all the Carmichael numbers less than one million in a matter of seconds:

```
> select(CarmichaelTest, [seq(2*i+1, i = 1 .. 500000)]);
[561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633,
  62745, 63973, 75361, 101101, 115921, 126217, 162401, 172081, 188461, 252601,
  278545, 294409, 314821, 334153, 340561, 399001, 410041, 449065, 488881, 512461,
  530881, 552721, 656601, 658801, 670033, 748657, 825265, 838201, 852841, 997633]
```

There are 43 Carmichael numbers below one million and their density decreases quickly as number size grows. We may do an experiment using the function `TestError` to compare the number of Carmichael numbers with the number of primes in a given interval. For example, we have:

```
> map(x ->
      TestError(x,CarmichaelTest), [2^20..2^21, 2^24..2^24+2^20, 2^28..2^28+2^20]);
  [[10, 73586, 0.0001358769498], [6, 63008, 0.00009521693592], [0, 54150, 0.]]
```

In this case we have considered three intervals of length 2^{20} increasing the number size by four bits each time, which is a small change. As predicted by the Prime Number Theorem, the number of primes (the second integer in each sublist) decreases slowly but the number of Carmichael numbers decreases much more quickly and there are none in the third interval. The third number in each sublist is the probability that a random integer from this interval that passes the Fermat tests for all bases is, nevertheless, composite (i.e., a Carmichael number).

For a long time, the problem whether there are infinitely many Carmichael numbers remained open: if there were only finitely many then perhaps they could all be found and then the Fermat test would become a primality test by using all the possible bases and excluding Carmichael numbers. This test would have been very inefficient for it would be necessary to use a non-polynomial number of bases but this no longer matters because it was proved in 1992 by Alford, Granville and Pomerance [5] that the set of Carmichael numbers is infinite (they actually proved that, for x sufficiently large, the number $C(x)$ of Carmichael numbers not exceeding x satisfies $C(x) > x^{2/7}$). However, as we are going to see now, the Fermat test admits a refinement that overcomes the difficulty posed by Carmichael numbers and gives a primality test if all—or even a fraction of—the possible bases are used.

6.2.2 The Strong Probable Prime Test

Next we present the refinement of the Fermat test we referred to above. It is based on the following theorem:

Theorem 6.6 *Let n be an odd prime and $n - 1 = 2^s t$, with t odd. If n does not divide b then either*

$$b^t \equiv 1 \ (mod\ n)$$

or

$$b^{2^i t} \equiv -1 \ (mod\ n) \text{ for some } i \text{ with } 0 \leq i \leq s - 1.$$

Proof Consider the sequence $b^t \mod n, \ldots, b^{2^i t} \mod n, \ldots, b^{2^s t} \mod n = b^{n-1} \mod n$, where i varies from 0 to s. The last of these values is equal to 1 by Fermat's little theorem and each term in the sequence is equal to the square root of the next one modulo n. But the only square roots of 1 modulo n are 1 and -1 because n is prime (this follows, for example, from Theorem 2.23). This means that if we go down the above sequence from $i = s$ to $i = 0$, the first term (if any) that is $\neq 1$ must be equal to -1. Thus either all of these terms are equal to 1, in which case $b^t \equiv 1 \pmod n$ holds, or there is an i such that $0 \leq i \leq s - 1$ and $b^{2^i t} \equiv -1 \pmod n$. □

This result is a strengthening of Fermat's little theorem because it shows that, when n is prime and does not divide b, then not only the congruence $b^{n-1} \equiv 1 \pmod n$

holds but also the successive square roots obtained starting with this element follow a very specific pattern. Thus if an integer n satisfies the assertion of the theorem, then it also passes the Fermat test and one might hope that this gives a necessary and sufficient condition for n to be prime. But it is easy to see that this is not the case. For example, if $n = 2047 = 23 \cdot 89$ and $b = 2$ then, with the same notation as above, $t = 1023$ and we have that

```
> Power(2, 1023) mod 2047;
                                    1
```

so that the composite number 2047 satisfies the assertion of Theorem 6.6 for $b = 2$, and hence 2047 is an odd composite that behaves like a prime as far as this theorem is concerned. This leads to the following definition, similar to that of (Fermat) pseudoprime above:

Definition 6.3 Let n be an odd composite number, $n - 1 = 2^s t$, with t odd, and b an integer. Then n is a *strong pseudoprime* to the base b if either $b^t \equiv 1 \pmod{n}$ or $b^{2^i t} \equiv -1 \pmod{n}$ for some i with $0 \leq i \leq s - 1$.

Example 6.3 It is clear that a strong pseudoprime to the base b is also a b-pseudoprime, but strong pseudoprimes are less abundant than pseudoprimes: for example, the Carmichael number 561 is not a strong pseudoprime to the base 2 for we have that $560 = 2^4 \cdot 35$, i.e., $s = 4$ and $t = 35$ in the above notation. Then, if we compute the sequence of values $b^{2^i t} \bmod n$, for $0 \leq i \leq 3$, we obtain:

```
> Power(2, 35) mod 561;
                                   263
> Power(263, 2) mod 561;
                                   166
> Power(166, 2) mod 561;
                                   67
> Power(67, 2) mod 561;
                                   1
```

Thus we see that 561 is not a strong pseudoprime to the base 2 (and this happens because 67 is a square root of 1 modulo 561).

Remark 6.4 Observe that if $\gcd(b, n) \neq 1$ and p is a prime that divides both n and b, then $b^t \equiv 0 \pmod{p}$ and $b^{2^i t} \equiv 0 \pmod{p}$ for all i with $0 \leq i \leq s - 1$, and this clearly implies that n cannot be a strong pseudoprime to the base b. Thus, although not explicitly required in the definition, a necessary condition for n to be a strong pseudoprime to the base b is that $\gcd(b, n) = 1$.

An odd integer n that satisfies the congruences in the definition of strong pseudoprime for some b is said to pass the *strong probable prime test* to the base b. This test is given in Algorithm 6.1.

According to the previous discussion, if n passes the strong probable prime test, then n is either prime or a strong pseudoprime to the base b. If, on the contrary, n fails the test to base b, then we know from Theorem 6.6 that n is composite and, in this case, b is said to be a *witness* (also called a *strong witness* or a *Miller–Rabin witness*) to the compositeness of n, meaning that b serves to guarantee the compositeness of

n. Note that, by Remark 6.4, any b such that $1 < b < n$ and $\gcd(b, n) \neq 1$ is a witness for n.

Algorithm 6.1. Strong probable prime test.

Input: An odd integer $n > 3$ and an integer b with $1 < b < n - 1$.
Output: Either "strong probable prime" or "composite".

1. `Compute` s and t such that $n - 1 = 2^s t$, with t odd.
2. `Set` $u := b^t \bmod n$; $i := 1$.
3. `if` $u = 1$ `then`
   `return` "strong probable prime"
   `end if`;
   `while` $u \neq 1$ `and` $u \neq n - 1$ `and` $i < s$ `do`
   $u := x^2 \bmod n$;
   $i := i + 1$
   `end do`;
   `if` $u = n - 1$ `then`
   `return` "strong probable prime"
   `else`
   `return` "composite"
   `end if`.

The strong probable prime test can be implemented in Maple as follows. First we give an auxiliary function that tests whether $b^{2^i t} \equiv -1 \pmod{n}$ for some i with $0 \leq i \leq s - 1$, where $n - 1 = 2^s t$:

```
> MinusOneTest := proc(x, s, n)
  local u, i;
  u := x;
  i := 1;
  while u <> n-1 and u <> 1 and i < s do
    u := modp(u^2, n);
    i := i+1
  end do;
  u = n-1
  end proc:
```

We will also use the compiled version of this function:

```
> CompMinusOneTest := Compiler:-Compile(MinusOneTest):
```

Now, the strong probable prime test is given by the next function. The function is indexed by the base so that, to submit an integer to the test to base b, this function will be called as `StrongProbablePrimeTest[b]`. The function is able to use the compiled version of `MinusOneTest` whenever the size of the integer to be tested allows it; this produces some speed gains that probably will only be noticeable, for integers of appropriate size, with 64-bit versions of Maple (with the external C compiler installed). The function takes as input the odd integer n to be tested and an optional Boolean argument compiled that tells the function whether or not to use the compiled version of `MinusOneTest`; the default value is `false` so that the procedure will work even if the C compiler is not installed. The output is either `true`

in case n is a strong probable prime to the base b or `false` otherwise, in which case we know that n is composite.

```
> StrongProbablePrimeTest := proc(n::odd, compiled::truefalse:=false)
  local b, c, s, k, x;
  if type(procname, 'indexed') then
    b := op(procname)
  else
    error "base not specified"
  end if;
  if compiled and n <= (kernelopts(wordsize)/2)-1 then
    c := CompMinusOneTest
  else
    c := MinusOneTest
  end if;
  s := 0;
  k := n-1;
  while k mod 2 = 0 do
    k := k/2;
    s := 1+s
  end do;
  x := Power(b, k) mod n;
  x = 1 or x = n-1 or c(x, s, n)
  end proc:
```

The strong probable prime test cannot be used to prove the primality of an integer but it can prove n to be composite if n actually is. The problem is that if we submit n to the test for a given base b and n passes it, we do not know if it is because n is prime or because n is a strong b-pseudoprime. We will see that, for large n, the first alternative is much more likely than the second one and this makes it possible to use this test to find large primes. We perform some computations to illustrate the behavior of strong pseudoprimes. The next Maple function tells us whether or not a given odd integer n is strong pseudoprime to the base b:

```
> StrongPseudoPrimeTest := proc(n::odd, compiled::truefalse := false)
  local b;
  if type(procname, 'indexed') then
    b := op(procname)
  else
    error "base not specified"
  end if;
  not isprime(n) and StrongProbablePrimeTest[b](n, `:-compiled`)
  end proc:
```

Example 6.4 The strong 2-pseudoprimes less than 10000 are:

```
> select(StrongPseudoPrimeTest[2], [seq(2*i-1, i = 2 .. 5000)]);
                      [2047, 3277, 4033, 4681, 8321]
```

We see that there are only five strong 2-pseudoprimes in this interval while, as we have seen, the number of 2-pseudoprimes in the same interval is 22. We also counted 94 2-pseudoprimes in the interval $[10^8..10^8 + 10^7]$ and now we see that the number of strong 2-pseudoprimes in the same interval is:

```
> count(10^8 ..10^8+10^7, StrongPseudoPrimeTest[2]);
                                    21
```

Exercise 6.8 Compute the strong pseudoprimes to the base b that are less than 3000000 for $b = 2, 3, 5$. Compare these sets with the corresponding sets of Fermat pseudoprimes.

From Exercise 6.8 we see that there are exactly 91 strong pseudoprimes to the base 2 that are less than three million. But if we consider additional bases and, in particular, bases that are relatively prime, then we see that the number of strong pseudoprimes drops steeply. For example, let us compute the numbers below 3000000 that are strong pseudoprimes to the bases 2 and 3:

```
> SPP2and3 := x -> StrongPseudoPrimeTest[2](x) and StrongPseudoPrimeTest[3](x):

> select(SPP2and3, [seq(2*i+1, i=2..15*10^5-1)]);
                    [1373653, 1530787, 1987021, 2284453]

> map(x -> TestError(x, SPP2and3), [2^22..2^23, 2^24..2^24+2^22, 2^26..2^26+2^22]);
  [[2, 268216, 0.000007456621107], [1, 250359, 0.000003994248282], [0, 232342, 0.]]
```

We see that there are only 4 of these strong pseudoprimes below 3000000 and also that the probability of error of the test drops sharply as the size of the numbers increases. Note that there are more efficient algorithms to compute the numbers that are strong pseudoprimes relative to several bases and some of them were used in [27] to compute all the integers up to 10^{16} that are strong pseudoprime relative to the bases 2 and 3. There are exactly 52593 such integers and, while this might seem a considerable number, it is actually small compared to the number of primes less than 10^{16} which is:

```
> numtheory:-pi(10^16);
                              279238341033925
```

This means that a random integer less than 10^{16} that passes the strong probable prime test to bases 2 and 3, has a probability equal to

```
> evalf(52593/(279238341033925+52593));
                             1.883444795 10^-10
```

of being composite, so it is very likely prime. Of course, if one considers more bases, then this likelihood increases. For example, it is shown in [27] that there is only one number less than 10^{16} that is a strong pseudoprime for all the prime bases ≤ 19. Thus a random integer less than 10^{16} that passes all these tests will have a probability greater than $1–10^{-14}$ of turning out to be prime. In addition to this, the above experiments suggest that the probability of error also decreases as the size of the tested number increases or, in other words, that the density of strong pseudoprimes drops much faster than that of primes as size increases. Although the above samples are too small to draw this conclusion with confidence, we shall cite below theoretical results that confirm that this is indeed the case.

Example 6.5 Let us compute the strong 2-pseudoprimes in two intervals of length 10^7, the first starting at 10^8 and the second starting at 10^{12}.

```
> select(StrongPseudoPrimeTest[2], [seq(2*i+1, i = 5*10^7 .. 5*(10^7+10^6)-1)]);
[100463443, 100618933, 100943201, 101270251, 101276579, 101649241, 102004421,
  102678031, 102690677, 104078857, 104852881, 104857391, 105305443, 105919633,
  106485121, 106743073, 106775761, 107543333, 108596953, 109118791, 109437751]

> select(StrongPseudoPrimeTest[2], [seq(2*i+1, i = 5*10^11 .. 5*(10^11+10^6)-1)]);
                              [1000002977551]
```

We see that, in contrast with the 21 strong 2-pseudoprimes found in the first interval, there is only one strong 2-pseudoprime in the second. For comparison, according to the PNT, the ratio of the expected number of primes in the first interval to the expected number of primes in the second one is, approximately, the ratio between the logarithms of 10^{12} and of 10^8, namely, about 1.5.

The preceding example again suggests that the number of strong pseudoprimes drops much faster than the number of primes in relation to the size of the involved numbers. On top of all this, notice also that the numbers in this experiment are still very small and this effect will be much stronger on numbers of the size required by public-key encryption schemes such as RSA, where primes of 1024 or 2048 bits are typically used.

6.2.3 The Miller–Rabin Test

We have noticed that strong pseudoprimes appear to be much scarcer than either primes or (Fermat) pseudoprimes and this suggests that the strong probable prime test may be useful for finding large primes. Moreover, the difference with the Fermat test is not only quantitative but also qualitative: we are going to see that now there is no analogue of Carmichael numbers and the integers that pass the probable prime test to all the bases are always prime or, in other words, there are no integers that are strong pseudoprime relative to all possible bases. For brevity, it is convenient to give a name to the bases for which a given composite number is a strong pseudoprime:

Definition 6.4 Let n be a composite odd integer and $n - 1 = 2^s t$ with t odd. The set of *strong liars* for n, $S(n)$, is defined as the set of all elements of $\mathbb{Z}_n^*$ that are not witnesses for n or, equivalently, the set of bases in $b \in \mathbb{Z}_n^*$ such that n is a strong pseudoprime to the base b. Thus

$$S(n) = \{b \in \mathbb{Z}_n^* | b^s \equiv 1 \pmod{n} \text{ or } b^{2^i t} \equiv -1 \pmod{n} \text{ for some } i,\ 0 \leq i < s\}.$$

Remark 6.5 Note that strong liars are the bases for which a composite number behaves in a prime-like fashion as far as the strong probable prime test is concerned. These bases might deceive us into thinking that a composite is prime, hence their name. On the other hand, witnesses are just the contrary: they tell us that a number is composite and never lie.

Example 6.6 The following Maple procedure computes $S(n)$ for n an odd composite integer:

```
> StrongLiars := proc(n::odd)
  local i, w;
  if isprime(n) then
    error "n is prime"
  end if;
  w := [];
  i := 1;
  for i to n-1 do
    if StrongProbablePrimeTest[i](n) then
      w := [op(w), i]
    end if
  end do;
  w
  end proc:
```

Let us compute the strong liars for $n = 91 = 7 \cdot 13$:

```
> StrongLiars(91);
   [1, 9, 10, 12, 16, 17, 22, 29, 38, 53, 62, 69, 74, 75, 79, 81, 82, 90]
```

There are 18 strong liars for this number. Let us now consider a bigger number:

```
> StrongLiars(1208183);
                                [1, 1208182]
```

We see that for $n = 1208183 = 599 \cdot 2017$, the only strong liars are 1 and $n - 1$. These are always strong liars so that, for this number, any base b such that $1 < b < n - 1$ will be a witness that proves that n is composite. Thus, for a number like this the strong probable prime test has an ideal behavior in the sense that a single application of the test for any nontrivial base suffices to prove that the number is composite. In contrast, if we submit 91 to the test with only one randomly chosen base b, $1 < b < 90$, we have a probability equal to $18/88 > 1/5$ that the number passes the test and is declared to be a probable prime.

Exercise 6.9 Write a Maple procedure that computes the ratio between the number of strong liars and the number of bases b, $1 < b < n - 1$ for a given odd composite number. Apply this function to a sample of composite numbers in order to get an intuitive idea of how this ratio varies.

In order to show that there is no analogue of the concept of Carmichael number for the strong probable prime test, we only have to prove that, for a given n, not all $b \in \mathbb{Z}_n^*$ are strong liars. But we will show that a much stronger condition holds, as follows from the following theorem, due to Monier and Rabin: the number of strong liars cannot be more than one-fourth the number of possible bases.

Theorem 6.7 (Monier–Rabin) *If $n > 9$ is an odd composite integer, then the number of strong liars for n is at most $\phi(n)/4$.*

Proof Let $n - 1 = 2^s t$, with t odd and let $n = \prod_{i=1}^{j} p_i^{e_i}$ be the prime factorization of n. Let $k = \max\{i \in \mathbb{N} |$ there exists $a \in \mathbb{Z}_n^*$ with $a^{2^i} \equiv -1 \pmod{n}\}$ (k is well defined because $(-1)^{2^0} \equiv -1 \pmod{n}$) and set $m = 2^k t$. Since $a^{2^k} \equiv -1 \pmod{n}$, we also have $a^{2^k} \equiv -1 \pmod{p_i}$ for each $i = 1 \ldots j$ and, by Proposition 2.6, the order of $a \bmod p_i$ in $\mathbb{Z}_{p_i}^*$ is precisely 2^{k+1}. Thus $2^{k+1} | p_i - 1$ by Corollary

2.1 and so $p_i \equiv 1 \pmod{2^{k+1}}$ for each $i = 1 \ldots j$, from which it follows that $n \equiv 1 \pmod{2^{k+1}}$. Therefore $2^{k+1} | n-1$, which implies that $k < s$ and hence that $m | \frac{n-1}{2}$. Let us now consider the following subsets of $\mathbb{Z}_n^*$:

$$\begin{aligned} J &= \{b \in \mathbb{Z}_n^* | b^{n-1} \equiv 1 \pmod{n}\}, \\ K &= \{b \in \mathbb{Z}_n^* | b^m \equiv \pm 1 \pmod{p_i^{e_i}} \text{ for all } i = 1 \ldots j\}, \\ L &= \{b \in \mathbb{Z}_n^* | b^m \equiv \pm 1 \pmod{n}\}, \\ M &= \{b \in \mathbb{Z}_n^* | b^m \equiv 1 \pmod{n}\}. \end{aligned}$$

Each of these subsets is a subgroup of $\mathbb{Z}_n^*$ and, since $m | \frac{n-1}{2}$, we have the following chain of subgroups:

$$M \subseteq L \subseteq K \subseteq J \subseteq Z_n^*.$$

Next we show that $S(n) \subseteq L$. Indeed, if $b \in S(n)$, then either $b^t \equiv 1 \pmod{n}$, in which case we have that also $b^m \equiv 1 \pmod{n}$ (since m is a multiple of t) and hence $b \in L$, or $b^{2^i t} \equiv -1 \pmod{n}$ for some i, $0 \leq i < s$. In this case we have that $i \leq k$ and hence $b^m \equiv b^{2^k t} \pmod{n}$ is a power of -1 in $\mathbb{Z}_n^*$ so that we also have $b^m \equiv \pm 1 \pmod{n}$ and hence $b \in L$. Now, since $S(n) \subseteq L$, it suffices to show that the index of L in $\mathbb{Z}_n^*$ is at least 4, for then we will have that $|S(n)| \leq |L| \leq |\mathbb{Z}_n^*|/4 = \phi(n)/4$.

Now let $G = \{b \in \mathbb{Z}_n^* | b \equiv \pm 1 \pmod{p_i^{e_i}} \text{ for all } i = 1 \ldots j\}$ which is clearly a subgroup of $\mathbb{Z}_n^*$. There are 2^j systems of congruences $x \equiv \pm 1 \pmod{p_i^{e_i}}, i = 1 \ldots j$ (where j is the number of distinct prime factors of n) and, by the Chinese remainder theorem, each of them corresponds to an element of G so that $|G| = 2^j$. Moreover, the function $f : K \to G$ defined by $f(b) = b^m \bmod n$ for all $b \in K$ is a group homomorphism and it is clear that $\text{Ker} f = M$. Next we show that f is surjective. Let $b \in G$ and, using the Chinese remainder theorem, define an element $c \in \mathbb{Z}_n^*$ by setting $c \equiv a \pmod{p_i^{e_i}}$ if $b \equiv -1 \pmod{p_i^{e_i}}$ and $c \equiv a^2 \pmod{p_i^{e_i}}$ if $b \equiv 1 \pmod{p_i^{e_i}}$, for all $i = 1, \ldots, j$ (where a is an element of $\mathbb{Z}_n^*$ that satisfies $a^{2^k} \equiv -1 \pmod{n}$). Then $c^{2^k} \equiv b \pmod{p_i^{e_i}}$ for all $i = 1, \ldots, j$ and hence also $c^m \equiv c^{2^k t} \equiv b \pmod{p_i^{e_i}}$ (since t is odd). From this it follows that $c^m \equiv b \pmod{n}$ and hence that $b = f(c)$, proving that f is surjective. Now, using Proposition 2.7 we obtain the index of M in K which is $(K : M) = |K/M| = |G| = 2^j$. On the other hand, we also have a surjective homomorphism $g : L \to \{-1, 1\}$ given by $b \mapsto b^m \bmod n$. It is clear that $\text{Ker} g = \{b \in L | b^m \equiv 1 \pmod{n}\} = M$, so that, again by Proposition 2.7, $L/M \cong \{-1, 1\}$ and $(L : M) = 2$. Therefore, $(K : L) = 2^{j-1}$.

Suppose now that $j \geq 3$. Then we have that $(Z_n^* : L) \geq (K : L) = 2^{j-1} \geq 4$ and we are done. Thus it remains to show that $(Z_n^* : L) \geq 4$ also when $j \leq 2$.

If $j = 2$ then, by Korselt's criterion (Theorem 6.5), n cannot be a Carmichael number. Thus there exists $b \in \mathbb{Z}_n^*$ such that $b^{n-1} \not\equiv 1 \pmod{n}$, which implies that J is a proper subgroup of $\mathbb{Z}_n^*$ and hence that $(\mathbb{Z}_n^* : J) \geq 2$. Since $(J : L) \geq (K : L) = 2^{j-1} = 2$ we have that $(\mathbb{Z}_n^* : L) = (\mathbb{Z}_n^* : J)(J : L) \geq 4$ and we are done in this case.

Finally, if $j = 1$ then n is a prime power, $n = p^e$ with $e \geq 2$. By Proposition 2.4, J consists of the elements of $\mathbb{Z}_{p^e}^*$ whose order divides $p^e - 1$. By Corollary 2.1, the orders of these elements also divide $|\mathbb{Z}_{p^e}^*| = \phi(p^e) = p^{e-1}(p-1)$. Hence J consists of the elements of $\mathbb{Z}_{p^e}^*$ whose order divides $\gcd(p^e - 1, p^{e-1}(p-1)) = p - 1$. Let g be a generator of $\mathbb{Z}_{p^e}^*$, which exists by Theorem 2.19. Then, using Proposition 2.5 we see that $J = \langle g^{p^{e-1}} \rangle$ and hence $|J| = p - 1$. Therefore, $(\mathbb{Z}_{p^e}^* : J) = |\mathbb{Z}_{p^e}^*|/|J| = (p^{e-1}(p-1))/(p-1) = p^{e-1} \geq 4$ except when $p^e = 9$. Since we are assuming that $n > 9$, the proof is complete. □

Exercise 6.10 Show that $S(n)$ is not necessarily a subgroup of $\mathbb{Z}_n^*$. (Hint: Use Maple to show that 65 is the smallest odd composite n such that there are elements $b_1, b_2 \in S(n)$ with $b_1 b_2 \bmod n \notin S(n)$).

Remarks 6.1

1. Note that, for any odd composite integer n, the number of strong liars is always $\leq (n-1)/4$. Indeed, by Theorem 6.7 we know that this number is actually $\leq \phi(n)/4$ if $n > 9$. The only case not covered by the theorem is when $n = 9$, for which $\phi(9) = 6$ and $|S(9)| = 2$ because:

```
> StrongLiars(9);
                                [1, 8]
```

   In this case the assertion of the theorem does not hold—this is the reason why the hypothesis $n > 9$ was required—but we see that $|S(n)| \leq (n-1)/4$ is verified, so that this weaker form of the theorem indeed holds for all odd composite integers.
2. The bound $\phi(n)/4$ on the size of $S(n)$ is attained for $n = 15$ and for $n = 91$. In the first case, $\phi(15)/4 = 2$ and there are always at least two strong liars (1 and -1), so that we have that $S(15) = \{1, 14\}$. In the second case, we have seen that $|S(91)| = 18 = \phi(91)/4$. But, as we shall indicate below, this bound is not tight and is very far from giving a good estimate of the value of $S(n)$ when n grows large.
3. Theorem 6.7 (and also Theorem 6.6) shows that a primality test may be obtained from the strong probable prime test. Indeed, given an odd integer n we select $\frac{n-1}{4}$ different bases in the interval $[2, n-2]$. In fact, a smaller number suffices because we are not including the trivial bases 1 and $n-1$ and, moreover, n might be replaced by $\phi(n)$ except that, of course, if we are testing n for primality, we do not know the exact value of $\phi(n)$ (although we might know an approximation sufficient for our purposes). We then submit n to the strong probable prime test for each of these bases. Then:

   a. If n fails any of these tests, we stop and declare n composite (because of Theorem 6.6).
   b. If n passes all these tests, then n is declared prime (because of Theorem 6.7).

   This will always produce the correct result but has an important drawback: the test is highly inefficient. The running time of this test may be analyzed as follows. In the worst case, performing the strong probable prime test amounts to a binary

exponentiation modulo n and so it has complexity $O(\text{len}(n)^3)$. The number of tests to be performed is $O(n)$ and hence the complexity of the test would be $O(n\text{len}(n)^3)$ which is exponential in the length of n. Even if we perform only $O(\phi(n))$ strong probable prime tests, taking into account that, by the PNT, $\phi(n) = O(n/\text{len}(n))$, we would have a running time of $O(n\text{len}(n)^2)$ which is, again, exponential and would require more than 10^{300} exponentiations to test a 1024-bit number.

As we have just seen, submitting an integer to a number of strong probable prime tests sufficient to produce a primality test according to Theorem 6.7 is not feasible because finding a witness for a composite number may require a number of tests that grows exponentially with the size of n. One possible way to overcome this difficulty is to find a tighter bound on the number of strong liars for, if we show that this number is polynomial in the size of n, then the method described in the preceding remarks would give a polynomial-time primality test. Looking at the problem from the point of view of witnesses, if we show that the *first witness*, i.e., the least base such that a given odd composite n fails the strong probable prime test, grows polynomially as a function of the size of n, we have a primality test just by submitting n to the first bases in increasing order until reaching one greater than the first witness.

Exercise 6.11 Write a Maple procedure that computes the smallest odd composite whose first witness is greater than a given integer and use it to compute the smallest composite whose first witness is one of the following values: 3, 5, 7.

The preceding exercise shows that the smallest odd composite integer n whose first witness is 3 is $n = 2047$ and, similarly, 1373653 and 25326001 are the smallest composites whose first witnesses are 5 and 7, respectively. This small set of data is greatly extended in [27, Table 3.2] where, for example, it is shown that the 56-digit number 68528663395046912244223605902738356719751082784386681071 is a strong pseudoprime for all the bases ≤ 100. We can easily confirm that the first witness for this number is 101, as follows:

```
> FirstWitness := proc(n::odd)
  local b;
  b := 2;
  if isprime(n) then
    error "%1 is prime", n
  else
    while StrongProbablePrimeTest[b](n) do
      b := b+1
    end do;
  end if;
  b
  end proc:

> FirstWitness(68528663395046912244223605902738356719751082784386681071);
                                    101
```

The values of the ratio between $\ln n$ and the first witness for n obtained from Bleichenbacher's data suggest the possibility that the first witness is $O(\text{len}(n))$ but this has not been proved. The strongest theoretical result we have in this regard is

the following theorem which, assuming the verification of the Extended Riemann Hypothesis, shows that the first witness does indeed have a polynomial growth:

Theorem 6.8 (Bach) *If the Extended Riemann Hypothesis is true and $n > 1$ is an odd composite integer, then there is a witness b for n such that $b < \ln^2 n$.*

From this it follows that, if we assume that the ERH holds, in order to test n for primality it suffices to perform the strong compositeness test on n for all bases up to $\ln^2 n$. Thus we obtain a polynomial-time primality test, conditional on the ERH, with running time $O(\ln^5 n)$. This test is often called *Miller's deterministic test*.

Miller's test, however, is not very satisfactory for a couple of reasons. The first is that the ERH remains unproven and the second that this test is inefficient in practice. For example, note that testing the primality of a 2048-bit number would require performing about 2 million strong probable prime tests. This number can be reduced since many of these tests are not independent—for example, the bases which are powers of integers should be disregarded and also, if n is a strong pseudoprime to the bases a and b, then it is very likely also a pseudoprime to the base ab [27]. But even considering only prime bases less than $\ln^2 n$, about 150000 strong probable prime tests would have to be performed in this case.

The fact that this test is not practical is not a big problem anyway. On the one hand, from the theoretical point of view this does not matter because, as we will indicate below, it is now known that the primality of an integer can be checked unconditionally in polynomial time (or, as in the title of the original paper where this is proved: PRIMES is in P). From a more practical point of view, the important thing is that we can turn the previous test into a very efficient one by trading off the theoretical assurance that the test always gives the correct answer and replacing it by a very high probability that this is indeed the case.

The idea to obtain a very efficient test starting with the strong probable prime test is just to choose k random bases (where $k > 0$) and submit the integer n to the strong probable prime test for each of these bases; if for any of these bases n fails the strong probable prime test, n is declared composite and if, on the contrary, n passes all k tests, n is declared prime. This is the *Miller–Rabin test*:

Algorithm 6.2. Miller–Rabin test.

Input: An odd integer $n > 3$ and a parameter $k > 0$.
Output: Either "prime" or "composite".

1. `for` i `from` 1 `to` k `do`
   $b \leftarrow [2, n-2]$;
   `if` n fails the strong probable prime test to base b `then`
   `return` "composite"
   `end if`
   `end do`;
2. `return` "prime".

The important facts about the Miller–Rabin test are captured in the following theorem:

Theorem 6.9 *The Miller–Rabin test runs in polynomial time* $O(k\text{len}(n)^3)$. *If* n *is an odd prime the test always outputs "prime". If* n *is an odd composite then the test outputs "prime" with probability at most* 4^{-k}.

Proof The running time follows from the fact that the strong probable prime test runs in time $O(\text{len}(n)^3)$. If n is prime then the algorithm returns "prime" because of Theorem 6.6. If n is composite then the probability that the algorithm fails to find a witness at each iteration is $< 1/4$ by Theorem 6.7, so that the probability that no witness has been found after k iterations (in which case the algorithm returns "prime") is $< (1/4)^k$. □

Remark 6.6 The Miller–Rabin test can be viewed as a search for a proof of the compositeness of n in the form of a witness: if n is prime then no such proof will be found and if n is composite then the proof will be found with probability $> 1 - 4^{-k}$. Thus the algorithm can prove that an integer is composite but it cannot prove that it is prime (since usually k will be much smaller than $(n-1)/4$). Because of this, the algorithm is often called the *Miller–Rabin compositeness test*. Actually, this test is a no-biased Monte Carlo algorithm for testing compositeness and a yes-biased Monte Carlo algorithm for testing primality. Thus it shows that the decision problem whether an integer n is composite is in the class $\mathcal{RP}$ and the decision problem whether n is prime is in co$\mathcal{RP}$ (both these facts are superseded by the AKS test mentioned below, which shows that both problems are actually in $\mathcal{P}$).

The next function implements the Miller–Rabin test in Maple, using the previously defined function `StrongProbablePrimeTest`. The input consists of the odd integer n to be tested and a positive integer k to set the number of random bases to be used. In practice, we will use Maple's default pseudo-random algorithm to generate these bases, so that they will not be really random but this is not a big concern here since unpredictability is not required. The output is `true` if n is either prime or a strong pseudoprime for the k bases considered (in which case n will be declared a probable prime) and `false` otherwise (so that `true` corresponds to "prime" and `false` corresponds to "composite" in the sense of Algorithm 6.2):

```
> MillerRabin := proc(n::And(posint, odd), k::posint)
  local t, i;
  i := 1;
  t := true;
  if 4 < n then
    while t and i <= k do
      t := StrongProbablePrimeTest[rand(2..n-2)()](n);
      i := i+1
    end do
  else
    t := evalb(n = 3)
  end if;
  t
  end proc:
```

If the number of bases k is kept constant, then the test running time is just $O(\ln^3 n)$. We are going to see that this is a viable way to proceed in practice because the probability that a large composite integer passes the test is actually much lower than the bound $1/4^k$. As we saw when discussing the strong probable prime test, the number of strong pseudoprimes relative to small bases is low in comparison with the number of primes, and the ratio between these numbers decreases as number size increases. We are now going to perform a similar experiment with pseudo-random bases, in order to see that the probability that a composite number passing the test is much lower than the 4^{-k} bound provided by Theorem 6.9. In order to see this, we will consider two intervals of length 10^6, the first starting at 10^7 and the second at 10^{15}, and we will measure the error produced by the Miller–Rabin test with one single base on these intervals. We will use the previously defined Maple function `TestError` which, in turn, uses `isprime` to distinguish between primes and composites, and we recall that the latter function is known to give always the correct result for numbers of this size, so the results of the experiment will be accurate.

```
> randomize();
  MillerRabin1 := x -> MillerRabin(x, 1):
  TestError(10^7 .. 10^7+10^6, MillerRabin1);
  TestError(10^15 .. 10^15+10^6, MillerRabin1);
                        [7, 61938, 0.0001130034708]
                              [0, 28845, 0.]
```

Let us analyze these results. Only seven out of the more than 430000 odd composites in the first interval passed the test and none of the more than 470000 odd composites in the second interval passed it! If the probability that a composite passes the test were close to $1/4$ we would expect over 100000 integers passing the test in either case. Note that, as the base is pseudo-randomly chosen each time the test is applied, different results can be obtained when this experiment is repeated, but the results will not be too far from the ones we have just obtained. This experiment suggests that the error probability of the Miller–Rabin test is indeed much lower than 4^{-k} and, furthermore, that it decreases sharply as the numbers tested get larger. We will have a few more words to say about this when we discuss algorithms to generate random primes of large size.

6.2.4 Other Primality Tests

The Miller–Rabin test is, due to its efficiency, the preferred one for selecting large (probable) primes for cryptographic use. There are, however, several other tests that can be used for this purpose and some of them are deterministic and always provide the correct result. Before mentioning some of these tests, let us go back to Maple's function `isprime` which we have used in the previous experiment as a "black box" to provide us with primes or with composites. But what is the algorithm underlying `isprime`? Well, the algorithm is, as of Maple v12, none other than Miller–Rabin's, combined with some trial division to detect small primes, a variant of the base-2 strong probable prime test and a *Lucas test*. This can be checked by printing Maple's

listing of the function and noting that it calls the built-in command `gmp_isprime`. This is a function from the *GNU Multiple-Precision Bignum Library (GMP)* and, as can be seen in the documentation of this library or by inspecting the code (available at http://gmplib.org/), it is just an implementation of the Miller–Rabin test. But wait, we have used `isprime` to provide us with the numbers required by the experiments with Miller–Rabin, so it seems that we are trapped in a kind of vicious circle here ... However, this does not invalidate these experiments because it is known that for the rather small numbers that we used in them, `isprime` always provides the correct answer. Moreover, even if `isprime` gave some erroneous results, the rough comparisons we made for different number of bases and for numbers of different size would still be meaningful.

The Lucas probable prime tests and their generalizations (not to be confused with the Lucas–Lehmer test which is deterministic and very inefficient except in particular cases) are described in [60] and [194] and they are often used in conjunction with Miller–Rabin because it is believed that these tests are in some sense independent and hence the probability that a composite number passes both tests should be very small. In fact, Pomerance, Selfridge and Wagstaff made in [159] the conjecture that no odd composite positive integer is both a strong pseudoprime to the base 2 and a Lucas pseudoprime for a certain choice of parameters (see, e.g., [194, p. 165]). In 1980, Pomerance, Selfridge and Wagstaff offered $30 for a proof or disproof of this conjecture. They later raised the prize to $620 so that this question became known as the "$620 *problem*".

More recently, the challenge has been simplified and it consists in finding a number $n \equiv \pm 2 \pmod 5$ that is simultaneously a base-2 (Fermat) pseudoprime and a Fibonacci pseudoprime (a special kind of Lucas pseudoprime; see below for the definition). A peculiar thing is that the prize money came from the three authors, but not equally: Selfridge offered $500, Wagstaff offered $100 and Pomerance offered $20 for finding such an n and they also agreed to pay $620, with Pomerance and Selfridge reversing their roles, for a proof that no such n exists (see [60, Research Problem 3.41, p. 168]). It is clear that this uneven distribution of the reward money to be paid was a reflection of the authors' feelings about the most likely outcome. In fact, Pomerance has an (unpublished) heuristic argument written in 1984 that suggests that, even for the stronger version of the problem, there is an infinite number of counterexamples n that pass these tests without being prime. This argument is available from the web page of Grantham [95], where there is also a list of primes compiled by Alford and Grantham that may be helpful in solving the problem. Grantham strongly believes that some sub-product of these primes is a Carmichael number and a Fibonacci pseudoprime that is also $\equiv 2$ or $\equiv 3 \pmod 5$.

Let us sketch a naive attack on the $620 problem using the Alford–Grantham list of primes. First we define a Fibonacci pseudoprime. Let $F_0 = 0, F_1 = 1, \ldots F_i = F_{i-1} + F_{i-2}$ be the sequence of Fibonacci numbers. An integer n such that $n \equiv \pm 2 \pmod 5$ is called a *Fibonacci pseudoprime* when it is composite and divides F_{n+1}. Thus we are looking for a composite number n that is simultaneously a 2-pseudoprime and a Fibonacci pseudoprime $\equiv \pm 2 \pmod 5$. Such a number should pass a test composed of the following ingredients:

1. The Fermat test to base 2 given by the previously defined function `Fermat2Test`.
2. The Fibonacci test for numbers $\equiv \pm 2 \pmod 5$:

```
> FibonacciTest := n -> (evalb(n mod 5 = 2) or evalb(n mod 5 = 3)) and
                       evalb(combinat:-fibonacci(n+1) mod n = 0):
```

Then the 'PSW test' would be:

```
> PSWTest := n -> Fermat2Test(n) and FibonacciTest(n):
```

We leave winning the $620 as a (not easy!) exercise:

Exercise 6.12

(i) Download the text file called `primes620.txt` from http://www.pseudoprime.com/pseudo.html.
(ii) Read this text file as a Maple list consisting of 2030 primes.
(iii) Construct a suitable (big) list of composite numbers by multiplying appropriate subsets of these primes.
(iv) Try to find a solution by applying the function `PSWTest` to the members of this list ...

The probabilistic primality tests (or compositeness tests) mentioned so far are the ones commonly used to select primes for cryptographic use but there are also deterministic tests that, while less efficient, provide also a proof of the primality of the numbers that are declared prime. One of the more powerful is CPP ("Cyclotomy Primality Proving", also called the Jacobi sum test, [52, 60]) whose running time is $O((\ln n)^{c \ln\ln\ln n})$ for some constant c. Although this is not polynomial time, the function $\ln\ln\ln n$ grows very slowly (its value is about 5 when n has five million decimal digits) and, for all practical purposes, it behaves as if it were bounded. Because of this it is sometimes said that the algorithm runs in *superpolynomial time* and it has been used to obtain primality proofs of primes of several thousand decimal digits.

An alternative test which is expected to be asymptotically more efficient is ECPP ("Elliptic Curve Primality Proving") based on the arithmetic of elliptic curves which, heuristically, runs in *expected polynomial time* $O(\ln^{5+\varepsilon} n)$ using fast multiplication techniques. This is a probabilistic Las Vegas algorithm which always gives the correct result but the running time could be much larger than the above estimate on some inputs (and, moreover, the expected running time has not been rigorously proved). However, the test works very well in practice and has the additional advantage of providing a *certificate* that permits the primality to be checked much more quickly. A distributed version of ECPP due to Morain [146] has been used to prove the primality of some very large primes, and there is also a nice version of ECPP for Windows operating systems due to Martin [137].

None of the preceding algorithms for primality testing provides an answer to a problem that remained open for many years: is there a deterministic polynomial-time algorithm for testing the primality of an integer?

The answer is yes and it only arrived in 2002 when Agrawal, Kayal and Saxena introduced their remarkable algorithm, now known as the *AKS algorithm* or *AKS test* [3]. This result is striking because the algorithm is rather simple and the proof, while being very ingenious, uses only elementary methods. The algorithm is based on the following theorem:

Theorem 6.10 *Let $n > 1$ be an integer. Then n is prime if and only if for all $a \in \mathbb{Z}_n^*$, the following identity in the ring $\mathbb{Z}_n[X]$ holds: $(X + a)^n = X^n + a$.*

The AKS algorithm shows that the decision problem 'Is n prime?' belongs to the class $\mathcal{P}$ but, despite being a major breakthrough, the algorithm is only of theoretical interest and is not currently useful for selecting primes for cryptographic applications. Its running time is $O(\ln^{12+\varepsilon} n)$ and, under a widely believed conjecture about *Sophie Germain primes*,[3] i.e., primes p such that $2p + 1$ is also prime, it comes down to $O(\ln^{6+\varepsilon} n)$. This is slow not only in comparison to Miller–Rabin's test but also in comparison to the expected time of ECPP. Several people are currently working on improving the algorithm (and its running time) but whether it will become useful to find large primes in practice remains to be seen. Detailed discussions of the AKS algorithm can be found in [60, 180].

Remark 6.7 The comparison between CPP and AKS provides an example of a rare situation in which a superpolynomial-time test is more efficient than a polynomial-time test in practice. Of course, this is not so surprising if one bears in mind the slow growth of the exponent $\ln \ln \ln n$ appearing in the time estimate of CPP.

6.3 Generating Random Primes

For many cryptographic applications it is necessary to generate large random primes. These can be either primes of a specified bit length k or primes less than a given bound. In both cases, the idea (sketched at the beginning of this chapter when studying the distribution of primes) is simple enough: given a primality test, integers in the required range are randomly generated and submitted to the primality test until a prime is found.

The simplest way to generate a random integer in the interval $[0, n - 1]$ (i.e., a random element of $\mathbb{Z}_n$), where $\text{len}(n - 1) = r$, is just to generate r random bits and to check whether the integer defined in the standard way by the r-bit string thus obtained is less than n, repeating the process until this condition is satisfied. This condition can be checked in polynomial time and, clearly, the probability that a random r-bit string satisfies it is equal to $n/2^r$ and hence $\geq 1/2$. Thus the expected number of trials is ≤ 2 and the algorithm runs in expected polynomial

[3] This conjecture postulates that Sophie Germain primes are fairly frequent, see [180, Conjecture 5.24] and the discussions in [180, 21.2, 21.3] and [60, 4.5.1, 4.5.2]. There is some heuristic evidence in favor of this hypothesis, however it is not even known whether or not there are infinitely many of these primes.

time. Similarly, if we want to generate an r-bit integer, we just set the first bit to 1 (and also the last bit if we want an odd integer) and generate the remaining bits at random.

The other ingredient we need to generate random primes is a primality test, so suppose that we have one called `PrimeTest` that returns "prime" or "composite" when performed on a positive integer n. The algorithm to generate an r-bit random prime is then:

Algorithm 6.3. `Randomprime.`

Input: A length parameter $r > 0$.
Output: A random r-bit prime.

1. `Initialize` a Boolean variable `found := false`;
2. `while not found do`
   $n \leftarrow [2^{r-2}, 2^{r-1} - 1]$;
   $n := 2n + 1$;
   `if PrimeTest`(n) = "prime" `then`
   `found := true`
   `end if`
   `end do`;
3. `return` n.

If the algorithm `PrimeTest` used here as a subroutine is a deterministic polynomial-time algorithm such as AKS or a Las Vegas algorithm such as ECPP, then `RandomPrime` is a Las Vegas algorithm that correctly generates a random prime in expected polynomial time. As already mentioned, the AKS test is not fast enough, in practice, for primes of cryptographic size, but the use of ECPP is viable for this purpose. However, the probabilistic tests—and, in particular, Miller–Rabin—are considerably faster and are preferred in many situations, in particular in environments where computational power is severely limited, as in the case of smart cards. Moreover, the Miller–Rabin test is usually complemented with some trial division that will get rid of the numbers with small prime factors and will improve efficiency even more. The price to pay for using a Monte Carlo algorithm such as Miller–Rabin is that the output will not necessarily be correct but the prime-generating algorithm can be made to run in polynomial time in such a way that it outputs a random prime except with negligible probability.

If the Miller–Rabin algorithm with k random bases is used, we know from Theorem 6.9 that, if n is composite, then the algorithm will mistakenly output "prime" with probability $\leq 4^{-k}$. Thus one could be tempted to think that `RandomPrime` will return a prime with probability $\geq 1 - 4^{-k}$ but this is not correct. The probability of error we are interested in here is not the probability that a composite number passes the test (and is therefore declared prime) but the probability that a number that passes the test (and is declared prime) turns out to be, nevertheless, composite. This probability cannot be estimated by using Theorem 6.9 alone as it depends strongly on the density of primes in the interval where the integers are chosen. To see this,

let $P(r, k)$ be the probability that a random r-bit integer that has passed the Miller–Rabin test with k random bases turns out to be composite. Using the Prime Number Theorem we see that, for example, if we choose a random odd 1024-bit integer, we have a probability of about $1/355$ of choosing a prime. Suppose, only for the sake of this argument, that the probability that a composite number passes the test with one base is approximately $1/4$. Then it would be completely erroneous to assume that $P(1024, 1) \approx 1/4$. Indeed, it is clear that, with these hypotheses, $P(1024, 1) > 1/4$ for it would be much more likely to have the "bad luck" of choosing a composite that passes the test with probability close to $1/4$ than to have the "good luck" of choosing a prime, with probability around $1/355$.

If we denote by X the event that n is composite, $\bar{X}$ the event that n is prime, and Y_k the event that Miller–Rabin with k bases declares n to be prime, then we have:

Proposition 6.1 $P(r, k) \leq 4^{-k}/\Pr(\bar{X})$.

Proof Using the inequality $\Pr(Y_k) \geq \mathrm{pr}(\bar{X})$, Bayes' formula, and Theorem 6.9 which tells us that $\Pr(Y_k|X) \leq 4^{-k}$, we have:

$$P(r, k) = \Pr(X|Y_k) = \frac{\Pr(Y_k|X)\Pr(X)}{\Pr(Y_k)} \leq \frac{\Pr(Y_k|X)}{\Pr(Y_k)} \leq \frac{\Pr(Y_k|X)}{\Pr(\bar{X})} \leq 4^{-k}/\Pr(\bar{X}).$$

□

Thus we see that the probability of error of the Miller–Rabin based `RandomPrime` algorithm might increase with the size r of n because the term 4^{-k} is multiplied by a factor equal to the inverse of the probability that a random odd integer of size r is prime. Since, by the PNT, $\Pr(\bar{X}) \approx 2/\ln 2^r$, this factor would be approximately equal to $(r \ln 2)/2$. If the error probability did indeed increase with the size of n, then it would also be necessary to increase the number of bases accordingly and this would make `RandomPrime` less efficient (although still polynomial). However, this does not take into account the fact that, as we have seen in the experiments above, $\Pr(Y_k|X)$ is usually much smaller than 4^{-k} and, moreover, it decreases as r increases. In fact, Burthe proved in [44] that $P(r, k) < 4^{-k}$ and, for large integers, the value of $P(r, k)$ is actually much smaller than this bound because $P(r, k)$, with k fixed, also decreases as r gets larger. For example, it it also shown in [44] that $P(500, 1) < 4^{-28}$ so that a random 500-bit integer that has passed Miller–Rabin with just one base has a very high probability of being prime. This probability might not be considered large enough, but from [44] it also follows that $P(500, 16) < 10^{-50}$, which can be considered negligible in the practical sense.[4]

From a more theoretical point of view, we note that, if we take, for example, $k = r = O(\mathrm{len}(n))$, then the algorithm runs in expected polynomial time $O(r^5)$ (since we expect to make $O(r)$ calls to Miller–Rabin, and for each of them we would perform $O(r)$ exponentiations each of which requires time $O(r^3)$). Moreover, since

[4] Regarding this probability we can mention the following quote from Borel cited in [118, p. 126]: *Un phénomène dont la probabilité est* 10^{-50} *ne se produira donc jamais, ou du moins ne sera jamais observé.*

$P(r,k) < 4^{-k}$, we have that the probability of error of the prime-finding algorithm will, in this case, be $P(r,r) = O(4^{-r})$ which is certainly negligible in the length parameter r. We stress that, in practice, there is no need to increase the number of bases used for the Miller–Rabin test and using a constant value (such as $k = 20$ or $k = 25$) is sufficient for, once the error probability is below, say 10^{-50}, there is no need to make it even smaller for larger numbers (and, as remarked before, this probability will anyway decrease as the number size r increases). If k is kept constant, then the prime-finding algorithm will run in polynomial time $O(r^4)$.

Miller–Rabin is, due to its efficiency, the test of choice to obtain large primes for cryptographic use. The "probable primes" furnished by the test have been called *industrial-grade primes* because they are very likely prime but have not been *proved* to be prime. In relation with this it might be pointed out that, from an industrial point of view, the quality of these probable primes is orders of magnitude above anything produced in the usual industrial manufacturing processes. It should also be pointed out that *any* large prime which is actually being used in cryptography is in some sense an industrial-grade prime. Even if a deterministic algorithm has been used to prove[5] that the number is prime, the algorithm must be implemented and run on a machine. The implementation might have bugs and the machine will most likely suffer from hardware errors. Both these facts might conceivably affect the result of applying the algorithm and, even if the probability that they lead to an erroneous result is low, will it be lower than the error probability of Miller–Rabin? It seems a bit naive to assume, prima facie, that this will indeed be the case...

All the variants of the `RandomPrime` algorithm are probabilistic since they require as input random integers of a given length, which are subsequently tested for primality. From a theoretical point of view it is an interesting problem to find a deterministic version of `RandomPrime` but there is no obvious way to "derandomize" this algorithm. Thus the following problem, as described by Tao (cf. [157]), is open[6]:

> Find a deterministic algorithm which, when given an integer r, is guaranteed to find a prime of at least r digits in length of time polynomial in r. You may assume as many standard conjectures in number theory (e.g. the Extended Riemann hypothesis) as necessary, but avoid powerful conjectures in complexity theory (e.g. $\mathcal{P} = \mathcal{BPP}$) if possible.

This problem is discussed in several threads accessible from [157] and also in [192].

6.3.1 Generating Safe Primes

Sometimes in cryptography it is desirable to generate random r-bit primes of a particular form. For example, a *safe prime* is a prime of the form $p = 2q + 1$, where q is also prime (so that q is a Sophie Germain prime). As we have mentioned, it is not known whether there are infinitely many Sophie Germain primes but the heuristic

[5] Let us assume here that the mathematical proof gives absolute certainty, something that is by no means obvious and is subject to much philosophical debate.

[6] This problem highlights again the difference between primality testing and finding random primes.

evidence suggests that this is the case and, in fact, that they are sufficiently dense to allow them—and hence also safe primes—to be found in probabilistic polynomial time, see [180, 5.5.5]. We will see that safe primes are useful for some public-key encryption schemes and hence we will briefly discuss how to generate them. In general, to generate primes of a special form we may use a method similar to the one just discussed to generate random primes, but the estimation of the running time or the error probability will be more difficult as it will depend on the density of primes of the particular form considered. In the specific case of safe primes, it is not obvious what the density of primes among the numbers of the form $2q + 1$, with q prime, is. Let us have a closer look at this in order to estimate the complexity of generating random safe primes. A first straightforward observation is the following:

Lemma 6.1 *If* $p = 2q + 1$ *is a safe prime, then* $q \equiv 5 \pmod 6$ *and* $p \equiv 11 \pmod{12}$.

Proof Since q is odd we have that $q \equiv 1 \pmod 2$ and since q is prime, $q \not\equiv 0 \pmod 3$. We also have that $q \not\equiv 1 \pmod 3$ for otherwise p would be a multiple of 3, in contradiction with the fact that p is also prime. Thus we see that $q \equiv 2 \pmod 3$ and this, together with $q \equiv 1 \pmod 2$ implies (using the Chinese remainder theorem) that $q \equiv 5 \pmod 6$. This clearly implies that $p \equiv 11 \pmod{12}$. □

Thus, in order to generate a random safe prime we can proceed as follows:

1. Choose a random integer n of appropriate size and set $q := 6n+5$ (alternatively, choose a random odd integer n of appropriate size and set $q := 3n + 2$).
2. Apply the Miller–Rabin test to q. If q fails the test go to 1). Otherwise, set $p := 2q + 1$.
3. Apply the Miller–Rabin test to p. If p fails the test go to 1). Otherwise, return p.

To estimate the running time of this algorithm we only have to estimate the average number of calls to the Miller–Rabin test. Suppose that we want a safe prime p of length r. Since the length of q is only one bit less and we intend to apply the algorithm to large primes, we have that $\text{len}(q) \approx r$. Then, if we choose q in the congruence class of 5 (mod 6) we have seen that, as a consequence of the PNT for arithmetic progressions, the expected number of draws until finding a prime is $\frac{\phi(6)}{6} r \ln 2 = (r \ln 2)/3$, so that it is just one-third of the expected number of draws to find a prime among all the integers of the same size. Once a prime q is found we have to check whether $p = 2q + 1$ is also prime. If the numbers of this form behave as random numbers in the congruence class of 11 modulo 12 (to which all of them belong), then one would expect that one in each $\frac{\phi(12)}{12} r \ln 2 = (r \ln 2)/3$ would be prime and hence the average number of draws of integers q would be, approximately, $r^2 \ln^2 2/9 = O(r^2)$. Since each draw requires either one of two calls to the Miller–Rabin test with complexity $O(r^3)$, the algorithm would run in expected polynomial time $O(r^5)$. We remark that the argument is not rigorous but one would expect it to hold if the primes follow the so-called *Cramér's model*, formulated by H. Cramér in the 1930s, which postulates that if we consider a sequence $\{X_n\}_{n=2}^{\infty}$ of independent

random variables taking values 0 and 1 with $\Pr(X_n = 1) = 1/\ln n$ (for $n \geq 3$), then the sequence $\{n \geq 2 | X_n = 1\}$ is a model for the sequence of primes.

Remarks 6.2

1. The algorithm above is more than twice as fast as the similar algorithm in which q is simply chosen to be a random odd integer, for then an argument similar to the above would give an expected number of draws equal to $r^2 \ln^2 2/4$.
2. The algorithm above has the drawback that if we set $q := 6n + 5$ for some randomly chosen n, q will not be uniformly distributed among all the integers $\equiv 5 \pmod 6$ which have a fixed binary length. If we want q to have this property—in which case p will also be uniformly distributed among the integers $\equiv 11 \pmod{12}$ of length equal to $\text{len}(q) + 1$—then we can modify step 1 in the algorithm by choosing a random odd q of the required length and testing whether $q \equiv 2 \pmod 3$. If this condition holds then the algorithm proceeds as above, otherwise this q is discarded and a new q is chosen.

6.3.2 Generating Pseudo-Random Primes with Maple

We are going to implement in Maple the above-outlined algorithm to generate a random prime of given length. Since we cannot generate true random integers with Maple, we will settle for generating them pseudo-randomly. For cryptographic use, it is important that an adversary be unable to guess the primes that will be generated and so we will use the cryptographically strong Blum–Blum–Shub PRG. For the primes to be generated pseudo-randomly it is also crucial that the PRG is supplied with a true random seed and we will assume that the seed has indeed been randomly generated. In order to prevent brute-force and other attacks, the seed must also have reasonable size: at least 128 bits or even 256 bits or more in some cases. We will often use 128-bit random seeds in our examples but the reader can easily replace them with, say, 256-bit seeds for increased security.

Note that if we choose a 128-bit string at random, the corresponding integer might have fewer than 128 bits because the string could have leading zeros. Thus we do not check the size of the number—it is impossible, anyway, to check whether an external seed was randomly generated or not—and we will assume that all these requirements are met. To handle the seed, the procedure `stringposint` from Appendix A will be used.

The function has two required parameters, `seed` (given either as a hexadecimal string or as an integer) and `length` (the bit length of the prime to be found). There are also two optional keyword parameters: `numbases` to specify the number of bases to be used by Miller–Rabin (with default equal to 20) and `bbslength` that specifies the bit length of the primes used by Blum–Blum–Shub (with default 1024). The function is the following:

```
> PseudoRandomPrime := proc(seed::{posint, string}, length::posint,
                        {numbases::posint:=20, bbslength::{512, 768, 1024}:=1024})
  local s, B, prime, n;
```

```
  s := stringposint(seed);
  B := RandomTools:-BlumBlumShub:-NewBitGenerator(s, primes = bbslength);
  prime := false;
  while not prime do
    n := convert(cat(1, seq(B(), i = 1 .. length-2), 1), decimal, binary);
    prime := MillerRabin(n, numbases)
  end do;
  n
  end proc:
```

For example, a 1024-bit pseudo-random prime may be generated as follows:

```
> PseudoRandomPrime(19980265180094725216637080723906857941 5, 1024);
15600648073135628308192446236674420732043764209149672683201135830883108286600298 65\
  0946481034992774931994501971114418497589191444442324357180913377120361250096787 1\
  8738073200531785977213640172391743853306808125388117223126830491474587731368775 7\
  3049096319346151670082885140689281246081222876119459701438239776611
```

Let us now give a similar function to generate pseudo-random safe primes of a given length. The parameters are the same as those of the previous function and a safe prime of the required length is produced as output:

```
> PseudoRandomSafePrime := proc(seed::{posint, string}, length::posint,
                          {numbases::posint:=20, bbslength::{512, 768, 1024}:=1024})
  local s, B, primep, primeq, q, p;
  s := stringposint(seed);
  B := RandomTools:-BlumBlumShub:-NewBitGenerator(s, primes = bbslength);
  primep := false;
  while not primep do
    primeq := false;
    while not primeq do
      q := convert(cat(1, seq(B(), i = 1 .. length-3), 1), decimal, binary);
      if q mod 3 = 2 then
        primeq := MillerRabin(q, numbases)
      end if
    end do;
    p := 2*q+1;
    primep := MillerRabin(p, numbases)
  end do;
  p
  end proc:
```

Example 6.7 Let us generate a 1024-bit safe prime:

```
> PseudoRandomSafePrime(29477983376442558187316294782580797080 8, 1024);
17328165925465747742852145265340236545082566432780000430958759185087352989563944 85\
  4566856147240835861342469150055623474555275412110913905702041424290444739391553 0\
  4265037506925142587301897565098750305444107807281488314244259629975103151218296 7\
  6339640034779641357853974926946295373058442933189109231598423704283
```

6.4 The Integer Factorization Problem

We have already mentioned the integer factorization problem on several occasions. In Chap. 2 we have seen that the factorization decision problem belongs to the classes $\mathcal{NP}$ and co$\mathcal{NP}$ and we have also mentioned that this problem is believed not to belong to the class $\mathcal{P}$. Later, in Definition 3.15, we have introduced the factoring assumption, which formalizes the widely believed hypothesis that "factoring is hard",

and we proved that it implies the existence of families of one-way functions (Theorem 3.8) which, in turn, leads to the existence of pseudo-random generators (Theorem 3.10). These results show that the conjectured hardness of the factorization problem plays an important role in private-key cryptography. As we will see in coming chapters, the factorization problem is also of crucial importance for public-key cryptography, where the factoring assumption is a necessary condition for the security of several encryption and digital signature schemes.

In order to better understand the reasons why the factorization problem is regarded as difficult, we will have a look at some of the most important factorization algorithms. It is important to understand how they work and what their limitations are, in order to be able to choose cryptographic parameters in a sensible way.

6.4.1 Trial Division as a Factoring Algorithm

We have already discussed an algorithm that is able to factor integers: trial division. In our discussion we used the algorithm as a primality test, but a straightforward modification converts it to a factoring algorithm. Of course, this algorithm also embodies an (expensive) primality test and it will just return the input number in case it is prime. As we have seen, the algorithm runs in exponential time $O(n^{1/2+\varepsilon})$ (without taking into account the time to generate the prime numbers up to a certain bound) and hence it cannot factor large composite numbers unless all their prime factors, except perhaps the largest one, are small. The algorithm plays a role, nevertheless, as an auxiliary step in other algorithms as it serves to quickly get rid of small factors.

We are now going to give a Maple implementation of the algorithm and, as we did when discussing it in Chap. 2, we will start by building an array containing the primes up to 2^{28}, by using the previous function `Sieve` (which implements Eratosthenes' sieve) as follows:

```
> primearray := Sieve(2^28):
```

Thus all these primes are stored in a Maple array, in the global variable `primearray`. Since now we are interested in trial division as a factoring algorithm and not as a primality test, we will use Maple's function `isprime` to check whether the number is prime and also, after factoring out a prime divisor, to check whether the remaining unfactored part is prime. This will significantly reduce computation time in many cases—at the cost of a slight time increase in others—and illustrates a principle that one should always follow when trying to factor a number whose primality/compositeness status is unknown: never try to factor a number unless it is already known to be composite. In other words, since primality testing is much cheaper than factoring, use it before trying to factor a number to avoid ending up with the much more expensive factoring algorithm being used as a primality test.

The next function implements trial division factoring in Maple. The input is a positive integer n and the output is a list containing all the prime factors of n less than 2^{28} (which may include repeated factors), and, possibly, a cofactor that may be

composite. This list will give the complete prime factorization of n provided that the second largest prime factor of n is smaller than 2^{28} and this will always be the case if all the factors appearing in the list are $\leq 2^{56}$. In particular, all integers $n \leq 2^{56}$ will be completely factored. When the last integer in the list is composite, the factorization is not complete and the function will then print a message informing that this has happened.

```
> TrialDivisionFactor := proc(n::posint)
  local s, p, t, r, f;
  if isprime(n) or n=1 then
    return [n]
  end if;
  s := isqrt(n);
  t := n;
  r := 1;
  f := [];
  for p in primearray while r <> 0 and p <= s do
    r := modp(n, p);
    while r = 0 do
      t := iquo(t, p);
      f := [op(f), p];
      r := modp(t, p);
      if isprime(t) then
        return [op(f), t]
      end if
    end do;
  end do;
  printf("Incomplete factorization:");
  [op(f), t]
  end proc:
```

Example 6.8 We start with a historical example that illustrates the dangers of making predictions about the future of factoring. In his 1874 treatise *The Principles of Science: A Treatise on Logic and Scientific Method*, the English economist, logician and statistician Jevons stated: *"Can the reader say what two numbers multiplied together will produce the number* 8616460799? I think it unlikely that anyone but myself will ever know." The number was already factored by D.N. Lehmer in 1903 and, as Knuth remarked in [115, p. 388], Fermat could have factored it in less than 10 minutes on the back of an envelope. We will see later the method that Fermat would have used but now we just remark that, with a computer, the above method factors this number in a tiny fraction of a second:

```
> TrialDivisionFactor(8616460799);
                              [89681, 96079]
```

Example 6.9 Another famous historical example of an integer factorization also happened in 1903. In the eighteenth century, the French monk and mathematician Mersenne gave a list of all the primes of the form $2^p - 1$, with p prime, for $p \leq 257$. The primes of this form are now called *Mersenne primes* and Mersenne's list had a few errors because it contained some composites while some primes were missing. Among the numbers included was $M_{67} = 2^{67} - 1$. At a 1903 meeting of the American Mathematical Society, F.N. Cole showed that this number is the product of two specific primes by first raising 2 to the 67th power and subtracting 1, and then multiplying the two primes to show that the result was precisely M_{67}. All these

operations were done on the blackboard and Cole famously did not pronounce a single word during his "talk". When he was later asked by E.T. Bell, in 1911, how long it had taken him to crack M_{67}, he answered: *"three years of Sundays"*. This episode is a nice illustration of the fact that the factorization problem belongs to the class $\mathcal{NP}$ but also of the enormous advances made in the problem during the last century—with a decisive role being played by computers. Even trial division, the humblest of all factoring algorithms, can now factor this number very quickly:

```
> TrialDivisionFactor(2^67-1);
                          [193707721, 761838257287]
```

Exercise 6.13 Modify the function `TrialDivisionFactor` so that the array containing the primes is computed and stored in a local variable each time the procedure is run. The size of this array should then be made dependent on the size of the number to be factored.

6.4.2 Pollard's Rho Method and Its Maple Implementation

We are going to describe a factorization algorithm introduced by Pollard in 1975. This algorithm is inspired by the birthday problem and, despite being exponential, is substantially faster than trial division. Although this algorithm cannot usually be directly used to factor numbers of cryptographic size, there are several good reasons to study it. One of them is that it may be used as a subroutine in more powerful methods in order to factor auxiliary numbers of small size. Another important reason is that the rho method is the basis for a similar algorithm to compute discrete logarithms. In the specific case of the discrete logarithm problem for elliptic curves we will see that the strongest known attacks are based precisely on this algorithm.

The idea underlying the rho method is the following. Suppose that n is a composite number and F is a random function from $\mathbb{Z}_n$ to itself that, for each prime factor p of n, satisfies that $F(x) \equiv F(x \mod p) \pmod p$, so that F factors through the canonical projection from $\mathbb{Z}_n$ to $\mathbb{Z}_p$ and induces a function $f : \mathbb{Z}_p \to \mathbb{Z}_p$, given by $f(x) = F(x) \mod p$ for all $x \in \mathbb{Z}_p$. All the polynomial functions F satisfy this condition and experiments suggest that, although not random, the function $F(x) = (x^2 + b) \mod n$ heuristically behaves "like a random function" provided that $b \neq 0$ and $b \neq -2$ (see [194] and [60] for a discussion of these issues). Now let $s \in \mathbb{Z}_n$ be a random element and consider the sequence obtained by iterating $F : \mathbb{Z}_n \to \mathbb{Z}_n$:

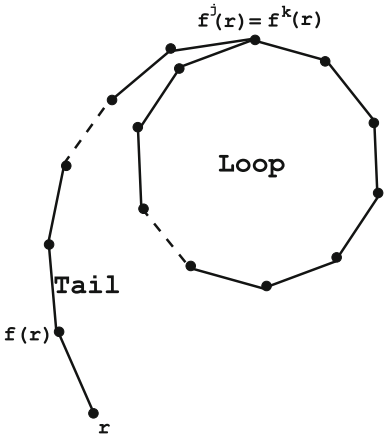
$$s, F(s), F(F(s)), F(F(F(s))), \ldots$$

Then, if p is an unknown prime factor of n and $r = s \mod p$, we also have an induced sequence of elements in $\mathbb{Z}_p$:

$$r, f(r), f(f(r)), f(f(f(r))), \ldots$$

This sequence consists of the values obtained by repeatedly applying the function f starting with the initial value $s \mod p$ and we expect them to be randomly and uniformly distributed in $\mathbb{Z}_p$. Since $\mathbb{Z}_p$ is finite, the sequence must eventually repeat a term and from this point on become cyclic. This behavior can be graphically represented by a letter ρ where the tail corresponds to the precyclic part of the sequence and the loop to the cyclic one. Now, the interesting question is: how long do we expect these parts to be?

It is clear from Theorem 5.4 that we expect that some number in $\mathbb{Z}_p$ will be repeated after roughly $\sqrt{p}$ iterations of f or, in other words, that the expected length of the tail and the loop together will be of the order of $\sqrt{p}$. That is, we expect there to be $k = O(\sqrt{p})$ steps with $0 \leq j \leq k$ and $f^j(r) = f^k(r)$. We do not know f because we do not know p but if $u = F^j(s)$ and $v = F^k(s)$ are elements in the sequence of iterates of F such that $f^j(r) = f^k(r)$, we have that $F^j(s) \equiv F^k(s) \pmod{p}$ or, in other words, $u \equiv v \pmod{p}$. Then p divides $u - v$ but, because of the randomness of the sequence, it is unlikely that another prime factor q of n also divides $u - v$. Thus we expect that $u \not\equiv v \pmod{n}$ and this means that the gcd of $u - v$ and n is a proper factor of n and, in fact, it is probably $\gcd(u - v, n) = p$.



Pollard's Rho

Now, the problem is that we cannot compute all the gcds for every pair $i, j < \sqrt{p}$ because there are about $\frac{1}{2}(\sqrt{p})^2 = p/2 = O(p)$ such pairs, which would lead to a running time estimate even worse than trial division. Fortunately, there is a wonderful solution to this problem which is given by *Floyd's cycle-finding algorithm*. This algorithm computes two iterates of F in the same loop with one instance running twice as fast as the other one, i.e., it computes $s_i = F^i(s)$ and $s_{2i} = F^{2i}(s)$ together and $\gcd(s_{2i} - s_i, n)$ hoping to find p. Suppose then that $s_j \equiv s_k \pmod{p}$ for some $j < k$ so that, as already mentioned, we have that $k = O(\sqrt{p})$. Let $l = k - j$, so that for $i \geq j$ we have $s_i \equiv s_{i+l} \equiv s_{i+2l} \equiv \ldots \pmod{p}$. Consider the value $i = l\lceil j/l \rceil$, the first multiple of l that exceeds j. Then for this i we have that

$i + \lceil j/l \rceil l = 2i$ and hence $s_i \equiv s_{2i} \pmod{p}$, with $i \leq k = O(\sqrt{p})$. Thus Pollard's rho method proceeds by computing the sequence $\gcd(s_{2i} - s_i, n)$ for $i = 1, 2, \ldots$, and it should output a nontrivial factor of n in $O(\sqrt{p}) = O(n^{1/4})$ steps which, if $O(\text{polylog}(n)) = O(\text{len}(n)^c)$ gives a complexity of $O(n^{1/4}\text{polylog}(n))$. This is a heuristic estimate and a rigorous analysis of the algorithm remains to be done. This estimate means that the rho method will find a t-bit prime factor of n with roughly the same work needed to find a $t/2$-bit prime factor by trial division. The smallest prime factor p will usually be found first, although this may not be always the case since the algorithm is probabilistic. The running time is more dependent on the size of p than on the size of the integer n itself and in some cases $\sqrt{p}$ can be substantially smaller than $n^{1/4}$.

The preceding discussion leads to the following function that implements the rho method in Maple. The input is the (positive) integer to be factored and the output (if successful) a nontrivial factor of n in case n is composite or n itself if n is prime. It may happen that, even if n is composite, the first factor greater than 1 that the algorithm finds is n itself and, in that case—which is unlikely if n is large—the procedure will ask the user whether to continue the algorithm restarting with a new s and a new F, or to stop the computation.

```
> RhoFactor := proc(n::posint)
  uses RandomTools;
  local b, s, u, v, f, g, finished, a;
  if isprime(n) or n = 1 then
    return n
  end if;
  randomize();
  finished := false;
  while not finished do
    b := Generate(integer(range = 1 .. n-3));
    s := Generate(integer(range = 0 .. n-1));
    f := x -> `mod`(x^2+b, n);
    u := s; v := s; g := 1;
    while g = 1 do
      u := f(u);
      v := f(f(v));
      g := igcd(u-v, n)
    end do;
    if g = n then
      a := readstat("No factor found, continue?, y/n");
      if a = y then
        print("Computation proceeds with new seeds")
      else
        finished := true;
        print("Computation interrupted")
      end if
    else
      return g
    end if
  end do
  end proc:
```

Example 6.10 As a first historical example, let us try M_{67}:

```
> RhoFactor(2^67-1);
                                193707721
```

The factorization is obtained instantaneously in this case.

Example 6.11 A much more difficult example is the Fermat number $F_8 = 2^{2^8} + 1$. This number was known to be composite long ago but remained unfactored until 1980 when Brent and Pollard factored it by means of the rho method [41], which constitutes the greatest success of this algorithm. They used several modifications of the method, one of them being a specific improvement that took advantage of the fact that the prime factors of a Fermat number $F_k = 2^{2^k} + 1$ are congruent to 1 modulo 2^{k+2} (see [60, Theorem 1.3.5] and the discussion on how to choose the function F in [60, p. 231]). Now, we use the basic Pollard rho algorithm on this number and we obtain:

```
> RhoFactor(2^(2^8)+1);
                              1238926361552897
```

This completely factors F_8 because the remaining cofactor is a 62-digit prime as can easily be checked by using a primality test. The fact that this factorization can now easily be accomplished with Maple on a home computer bears testimony to the enormous advances (indeed exponential because of Moore's law) in computing power during the last years. One should bear in mind, however, that the rho method is probabilistic and a "chance factor" will influence the actual time taken to factor a number in each run of the algorithm.

Exercise 6.14 In their factorization of F_8 [41], Brent and Pollard applied a heuristic argument of the latter that says that if the factors p satisfy $p \equiv 1 \pmod{m}$ then the use of the polynomial $x^m + b$ (instead of $x^2 + b$) for the function F reduces the expected number of steps in the rho method by a factor of $\sqrt{m-1}$. In the case of F_8, the polynomial $x^{2^{10}} + 1$ and the initial value $s = 3$ were used. Modify the function `RhoFactor` so that it uses the polynomial $x^{1024} + 1$ and either $s = 3$ or s equal to a random value in $[2, n-1]$ and use it to factor F_8 in a matter of seconds.

6.4.3 Fermat's Factorization Method

We will now describe a factoring algorithm that, although rather old and of exponential complexity, contains the basic idea on which the fastest currently known factorization methods are based. This wonderful idea is due to Fermat and it starts with the basic remark that, given a composite integer n, if we are able to write it as a difference of squares:

$$n = x^2 - y^2,$$

where x and y are non-consecutive integers, then we have a nontrivial factorization of n:

$$n = (x + y)(x - y).$$

Proposition 6.2 *Let n be an odd positive integer. Then there exists a bijective correspondence between factorizations*

$$n = ab, \text{ with } a \geq b > 0$$

and representations of n of the form:

$$n = x^2 - y^2, \text{ with } x > y \geq 0.$$

Proof The correspondence is as follows. Given a factorization $n = ab$ we define $x = \frac{a+b}{2}$ and $y = \frac{a-b}{2}$ and then we have that $n = ab = (\frac{a+b}{2})^2 - (\frac{a-b}{2})^2 = x^2 - y^2$. For the inverse map, given $n = x^2 - y^2$, define $a = x + y$ and $b = x - y$ and note that then $n = x^2 - y^2 = (x + y)(x - y) = ab$. □

In order to represent n in the form $n = x^2 - y^2$, we can proceed by exhaustive search as follows. We start with $x_0 = \lceil \sqrt{n} \rceil$ and for $x = x_0, x_0 + 1, x_0 + 2, \ldots$ we compute $x^2 - n$ until finding a square y^2. It is clear that if $n = ab$ then the number of steps increases with the size of $x = \frac{a+b}{2}$ (clearly, the number of iterations is $x - \lceil \sqrt{n} \rceil + 1 = \frac{a+b}{2} - \lceil \sqrt{n} \rceil + 1$) and this number also increases as the difference between the factors $a - b = 2y$ increases. If n is odd and composite, the difference between the factors is the largest possible when $n = 3p$ with p prime. In this case, if we solve the equations $x + y = n/3$ and $x - y = 3$ we find that $x = (n + 9)/6$ and $y = (n - 9)/6$. Thus we see that in the worst case this algorithm is much less efficient even than trial division, as it will require $(n + 9)/6 - \lceil \sqrt{n} \rceil + 1 = \Theta(n)$ steps. Next we discuss the complexity of the method in more detail.

6.4.3.1 The Complexity of Fermat's Method

From the preceding discussion it follows that Fermat's method is most useful when n has two (not necessarily prime) factors whose difference is small. Small prime factors of a number are easily factored out either by trial division or by the rho method (and even more so by the Elliptic Curve Method, which we shall not discuss in detail) so, if one wants to build a composite integer n that it is hard to factor, the size of the prime factors should be as large as possible relative to the size of n. Bearing this in mind we see that the integers of a given size which are most difficult to factor are those that are the product of only two primes which have approximately the same length. But Fermat's method tells us that these primes should not be too close to each other for, in that case, the number will be easily factored.

Suppose, for simplicity, that $n = pq$ is the product of two primes and we want to factor it (as far as Fermat's method is concerned, the argument applies equally well if the two factors are the closest to $\sqrt{n}$ even if they are not primes but then n would probably be easier to factor by other methods). In several texts it is claimed that the number of iterations in Fermat's algorithm when trying to factor $n = pq$ is of the order of $|p - q|$ and so one should make this difference greater than a fixed bound large enough to make the computation of $|p - q|$ steps infeasible. As we are going to see, this analysis is erroneous and it is not sufficient to take $|p - q|$ greater than a fixed bound to prevent the success of Fermat's method (see [198]). In fact, the

number of iterations grows like $(\sqrt{p}-\sqrt{q})^2$ and, as p and q get larger, this number may stay fixed even as $|p-q|$ grows. Let us analyze this question.

Proposition 6.3 *Let $n = pq$ with $p > q$ primes and let t be the number of iterations in applying Fermat's method to the factorization of n. Then*

(i) $t = \lceil (\sqrt{p}-\sqrt{q})^2/2 \rceil = \lceil (p-q)^2/2(\sqrt{p}+\sqrt{q})^2 \rceil$,

(ii) $t < \frac{(p-q)^2}{8\lfloor\sqrt{n}\rfloor} + 1$,

(iii) If p and q are large primes of approximately the same length, then $len(t) \approx 2len(p-q) - len(p) - 3$. If $len(p-q) \leq (len(p)+3)/2$, then one iteration of the algorithm suffices to factor n.

Proof To prove (i) note that the algorithm proceeds to compute $x^2 - n$ for $x = x_0$, $x = x_0+1, \ldots$, where $x_0 = \lceil\sqrt{n}\rceil = \lfloor\sqrt{n}\rfloor+1$. When $x = \frac{p+q}{2}$ we have that $x^2-n = (\frac{p+q}{2})^2 - n = (\frac{p-q}{2})^2$ and n is then factored. The number of steps is $t = \frac{p+q}{2} - (\lfloor\sqrt{n}\rfloor+1)+1 = \frac{p+q}{2} - \lfloor\sqrt{n}\rfloor = \lceil \frac{p+q}{2} - \sqrt{p}\sqrt{q} \rceil = \lceil (\frac{(\sqrt{p})^2+(\sqrt{q})^2}{2} - \sqrt{p}\sqrt{q}) \rceil = \lceil (\sqrt{p}^2 + \sqrt{q}^2 - 2\sqrt{p}\sqrt{q})/2 \rceil = \lceil (\sqrt{p}-\sqrt{q})^2)/2 \rceil = \lceil (p-q)^2/2(\sqrt{p}+\sqrt{q})^2 \rceil$.

For (ii) observe that, as before, $t = \frac{p+q}{2} - \lfloor\sqrt{n}\rfloor$ satisfies $(\lfloor\sqrt{n}\rfloor+t)^2 - n = (\frac{p-q}{2})^2$ which in turn gives $t^2 + 2t\lfloor\sqrt{n}\rfloor = (\frac{p-q}{2})^2 + n - (\lfloor\sqrt{n}\rfloor)^2$. Discarding the term t^2 (which, moreover, in the cases of interest when t is much smaller than $\sqrt{n}$, will be small in comparison to $2t\lfloor\sqrt{n}\rfloor$), we obtain the bound $t < \frac{(p-q)^2}{8\lfloor\sqrt{n}\rfloor} + \frac{n-(\lfloor\sqrt{n}\rfloor)^2}{2\lfloor\sqrt{n}\rfloor}$. Since the latter fraction is clearly ≤ 1, we obtain the required bound. Observe that this already tells us that if $p - q < \sqrt{8}\sqrt[4]{n}$ then Fermat's method is successful after only one iteration.

Finally, to prove (iii) observe that, taking binary logarithms in (i) we obtain the approximation $\text{len}(t) = 2\text{len}(p-q) - (1+2\text{len}(\sqrt{p}+\sqrt{q}))$. Now, since p and q have approximately the same length, $\text{len}(\sqrt{p}+\sqrt{q})$ can be approximated by $\text{len}(\sqrt{p})+1$ and $2\text{len}(\sqrt{p}) \approx \text{len}(p)$. Thus we obtain $\text{len}(t) \approx 2\text{len}(p-q) - (1 + 2(\text{len}(\sqrt{p}) + 1)) = 2\text{len}(p-q) - \text{len}(p) - 3$. Moreover, by (ii) we have that one iteration will suffice if $(p-q)^2 \leq 8\sqrt{n}$. Taking binary logarithms we see that this happens if $2\text{len}(p-q) < 3 + \text{len}(n)/2$ which, approximating $\text{len}(n)/2$ by $\text{len}(p)$, gives that one iteration suffices when $\text{len}(p-q) \leq (\text{len}(p)+3)/2$. □

Remark 6.8 If $n = ab$ and a is the smallest factor of n greater than $\sqrt{n}$, then the bounds given in Proposition 6.3 work as well even if a and b are composite. The case $n = a^2$ is not included in the proposition but then $x_0 = \lceil\sqrt{n}\rceil = a$ and n is factored in one iteration.

Proposition 6.3 shows that the difference $p-q$ must increase as p and q grow, in order to prevent Fermat factoring. For example, if p and q are 1024-bit primes, (iii) shows that one iteration suffices to factor $n = pq$ even if $p-q$ is a 513-bit number, i.e., greater than 10^{150}. Let us then estimate how the length of $p-q$ should be related to the length of p (and q) to make Fermat's method infeasible. A computation involving 2^{64} operations like those performed in each iteration of Fermat's method seems feasible,

so that if we use the approximation $\text{len}(t) \approx 2\text{len}(p-q) - \text{len}(p) - 3$, given by Proposition 6.3 for the number of iterations t, then if $\text{len}(p) = \text{len}(q) = 1024$ and $\text{len}(t) = 64$, we obtain $2\text{len}(p-q) - 1027 = 64$. This tells us that if $\text{len}(p-q) \leq 546$ then the factorization will be feasible in practice.

We see that if the difference $p-q$ is close to $\sqrt[4]{n}$ then Fermat's method is effective to factor n. Thus, to obtain a security margin it seems reasonable to take $p-q > \sqrt[3]{n}$. Then we would have, by Proposition 6.3, that $t < \frac{(p-q)^2}{8\sqrt{n}} \approx \frac{n^{2/3}}{8n^{1/2}} = \frac{n^{1/6}}{8}$. If we approximate t by this bound we obtain that $\text{len}(t) \approx \text{len}(n)/6 - 3$ and then the computation would be infeasible for n a 1024-bit number, as this would imply a number of iterations close to 2^{167}. On the other hand, it has recently been shown in [72] that, by combining it with an algorithm of Coppersmith, Fermat's method can factor n in polynomial time whenever $|p-q| < n^{1/3}$. But one should not worry much about this when choosing large primes p, q aiming at $n = pq$ being hard to factor. The reason is that the probability that two randomly chosen k-bit primes p, q, satisfy $|p-q| < (pq)^{1/3}$, decreases exponentially as a function of k and is, therefore, negligible.

6.4.4 Fermat's Factorization Method in Maple

We are going to implement Fermat's factoring method in Maple and perform some experiments to check that the implementation behaves as predicted by the previous estimates. For efficiency reasons, we are not going to work with x and y when trying to find a value such that $x^2 - n = y^2$ but we are going to work instead with $k = 2x+1$ and $r = x^2 - n$. In the passage from one step of the algorithm to the next one, after a value of $r = x^2 - n$ has been computed, the new value of r is obtained by adding $k = 2x+1$ to the old value (because $k = 2x+1 = (x+1)^2 - x^2$). Also, the value of k increases by 2 in each step (as the value of x is increased by 1). This is somewhat more efficient because this way the computation of the square of x is replaced by an addition. The algorithm is then the following:

Algorithm 6.4. Fermat's factorization.

Input: n, an odd composite integer.
Output: Two nontrivial factors of n.
$x := \lceil \sqrt{n} \rceil$;
$k := 2x + 1$;
$r := x^2 - n$;
`while` r is not a square `do`
    $r := r + k$;
    $k := k + 2$
`end do`;
$x := (k-1)/2$;
$y := \sqrt{r}$;
`return` $x - y, x + y$.

The next function implements this algorithm in Maple. The function takes as input the integer to be factored, and as optional inputs, the maximum number of steps to be performed and an argument that tells the function whether to print the number of iterations used to find the factors. The output is a sequence consisting of two factors a, b such that $n = ab$ and, in case `numsteps` is set to `true`, with an additional term indicating the number of iterations used.

```
> FermatFactor := proc(n::And(posint,odd), maxnumsteps::posint:= 10^7,
                {numsteps::truefalse := false})
  local x, k, r, i, y;
  x := isqrt(n);
  if x^2 < n then
    x := x+1
  end if;
  k := 2*x+1;
  r := x^2-n;
  for i to maxnumsteps while not issqr(r) do
    r := r+k;
    k := k+2
  end do;
  if issqr(r) then
    x := (k-1)/2;
    y := isqrt(r)
  else
    error "%1 could not be factored in %2 iterations", n, maxnumsteps
  end if;
  if not numsteps then
    x-y, x+y
  else
    x-y, x+y, i
  end if;
  end proc:
```

As a first, easy example, let us try Jevons' number 8616460799:

```
> FermatFactor(8616460799, numsteps = true);
                              89681, 96079, 56
```

We see that the number was easily factored in 56 steps. Thus Fermat could indeed have done it by hand, especially bearing in mind that he already knew some tricks to speed up the verification of whether an integer is a square (we will mention these tricks below). Next, we do a couple of experiments to test how well the number of iterations agrees with that expected according to Proposition 6.3.

Example 6.12 We first give a Maple function to generate a composite number n which is the product $n = pq$ of two primes p, q of the same given length and such that the approximate length of $p - q$ is also given.

```
> composite := proc(length, lengthdiff)
  local p, q;
  randomize();
  p := 0;
  while not isprime(p) do
    p := (rand(2^(length-1) .. 2^length-1))()
  end do;
  q := nextprime(p+(rand(2^(lengthdiff-1) .. 2^lengthdiff-1))());
  p*q
  end proc:
```

Next we generate a product of two 1024-bit primes whose difference is a 513-bit number but we do not print it:

```
> c1 := composite(1024, 513):
```

According to part (iii) of Proposition 6.3, we should expect that `c1` is factored in just one iteration of the method. We check that this is indeed the case:

```
> FermatFactor(c1, numsteps = true);
937611948159818967017818470824998782058744850973202387353582653724080339337366058819380513 71\
  193805137122881531881292800245142334726030953438222090238195191940281260071669855639848217\
  307345696486234010500273315049088469619125158007322076218786532734644603403232 63\
  640473953205028640607039576812581554998809323921247145004288124531, 937611948159\
  818967017818470824998782058744850973202387353582653724080339337366058819380513 71\
  228815318812928002451423347260309534382220902381951919402812600716698556398482 17\
  009790549863496148711735941578889172346854560773416140751557209211434629523867 82\
  805366254306512176147775504898552084148460851348294061 63, 1
```

We obtain the two prime factors and we see that the number was indeed factored in one iteration. Now, let us generate a product of 1024-bit primes whose difference is a 526-bit number. According to Proposition 6.3, (iii), the number of iterations should have an approximate length of $2 \cdot 526 - 1024 - 3 = 25$. Let us perform the experiment:

```
> c2 := composite(1024, 526):
```

Now, we factor it using `FermatFactor`. As the expected number of iterations is more than the default 10^7 we specify 10^8 as the maximum number to be tried:

```
> f := FermatFactor(c2, 10^8, numsteps = true);
137109075637265980681834372316308092991085264939294038602720108235152529362319514 0\
  788246070512433780992106609920470393659427973357388700000909226757412189237670 96\
  399615355681199013789439954147900950784313466453387041041759446219882935868071 62\
  303845524002840436479573832721716053415787686315084460917913326601 3, 13710907563\
  726598068183437231630809299108526493929403860272010823515252936231951407882460 70\
  512433780992106609920470393659427973357388700000909226757413886093510925178320 31\
  638464471625516668844381187104354188124871768918912291954048926015896154924884 29\
  206950076998651503224651839897804266409752537833761124090 1, 26250266
```

Next we check the length of the involved primes and of the number of iterations, as well as the length of the difference $q - p$:

```
> ilog2~([f])+~1;
  ilog2(f[2]-f[1])+1;
                              [1024, 1024, 25]
                                     526
```

We see that these are indeed 1024-bit primes and their difference is a 526-bit number. More importantly, we also see that the number of iterations is a 25-bit integer, agreeing exactly with the expected value.

Exercise 6.15 Make a more exhaustive analysis of the behavior of the number of iterations in Fermat's method by generating composite numbers which are a product of two 1024-bit primes whose difference ranges from 513 to 526 bits and comparing the number of iterations required in each case with that expected according to Proposition 6.3.

Fermat's method admits many variants that improve its running time. The costliest part in each iteration of the algorithm is the "squareness test" which, in Maple, consists of a computation of the integral part of the square root of the number followed by its squaring to check whether it agrees with the number. A trick that Fermat already knew in order to speed up this process was to look at the last digit or the last two digits of the number r one wants to check for squareness. The last decimal digit of a square must be one of the following: 0, 1, 4, 5, 6 or 9. Thus if the number ends in 2, 3, 7 or 8, one already knows that it cannot be a square. Similarly, only 22 two-digit numbers may occur as the last two decimal digits of a square. The reason is that the residue class modulo m of a square must be a square in $\mathbb{Z}_m$ for each integer $m \geq 2$ and, for example, the six digits mentioned above are the squares in $\mathbb{Z}_{10}$ while there are only 22 squares in $\mathbb{Z}_{100}$. There is no need to take 10 or 100 as the modulus and, in fact, there are other moduli which are more advantageous for this purpose. For example, there are only four squares in $\mathbb{Z}_{16}$ so checking the least non-negative residue of r modulo 16 will detect 75 % of non-squares.

In order to apply this idea to speed up Fermat's method one needs to find a modulus m such that the ratio between the number of squares in $\mathbb{Z}_m$ and m is as low as possible (we stress that here we are counting squares in $\mathbb{Z}_m$ and not quadratic residues in $\mathbb{Z}_m^*$). Moreover, this m should not be too large because in that case the computation of the squares would take a long time. Using the Chinese remainder theorem and, in particular, Theorem 2.14, it is easy to see that the function that to each $m \geq 2$ assigns the number of squares in $\mathbb{Z}_m$ is multiplicative, i.e., if $m = uv$ with u, v, relatively prime, then the number of squares in $\mathbb{Z}_m$ equals the product of the number of squares in $\mathbb{Z}_u$ by the number of squares in $\mathbb{Z}_v$. This follows from the isomorphism $\mathbb{Z}_m \to \mathbb{Z}_u \times \mathbb{Z}_v$, in which a square in $\mathbb{Z}_m$ corresponds to a pair formed by a square in $\mathbb{Z}_u$ and a square in $\mathbb{Z}_v$. Thus the computation of the number of squares modulo m, and also the computation of the squares themselves, reduce to the analogous computations modulo the prime powers dividing m.

Exercise 6.16 Use Maple to compute the value of m below a given bound, which has the smallest ratio of squares modulo m to m. Show that these ratios have values 0.038095, 0.020779, 0.011189 for $m = 5040$, $m = 55440$ and $m = 720720$, respectively, and that no other value of m less than one of these has a smaller ratio. (Hint: One may use either brute force or the function `numtheory:-quadres` to compute the squares. A more efficient alternative would be to use the formulas in [187] for the number of squares modulo a prime power and the fact that the number of squares is a multiplicative function.)

The preceding exercise shows that by checking the congruence class of r modulo 55440 we can detect almost 98 % of the tested numbers as being non-squares. Even more will be detected if we take 720720 as modulus at the cost of computing many more modular squares; this would only be advantageous if the number of iterations is very large. So we will take $m = 55440$ for factorizations that require several million iterations such as, for example, that of the number `c2` above. The function that does Fermat factoring with the modified squareness test that checks first whether

the residues are squares modulo 55440 is given below. This function is similar to `FermatFactor` except that before starting the iterations of the method it computes the elements $a \in \mathbb{Z}_{54440}$ such that $a^2 - n$ is a square in $\mathbb{Z}_{54440}$. This is done by using Maple's function `numtheory:-quadres` which computes a sort of "generalized Legendre symbol" that detects squares modulo m. Then one starts at $x = \lceil \sqrt{n} \rceil$ and keeps increasing it by 1 in each iteration, while checking whether $x \bmod 55440$ is one of these pre-computed values and, if this is the case, whether $x^2 - n$ is indeed a square. This way only a small fraction of the values $x^2 - n$ need to be computed and submitted to the `issqr` test.

```
> FermatFactorR := proc(n::(And(posint, odd)), maxnumsteps := 10^8,
                  {numsteps::truefalse := false})
  local x, fact, m, i, y, q;
  x := isqrt(n);
  if n <= x^2 then
    x := x-1
  end if;
  fact := false;
  m := n mod 55440;
  q := Array(0 .. 55439, proc(a) numtheory:-quadres(a^2-m, 55440) end proc);
  for i to maxnumsteps while not fact do
    x := x+1;
    if q[x mod 55440] = 1 then
      fact := issqr(x^2-n)
    end if
  end do;
  if fact then
    y := isqrt(x^2-n)
  else
    error "unable to factor %1 in %2 iterations", n, maxnumsteps
  end if;
  if not numsteps then
    x-y, x+y
  else
    x-y, x+y, i-1
  end if
  end proc:
```

One can readily check that this function is more than three times faster than `FactFermat` for numbers similar to `c2` above.

Exercise 6.17 Write a Maple function to test whether an integer is a square, which uses an array of precomputed data and is faster than `issqr` on large integers. (Hint: The function should use a modulus like 55440 and Maple's `issqr`; it will actually be slower than the latter on numbers that turn out to be squares but it will be much faster on the majority of non-squares, so faster on average.)

Exercise 6.18 Experiment with variants of the procedure `FermatFactorR` based on other moduli (like, for example, 720720) and find numbers for which these variants are faster than `FermatFactorR`.

6.4.5 Factor Bases

As we have seen, Fermat's factorization method only works well when the difference between the factors of n is relatively small because in that case the number of steps will also be small. This is related to the fact that we want to write n as a difference of squares, $n = x^2 - y^2$, but there is another idea, originally due to Legendre, that relaxes this requirement in a way that allows much more flexibility. Legendre realized that if we have a congruence $x^2 \equiv y^2 \pmod{n}$ such that $x \not\equiv \pm y \pmod{n}$, then we also obtain a nontrivial factorization of n. The reason is that in this case n divides $x^2 - y^2 = (x + y) \cdot (x - y)$ but n divides neither $x + y$ nor $x - y$ and hence both $\gcd(x + y, n)$ and $\gcd(x - y, n)$ are proper factors of n. This situation always arises whenever n is an odd composite with at least two different prime factors:

Proposition 6.4 *Let n be an odd positive integer which is composite and is not a prime power. Then there exist $x, y \in \mathbb{Z}$ such that $x^2 \equiv y^2 \pmod{n}$ and $x \not\equiv \pm y \pmod{n}$.*

Proof Since n is not a prime power we can write $n = ab$ with $a, b > 1$ and $\gcd(a, b) = 1$. Let $y \in \mathbb{Z}$ such that $\gcd(y, n) = 1$ and $x \in \mathbb{Z}$ such that $x \equiv y \pmod{a}$ and $x \equiv -y \pmod{b}$ (such an x exists by the Chinese remainder theorem). Then a divides $x - y$ and b divides $x + y$ and, since a and b are relatively prime, their product divides $(x - y)(x + y)$ and so $x^2 \equiv y^2 \pmod{n}$. Suppose now that $x \equiv y \pmod{n}$. Then we also have that $x \equiv y \pmod{b}$ and hence $y \equiv -y \pmod{b}$ and $2y \equiv 0 \pmod{b}$. Since $\gcd(y, b) = 1$, this implies that $b = 2$, which is a contradiction because we are assuming that n, and hence also b, is odd. The case $x \equiv -y \pmod{n}$ may be discarded in a similar manner. □

In the remainder of this section we will assume that n is an odd composite integer which is not a perfect square; the above proposition reduces the factorization of such an n to the problem of finding congruences of the form $x^2 \equiv y^2 \pmod{n}$. The condition $x \not\equiv \pm y \pmod{n}$ shows that this may not be enough to factor n but we shall not worry much about this because, assuming some kind of statistical independence, one expects that at least half of these congruences lead to a nontrivial factorization of n. Indeed, we have seen in Corollary 2.7 that if n is a product of two different odd primes, then any quadratic residue modulo n has exactly four square roots and the same proof (using the Chinese remainder theorem) serves to show that if n has r different prime factors then each square modulo n has 2^r square roots. Thus a random square root of x^2 modulo n (like $y \bmod n$ if $x^2 \equiv y^2 \pmod{n}$) has only a $2/2^r \leq 1/2$ chance of being equal to either x or $-x$ modulo n and so, assuming that x and y have a random behavior, there is at most a probability equal to 2^{-t} that a nontrivial factor of n will not be found after trying t congruences $x^2 \equiv y^2 \pmod{n}$ (and this probability will be even lower if the number of prime factors of n is greater than 2). This is only a heuristic, non-rigorous argument but it seems to work well in practice and so, given a reasonable number of congruences $x^2 \equiv y^2 \pmod{n}$, we expect that there is a high probability that they produce a factorization of n.

The problem that arises now is how to generate these congruences. Here a new important idea—due to Kraitchik who introduced it in the 1920s—comes into play: just generate congruences of the form $u^2 \equiv v \pmod{n}$ such that v is small, and factor these residues v as products of (small) primes. Then search among them for a set of congruences $x_t^2 \equiv z_t \pmod{n}$ such that the product of the z_t is a square (which can easily be detected because the factorizations of the z_t are known) and multiply them to obtain a congruence of the required form $x^2 \equiv y^2 \pmod{n}$, where $x = \prod_t x_t$ and $y = \sqrt{\prod_t z_t}$. The form in which we will generate the required congruences is inspired by Fermat's method. We will consider the polynomial $Q(t) = (t + \lceil\sqrt{n}\rceil)^2 - n$ and compute the values $x_t = t + \lceil\sqrt{n}\rceil$ and $z_t = Q(t) = x_t^2 - n$ for $t = 0, 1, \ldots$; these z_t are just the integers that are submitted to the squareness test in Fermat's method. We illustrate this with a simple example.

Example 6.13 Let us factor $n = 1261$. We compute $\lceil\sqrt{1261}\rceil = 36$ and the values of x_t and z_t for $t = 0, \ldots 5$. Then we have:

t	$x_t = t + 36$	x_t^2	$z_t = x_t^2 - n$	Factorization of z_t
0	36	1296	35	$5 \cdot 7$
1	37	1369	108	$2^2 \cdot 3^3$
2	38	1444	183	$3 \cdot 61$
3	39	1521	260	$2^2 \cdot 5 \cdot 13$
4	40	1600	339	$3 \cdot 113$
5	41	1681	420	$2^2 \cdot 3 \cdot 5 \cdot 7$

We see that none of the z_t is a square since all of them have some odd exponent in their prime factorization. We look for a product of the z_t which is a square, i.e., we look for a product in which all prime factors have even exponents. We see that if we multiply the prime factors in rows 1, 2 and 6 (corresponding to $t = 0, 1, 5$), we obtain $z_0 z_1 z_5 = 2^4 \cdot 3^4 \cdot 5^2 \cdot 7^2$ which is obviously a square with square root equal to $2^2 \cdot 3^2 \cdot 5 \cdot 7 = 1260$. Thus if we set $x = x_0 x_1 x_5 = 36 \cdot 37 \cdot 41 = 54612$ and $y = \sqrt{z_1 z_2 z_6} = 1260$, then $x^2 \equiv y^2 \pmod{1261}$ and $x \not\equiv \pm y \pmod{1261}$. Now we find a proper factor of 1261 by computing $\gcd(x - y, 1261) = \gcd(53352, 1261) = 13$ and we have that $1261 = 13 \cdot 97$. Using `FermatFactor` we can find the number of steps required by Fermat's method on this integer:

```
> FermatFactor(1261, numsteps = true);
                                13, 97, 20
```

We see that Fermat's method required 20 iterations to factor 1261 while Kraitchik's method required only the computation of the first 6 iterations.

Although Kraitchik's method is still far from satisfactory, we are going to see that it is a big improvement on Fermat's method and an important step towards the more powerful modern factoring algorithms. The strategy applied in the preceding example was to chose the z_t as small as possible in order to maximize the probability that they factor as a product of small primes. Then we discarded the z_t with 'large' prime factors, namely, z_2, z_3 and z_4, which were not used to obtain the final congruence. The systematic way to do this is to use a *factor base* consisting of primes less than a

given bound B and factor as many z_t as possible over the factor base. In our example, we may choose as factor base the set of the primes ≤ 7, i.e., $\{2, 3, 5, 7\}$, and we find that z_0, z_1 and z_5 can be completely factored over the factor base and they give the required congruence. But note that, in contrast with what happens in this small example, not all primes $\leq B$ should be members of the factor base in general. This is because the only odd primes that may occur in the factorization of an integer of the form $x_t^2 - n$ are those p for which the congruence $x^2 \equiv n \pmod{p}$ has a solution ($x_t \mod p$ will, in this case, be a solution) and these are precisely the primes p such that n is a quadratic residue modulo p. As we have seen, this property is easy to verify by using the Legendre symbol and so we only include in the factor base the odd primes $\leq B$ that satisfy $\left(\frac{n}{p}\right) = 1$. For example, there would be no point in including 11 in the factor base when trying to factor 1261 because

```
> numtheory:-legendre(1261, 11);
                                    -1
```

and hence 11 cannot appear as a factor of the z_t in this case. But, in addition to 2 and the odd primes modulo which n is a quadratic residue, we may include something else in the factor base, namely, the integer -1. This may seem strange at first sight because we are looking for squares, which are always positive, but it is easy to realize that, when searching for squares, -1 behaves in a similar way to a prime factor because what we need is, again, that its exponent is even. Thus including -1 in the factor base allows us to consider negative values for the z_t. The advantage is that, instead of starting at $\lceil\sqrt{n}\rceil$ and letting the x_t run up through the integers, we may let the x_t run through a sequence of integers centered at $\lceil\sqrt{n}\rceil$ by letting t also run through negative values, ie., $t \in [-M, M]$ for an appropriate positive integer M. This will cause about half of the z_t, namely those corresponding to negative values of t and hence to values of x_t less than $\lceil\sqrt{n}\rceil$, to be negative. These negative values are easily dealt with because we have included -1 in the factor base and, on the other hand, this method has the advantage that the absolute values of the z_t are now somewhat smaller on average and this makes it more likely that these values are completely factored over the factor base. We return to our earlier example to see what happens if we add -1 to the factor base.

Example 6.14 Consider again $n = 1261$ and take as factor base the set $\{-1, 2, 3, 5, 7\}$. We let t run through the interval $[-2,2]$ which will produce two values for the x_t preceding $\sqrt{1261} \approx 35.51$ and three following it. Thus we will obtain two negative values and three positive values for the $z_t = x_t^2 - n$. Only the last of these values does not completely factor over the factor base (as we have seen in the preceding example, $z_2 = 3 \cdot 61$) and the result of factoring the first four z_t is given in the following table:

t	x_t	x_t^2	$z_t = x_t^2 - n$	Factorization of z_t
-2	34	1156	-105	$-1 \cdot 3 \cdot 5 \cdot 7$
-1	35	1225	-36	$-1 \cdot 2^2 \cdot 3^2$
0	36	1296	35	$5 \cdot 7$
1	37	1369	108	$2^2 \cdot 3^3$

Now we only had to compute four x-values to obtain a congruence of squares modulo n. Indeed, it is clear from the factorizations in the rightmost column of the table that $z_{-2}z_{-1}z_0z_1 = 2^4 \cdot 3^6 \cdot 5^2 \cdot 7^2$ is a square. Hence, taking $x = 34 \cdot 35 \cdot 36 \cdot 37 = 1585080$ and $y = 2^2 \cdot 3^3 \cdot 5 \cdot 7 = 3780$ we have that $x^2 \equiv y^2 \pmod{n}$. Then we try $\gcd(x - y, n) = \gcd(1581300, 1261) = 1$, so that in this case we do not obtain a nontrivial factor of 1261 (from this also follows that $\gcd(x + y, n) = n$ does not give a nontrivial factor either). The congruence between squares was found very quickly (suggesting that the inclusion of -1 in the factor base is a good idea) but we had the bad luck that $x \equiv -y \pmod{n}$. Another congruence that works can be obtained if we add the value $x_5 = 41$ which, as we have seen in the previous example, gives $z_5 = 41^2 - n = 420 = 2^2 \cdot 3 \cdot 5 \cdot 7$. Then $z_{-2} \cdot z_{-1} \cdot z_5 = 2^4 \cdot 3^4 \cdot 5^2 \cdot 7^2$ is a square and so we may take $x = 34 \cdot 35 \cdot 41 = 48790$ and $y = 2^2 \cdot 3^2 \cdot 5 \cdot 7 = 1260$ and we obtain a factor as $\gcd(x - y, n) = \gcd(47530, 1261) = 97$.

Example 6.15 Let us now consider another example that can be done by hand (with a little patience), where the advantage of including -1 in the factor base shows up much more clearly. Let $n = 1937$ and consider the factor base that, in addition to 2, includes all the odd primes $p \leq 11$ such that $\left(\frac{1937}{p}\right) = 1$. We see that

```
> numtheory:-legendre ~(1937, [3, 5, 7, 11])
                              [-1, -1, -1, 1]
```

and hence our factor base will be $\{2, 11\}$. It is intuitively clear that the factor base is not of "good quality" because relatively many small odd primes (3, 5, 7) are missing and this will make it more difficult to obtain the factorizations we are looking for; later we will see how to evaluate more precisely the quality of the factor base and how to modify the method to deal with "bad" factor bases. Now, $\sqrt{1937} \approx 44.11$ and if we use only positive values for the x_t we have to let t run on the interval $[0, 42]$, which corresponds to x_t running through the interval $[45, 45 + 42] = [45, 87]$, in order to obtain sufficient fully factored numbers to get a congruence between squares:

t	x_t	x_t^2	$z_t = x_t^2 - n$	Factorization of z_t
0	45	2025	88	$2^3 \cdot 11$
42	87	7569	5632	$2^9 \cdot 11$

Then we see that $z_0z_{42} = 2^{12} \cdot 11^2$ is a square, which gives us $x = 45 \cdot 87 = 3915$ and $y = \sqrt{z_0z_{42}} = 2^6 \cdot 11 = 704$. Then we obtain a factor by computing $\gcd(3915 - 704, 1937) = 13$. For this we had to try to factor several numbers greater than n, up to $z_{42} = 5632$, so that the method required a lot of work and was not helpful in this case. However, if we include -1 and consider the factor base $\{-1, 2, 11\}$, and we let t run on the interval $[-4, 4]$, then we obtain the following factorizations:

t	x_t	x_t^2	$z_t = x_t^2 - n$	Factorization of z_t
-4	41	1681	-256	$-1 \cdot 2^8$
-2	43	1849	-88	$-1 \cdot 2^3 \cdot 11$
0	45	2025	88	$2^3 \cdot 11$

Now we see that $z_{-4}z_{-2}z_0 = 2^{14} \cdot 11^2$ is a square and hence we obtain the required congruence by taking $x = x_{-4}x_{-2}x_0 = 41 \cdot 43 \cdot 45 = 79335$ and $y = \sqrt{z_{-4}z_{-2}z_0} = 2^7 \cdot 11 = 1408$. Now, $\gcd(x - y, n) = \gcd(77927, 1937) = 149$ is a nontrivial factor.

In this example we used an extremely small factor base consisting of only two primes. We leave as an exercise for the reader to show that if we use the factor base $\{2, 11, 13, 17\}$, then it is enough to let t run through the interval $[0, 10]$ to find a congruence that serves to factor n.

The preceding examples use very small numbers and so are not very representative of what might happen when trying to factor large integers. We will later handle larger examples with Maple but now it is worthwhile to consider what we did in these examples in order to find the required congruences. One of the things we had to do was to generate a factor base containing the primes $p \leq B$ such that n is a quadratic residue modulo p. This poses the problem of how to choose B and we will give some guidelines later on.

Once the factor base was ready, we had to generate the auxiliary integers z_t and obtain a number of them that factor as a product of members of the factor base, so that these z_t are, in particular, *B-smooth* in the sense that all their prime factors are $\leq B$. In one case, several of the integers z_t were even greater than the number n being factored but this was an artifact of the small size of n and of the relatively small factor base used in our example. In real life this will not be a problem because the values of t will be small enough to guarantee that the x_t^2 are much smaller than $2n$ and hence that the z_t are much smaller than n. Then the B-smooth z_t can be factored by trial division over the factor base, although we will later see that there is a much better alternative.

The next step was to look for a subset of the B-smooth z_t whose product was a square and this produced a congruence of the required type. This part of the algorithm also poses some problems. One of them is how to recognize how many B-smooth values should be generated in order to have assurance (or, at least, a high probability) that a square may be obtained by multiplying some of these smooth values (actually, several of these squares will be required in order to factor the number with high probability). Another related problem is how to find which of these smooth values we have to multiply in order to get the squares we are looking for. In the previous examples this was done by direct inspection of the factorizations of the smooth values until several of them were found such that the sums of their exponents were all even but it is clear that a much more efficient way is required for the method to work with large numbers.

In order to deal with this problem, suppose that our factor base is $FB = \{p_1, \ldots, p_K\}$, where $p_1 = -1$, $p_2 = 2$, and $p_3, \ldots, p_K$ are odd primes $\leq B$. Each of the factorizations of a B-smooth value z_t is called a *relation*:

$$z_t = \prod_{j=1}^{K} p_j^{e_{tj}},$$

and each relation, where the exponents e_{ij} are non-negative integers, has an associated exponent vector:

$$e(z_t) = (e_{t1}, e_{t2}, \ldots, e_{tK}).$$

To find a set of values z_t whose product is a square, it is clear that we have to find values such that the sum of the corresponding exponent vectors is even, because if $\{t_i | i = 1 \ldots l\}$ is a set of integers in the interval $[-M, M]$ then:

$$\prod_{i=1}^{l} z_{t_i} = \prod_{j=1}^{K} p_j^{\sum_{i=1}^{l} e_{t_i j}},$$

and hence $e(\prod_{i=1}^{l} z_{t_i}) = \sum_{i=1}^{l} (e_{t_i 1}, e_{t_i 2}, \ldots, e_{t_i K})$. This was done by hand in our examples but now it is easy to realize that finding a set of relations such that the sum of their exponent vectors has all even components is the same as finding a set of relations whose exponent vectors sum to the zero vector when regarded as vectors modulo 2. Therefore, once we have generated a number of relations, we may consider the associated vectors over the binary field $\mathbb{F}_2 = \{0, 1\}$:

$$e(z_t) \bmod 2 = (e_{t1} \bmod 2, e_{t2} \bmod 2, \ldots, e_{tK} \bmod 2),$$

and look for a subset of these vectors whose sum modulo 2 is the 0 vector. Thus we are working in the $\mathbb{F}_2$-vector space $\mathbb{F}_2^K$ and we look for a way to write the 0 vector as a nontrivial linear combination of these vectors (which is the same as a sum of a subset of these vectors since the only nonzero coefficient in the field is 1). Such a linear combination is called a *linear dependency* or, simply, a *dependency* and can be found by using standard linear algebra algorithms such as Gaussian elimination (although more efficient algorithms are known for the big sparse matrices that arise in the factorization of large integers). Indeed, if we consider the $r \times K$ matrix M over $\mathbb{F}_2$ whose rows are the exponent vectors corresponding to r relations, then we want to find a linear dependency between the rows of this matrix. These linear dependencies correspond to the null space of the transpose M^t of this matrix, in the usual convention where the null space of M^t consists of the column vectors $v \in \mathbb{F}_2^r$ such that $M^t \cdot v = 0$. We will use Maple's command `MatBasis` from the `LinearAlgebra:-Modular` package to compute a basis of the null space of this matrix and each vector in this basis will give a dependency and hence a congruence of the form $x^2 \equiv y^2 \pmod{n}$. There are, in general, many more dependencies in the null space of M^t—which can be obtained as subset sums of the basis—but it can easily be shown that if one of them gives a nontrivial factor of n then there is already a dependency in the basis that also gives the nontrivial factor, so we will not need these additional dependencies.

We remark that the linear algebra approach also solves another of the problems mentioned above because it provides an upper bound to the number of relations needed to create a congruence between squares modulo n. Since the r rows of M live in the K-dimensional vector space $\mathbb{F}_2^K$, we know for sure that a linear dependency

among them will be found as soon as $r \geq K + 1$. In practice, when K is large, fewer relations will be sufficient to obtain a dependency with high probability.

The preceding discussion gives an outline of the factor base algorithm but there is still an important question that we have not addressed: how the smoothness bound B is to be chosen. If B is chosen small then fewer relations will be needed to obtain a dependency, which is good, but this must be weighed against the fact that a small B makes it more difficult to find B-smooth numbers (and hence relations); in fact, if B is chosen too small, then we may not find any B-smooth numbers. The choice of B ultimately affects the running time of the algorithm in a decisive way and we will examine it in more detail later on, when discussing the quadratic sieve factoring algorithm. For now, we just remark that the optimum value of B in the quadratic sieve is, heuristically, close to $L(n)^{1/2}$, where, as in Definition 2.12:

$$L(n) = e^{\sqrt{(\ln n)(\ln \ln n)}}$$

(see also the discussion in [102, pp. 146–151] or [60, pp. 228–229]). The optimal value of B in the factor base method is, as we shall see, also related to this quantity and, for simplicity, we will adopt a suitable constant multiple of $L(n)^{1/2}$ in the Maple implementation that follows.

6.4.6 The Factor Base Method in Maple

We start with a function that computes the factor base. The input is the number n to be factored, which should be composite and a non-square and, in addition, there is an optional input parameter, `c`, for a constant which takes `1.5` as default value and which, multiplied by the above mentioned term $L(n) = e^{\sqrt{(\ln n)(\ln \ln n)}}$, is used to set the value of the smoothness bound B. The output is a list with the elements of the factor base.

```
> FactorBase := proc(n, c := 1.5)
  local B, bound;
  B := ceil(evalf(c*sqrt(exp(sqrt(ln(n)*ln(ln(n)))))));
  bound := numtheory:-pi(B);
  [-1, 2, op(select(x -> evalb(numtheory:-legendre(n, x) <> -1),
        [seq(ithprime(i), i = 2 .. bound)]))]
  end proc:
```

The next function implements trial division over the factor base for integers of the form $x^2 - n$, where n is the number to be factored. The first two input parameters `n` and `x` are for these integers and the third input parameter `fb` is for the factor base given as a one-dimensional Maple `Array`. The output is either a list consisting of the values `x`, $\texttt{x}^2 - \texttt{n}$ and the exponent vector corresponding to $\texttt{x}^2 - \texttt{n}$, in case the latter is a B-smooth value, or the `NULL` sequence otherwise.

```
> FBTrialDivision := proc(n::posint, x::integer, fb::Array)
  local l, r, e, q, i;
  r := 1;
  q := x^2-n;
  l := ArrayNumElems(fb);
  e := Vector(l, datatype = integer[1]);
```

```
if type(q, negint) then
  e[1] := 1;
  q := -q
end if;
for i from 2 to l while r <> 0 do
  r := modp(q, fb[i]);
  while r = 0 do
    q := iquo(q, fb[i]);
    e[i] := 1 + e[i];
    r := modp(q, fb[i]);
    if q = 1 then
      return [x, x^2-n, e]
    end if
  end do
end do
end proc:
```

Next we give the function that finds the relations corresponding to smooth values factored over the factor base. The required input parameters are `n`, for the integer being factored, and `fb` for the factor base given as a Maple `Array`. There are also two optional keyword parameters. The first of them, called `mindeps`, specifies the minimum number of relations to be found. With the default `mindeps = 5` the function searches for relations until $K + 5$ of them are found, where K is the size of the factor base. This guarantees that the basis of the null space of the relations matrix will contain at least 5 dependencies but, in general, it will contain more and this number can be lowered when the integer being factored is large. The second keyword parameter, `verbose`, specifies the way information is to be printed and has `true` as default value. The output is a list containing three entries. The first is a vector with the x-values, the second a vector with the $x^2 - n$ values (also called the z-values in our previous notation) and the third is the relations matrix, an $r \times K$ matrix whose rows are the exponent vectors corresponding to the r B-smooth values found.

```
> Relations := proc(n::posint, fb::Array,
                    {mindeps::posint:=5, verbose::truefalse:=false})
  local s, np, f, j, g, f1, f2, i;
  s := isqrt(n);
  np := ArrayNumElems(fb);
  f := [];
  j := 1;
  g := np + mindeps;
  while nops(f) < g do
    f1 := FBTrialDivision(n, s-j+1, fb);
    f2 := FBTrialDivision(n, s+j, fb);
    f := [op(f), f1, f2];
    j := j+1
  end do;
  if verbose then
    printf("%d smooth values found out of %d numbers tried:\n", g, 2*j-2)
  else
    print(`Numbers tried to factor: `);
    print(2*j-2)
  end if
  [Vector([seq(f[i][1], i = 1..nops(f))]), Vector([seq(f[i][2], i = 1..nops(f))]),
   LinearAlgebra:-Transpose(Matrix([seq(f[i][3], i = 1..nops(f))]))]
  end proc:
```

The next function searches for a basis of the null space of the matrix obtained by reducing the relations matrix modulo 2. The input is the relations matrix and the output is a list of the dependencies, each of which is given as a one-dimensional array over $\mathbb{F}_2$.

```
> Dependencies := proc(A::Matrix)
  uses LinearAlgebra:-Modular;
  local M, rd, cd, z, rk;
  M := Mod(2, LinearAlgebra:-Transpose(A), integer[1]);
  rd := LinearAlgebra:-RowDimension(M);
  cd := LinearAlgebra:-ColumnDimension(M);
  if rd < cd then
    #(MatBasis requires as many rows as columns; 0-rows are added if needed)
    z := Create(2, cd-rd, cd, integer[1]);
    M := ArrayTools:-Concatenate(1, M, z)
  end if;
  rk := MatBasis(2, M, rd, true);
  [LinearAlgebra:-Row(M, rk+1 .. cd)]
  end proc:
```

The final ingredient we need is a function that computes two nontrivial factors of n if the relations and the dependencies previously assembled allow it. The input is the integer n being factored, the relations in the format provided by the function `Relations` and the dependencies as given by the function `Dependencies`. The output is either two nontrivial factors of n or a message informing that no nontrivial factors have been found.

```
> FindFactors := proc(n, rels, deps)
  local fact, i, x, y;
  fact := 1;
  for i to nops(deps) while fact = 1 or fact = n do
    x := mul(j, j = rels[1]~deps[i]);
    #(use x:= mul(j, j=zip(`^`,rels[1],deps[i])) in Maple versions prior to v.13)
    y := isqrt(mul(j, j = rels[2]~deps[i])); #(use zip in versions prior to v.13)
    fact := igcd(x+y, n)
  end do;
  if fact <> 1 and fact <> n then
    ``(fact)*``(iquo(n, fact))
  else
    print(`No nontrivial factors found`)
  end if
  end proc:
```

Example 6.16 Let us factor the integer:

```
> n := 516378077600327:
```

We start by computing the factor base; we display it as a list and then we convert it to an array, as required by the `Relations` function:

```
> f := FactorBase(n);
  f := Array(f):
[-1, 2, 29, 31, 37, 41, 47, 59, 71, 73, 79, 107, 127, 131, 137, 157, 181, 193, 197,
  199, 223, 227, 229, 239, 241, 257, 277, 283, 307, 311, 313, 317, 331]
```

Observe that, once again, this factor base does not look particularly good as many small odd primes are missing. Since small primes contribute the most to the factorization of the z-values, we expect that relatively many of these values will have to be submitted to trial division in order to find the required number of relations.

The result of computing the relations is given in Table 6.1. The output consists of two vectors and a matrix. The first vector contains the 'x-values' and the second the

Table 6.1 Relations for $n = 516378077600327$

```
> interface(rtablesize = 50):
  r := Relations(n, f, verbose = true): r;

38 smooth values found out of 2451834 numbers tried:
```

22721368	−117513808903	1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1
22730503	297689032682	0 1 1 1 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0
22741149	781780239874	0 1 2 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
22705480	−839255569927	1 0 0 0 2 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1
22742582	846958426397	0 0 0 0 2 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0
22799571	3442360183714	0 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0
22814991	4145736729754	0 1 0 0 1 1 0 0 0 0 0 0 0 0 1 0 0 0 2 0 0 0 0 0 1 0 0 0 0 0 0 0
22631452	−4195457972023	1 0 0 0 0 0 0 0 0 0 0 2 1 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0
22838412	5214985081417	0 0 1 1 2 0 0 0 0 0 0 1 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
22840744	5321508873209	0 0 0 0 0 0 1 0 0 0 0 1 0 2 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
22886291	7404238136354	0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0
22548332	−7950801618103	1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0
22904041	8217016529354	0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0
22929251	9372473820674	0 1 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0
22507873	−9773730616198	1 1 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 2
22944494	10071727315709	0 0 3 2 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
22946139	10147217406994	0 1 0 2 0 0 1 0 0 2 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
22948061	10235426059394	0 1 1 2 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
22966716	11091966224329	0 0 0 0 0 2 0 1 1 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
22984608	11914127313337	0 0 0 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0
23010092	13086256248137	0 0 2 1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0
22414585	−13964456878102	1 1 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0
22297612	−19194576697783	1 0 0 0 0 1 1 1 2 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
23154422	19749180553757	0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0 0 0
23191084	21448299494729	0 0 2 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
23220023	22791390520202	0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0
22076465	−29007770704102	1 1 0 1 0 0 0 1 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
21986689	−32963584417606	1 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0
21955851	−34318684466126	1 1 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0
23592013	40204999791842	0 1 0 0 0 0 2 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
23630879	42040364712314	0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1
23678164	44277372810569	0 0 0 0 1 1 2 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0
23728480	46662685510073	0 0 1 0 0 0 1 1 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0
21718838	−44670153530083	1 0 1 0 1 0 0 0 1 1 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
21677333	−46471311607438	1 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0
23795241	49835416647754	0 1 1 0 1 0 2 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
21523146	−53132263863011	1 0 1 1 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0
23949871	57218243316314	0 1 0 0 0 1 2 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0

'z-values' of the form $x^2 - n$. Each column of the matrix corresponds, in order, to a member of the factor base, and the rows are the exponent vectors of the relations.

For example, the numbers in the first row mean that if $x = 22721368$, then $z = x^2 - n = -117513808903 = (-1)\cdot 29\cdot 181\cdot 239\cdot 283\cdot 331$ and, similarly, the numbers in the ninth row mean that $22838412^2 - n = 5214985081417 = 29\cdot 31\cdot 37^2\cdot 107\cdot 199^2$. As already mentioned, a high number of trials was expected in this case to obtain the required number of 38 B-smooth integers. Now we see that the number of trials was 2451834 which is indeed high taking into account the size of n. Later we will see how to deal with this difficulty but now we continue the process of factoring n.

```
> d := Dependencies(r[3]): d;
 [[0 1 0 1 1 0 0 1 0 0 0 0 1 0 1 1 0 1 1 0 0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 0 0 0],
  [1 0 1 0 0 1 1 0 1 0 0 1 0 1 1 1 0 0 0 1 0 1 1 1 1 1 1 0 0 1 1 0 1 0 0 0 0 0],
  [0 0 1 1 1 1 1 1 1 0 0 0 1 0 0 0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0],
  [1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 1 0 0 0 1 0 0 0],
  [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0],
  [0 1 1 1 0 1 1 1 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 0],
  [0 1 0 1 1 0 0 0 1 1 0 1 0 1 0 0 1 0 0 1 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1]]
```

We have obtained 7 linear dependencies which form a basis of the null space of the relations matrix. The entries with value 1 in each of these dependencies correspond to the rows (i.e., the relations) used to obtain a congruence between squares modulo n. For example, the first of these relations has value 1 entries corresponding to the following rows: 2, 4, 5, 8, 13, 15, 16, 18, 19, 27, 28, 29, 31, 32. The x-values and the z-values corresponding to these rows give a congruence of the form $x^2 \equiv y^2 \pmod{n}$. Now we can search for nontrivial factors of n with the function `FindFactors` using these dependencies and the relations previously produced.

```
> FindFactors(n, r, d);
                          (30778453) (16777259)
```

We have used the list of all the dependencies to factor the number but, as we have noticed before, it is possible that some of these dependencies give a congruence $x^2 \equiv y^2 \pmod{n}$ with $x \equiv \pm y \pmod{n}$ that does not produce a nontrivial factor. In order to appreciate what happens in more detail we may also check the dependencies one by one. For example, if we try with the third dependency we have:

```
> FindFactors(n, r, [d[3]]);
                       No nontrivial factors found
```

This dependency produced no factors but if we use the first one instead we obtain the factorization of n:

```
> FindFactors(n, r, [d[1]]);
                          (30778453) (16777259)
```

When the factor base has, as happens in this case, few small primes, there is a trick that serves to speed up the process considerably. The idea is to use a *multiplier*, i.e., to multiply the integer being factored by a small square-free integer m, so that we will factor mn instead of n. This seems a rather peculiar way to proceed and, as

mentioned in [115, pp. 399–400], it may even look downright stupid at first sight, but notice what happens in our example if we multiply n by 2 and compute the factor base corresponding to $2n$:

```
> f2 := FactorBase(2*n);
  f2 := Array(f2):
[-1, 2, 3, 5, 11, 13, 19, 31, 41, 43, 47, 53, 61, 67, 71, 73, 79, 83, 101, 109,
  127, 137, 139, 149, 163, 173, 179, 193, 199, 211, 223, 239, 241, 251, 257, 269,
  293, 311, 313, 347, 349, 367]
```

We see at a glance that the factor base looks much better now because it contains many more small primes. We shall discuss how to choose the multiplier later but now we shall factor n by factoring $2n$ instead, in order to appreciate the difference.

The relations obtained for $2n$ with the factor base `f2` are displayed in Table 6.2. Notice the dramatic decrease in the number of integers that we had to submit to trial division: just 12826 against 2451834 without using the multiplier. The reason is readily apparent if one inspects the first columns of the matrix and compares them with those in the earlier relations matrix in which the multiplier was not used: now the exponents of the first primes in the factor base are significantly larger making their contribution to the factorization also larger, even taking into account the smaller size of these primes. We may now use these relations to factor n as we did without the multiplier.

Exercise 6.19 Use the relations matrix for $2n$ found in the previous example to compute the linear dependencies and factor n.

Exercise 6.20 In his treatise on number theory published in 1926, M. Kraitchik used the method sketched above (with a different way to obtain the relations) to factor the number 193541963777. Use the previous Maple procedures to factor this number without multiplier and also using the multiplier 2.

In the previous example we analyzed the factorization of an integer by means of the factor base method and we saw the importance of using an appropriate multiplier but we did not give any clues on how to choose this. We do it next.

6.4.6.1 The Multiplier

We want to select the multiplier m in such a way that m is square-free and the contribution of the primes in the factor base to the factorization of the z-values is maximized. In order to do that, we will first estimate the average number of times that each prime in the factor base divides each of these z-values. Note that if we try to factor mn instead of n, then we should add the prime divisors of m to the factor base because a z-value of the form $x^2 - mn$ will be divisible by such a prime p whenever x is. This was already done in our previous code since when computing the factor base we included not only the odd primes p such that $\left(\frac{mn}{p}\right) = 1$ but also those that satisfy $\left(\frac{mn}{p}\right) = 0$, which are just the divisors of m because we are assuming that n

Table 6.2 Relations for $2n = 2 \cdot 516378077600327$

```
> r2 := Relations(2*n, f2, verbose = true): r2;

47 smooth values found out of 12826 numbers tried:
```

32136530	405240246	0 1 4 0 0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
32136576	3361803122	0 1 0 0 0 0 0 1 0 0 1 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
32136441	−5315054173	1 0 0 0 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
32136430	−6022055754	1 1 3 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0
32136638	7346742390	0 1 3 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0
32136667	9210668235	0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
32136727	13067071875	0 0 4 5 1 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
32136817	18851690835	0 0 2 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
32136100	−27231990654	1 1 1 0 2 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0
32136040	−31088319054	1 1 1 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0
32137043	33377583195	0 0 3 1 2 0 1 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
32135998	−33787744650	1 1 4 2 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
32137164	41154762242	0 1 0 0 3 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
32137193	43018718595	0 0 1 1 0 0 0 0 0 0 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
32135758	−49212966090	1 1 1 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
32137328	51695778930	0 1 1 1 0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
32135713	−52105182285	1 0 1 1 0 1 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
32137348	52981272450	0 1 3 2 0 0 0 0 0 0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
32135548	−62709940350	1 1 2 2 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
32137663	73227900915	0 0 2 1 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
32137772	80233923330	0 1 3 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
32138158	105044432310	0 1 3 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
32134777	−112262360925	1 0 1 2 1 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0
32138452	123941755650	0 1 1 2 0 0 0 2 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
32134298	−143047247850	1 1 2 2 0 1 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
32138755	143417749371	0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 2 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
32134279	−144268350813	1 0 2 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0
32134054	−158728725738	1 1 4 0 1 0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
32139298	178320732150	0 1 1 2 1 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0
32133737	−179101615485	1 0 1 1 0 1 1 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
32139323	179927697675	0 0 1 2 1 0 2 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
32133594	−188291843818	1 1 0 0 0 2 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
32139537	193683373715	0 0 0 1 0 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0
32133427	−199024436325	1 0 1 2 1 1 0 0 0 0 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
32140148	232958261250	0 1 3 4 1 2 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
32141128	295953911730	0 1 3 1 0 0 2 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0
32141281	305789120307	0 0 2 0 1 0 0 0 1 0 0 0 0 1 0 2 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
32141450	316652901846	0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
32141527	321602691075	0 0 1 2 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
32131403	−329096452245	1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0
32131312	−334944359310	1 1 1 1 1 0 2 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
32141773	337416382875	0 0 1 3 1 1 0 0 1 2 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
32142007	352458787395	0 0 1 1 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0
32142018	353165911670	0 1 0 1 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
32142142	361137147510	0 1 1 1 0 0 2 0 2 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
32142637	392958113115	0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0
32130112	−412058068110	1 1 3 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

has no prime factors less than or equal to the smoothness bound B (these factors are easily obtained by trial division). Since their behavior is different, we will analyze separately the following three cases: odd primes in the factor base not dividing m, odd primes dividing m, and the prime 2.

Suppose first that p is an odd prime $\leq B$ such that $\left(\frac{mn}{p}\right) = 1$ (so that p does not divide m). In this case the equation $x^2 \equiv mn \pmod{p}$ has, as we have seen, exactly two solutions, and this means that in the sequence $x_0, x_1, \ldots, x_{p-1}$ there are two values, say x_{t_1} and x_{t_2}, such that p divides $x_{t_1}^2 - mn$ and $x_{t_2}^2 - mn$. Then p also divides each of the values $x_{t_1+sp}^2 - mn = (x_{t_1} + sp)^2 - mn$ and $x_{t_2+sp}^2 - mn = (x_{t_2} + sp)^2 - mn$ for all integers s, i.e., p appears as a factor twice for each p numbers in the sequence of z-values. This would give an average of $\frac{2}{p}$ for the exponent of p in each z-value, except that it does not take into account the possibility that these values may be divisible by higher powers of p. But the power p^2 occurs twice for each p^2 z-values being tried and this adds two more p-factors for each p^2 z-values. Similarly, if $i \geq 2$, each p^i occurs twice for each p^i values and so the total contribution of p or, in other words, the average exponent of p is:

$$\frac{2}{p} + \frac{2}{p^2} + \frac{2}{p^3} + \cdots = \frac{2}{p}\left(1 + \sum_{i=1}^{\infty} \frac{1}{p^i}\right) = \frac{2}{p}(1 + \frac{1}{p-1}) = \frac{2}{p-1}.$$

Now suppose that p is an odd prime that divides m. Then p is a factor of $z_t = x_t^2 - mn$ precisely when p divides x_t and, since m is square-free, p^2 does not divide $x_t^2 - mn$. Thus the average exponent of p per z-value is just $\frac{1}{p}$ in this case.

Finally, let us look at the prime 2, whose contribution also depends, as we are going to see, on the value of mn. Notice first that just one-half of the values $z_t = x_t^2 - mn$ are even. Assuming that z_t is even, we distinguish two cases: mn odd—in which case x_t is also odd and hence $x_t^2 \equiv 1 \pmod{8}$—and mn even. In the first of these cases we have three subcases. The first is that $mn \equiv 1 \pmod{8}$ and then, since $x_t^2 \equiv 1 \pmod{8}$ too, we have that $z_t \equiv 0 \pmod{8}$. Moreover, we shall see that this is the only situation in which $z_t \equiv 0 \pmod{8}$. Indeed, the second subcase is when $mn \equiv 5 \pmod{8}$ and then $z_t \equiv 0 \pmod{4}$ but $z_t \not\equiv 0 \pmod{8}$. These two subcases exhaust the possibility that $mn \equiv 1 \pmod{4}$ and hence the remaining subcase will be $mn \equiv 3 \pmod{4}$ which implies that $z_t \equiv 2 \pmod{4}$, so that z_t is even (as assumed by hypothesis) but not divisible by 4.

Let us now compute the average exponent of 2 in the z-values for each of these three subcases. In the first, namely, when $mn \equiv 1 \pmod{8}$, we have 2^3 as a factor in one of each two z-values (in the even one) which gives a contribution to the average exponent of $\frac{3}{2}$. But there will be also some higher powers of 2 as factors in this case, so there will be one more factor 2 for each 2^i z-values, for $i \geq 2$, and hence the average exponent of 2 will be:

$$\frac{3}{2}+\frac{1}{2^2}+\frac{1}{2^3}+\cdots=\frac{3}{2}+\sum_{i=2}^{\infty}\frac{1}{2^i}=2.$$

Next, if $mn \equiv 5 \pmod 8$, there is a factor 2^2 for each two z-values (and no higher power factors) so we obtain an average exponent equal to 1. Finally, if $mn \equiv 3 \pmod 4$ then we have that the even z-values are not divisible by 4 and hence we have one factor 2 in one of each two z-values, giving an average exponent of $\frac{1}{2}$.

It remains to evaluate the case when mn is even, which only happens when m itself is even, as we are assuming throughout that n is odd. In this case we have $mn \equiv 2 \pmod 4$ since m is square-free and hence the even z-values are divisible by 2 but not by 4. Since half of the z-values are even, the expected average exponent is, again, $\frac{1}{2}$ in this case.

Summing up the previous discussion we obtain the following function that gives the expected average exponent for each prime p in the factor base (cf. [160]):

$$g(p,m,n)=\begin{cases}\frac{2}{p-1} & \text{if } p \text{ odd and } \left(\frac{mn}{p}\right)=1,\\ \frac{1}{p} & \text{if } p \text{ odd and } p|m,\\ 2 & \text{if } p=2 \text{ and } mn\equiv 1 \pmod 8,\\ 1 & \text{if } p=2 \text{ and } mn\equiv 5 \pmod 8,\\ \frac{1}{2} & \text{if } p=2 \text{ and } mn\not\equiv 1,5 \pmod 8.\end{cases}$$

Now, let $FB_m=\{p_1,\ldots,p_K\}$, where $p_1=-1$, $p_2=2$, etc, be the factor base corresponding to mn, when m is one of the possible values of the multiplier. Then each z_t (corresponding to mn) factors in the form

$$z_t=(\pm 1)\cdot\prod_{j=2}^{K}p_j^{e_{tj}}\cdot q,$$

where q is the product of the prime factors of z_t which do not belong to the factor base. If we denote by $FB_m(z_t)$ the factor $\prod_{j=2}^{K}p_j^{e_{tj}}$, then the most favorable situation is when z_t is B-smooth, i.e., $FB_m(z_t)=\pm z_t$ and $q=1$. Thus, when choosing the multiplier, we try to maximize the ratio between the factor $FB_m(z_t)$ and the average value of z_t itself (maximizing $FB_m(z_t)$ is not enough because, when using a multiplier, the size of the z-values grows relative to the size of the z-values corresponding to n and this growth must be taken into account). The average value of $FB_m(z_t)$ is obtained by raising the primes in the factor base to the average exponents given by the previous formula and multiplying these factors. Since a rough approximation to z_t is given by $t^2+2t\sqrt{mn}$ and, for large integers n, t is small compared to $2\sqrt{mn}$, we see that the size of the z-values corresponding to mn are larger by a factor of $\sqrt{m}$ than those corresponding to n and hence instead of maximizing the expected average value of $FB_m(z_t)/z_t$ we may just choose the m that maximizes $FB_m(z_t)/\sqrt{m}$ or,

equivalently, that maximizes the logarithm of this quantity. Thus we want to choose the value of m that maximizes the function:

$$f(m,n) = -\frac{1}{2}\ln(m) + \sum_{-1 \neq p \in FB_m} g(p,m,n) \cdot \ln(p).$$

Next we give a Maple implementation of the function g defined above:

```
> g := proc(p, m, n)
  if p = 2 then
    if m*n mod 8 = 1 then
      2.
    elif m*n mod 8 = 5 then
      1.
    else
      .5
    end if
  else
    if m mod p <> 0 then
      2./(p-1)
    else
      1./p
    end if
  end if
  end proc:
```

The next function selects the multiplier. It takes as input the integer n to be factored and has two optional keyword parameters. The first, `c`, specifies the constant to use when choosing the factor base (with default `c := 1.5`) and the second, `verbose`, specifies whether information about the possible multipliers is printed on screen (with `verbose := false` as default). The output is a list $[m, fb]$, where m is the selected multiplier and fb is the factor base corresponding to mn. The value of m is selected as the square-free positive integer ≤ 23 that maximizes the value of the function f defined above (called '`f-value`' in the messages printed by the function).

```
> MultSelect := proc(n, {c := 1.5, verbose::truefalse := false})
  local mults, i, mn, fbi, f, h, fb, m;
  mults := {1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15, 17, 19, 21, 22, 23};
  h := 0;
  for i in mults do
    mn := i*n;
    fbi := FactorBase(mn, c);
    f := evalf((-1)*.5*ln(i)+add(g(pr, i, n)*ln(pr), pr = fbi[2 .. -1]));
    if verbose then
      printf("Multiplier: %2d, f-value: %f\n", i, f)
    end if;
    if h < f then
      h := f;
      fb := fbi;
      m := i
    end if
  end do;
  [m, fb]
  end proc:
```

Now we have all the ingredients needed to implement the factor base factorization method. We put all of them together in the next Maple function.

```
> FBFactor := proc(n::And(posint,odd), mult::nonnegint:=0,
                   {mindeps::posint:=5, c:=1.5})
  local mfb, m, x, fb, nfb, r, M, d;
  if isprime(n) then
    return ``(n)
  elif issqr(n) then
    return ``(isqrt(n))*``(isqrt(n))
  elif n < 1000000 then
    return ifactor(n)
  end if;
  if mult = 0 then
    mfb := MultSelect(n, ':-c' = c)
  else
    mfb := [mult, FactorBase(mult*n, c)]
  end if;
  m := mfb[1];
  if m > 1 then
    print(`Using multiplier: `);
    print(m)
  end if;
  x := m*n
  print(`Using smoothness bound: `);
  print(ceil(evalf(c*sqrt(exp(sqrt(ln(x)*ln(ln(x))))))));
  fb := Array(mfb[2], datatype = integer[4]);
  nfb := ArrayNumElems(fb);
  print(`Size of factor base: `);
  print(nfb);
  r := Relations(x, fb, ':-mindeps' = mindeps);
  M := r[3];
  print(`Smooth values found: `);
  print(nfb + mindeps);
  print(`Solving a matrix of size: `);
  print(LinearAlgebra:-Dimension(M));
  d := Dependencies(M);
  print(`Number of linear dependencies found: `);
  print(nops(d));
  print(`Factors: `);
  FindFactors(n, r, d)
  end proc:
```

There is one required input parameter, `n`, for the integer to be factored, and three optional parameters. The first of them is an optional ordered parameter `mult` which specifies the multiplier to be used (in order to allow experimentation) and takes, by default, the optimal value selected by the function `MultSelect`. The remaining optional parameters are two keyword parameters: `mindeps` specifies the minimum number of linear dependencies to be searched for (with default value 5) and `c` specifies the constant to be used for computing the smoothness bound B (with default value 1.5). The output is the number itself if n is prime (checked with `isprime`) and, in case n is composite, either two nontrivial factors of n or a message informing that no nontrivial factors have been found. In order to exclude cases that are too small for the smoothness bound to be reliably computed, we use Maple's `ifactor` for numbers less than 10^6 (of course, the previously defined function `TrialDivisionFactor` could have been used instead).

Example 6.17 Let us go back to the number factored in Example 6.16 and look at the result of evaluating the possible multipliers:

```
> n := 516378077600327:
  MultSelect(n, verbose = true);
Multiplier:  1, f-value: 3.003686
Multiplier:  2, f-value: 6.066766
Multiplier:  3, f-value: 5.551260
Multiplier:  5, f-value: 5.479701
Multiplier:  6, f-value: 3.930504
Multiplier:  7, f-value: 6.235948
Multiplier: 10, f-value: 4.371940
Multiplier: 11, f-value: 4.268430
Multiplier: 13, f-value: 3.754059
Multiplier: 14, f-value: 3.575559
Multiplier: 15, f-value: 3.866392
Multiplier: 17, f-value: 4.323051
Multiplier: 19, f-value: 4.230414
Multiplier: 21, f-value: 3.580619
Multiplier: 22, f-value: 2.993790
Multiplier: 23, f-value: 4.874019

[7, [-1, 2, 5, 7, 11, 13, 17, 23, 29, 31, 37, 43, 47, 59, 61, 67, 89, 97, 101, 131,
137, 151, 163, 173, 179, 191, 193, 197, 199, 211, 223, 227, 263, 269, 277, 283, 293,
307, 311, 317, 347, 349, 353, 359, 367, 373, 383, 389, 397, 401, 409, 419, 421]]
```

We see that the best multiplier, and hence the one that this function selects, is $m = 7$, although the multiplier 2, which we used in our earlier computation of the relations for this number, is a close second. Note also that not using a multiplier (or, equivalently, using multiplier 1) gives the worst result except for the case $m = 22$. Now the function `FBFactor` will do all the work including, of course, the selection of the multiplier. The result of applying it to this number is:

```
> FBFactor(n);
                              Using multiplier:
                                       7
                            Using smoothness bound:
                                      432
                              Size of factor base:
                                      53
                            Numbers tried to factor:
                                     7446
                              Smooth values found:
                                      58
                            Solving a matrix of size:
                                    58, 53
                      Number of linear dependencies found:
                                       9
                                   Factors:
                              (16777259) (30778453)
```

Example 6.18 We give an even more extreme example of the benefits of using a multiplier. Consider the integer $n2 = 5541000035610947$ and apply the `MultSelect` function to it:

```
> n2 := 5541000035610947:
  MultSelect(n2, verbose = true);
Multiplier:  1, f-value: 1.573015
Multiplier:  2, f-value: 5.714888
Multiplier:  3, f-value: 6.147232
Multiplier:  5, f-value: 5.501545
Multiplier:  6, f-value: 4.644188
Multiplier:  7, f-value: 5.113436
Multiplier: 10, f-value: 4.215535
Multiplier: 11, f-value: 5.706725
```

```
Multiplier: 13, f-value: 4.474718
Multiplier: 14, f-value: 4.740725
Multiplier: 15, f-value: 4.187766
Multiplier: 17, f-value: 5.722923
Multiplier: 19, f-value: 5.106755
Multiplier: 21, f-value: 3.697481
Multiplier: 22, f-value: 3.995631
Multiplier: 23, f-value: 4.099556

[3, [-1, 2, 3, 5, 7, 17, 19, 29, 31, 41, 43, 53, 67, 79, 89, 101, 103, 109, 113,
 127, 157, 163, 181, 199, 229, 239, 257, 269, 271, 281, 283, 317, 337, 347, 353,
 367, 373, 379, 389, 397, 409, 433, 439, 443, 457, 487, 491, 503]]
```

Here the multiplier 1 gives the worst result *by far* and the reason stands out clearly if we compute the factor base corresponding to $n2$ without using a multiplier:

```
> FactorBase(n2);
[-1, 2, 109, 137, 139, 149, 151, 157, 173, 181, 197, 211, 223, 229, 233, 239, 293,
  307, 331, 337, 347, 373, 397, 401, 409, 433, 443, 449]
```

Already at first sight we notice that this factor base leaves a lot to be desired. It is small, with only 28 elements, in contrast with the one corresponding to the multiplier 3 above, which has 48 elements. Even worse, the smallest odd prime in the factor base is 109. On top of all this, we have that $n2 \equiv 3 \pmod 4$, which makes the contribution of the prime 2 as small as possible, as we saw when discussing the choice of multiplier. These inconveniences are reflected in the low f-value for the multiplier 1 and make it difficult to factor $n2$ without using a multiplier; we would probably have to choose a smoothness bound larger than the usual one and it would take a lot of time to factor the number. Of course, you are not likely to come across a number like this one just by chance...

Let us now try `FBFactor` on $n2$:

```
> FBFactor(n2);
                              Using multiplier:
                                       3
                            Using smoothness bound:
                                      503
                              Size of factor base:
                                      48
                            Numbers tried to factor:
                                     25704
                              Smooth values found:
                                      53
                           Solving a matrix of size:
                                    53, 48
                     Number of linear dependencies found:
                                       8
                                    Factors:
                          (131564099) (42116353)
```

We see that `FBFactor` factors $n2$ quickly, the reason being that it uses the multiplier 3, whose f-value is almost four times as big as that of the multiplier 1.

Exercise 6.21 Construct a product of two 10-digit primes such that the multiplier 23 has a higher f-value than all the preceding multipliers, and factor it using `FBFactor`.

The factor base method is a clever combination of two of the factoring methods that we have discussed, Fermat's method and trial division. While the factor base method can factor integers that are beyond the reach of these two methods, its main interest lies in the fact that the idea of using a factor base can be further refined and underlies the more powerful factoring algorithms currently available.

Example 6.19 We may see that Mersenne's number M_{67} is easily factorable by this method (but, as we have seen, this number is also within reach of trial division). In fact, substantially larger numbers can also be factored with `FBFactor`. Consider, for example, the 27-digit number 672333687808859742277210981, which is beyond the reach of our previous implementations of trial division and Fermat's method (although quickly factorable by the rho method). We can easily factor this number with `FBFactor`, and in order to give an idea of the parameters involved we print the output of `FBFactor`, which looks like this:

```
> FBFactor(672333687808859742277210981);
                              Using multiplier:
                                    21
                            Using smoothness bound:
                                  5582
                              Size of factor base:
                                   361
                            Numbers tried to factor:
                                 987418
                              Smooth values found:
                                   366
                          Solving a matrix of size:
                                366, 361
                    Number of linear dependencies found:
                                   25
                                Factors:
                  (36335009715479) (18503743168739)
```

Exercise 6.22 Make an experimental study of the time required by the function `FBFactor` to factor numbers of several sizes between 15 and 30 decimal digits. Determine the value of the constant `c` in the input of `FBFactor` that gives the best results in this range.

6.4.7 The Quadratic Sieve

The factor base algorithm outlined above is a nice extension of Fermat's method but is still inefficient because trial division, which runs in exponential time, lies at its heart. Thus the crucial idea to obtain a fast factorization method is to get rid of trial division while preserving the basic procedure of generating many B-smooth numbers. These numbers have to be factored to find the relations but, since they are the values of a (quadratic) polynomial, there is an alternative. We have, on one side, a bunch of numbers from which we want to extract smooth values and factor them over the factor base. On the other, we have the factor base primes. In the previous algorithm we considered the polynomial values individually and

tried to factor each of them by using all the primes in the factor base. But, instead of doing this, we may consider all the polynomial values corresponding to an interval together and find which ones are divisible by a given prime in the factor base.

Following an idea of C. Pomerance, inspired by an earlier algorithm due to Schroeppel, this can be accomplished by means of a sieve similar to the Eratosthenes sieve we used to generate small primes. The reason this is possible is the fact that the numbers where we try to find the smooth values are a sequence of values of a polynomial and in this sequence the locations of the multiples of a prime p follow a regular pattern, exactly as happens in the sieve of Eratosthenes. Consider the polynomial $Q(t) = (t + b)^2 - n$, where $b = \lceil \sqrt{n} \rceil$, that we used to generate the z-values $z_t = Q(t)$. Then $Q(a) \equiv 0 \pmod{p}$ if and only if $Q(a + sp) \equiv 0 \pmod{p}$ for every integer s. Thus the idea is to fix an interval $[-M, M]$ and, for each prime p in the factor base, divide $Q(t)$ by the highest possible power of p (without trying to divide those $Q(t)$ that are not multiples of p) until all the B-smooth values have been factored. But how do we determine the $t \in [-M, M]$ such that p divides $Q(t)$? We first determine the integers $t \in [0, p - 1]$ such that $Q(t)$ is divisible by p, and this is easy to do for we only have to solve the equation $Q(t) \equiv 0 \pmod{p}$, which we know has exactly two roots in $\mathbb{Z}_p$ if n is a quadratic residue modulo p. These roots are quickly found by using Algorithm 2.8, and we observe that there is only one root if we are using a multiplier m and p divides m (as the only square root of 0 modulo p is 0 itself) but we shall explain later how to proceed with these primes, so for now we will just assume that there is no multiplier. Once we know the two zeros of this polynomial, say $a, b \in \mathbb{Z}_p = [0, p - 1]$, the other values of the polynomial $Q(t)$ that are divisible by p are found from these zeros by adding multiples of p. Thus one possible way to proceed is as follows: We compute the zeros a, b and, starting at them, walk through $[-M, M]$ in both directions in steps of length p. At each point $a + sp$ we divide $Q(a + sp)$ by p and we record the factor p (or we keep a running product and multiply by p). We may also sieve with higher powers of p or we can simply divide out the highest power of p at each one of these values. After we do this with all the primes, the values that have been completely divided out are the smooth values and we know their factorizations. The important thing to observe is that there is no trial division in this process as the only divisions are performed when we already know that the remainder will be 0.

Now, the number of steps taken by the sieve for each $t \in [-M, M]$ is, like in the sieve of Eratosthenes, $O(\ln \ln B)$ (see the discussion in [60, 3.2.6]) and this is much less than the $O(B/\ln B)$ steps required for trial division—taking into account the number of primes $\leq B$ given by the Prime Number Theorem—per t-value. Thus the use of the sieve instead of trial division saves much time.

In practice, the sieve will be done a bit differently. In order to speed it up, the arithmetic will be done with logarithms. Exact arithmetic with logarithms involves infinite precision but we do not aim for exactness and will be satisfied with an approximate result. For this, integer approximations for the base 2 logarithms of the

factor base primes are used and divisions of the polynomial values by the primes in the factor base are replaced by subtractions of these logarithms, which is a much more efficient method for two reasons: subtraction is much faster than division and the numbers involved are much smaller. Therefore, in order to do the sieving, a $[-M, M]$ array is initialized and, for each index $t \in [-M, M]$, the integer approximation of the binary logarithm of $Q(t)$ is stored (in practice, the sieving interval $[-M, M]$ may be divided into smaller subintervals to save storage space). Then, during the sieve, $\texttt{round}(\log_2(p))$ is subtracted from the array entries for the values of t such that p divides $Q(t)$. If the sieve were an exact process then, once sieving is finished, the smooth values would correspond to the t-values for which the corresponding array entry is 0. But this is not the case because approximations for the logarithms are used and, moreover, small primes in the factor base will not be used for sieving since they are the most time-consuming while their logarithms contribute little to the sum. Higher powers of factor base primes will not be used for sieving either, and to compensate for all these inaccuracies a certain threshold T will be set and the array scanned for values below this threshold. The indices t corresponding to these values will have a high probability that $Q(t)$ is B-smooth and these will be submitted to trial division. Thus we do not get rid of trial division completely but many fewer numbers will have to be tried. The final part of the algorithm proceeds exactly as in the factor base method we discussed. Because the core of this algorithm consists of sieving the values of the quadratic polynomial $Q(t)$, the algorithm receives the generic name of *quadratic sieve* or, for short, QS.

6.4.7.1 The Complexity of the Quadratic Sieve

Let us now look more closely at the parameters involved in the algorithm in order to give an idea of its complexity and also to have some guidelines for its implementation. The starting point for this is a result of Canfield–Erdös–Pomerance according to which, if $\psi(x, y)$ denotes the number of y-smooth positive integers less than or equal to x, and $u = \ln x / \ln y$, then

$$\psi(x, y) = \psi(x, x^{1/u}) = xu^{-u(1+o(1))}$$

as $u \to \infty$ and $u < (1 - \varepsilon) \ln x / \ln \ln x$ (for any fixed $\varepsilon > 0$, see the discussions in [60, pp. 48–49, 264–265] and [194, Sect. 4.4]). This suggests that the "probability" that an integer of the form $x^2 \mod n$ is B-smooth is about u^{-u}, with $u = \ln n / \ln B$, but this is only a heuristic conjecture as the above mentioned result applies to all integers up to a certain bound and not to a special subset. However, experience has shown that, in practice, the method works pretty much as predicted by this conjecture. On the other hand, if we consider the z-values $Q(t)$ which we have to factor in order to obtain the B-smooth numbers required to factor n, we see that they are of the form $x_t^2 - n$ and, when n is large, much smaller than n and, in fact, much closer to $\sqrt{n}$. Then, if we assume

that $x_t^2 - n$ is of order of magnitude $n^{1/2+\varepsilon}$ where $\varepsilon > 0$ is small, we can revise the previous heuristic estimate replacing n by $n^{1/2}$ in the value of u, leading to $u \approx \ln n/2 \ln B$.

If K is the size of the factor base, then we have to find about K relations and, heuristically, n will be a quadratic residue modulo about half of the primes $\leq B$, so that by the Prime Number Theorem $K \approx \pi(B)/2 \approx B/2 \ln B$. On the other hand, the complexity of the sieve is similar to that of the Eratosthenes sieve, namely, $\ln \ln B$ per sieved value. This leads to the following estimate of the amount of work for the sieving stage of the algorithm: we expect to try u^u values to get one B-smooth value and we require about K smooth values, so the total sieving work will be $W(B) \approx Ku^u(\ln \ln B) \approx Bu^u(\ln \ln B)/2 \ln B$. This expression is dominated by the factor Bu^u which is much larger than $(\ln \ln B)/2 \ln B$ so we will try to find the value of B (as a function of n) that minimizes the term Bu^u. Defining, as before, $L(x) = \exp(\sqrt{(\ln x)(\ln \ln x)})$ and $a = \ln B/\ln L(n)$, so that $B = L(n)^a$, we express Bu^u as a function of n and a. Since $\ln B = a \ln L(n) = a\sqrt{(\ln n)(\ln \ln n)}$ we have:

$$u = \frac{\ln n}{2 \ln B} = \frac{\ln n}{2a\sqrt{(\ln n)(\ln \ln n)}} = \frac{1}{2a}\sqrt{\frac{\ln n}{\ln \ln n}}.$$

Then, taking logarithms we obtain $\ln u = \frac{1}{2}(\ln \ln n) - \frac{1}{2}(\ln \ln \ln n) - \ln 2a \approx \frac{1}{2}(\ln \ln n)$ since the other terms are much smaller; hence $u \ln u \approx \frac{1}{2a}\sqrt{\frac{\ln n}{\ln \ln n}} \cdot \frac{1}{2}(\ln \ln n) = \frac{1}{4a} \ln L(n)$. Therefore, $u^u = e^{u \ln u} = L(n)^{1/4a}$ and hence $Bu^u \approx L(n)^a \cdot L(n)^{1/4a} = L(n)^{a+\frac{1}{4a}}$. Thus we have to minimize the exponent $a + \frac{1}{4a}$ and by the standard calculus procedure we see that in the interval $(0, 1)$ the minimum value occurs at $a = \frac{1}{2}$, which corresponds to $B = L(n)^{1/2}$.

From this we conclude that we should choose the smoothness bound B equal to about $L(n)^{1/2}$ and that, since in this case we also have $u^u \approx L(n)^{1/2}$, the amount of work for the sieving is $W(B) \approx L(n)$. This leads to a running time estimate of $O(L(n) \cdot \text{polylog}(n))$ which, since $\text{polylog}(n) = O(\ln^c n) = O(e^{c \ln \ln n})$ for a constant c, and $e^{c \ln \ln n} = L(n)^{c\sqrt{\frac{\ln \ln n}{\ln n}}}$, where $c\sqrt{\frac{\ln \ln n}{\ln n}} = o(1)$ (so that $\text{polylog}(n) = L(n)^{o(1)}$), gives that the complexity of the sieving step of the algorithm is $L(n)^{1+o(1)}$. On the other hand, this also gives an estimate for the size of the interval $[-M, M]$ to be sieved. It should contain about Ku^u values which, with $B \approx u^u \approx L(n)^{1/2}$ and $K \approx B/2 \ln B$ gives $Ku^u \approx L(n)/\ln L(n)$. Thus we should take $M \approx L(n)/2 \ln L(n)$.

To estimate the complexity of QS we have to take into account other steps of the algorithm which were not considered in the preceding argument. First of all, we have to compute the roots of $Q(t) \equiv 0 \pmod{p}$ for each prime p in the factor base, which can be done in time polynomial in the size of p (and hence in the size of n) using Algorithm 2.8. Since there are fewer than $B \approx L(n)^{1/2}$ primes in the factor base and the complexity of Algorithm 2.8 is, as seen above, $L(n)^{o(1)}$, this computation has complexity $L(n)^{1/2+o(1)}$, a term swallowed by the complexity of the sieving step,

so the overall complexity of the algorithm including the computation of the roots remains $L(n)^{1+o(1)}$.

The other major component of QS is the linear algebra step, where the null space of a matrix of approximate size $K \times K$ over $\mathbb{F}_2$ is computed. One alternative is to use Gaussian elimination, whose complexity is $O(K^3)$ (see [60, Sect. 6.1.3] for a discussion of this and references). As we have seen $K = O(L(n)^{1/2})$, giving a complexity of $O(L(n)^{3/2})$ for this step, which is higher than $L(n)^{1+o(1)}$ and raises the overall complexity of QS. As discussed in [60], this is not a problem in practice for numbers which are not too large but, for really large factorizations, there are other methods that perform the linear algebra step without increasing the above estimate for the complexity of QS. These methods take advantage of the fact that the matrix obtained is very sparse, which allows the use of special encoding methods that also reduce the amount of storage required. In particular, the *block Lanczos method* and the *Wiedemann method* run in time $O(KN)$ where N is the number of nonzero entries in the matrix. Clearly, this matrix has at most $O(\ln n)$ nonzero entries per row, for a total of $O(K \ln n)$ nonzero entries and hence a running time for these methods of $O(K^2 \ln n)$. Since $K^2 = O(L(n))$ we have that $O(K^2 \cdot \text{polylog}(n)) = L(n)^{1+o(1)}$ and hence, with these methods, the overall complexity of QS is still $L(n)^{1+o(1)}$, which is subexponential according to Definition 2.13. We emphasize again that this is only a heuristic estimate although, as already mentioned, the running times obtained in practice agree.

Exercise 6.23 Show that if instead of the sieve we use trial division to find the relations, which means that the work per z-value is $(B/\ln B) \cdot \text{polylog}(n)$ (instead of $\ln \ln B$) and we use the above argument, we obtain that, after minimizing $B^2 u^u$, the optimal value of B is about $B = L(n)^{1/2\sqrt{2}}$ and the complexity of the Factor Base algorithm is then $L(n)^{\sqrt{2}} \cdot \text{polylog}(n) = L(n)^{\sqrt{2}+o(1)}$.

Exercise 6.24 Modify the previous implementation of the factor base method so that it uses the smoothness bound $B = c \cdot L(n)^{1/2\sqrt{2}}$ for a suitable constant c, as suggested by the previous exercise. Make an experimental study of running times for integers of several sizes in the $2^{15}..2^{30}$ range, in order to determine the optimal value for c.

After the preceding discussion, we are now ready to give a schematic description of the QS algorithm. We stress that this is a basic scheme and many improvements are possible:

Algorithm 6.5. Quadratic sieve.

Input: An odd composite integer n which is not a perfect power.
Output: Either a nontrivial factorization of n or a message informing that no nontrivial factors were found.

`Initialization`:
Select a multiplier m and for $B = \lceil c \cdot L(mn)^{1/2} \rceil$, where c is a small constant, compute $FB := \{-1, 2\} \cup \{p | p \text{ odd prime}, p \leq B, \left(\frac{mn}{p}\right) \neq -1\}$. Let $SP = \{p \in FB | \left(\frac{mn}{p}\right) = 1\}$ (excluding perhaps some small primes) and, for each $p \in SP$ compute the values $t_{ip} \in \mathbb{Z}_p$, $i = 1, 2$ such that $Q(t_{ip}) \equiv 0 \pmod{p}$, where $Q(t) = (t + b)^2 - mn$, with $b = \lceil \sqrt{mn} \rceil$. Initialize a one-dimensional $[-M, M]$ array SA where $SA[t] = \lfloor \log_2(Q(t)) \rfloor$.

`Sieving`:
Sieve the array SA by subtracting $\texttt{round}(\log_2(p))$, for each $p \in SP$, from the values $SA[t_{ip} + sp]$ such that s is an integer and $t_{ip} + sp \in [-M, M]$.

`Collecting Relations`:
Find $K + h$ B-smooth numbers, where $K = |FB|$ and h is a small integer, by trial dividing the values $Q(t)$ by the primes in FB for those $t \in [-M, M]$ such that the value of $SA[t]$ falls below a certain threshold T. This way values $t_j \in [-M, M]$, $j = 1, \ldots, K + h$, are found such that $Q(t_j) = x_j^2 - mn$ (with $x_j = t_j + b$) is B-smooth and has an associated modulo 2 exponent vector $e_j = (e_{j1}, e_{j2}, \ldots e_{jK}) \in \mathbb{F}_2^K$. The exponent vectors $e_1, e_2, \ldots, e_{K+h}$ are the rows of a $(K + h) \times K$ matrix over $\mathbb{F}_2$ called the relations matrix.

`Linear Algebra`:
Compute the dependencies, i.e., a basis $\{d_1, d_2, \ldots, d_j\}$ of the null space of the transpose of the relations matrix. If $d = (u_1, u_2, \ldots, u_{K+h}) \in \mathbb{F}_2^{K+h}$ is one of these dependencies then the sum modulo 2 of the rows of the relations matrix corresponding to the indices j such that $u_j = 1$ is equal to 0.

`Factorization`:
Each of the dependencies obtained in the Linear Algebra step is used to get a congruence between squares modulo mn. If $d = (u_1, u_2, \ldots, u_{K+h}) \in \mathbb{F}_2^{K+h}$ is a dependency with the 1-components in positions $j_1, j_2, \ldots, j_q$, then compute $x = x_{j_1} x_{j_2} \cdots x_{j_q}$ and $y = \sqrt{(x_{j_1}^2 - mn)(x_{j_2}^2 - mn) \cdots (x_{j_q}^2 - mn)}$ (y may also be computed from the prime factorizations of the $x_{j_i}^2 - mn$, given by the corresponding relations) and obtain a factor of n by computing $g = \gcd(x - y, n)$. If g is a nontrivial factor then return g else try with another relation. If none of the relations produces a nontrivial factor print an informative message.

6.4.8 The Basic QS Algorithm in Maple

We are now ready to implement in Maple the basic QS algorithm described above. We will take advantage of the previous implementation of the factor base method, some of whose functions will be used without modification. When selecting the value of parameters like the smoothness constant B we will follow the guidelines explained in the discussion of the complexity of the algorithm. In particular, B will be of the form $c \cdot L(n)^{1/2}$ but we point out that often somewhat lower values are

used. For this and other related issues see, e.g., the discussion following Algorithm 6.1.1 in [60], where the authors say: *"The optimal B-value is more of an art than a science, and is perhaps best left to experimentation"*. We also remark that, from version 12 onwards, Maple has an excellent implementation of the quadratic sieve (the multiple polynomial version that we will mention later) which constitutes the core of the `ifactor` function and is used to factor the most difficult numbers. Maple's implementation is very fast, among other things because the most critical parts are written in C, but this makes it difficult to learn much by looking at the Maple code. A description of this implementation can be found in [40].

In the implementation that follows we will use several of the functions previously defined for the factor base method, namely, `FactorBase`, `g`, `MultSelect`, `FBTrialDivision`, `Dependencies` and `FindFactors`. So we start with a function called `Initialization`, which has three required input parameters: `n` (for the number being factored), `c` and `cutoff`. The values assigned to these parameters will be furnished by the main function of the algorithm, `QS`, to be defined later, where there will be two optional parameters corresponding to the last two parameters of `Initialization`. `c` is used when computing the smoothness bound B, which has the form $c \cdot L(n)^{1/2}$, where the value assigned to the constant c can be set through the parameter `c`. The value assigned to `cutoff` is used to determine which small primes in the factor base will not be used for sieving, namely, those that are less than or equal to the value of `cutoff`, plus the prime divisors of the multiplier independently of their size. The output of this function is a list of the following 11 values:

1. `m`: The multiplier.
2. `mn`: The product of the multiplier by the integer n being factored.
3. `B`: The smoothness bound B.
4. `b`: The value of $\lceil\sqrt{mn}\rceil$.
5. `M`: Half the size of the sieving interval $[-M, M]$.
6. `fb`: The factor base given as a one-dimensional array.
7. `sievingprimes`: The array containing the primes used for sieving.
8. `svplogs`: An array with the integer approximations of the binary logarithms of the sieving primes.
9. `nsvp`: The number of sieving primes.
10. `polroots`: A 2-dimensional array containing the roots of the polynomial $Q(t) = (t + b)^2 - mn$ modulo the sieving primes.
11. `sievearray`: A one-dimensional array of length $2M + 1$ filled with an integer approximation of $\log_2(M\sqrt{mn})$ minus the expected contribution of the small primes not used for sieving.

The multiplier, the smoothness bound and the factor base are computed in a way similar to that used for the factor base method. The value of M, which determines the length of the sieving interval, is chosen somewhat lower than suggested by our previous discussion in order to allow the algorithm to deal with larger numbers without hitting memory limits. The roots of the polynomials are computed with

Maple's function `numtheory:-msqrt` which, as already remarked, computes modular square roots (using Algorithm 2.8 for prime moduli) and is here applied to compute the square roots of the polynomial values modulo the sieving primes. The array `sievearray` is filled with an integer approximation of the value of the binary logarithm of $Q(t)$ minus the expected contribution of the prime factors of $Q(t)$ corresponding to the small primes that are not used for sieving. The logarithm is approximated by $\texttt{round}(\log_2(M\sqrt{mn}))$ because the value of $Q(t) = (t+b)^2 - mn$ on the interval $[-M, M]$ is approximately bounded by $2M\sqrt{mn}$. We disregarded the factor 2 which would add 1 to the logarithm because this can be compensated later by setting the threshold to trial divide the polynomial values one unit lower. The expected contribution of the non-sieving primes is computed using the function g and is subtracted from the log approximation to compensate not sieving with these primes. The function that does the initialization is then the following:

```
> Initialization := proc(n, c, cutoff)
  local z, m, mn, b, B, M, fb, pr, sp, svp, nsvp, sievingprimes,
  i, svplogs, splog, sqroots, polroots, logapprox, sievearray;
  z := MultSelect(n, ':-c' = c);
  m := z[1];
  mn := m*n;
  B := ceil(evalf(c*sqrt(exp(sqrt(ln(mn)*ln(ln(mn)))))));
  fb := z[2];
  b := isqrt(mn);
  if b^2 < mn then
    b := b+1
  end if;
  pr := selectremove(x -> evalb(x<=cutoff) or member(x,numtheory:-factorset(m)),fb);
  sp := pr[1];
  svp := pr[2];
  nsvp := nops(svp);
  splog := add(g(i, m, n)*log[2.](i), i = sp[2 .. -1]);
  M := round(evalf(c*exp(sqrt(ln(mn)*ln(ln(mn))))/(ln(B)*ln(ln(B)))));
  fb := Array(fb, datatype = integer[4]);
  sievingprimes := Array(svp, datatype = integer[4]);
  svplogs := Array(1 .. nsvp, datatype = integer[2]);
  for i to nsvp do
    svplogs[i] := round(log[2.](sievingprimes[i]))
  end do;
  sqroots := Array(map(i -> numtheory:-msqrt(mn, i), svp), datatype = integer[4]);
  polroots := Array(1 .. nsvp, 1 .. 2, datatype = integer[4]);
  for i to nsvp do
    polroots[i, 1] := (sqroots[i]-b) mod sievingprimes[i];
    polroots[i, 2] := (-sqroots[i]-b) mod sievingprimes[i]
  end do;
  logapprox := round(log[2.](M)+log[2.](isqrt(mn))-splog);
  sievearray := Array(1 .. 2*M+1, logapprox, datatype = integer[2]);
  [m, mn, B, b, M, fb, sievingprimes, svplogs, nsvp, polroots, sievearray]
  end proc:
```

The next function implements the sieve. No arrays or other data structures are created and no Maple functions except basic programming ones and arithmetic operations are used within this procedure in order to allow it to be compilable by the external C compiler. The only modification that has to be done for it to be compilable from Maple is to change the data type of the arrays `svplogs` and `sievearray` from `integer[2]` to `integer[4]`. The function has no output but the array `sievearray` is modified by subtracting the logarithms of the sieving primes at the appropriate places as explained in Algorithm 6.5.

```
> sieve := proc(svprimes::(Array(datatype = integer[4])),
                svplogs::(Array(datatype = integer[2])), nsvprimes::integer[4],
                M::integer[4], sievearray::(Array(datatype = integer[2])),
                polroots::(Array(datatype = integer[4])))
  local p, logp, i, j, r1, r2;
  for i to nsvprimes do
    p := svprimes[i];
    logp := svplogs[i];
    r1 := polroots[i, 1];
    r2 := polroots[i, 2];
    j := 1 + ((M+r1) mod p);
    while j <= 2*M+1 do
      sievearray[j] := sievearray[j]-logp;
      j := j+p
    end do;
    j := 1 + ((M+r2) mod p);
    while j <= 2*M+1 do
      sievearray[j] := sievearray[j]-logp;
      j := j+p
    end do
  end do
  end proc:
```

The function `FindRelations` is similar to the previous `Relations` with minor variations to adapt it to the sieve. Its four input parameters correspond to the outputs of the same name produced by `Initialization` and it builds the relations after performing trial division by the factor base primes. The function scans `sievearray` (after this array has been sieved) and tries to factor the integers $(t+b)^2 - mn$ (for $t \in [-M, M]$ or, in terms of the local variable `j`, for $j = t + M + 1 \in [1, 2M+1]$) in case the value `sievearray[j]` is below a threshold which is set here to 14 (the threshold should perhaps grow proportionally to $\ln B$ but, for simplicity, we use a fixed value that works well for integers with less than 50 digits). Because of these low values, these integers have a high probability of being B-smooth. The output of the function consists of the relations in the same format as in `Relations`.

```
> FindRelations := proc(mn, b, M, fb, sievearray)
  local f, start, j, x, r;
  f := [];
  start := b-M-1;
  for j to 2*M+1 do
    x := start+j;
    if sievearray[j] < 14 then
      r := FBTrialDivision(mn, x, fb);
      f := [op(f), r]
    end if;
  end do;
  [Vector([seq(f[i][1], i = 1..nops(f))]), Vector([seq(f[i][2], i = 1..nops(f))]),
   LinearAlgebra:-Transpose(Matrix([seq(f[i][3], i = 1 .. nops(f))]))]
  end proc:
```

We have in place all the ingredients we need to implement the basic QS algorithm, which is given by the next function. The only required input is the integer n to be factored, which should be an odd composite that is not a power. QS is aimed at difficult numbers whose factors are large, so small factors should have been factored out previously with other methods. There are two optional keyword parameters, `c` and `cutoff`, whose arguments are passed to the function `Initialization` where they are used as explained above. The default values are 1.2 for `c` and 7 for `cutoff`,

so that only factor base primes which are ≥ 7 and do not divide the multiplier will be used for sieving. The output is either two nontrivial factors whose product is n or a message informing that no nontrivial factors were found.

```
> QS := proc(n::posint, {c := 1.2, cutoff := 7})
  local st, nfb, svprimes, svplogs, nsvprimes, M, sievearray, polroots,
        kn, b, fb, rels, A, deps;
  st := Initialization(n, c, cutoff);
  print(`Using multiplier: `);
  print(st[1]);
  print(`Using smoothness bound: `);
  print(st[3]);
  fb := st[6];
  print(`Size of factor base: `);
  print(ArrayNumElems(st[6]));
  svprimes := st[7];
  svplogs := st[8];
  nsvprimes := st[9];
  M := st[5];
  sievearray := st[11];
  polroots := st[10];
  print(`Sieving interval of length: `);
  print(2*M+1);
  sieve(svprimes, svplogs, nsvprimes, M, sievearray, polroots);
  print(`Sieving done, searching for smooth values ...`);
  kn := st[2];
  b := st[4];
  rels := FindRelations(kn, b, M, fb, sievearray);
  A := rels[3];
  print(`Number of smooth values found: `);
  print(LinearAlgebra:-RowDimension(A));
  gc();
  print(`Solving a matrix of size: `);
  print(LinearAlgebra:-Dimension(A));
  deps := Dependencies(A);
  print(`Number of linear dependencies found: `);
  print(nops(deps));
  print(`Factors: `);
  FindFactors(n, rels, deps)
  end proc:
```

Example 6.20 Let us factor Fermat's number F_7 with default settings:

```
> QS(2^128 + 1);
                              Using multiplier:
                                     5
                           Using smoothness bound:
                                  28771
                             Size of factor base:
                                  1592
                          Sieving interval of length:
                                57693817
                 Sieving done, searching for smooth values ...
                        Number of smooth values found:
                                  1662
                            Solving a matrix of size:
                                 1662, 1592
                     Number of linear dependencies found:
                                   144
                                 Factors:
                (570468920068512905472l) (59649589127497217)
```

Before giving one more example, let us remark that this implementation admits a lot of tweaking. Many integers will be factored with the default values but others will

need the constant c to be raised in order to collect enough relations. Of course, many other changes can be made to the parameters such as B, M or to the threshold value of 14 used inside the function `FindRelations`, allowing much more flexibility than is permitted by simply changing the value of `c` in `QS`. Numbers of over 40 decimal digits can be factored with this function but we cannot go much further above this size before hitting storage limitations, due to the size of the sieve array which grows very fast and requires a lot of memory. One straightforward way to mitigate this difficulty is to segment the sieve by dividing the sieved interval into smaller subintervals and, as we will mention later, this automatically happens when the basic algorithm is replaced by the "multiple polynomial" one. Segmenting the sieve also has the added benefit that, after sieving each subinterval, it may be checked whether the number of relations collected is already sufficient, in which case there is no need to go further. In the current simple version, if the relations found are not sufficient, one has to start again from the beginning.

Exercise 6.25 Modify the function `sieve` and the function `QS` so that the sieve is segmented. After sieving each subinterval it should be searched for smooth values and the corresponding relations collected. At this point it should be checked whether the collected relations are sufficient and, if not, go on with the remaining subintervals until enough relations have been found.

Example 6.21 Let us factor the 45-digit number:

```
> n45 := 538096101322932978757272139986951552262989443:
```

If we try to factor $n45$ using `QS` with default values, the function uses a factor base of size 3716 but finds only 2929 relations, which are insufficient to generate any dependencies and hence no nontrivial factors are found. However, if we raise the value of `c` to 1.4, the result is the following (with a 64-bit version of Maple, using almost 1 GB of memory; some versions of Maple may give here an '`object too large`' error):

```
> QS(n45, c = 1.4);
                              Using multiplier:
                                       5
                           Using smoothness bound:
                                    86177
                             Size of factor base:
                                    4242
                          Sieving interval of length:
                                  384109867
                 Sieving done, searching for smooth values ...
                        Number of smooth values found:
                                    4213
                           Solving a matrix of size:
                                  4213, 4242
                     Number of linear dependencies found:
                                     244
                                  Factors:
          (5329630764382421350l501) (10096311078789820074943)
```

Exercise 6.26 Modify the procedure `sieve` in order to make it compilable from Maple by the external C compiler and also modify `QS` to include as an option the use of the compiled version of the sieve.

Exercise 6.27 Modify `FindRelations` and `QS` so that the latter admits an optional parameter where the value of the threshold used in `FindRelations` (therein set to 14) can be modified (and perhaps set as proportional to the logarithm of B) and use it to find the optimum threshold for integers of different sizes.

The basic QS algorithm admits many modifications that, without changing its asymptotic running time, have a big impact in practice and permit the factorization of numbers of more than 130 decimal digits. We will not go into the details of these improvements as this would lead us too far into technical aspects of the algorithm but we will give a brief summary of the more important ones.

6.4.9 Some Improvements on the Basic QS

Next we give a brief description of some modifications of the basic QS algorithm that are relevant in practice.

6.4.9.1 Large Prime Variations

The idea of the large-prime variation is to increase, in an indirect way, the smoothness bound B without increasing the size of the factor base, allowing the algorithm to obtain more relations with little additional work. The method works by saving the relations that have at most one prime factor larger than B (for example, in the interval $(B, B^2]$) with the idea of combining them to obtain more B-smooth numbers. These relations are called *partial relations* or, simply, *partials*, in contrast to the ordinary relations which are called *fulls*. The partials are stored and sorted in order of their large primes. If one large prime factor appears only once in this list, then the corresponding partial is discarded as it cannot be used to make a square. But as soon as the large prime appears in two partials, these relations can be multiplied and the large prime removed to form a full relation. Observe that it takes no extra effort to find partials when the large prime is $\leq B^2$ because if all the prime factors $\leq B$ have been removed from the prime factorization of a number and the remaining factor is larger than 1 but smaller than B^2, then it must necessarily be a prime. In addition, the birthday paradox suggests that duplicate large primes will not be uncommon in the partials and this is confirmed by practice.

The idea of considering primes larger than B when building the relations can be pushed further and it has been found that better performance is obtained in very large factorizations by considering two large primes instead of one; for example the factorization of RSA129, a 129-digit number factored in 1994, was accomplished with this

method. This variant includes *partial-partial relations*, i.e., relations corresponding to values of t such that $Q(t)$ can be fully factored over the factor base plus two large primes less than a given bound. The double large-prime method works similarly to the single large-prime variation but with some additional complications derived from the fact that the remaining cofactor in the interval $(B, B^3]$, after the primes in the factor base have been factored out, can be either a prime or a product of two large primes exceeding B. If the cofactor belongs to the interval $(1, B^2]$ then it must be prime, and if it belongs to $(B^2, B^3]$ then a fast pseudoprimality test is applied to it. If it passes the test it is discarded (as it is too large) but if proved composite then it is factored by a method like Pollard rho, giving a partial-partial relation. Finally, these double partial relations are combined by a cycle-finding algorithm in order to produce additional full relations (see [60, 194] and their references for details).

6.4.9.2 The Multiple Polynomial Quadratic Sieve

In the basic QS algorithm the values $x^2 - n$, where $x = t + \lceil \sqrt{n} \rceil$ is an integer close to $\sqrt{n}$, are searched for B-smooth values. But as x goes away from $\sqrt{n}$, the size of $x^2 - n$ increases making it less likely to be B-smooth. This makes the size of the sieve interval $[-M, M]$ grow very fast, which has a considerable impact on performance. A nice solution to this problem was proposed, independently, by Davis, Holdridge, and Montgomery, and it consists of using a family of polynomials instead of the single polynomial $Q(t) = (t + \lceil \sqrt{n} \rceil)^2 - n$. Using multiple polynomials leads to much shorter sieve intervals and the number of relations found per sieved value is much higher; the resulting algorithm is known as the *multiple polynomial quadratic sieve* (MPQS). We refer to [60] for the details on MPQS and the Montgomery method, as well as for the so-called *self initialization*, which is a clever method to minimize the amount of work done when generating the multiple polynomials used by MPQS and finding their zeros, and gives rise to the *self-initializing quadratic sieve* algorithm (SIQS, for short). The time savings provided by MPQS and SIQS with respect to the basic QS are important in practice and a "thought experiment" presented in [60, 6.1.5] suggests that MPQS should run about 17 times as fast as the basic QS method when n is a 100-digit number. Apart from being more efficient, MPQS and SIQS have another important advantage over basic QS: the use of multiple polynomials makes them easily parallelizable.

The previous implementation of QS can be modified without too much difficulty to become an implementation of SIQS with the large prime variation, able to factor integers of more than 70 decimal digits, but it cannot go much further beyond that size because then the linear algebra step as implemented here requires a lot of memory.[7] Making these modifications could be a fine—and challenging—exercise for the reader wishing to acquire a deeper understanding of the method.

[7] To overcome this difficulty a more efficient encoding of the binary matrix—such as storing eight entries per byte—should be used or, better still, sparse encoding in the form of a listing where the positions of the 1s appear, since most of the coefficients are 0. This also requires the use of a sparse-matrix method to compute the null space.

6.4.10 The Current Status of Factorization

6.4.10.1 The Number Field Sieve

The quadratic sieve was the most efficient factoring algorithm available until the 1990s but this is no longer the case since it has been superseded by the *number field sieve* (NFS). The origin of NFS can be traced back to 1988, when John Pollard suggested it as a method to factor integers that are close to a power, such as for example Fermat numbers, but it was later extended to become a general factoring method, also known as the *general number field sieve* (GNFS). NFS relies on the same basic tools as QS, using factor bases, smoothness, sieving and linear algebra, so it is not a big departure from a conceptual standpoint but, on the other hand, it also makes use of much more sophisticated algebraic methods which are outside the scope of this book. The main idea is to go beyond the rings $\mathbb{Z}$ and $\mathbb{Z}_n$ and to use the ring of integers of an algebraic number field to generate the congruences between squares modulo n. We refer to [60, 102, 194, 202] for details. The relevance of NFS for cryptography lies in the fact that it is currently the fastest algorithm to factor "difficult" numbers that are the product of two large primes which, as we will see, are commonly used in the implementation of cryptographic schemes. Until the arrival of NFS the fastest factoring algorithms (such as QS) had an expected running time of $O(\exp((1+o(1))(\ln n)^{1/2}(\ln\ln n)^{1/2}))$ and, because of the Canfield–Erdös–Pomerance theorem we mentioned, some people believed that this bound could not be improved except perhaps for the $1+o(1)$ term. But NFS shattered this belief because, under reasonable heuristic assumptions, it can be proved that its complexity is:

$$O(\exp(((64/9)^{1/3}+o(1))(\ln n)^{1/3}(\ln\ln n)^{2/3})),$$

ie., $O(L_n[1/3, (64/9)^{1/3}+o(1)])$, as defined in Definition 2.12. The complexities of both QS and NFS are *subexponential* in the sense of Definition 2.13 and we recall that this means that they are intermediate between polynomial and exponential. However, as n gets larger, the complexity function of QS grows much faster than that of NFS. We can use Maple to obtain a more precise idea about this. First we define these complexities in Maple (ignoring the $o(1)$ term) as approximate functions of the number of bits of n:

```
> qsc := k -> evalf(exp(ln(2^k)^(1/2)*ln(ln(2^k))^(1/2))):
  nfsc := k -> evalf(exp((64/9)^(1/3)*ln(2^k)^(1/3)*ln(ln(2^k))^(2/3))):
```

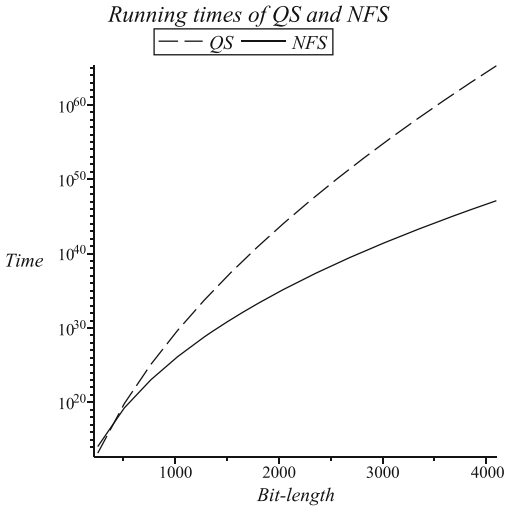
Then we obtain a table (in the form of a matrix) with the values of these functions for integers of $256 \cdot i$ bits, with i ranging from 1 to 8 as follows:

```
> Matrix([x->x, qsc, nfsc]~([seq(256*i, i = 1 .. 8)]))
```

$$\begin{bmatrix} 256 & 1.462692684 \cdot 10^{13} & 1.115419155 \cdot 10^{14} \\ 512 & 6.686872748 \cdot 10^{19} & 1.756508010 \cdot 10^{19} \\ 768 & 1.274022968 \cdot 10^{25} & 1.077547366 \cdot 10^{23} \\ 1024 & 4.423742482 \cdot 10^{29} & 1.315844299 \cdot 10^{26} \\ 1280 & 5.053525463 \cdot 10^{33} & 5.889340171 \cdot 10^{28} \\ 1536 & 2.588065924 \cdot 10^{37} & 1.311509950 \cdot 10^{31} \\ 1792 & 7.161901227 \cdot 10^{40} & 1.736648126 \cdot 10^{33} \\ 2048 & 1.211314590 \cdot 10^{44} & 1.532970529 \cdot 10^{35} \end{bmatrix}$$

The first column gives the size of the number in bits, the second the value of the QS running time function and the third the value of the NFS function. The following picture gives a graphic of both functions up to 4096 bits (a logarithmic scale was used for the y-axis, otherwise the curves would grow too steeply to fit in the picture):

```
> plots:-logplot([[x->x, qsc]~([seq(256*i, i = 1..16)]),
  [x->x, nfsc]~([seq(256*i, i=1..16)])], linestyle=[3,1], color=[black,black],
  legend=[QS,NFS], legendstyle=[location=top], title=[`Running times of QS and NFS`,
  font=[helvetica,14]], labels=[`Bit-length`,`Time`], labelfont=[helvetica,12])
```



We see that for 256-bit numbers QS is faster than NFS but the opposite is true already for 512-bit numbers. This agrees pretty well with the results obtained in practice, when NFS overtakes QS after 120–130 decimal digits. But notice also that, for 2048-bit numbers (which, as we will see, is a common size for RSA moduli), NFS is expected to be almost 10^9 times faster than QS.

There are a couple of excellent open-source implementations of NFS available, which can be used to check the running time on integers of moderate size. They are *Msieve* [147] and GGNFS [85], both written in C, and can be downloaded from the links given in the references.

6.4.10.2 A Factorization Record

Since NFS is the fastest factoring algorithm known so far, its running time plays a crucial role in the choice of cryptographic parameters for schemes whose security relies on the hardness of the integer factorization problem. We will examine this question when discussing the security of these schemes but, for now, we will mention the current factorization record (for an integer with two prime factors of approximately the same size) which corresponds to the following 768-bit, 232-digit number:

```
> RSA768 :=
  12301866845301177551304949583849627207728535695953347921973224521517264005072636\
  57518745202199786469389956474942774063845925192557326303453731548268507917026122\
  14291346167042921431160222124047927473779408066535141959745985690214341 3:
```

This number belonged to the now discontinued "RSA Factoring Challenge" [170] and is the product of the primes:

```
> p := 33478071698956898786044169848212690817704794983713768568912431388982883793 8\
       78002287614711652531743087737814467999489:
  q := 36746043666799590428244633799627952632279158164343087642676032283815739666 5\
       11279233373417143396810270092798736308917:
```

The bit length of n, p, and q is, respectively:

```
> ilog2~([RSA768, p, q])+~1;
                                [768, 384, 384]
```

and the verification that p and q are indeed the factors is easy:

```
> evalb(RSA768 = p*q);
                                     true
```

The factorization of RSA768 was obtained by a team of researchers from several countries led by T. Kleinjung and the method used was NFS. The factorization effort is described in detail in [113] where it is estimated that the computation required more than 10^{20} operations, the equivalent of almost 2000 years of computing on a single core 2.2 GHz AMD Opteron. To give an idea of the size of the computation involved we will only mention that the number of relations obtained was 64334489730 which, after filtering and discarding some of them, produced a matrix of size $192796550 \times 192796550$ with more than $27 \cdot 10^9$ nonzero entries. The sieving step took a little less than two years of computing time and the matrix step about four months, and both were distributed on a variety of machines. The computation began in 2005 and was finished in December, 2009, but discarding some time spent on additional computations and other work, total calendar time may be estimated at about 2.5 years. Taking this as a reference, we can use the expected running time of GNFS to obtain an estimate of the time it would require to factor, say, a 2048-bit integer (having only two prime factors of approximately the same length) with the same computational resources. The number of years would be:

```
> 2.5*nfsc(2048)/nfsc(768);
                                3.556619822*10^12
```

or, taking into account the more recent estimates, *more than 250 times the age of the universe!*

Exercise 6.28 Estimate how many years would it take to factor a 4096-bit composite integer which is a product of two primes of approximately the same length in the following cases:

(i) With the same computational resources used in the factorization of RSA768.
(ii) With computational resources initially equivalent to those in the factorization of RSA768 but assuming that "computing power" doubles each 2 years.

The previous estimate seems to suggest that factoring a "difficult" 2048-bit integer is completely out of reach—which is good for some cryptographic schemes—but there are reasons to be cautious. Some of these reasons are mentioned below.

6.4.10.3 Final Remarks on Factorization

There are other interesting factoring algorithms that we will not study in detail because they are slower than QS and NFS when factoring "difficult" numbers. The most important of these algorithms is the *Elliptic Curve Method* (ECM) of H.W. Lenstra which, as the name implies, is based on the arithmetic of elliptic curves (we will study elliptic curves in Chap. 11 because they also have interesting cryptographic applications). The heuristic complexity of ECM is also subexponential but, as mentioned in [60], this complexity comes close to being rigorous because the only unproved hypothesis used in its proof is a reasonable conjecture about the distribution of smooth numbers in short intervals. Under this hypothesis, Lenstra showed that the expected running time of ECM to find the least prime factor p of a composite number n is $(\ln n)^2 \cdot \exp((\sqrt{2} + o(1))\sqrt{\ln p \ln\ln p})$ which in the worst case $p \approx \sqrt{n}$ becomes $L(n)^{1+o(1)}$ as the term $\ln^2 n$ is absorbed by the $o(1)$. This is the same heuristic complexity as QS but the latter is faster for worst-case numbers. However, ECM is very well suited to finding relatively small factors of huge numbers and it has been successfully used to find factors of several Fermat numbers.

We have mentioned that NFS is the fastest factoring algorithm known but now we must make an important qualification to this assertion: this is only true in practice but not in theory and it will only remain true in practice while no fully functional quantum computers are built. Quantum computers are machines that use the principles of quantum mechanics for their basic operations. Instead of the bits used by classical computers, quantum computers operate with *quantum bits* (*qubits*, for short) and the use of quantum properties such as superposition and entanglement allows them, using a *quantum register* composed of n qubits, to execute in a single computational step the same mathematical operation on 2^n numbers. Thus quantum computers have an "exponential advantage" over classical computers whenever applying a *quantum algorithm* is possible. This is the case with the integer factorization problem and, as Shor [177] proved, *a quantum computer can factor integers in polynomial time*.

Shor found a polynomial-time quantum algorithm to find the order r of an integer x modulo n, i.e., the smallest $r \geq 1$ such that $x^r \equiv 1 \pmod{n}$. Then a factoring algorithm is obtained by finding the orders of random residues modulo n. Indeed,

if x is randomly chosen in $\mathbb{Z}_n^*$ and has even order r, then $x^r - 1 \equiv 0 \pmod{n}$ if and only if $(x^{r/2} + 1)(x^{r/2} - 1) \equiv 0 \pmod{n}$. In this case $x^{r/2} - 1 \not\equiv 0 \pmod{n}$ for otherwise the order of x would divide $r/2$, and hence if $x^{r/2} + 1 \not\equiv 0 \pmod{n}$ we find a nontrivial factor of n by computing $\gcd(x^{r/2} - 1, n)$. So factoring n is equivalent to finding an $x \in \mathbb{Z}_n^*$ with even order r and such that $x^{r/2} \not\equiv -1 \pmod{n}$ and the probability that a randomly chosen x satisfies these conditions is at least $1 - (1/2^{k-1})$ if n is an odd integer with exactly k distinct factors. This probability is $\geq 1/2$ if n has at least two distinct prime factors and hence it suffices to choose several random x to have a high probability of factoring n. Moreover, observe that, once we know the order of x, the remaining operations to find a factor of n can all be done in polynomial time even by a classical computer.

Shor's quantum algorithm to find the order of an integer modulo n is based on a quantum version of the Fast Fourier Transform, which we shall not describe, and we refer to Shor's original article or to [6, 60, 102, 202] for the details.

Exercise 6.29 Let n be an odd integer which is the product of two distinct prime factors. Show that if $x \in \mathbb{Z}_n^*$ is chosen at random and has order r modulo n, then the probability that r is even and $x^{r/2} \not\equiv -1 \pmod{n}$ is at least $1/2$. (Hint: Choosing a random $x \in \mathbb{Z}_n^*$, where $n = p_1 p_2$, amounts to choosing random elements $x_1 \in \mathbb{Z}_{p_1}^*$, $x_2 \in \mathbb{Z}_{p_2}^*$. If the orders of x_1 and x_2 are, respectively, $r_1 = 2^{k_1} s_1$, $r_2 = 2^{k_2} s_2$ with s_1, s_2 odd, then $r = \operatorname{lcm}(r_1, r_2)$ (the least common multiple of r_1 and r_2). Moreover, r is odd if and only if $k_1 = k_2 = 0$ and if r is even and $x^{r/2} \equiv -1 \pmod{n}$, then $k_1 = k_2$.)

Shor's algorithm makes it easy to factor integers except for one thing: a quantum computer is required to do it. In principle, there is nothing in the laws of physics to prevent us from building quantum computers but the practical difficulties seem very great. For now, only very small numbers such as 15 have been factored (see [103]) with experimental quantum computers that use a few qubits—often just four qubits. But factoring integers of several thousands of bits such as those currently being used in cryptography (especially for RSA keys) would require quantum computers with trillions of qubits and there are many problems (some of them related to noise and in particular to the errors produced by the interaction of the quantum computer with the environment) that make it unclear whether such computers will be built and suggest that, in any case, this will not happen in the near future. But, as some of the previous examples of historical factorizations corroborate, predicting the future of factorization is very difficult. There are, however, a few papers that present estimates of factorizations in the near future based mainly on extrapolating from the largest NFS factorizations so far, taking into account other factors such as Moore's law and massive parallelism. We will say a few words about this in Sect. 8.3, when discussing the security of RSA encryption.

6.5 The Discrete Logarithm Problem

We continue our examination of number-theoretic "difficult problems" with a view towards using them as the basis of cryptographic schemes. After the integer factorization problem, the most widely used problem of this type is the discrete logarithm problem (DLP).

We have already met the DLP when discussing one-way functions in Sect. 3.3.2 and we will meet it again soon when discussing the security of the DH protocol in Chap. 7. The most general version of the DLP takes place in a group G whose operation can be carried out efficiently. More explicitly, this implies that we have representations of the group elements by strings in such a way that it is easy to check whether a string represents the identity element and whether two strings represent the same element. Moreover, this also means that we have efficient algorithms to compute the string representing the product of two elements or the inverse of an element. This assumption, together with the binary exponentiation method discussed in Chap. 2 (therein used for modular exponentiation but obviously working in any group) implies that exponentiation in G is also efficient.

An instance of the DLP in a group G is then as follows. Given G and elements $g \in G$, $a \in G$, such that $a \in \langle g \rangle$, find a non-negative integer x such that $a = g^x$. If the order of g is n then, as we have seen in Chap. 3, the discrete logarithm function $\mathtt{dlog}_{(\langle g \rangle, g)} : \langle g \rangle \to \mathbb{Z}_n$ is the inverse of the discrete exponential function $\mathtt{dexp}_{(\langle g \rangle, g)} : \mathbb{Z}_n \to \langle g \rangle$, given by $\mathtt{dexp}_{(\langle g \rangle, g)}(x) = g^x$, where both functions are group isomorphisms. An instance of the DLP is then to find the image of an element $a \in \langle g \rangle$ by the function $\mathtt{dlog}_{(\langle g \rangle, g)}$. This image is called the *discrete logarithm—or the index—of a to the base g* and is denoted $\log_g a$. The interest of the DLP for cryptography comes from the fact that, while computing the function `dexp` is easy with the binary method, computing the function `dlog` is regarded as difficult for some groups.

Remark 6.9 Observe that $\log_g$ is just a shorter name for the function $\mathtt{dlog}_{(\langle g \rangle, g)}$: $\langle g \rangle \to \mathbb{Z}_n$, where the cyclic group and its order are understood, and the fact that this function is a group homomorphism can be expressed as the usual logarithmic formula $\log_g ab = \log_g a + \log_g b$ for all $a, b \in \langle g \rangle$, where $g \in G$ has order n and the sum is in $\mathbb{Z}_n$, i.e., sum of integers modulo n.

Exercise 6.30 Prove that the following "change of base" formula holds: if $\langle g \rangle = \langle g' \rangle \subseteq G$ is a cyclic subgroup of G of order n and $a \in \langle g \rangle$, then $\log_g a = (\log_g g' \cdot \log_{g'} a) \bmod n$.

We will study a few algorithms to solve the DLP and we start by pointing out that these algorithms fall into two classes: the *generic algorithms* that do not depend on any special properties of the encoding of the elements of the group as strings and hence work for arbitrary groups, and the *group-specific* algorithms which work only for specific groups whose additional properties they exploit. In the case of generic algorithms we may assume that $G = \langle g \rangle$ is cyclic because otherwise the elements

of G that do not belong to $\langle g \rangle$ will not be exploited by the algorithm. Moreover, we will generally assume that the order n of g is known (although in some cases it is enough to know an upper bound on n). The most basic generic algorithm for the DLP is a brute-force search in which the elements g^x are computed successively for $x = 1, 2, \ldots$ and compared to a until one such that $g^x = a$ is found (note that here, in contrast with the binary exponentiation method, we cannot take a shortcut and we must compute all the powers g^x between $x = 1$ and $x = \log_g a$). Since this process stops at the latest when $x = n - 1$, it requires $O(n)$ steps each of which consists of a group multiplication. For example, in the case of the group $G = \mathbb{Z}_p^*$, we obtain that the running time is $O(n\text{len}(p)^2)$. But we will see that, even for generic groups, there are algorithms which are more efficient.

At the opposite extreme to the brute-force search, we have seen in Example 3.3 an efficient algorithm for the DLP in the additive group $\mathbb{Z}_n$ which, given a generator $g \in \mathbb{Z}_n$ (i.e., an element $g \in \mathbb{Z}_n$ such that $\gcd(g, n) = 1$), and $a \in \mathbb{Z}_n$, returns $\log_g a$. This is just the element $(g^{-1} \bmod n \cdot a) \bmod n$ which may be computed in time $O(\text{len}(n)^2)$ using the extended Euclidean algorithm. This discrete log algorithm is specific to the groups $\mathbb{Z}_n$ because we have gone beyond the group structure and used the fact that $\mathbb{Z}_n$ is actually a ring in which the generators of its additive group are the same as the units of the ring.

6.5.1 The Baby-Step Giant-Step Algorithm and Its Maple Implementation

The naive brute-force search algorithm for the DLP can be improved by using methods inspired by the birthday paradox. One possibility is to build, instead of just one list of elements of G, two appropriate lists and search for a collision between them. This is the idea underlying Shank's *baby-step giant-step* (BSGS) algorithm. As mentioned before, we will assume that the order of the cyclic group $G = \langle g \rangle$ is known, although knowing an upper bound on this order is sufficient for this algorithm. So let n be the order of G (or an upper bound) and $t = \lceil \sqrt{n} \rceil$, $h = (g^{-1})^t$ and suppose that we want to compute the discrete logarithm of $a \in G$. Consider the lists of elements of G:

1. $1, g, g^2, \ldots, g^{t-1}$,
2. $a, ah, ah^2, \ldots, ah^{t-1}$,

where the first one is constructed with multiplications by g ("baby steps") and the second with multiplications by $h = g^{-t}$ ("giant steps"). Then we look for a collision between the lists, i.e., for a group element in their intersection, say $g^i = ah^j = a(g^{-1})^{tj} = a(g^{tj})^{-1}$. Multiplying both terms by g^{tj} we obtain $a = g^{i+tj}$ and hence we have that $\log_g a = i + tj$. We will assume that the elements of the group G are represented by strings that can be sorted (such will be the case in all our applications). The algorithm may be described as follows:

Algorithm 6.6. Baby-step giant-step.

Input: A finite cyclic group G, a generator $g \in G$, the order n of G and an element $a \in G$.
Output: $\log_g a$.

```
Walk the baby steps (precomputation):
    t := ⌈√n⌉;
    b := 1;
    for i from 0 to t − 1 do
        A[b] := i; (A is a table indexed by elements of G with entries in {0, 1, ..., t − 1})
        b := bg;
    end do;
    Sort the list of indices of A.

Walk the giant steps and find collision:
    h := (g^{-1})^t;
    b := a;
    for j from 0 to t − 1 do
        if b is an index of A (use binary search to find out) and A[b] = i then
            return i + tj
        end if;
        b := bh;
    end do.
```

The first part of the algorithm is a precomputation in the sense that it does not depend on the element $a \in G$ whose discrete logarithm we are trying to compute and hence it is not necessary to repeat it each time the logarithm of a new element of G is computed. After walking the baby steps, we have a table A where $A[b] = i$ just means that $g^i = b \in G$, for $i = 0, 1, \ldots t - 1$, and then the list of indices of this table is sorted. In the second part, after each giant step we can use binary search on the sorted list of indices of A to check whether the group element just generated is an index of A and, if so, we have found the discrete logarithm. This way it is not necessary in general to walk all the giant steps and some space is saved. The following proposition establishes the correctness of the algorithm and its running time:

Proposition 6.5 *The baby-step giant-step algorithm correctly finds the discrete logarithm of an element $a \in G = \langle g \rangle$ and its running time is $O(\sqrt{n} \cdot polylog(n))$ if $n = |G|$.*

Proof Let $x = \log_g a$, so that $g^x = a$, and let $t = \lceil \sqrt{n} \rceil$. Then $0 \leq x < n \leq t^2$ and hence x is a two-digit number in base t, i.e., $x = i + tj$ where i, j are base-t digits such that $0 \leq i, j < t$. Hence $a = g^x = g^{i+tj} = g^i (g^t)^j$, from which it follows that, letting $h = (g^{-1})^t \in G$ as above and multiplying both terms of the equation by $h^j = (g^{tj})^{-1}$, we have that $g^i = ah^j$. The second term of this equality is one of the elements of G computed in the giant steps so the algorithm finishes when this value of j is reached in the final loop with the output $i + tj = \log_g a$.

The total number of group operations performed is clearly $O(\sqrt{n})$ but we also have to take into account the time taken by sorting and searching. Sorting the $O(\sqrt{n})$

table indices takes time $O(\sqrt{n}\ln n)$ and binary search takes time $O(\ln n)$ for every searched element, so total searching time is again $O(\sqrt{n}\ln n)$. Since we are assuming that group operations are efficient, each of them requires time $O(\text{polylog}(n))$. Thus the total running time of the algorithm is $O(\sqrt{n}\cdot\text{polylog}(n))$. □

Remark 6.10 The main drawback of the BSGS algorithm is that it uses a lot of space. Although we have not discussed space complexity, we may mention that the space required to store A is $O(\sqrt{n})$ group elements. In contrast, Pollard's rho, studied in Sect. 6.5.2, is a heuristic algorithm for the DLP that only requires minimal (and constant) storage space.

6.5.1.1 The Baby-Step Giant-Step Algorithm in Maple

Although the BSGS algorithm is generic, we are going to give an implementation[8] for cyclic subgroups of $\mathbb{Z}_m^*$, where $m \geq 2$ (as we will see later, more efficient algorithms are known in this case). The next function takes as input a modulus m and two elements $g, a \in \mathbb{Z}_m^*$ and, optionally, the order n of g in $\mathbb{Z}_m^*$ which, if not supplied, is by default computed inside the function. The output is $\log_g a$ (in case $a \in \langle g\rangle \subseteq \mathbb{Z}_m^*$) or a message informing that a does not belong to the cyclic subgroup generated by g (in which case the discrete logarithm is not defined). Note that we do not require g to be a generator of $\mathbb{Z}_m^*$ which may not even be cyclic, in which case all work will be done within the subgroup $\langle g\rangle$. The function closely follows the steps in Algorithm 6.6, storing the result of computing the baby steps in a table indexed by the elements g^i whose entries are the exponents i. Then the list of indices is sorted and, after computing each giant step, the result obtained is searched for in the list using binary search. The most interesting case is when $m = p$ is prime, so that we know that $\mathbb{Z}_p^*$ is cyclic and there is always a solution if g is a primitive root modulo p.

We remark that, by Proposition 6.5, the baby-step giant-step algorithm is deterministic and always terminates outputting the discrete logarithm if the hypotheses in Algorithm 6.6 hold. Thus, assuming that $g, a \in \mathbb{Z}_m^*$ (otherwise the function will return an error indicating this) the next function always gives the desired result unless $a \notin \langle g\rangle$, in which case an error message pointing this out is printed at the end.

```
> BSGS := proc(m::posint, g::posint, a::posint, n := numtheory:-order(g,m))
  local t, b, A, i, l, h, j, s;
  if igcd(m, g) <> 1 then
    error "%1 is not a unit modulo %2", g, m
  end if;
  if igcd(m, a) <> 1 then
    error "%1 is not a unit modulo %2", a, m
  end if;
  t := isqrt(n);
  if t^2 < n then
    t := t+1
  end if;
```

[8] Maple's built-in function `numtheory:-mlog` computes modular discrete logarithms but we give explicit implementations of the different algorithms for this purpose. In Chap. 11, some of these implementations are adapted to deal with elliptic curve discrete logarithms.

```
b := 1;
A := table();
for i from 0 to t-1 do
  A[b] := i;
  b := b*g mod m
end do;
l := sort(ListTools:-Flatten([indices(A)]));
h := g^(-t) mod m;
b := a mod m;
for j from 0 to t-1 do
  s := ListTools:-BinarySearch(l, b);
  if s <> 0 then
    return A[l[s]]+t*j
  end if;
  b := b*h mod m
end do;
printf("%d does not belong to the cyclic subgroup generated by %d", a, g)
end proc:
```

Example 6.22 Consider the following values of p, g and a:

```
> p := 439809843839:
  g := 139046225820:
  a := 310669347635:
```

Then p is prime and g is a primitive root modulo p. Let us compute $\log_g a$ in $\mathbb{Z}_p^*$:

```
> x := BSGS(p, g, a);
                              x:= 54679028452
```

We can check that x is indeed the discrete logarithm of a to the base g:

```
> Power(g, x) mod p;
                              310669347635
```

Example 6.23 We now compute a discrete logarithm corresponding to a composite modulus. Let

```
> m := 1007916624383:
```

m is composite and, since $\gcd(10^{10}, m) = 1$, we see that $10^{10} \in \mathbb{Z}_m^*$. Now consider the element $111111 \in \mathbb{Z}_m^*$ and let us try to compute its discrete logarithm to the base 10^{10} modulo m:

```
> BSGS(m, 10^10, 111111);
111111 does not belong to the cyclic subgroup generated by 10000000000
```

We see that 111111 does not belong to the cyclic subgroup of $\mathbb{Z}_m^*$ generated by 10^{10}. But the function will compute discrete logarithms in the subgroup $\langle 10^{10} \rangle \subseteq \mathbb{Z}_m^*$ and, for example:

```
> Power(10^10, 10^8) mod m;
                              879981576058
> BSGS(m, 10^10, %);
                                100000000
```

Exercise 6.31 Show that the function `BSGS` can be modified by separating the precomputation from the main function, allowing the computation of several discrete logarithms in the same group without recomputing the table `A`.

Exercise 6.32 Modify the function `BSGS` so that it uses Maple's commands `compiletable` and `tablelook` (it should work well for small numbers but for larger numbers it will require much more memory and will be significantly less efficient than `BSGS`).

6.5.2 Pollard's Rho Method for Discrete Logarithms

As already mentioned, the main drawback of the baby-step giant-step algorithm is that it requires a lot of storage. Pollard devised a method to compute discrete logarithms that is analogous to the rho factoring method and that, while having the same running time as the BSGS method, requires almost no storage. The original formulation was for the group $\mathbb{Z}_p^*$ but the method can be applied to any group whose elements can be encoded as strings. So let G be a finite cyclic group of order n, $g \in G$ a generator, and let $a \in G$ be an element whose discrete logarithm to the base g, $l = \log_g a \in \mathbb{Z}_n$, we want to compute.

The idea is then to find a collision between elements $g^r a^s$ and $g^t a^u$. Since we can compute inverses and exponentiations efficiently in G, this gives us $g^{r-t} = a^{u-s}$ and hence $g^{r-t} = g^{l(u-s)}$. The exponents here are uniquely determined modulo the order n of g so that this equality is equivalent to $l(u-s) \equiv r - t \pmod{n}$ and we can recover the discrete logarithm l by solving this congruence as explained below.

As in the rho factoring method, the procedure to obtain a collision consists of using a function $F : G \to G$ which is sufficiently simple to keep track of its images but behaves heuristically like a random function. Following the original idea of Pollard, we may consider a partition of G into three disjoint subsets which are not related to the group structure of G, $G = S_1 \cup S_2 \cup S_3$, and define $F : G \to G$ as follows:

$$F(x) = \begin{cases} gx & \text{if } x \in S_1, \\ x^2 & \text{if } x \in S_2, \\ ax & \text{if } x \in S_3. \end{cases}$$

Now, if we repeatedly apply the function F starting at $x_0 = 1$, we obtain, after i steps, an element of the form $x_i = g^{r_i} a^{s_i}$ and the idea is to compute the values of x_i and of the exponents r_i and s_i at the same time until a collision is found which, by the birthday paradox, is to be expected after $O(\sqrt{n})$ steps (note that we should assume that $1 \notin S_2$ if we want to start at $x_0 = 1$ for, otherwise, this would be a fixed point and the sequence would remain stationary at this value).

The procedure to keep track of the exponents r_i, s_i is to lift the sequence $x_0 = 1$, $x_1 = F(x_0), \ldots, x_i = F(x_{i-1}), \ldots$ to a sequence $\{(r_i, s_i)\}$ in $\mathbb{Z}_n \times \mathbb{Z}_n$ such that $g^{r_i} a^{s_i} = x_i$. In fact, these two sequences are related by the group homomorphism $h : \mathbb{Z}_n \times \mathbb{Z}_n \to G$ given by $h(r, s) = g^r a^s$, which is actually surjective because the generator g belongs to its image which, being a subgroup, must be equal to all G.

If we use the function $F : G \to G$ defined above, then the sequence $\{(r_i, s_i)\}$ of elements of $\mathbb{Z}_n \times \mathbb{Z}_n$ such that $h((r_i, s_i)) = x_i$ for each i, is computed as follows. It

is clear that $r_0 = s_0 = 0$ and a straightforward induction shows that the subsequent values for the exponents are given by:

$$r_{i+1} = \begin{cases} r_i + 1 \bmod n & \text{if } x_i \in S_1, \\ 2r_i \bmod n & \text{if } x_i \in S_2, \\ r_i & \text{if } x_i \in S_3. \end{cases} \qquad s_{i+1} = \begin{cases} s_i & \text{if } x_i \in S_1, \\ 2s_i \bmod n & \text{if } x_i \in S_2, \\ s_i + 1 \bmod n & \text{if } x_i \in S_3. \end{cases}$$

Then we have that $x_i = g^{r_i} a^{s_i}$ for all $i \geq 0$. As we did in the case of the rho factoring method, we can use Floyd's cycle-finding algorithm to find a collision $x_i = x_{2i}$, so that $g^{r_i} a^{s_i} = g^{r_{2i}} a^{s_{2i}}$ and $a^{s_i - s_{2i}} = g^{r_{2i} - r_i}$. Taking base-$g$ logarithms we obtain $(s_i - s_{2i}) \log_g a \equiv (r_{2i} - r_i) \pmod{n}$. It may happen that $s_i \equiv s_{2i} \pmod{n}$ but the probability is small. In this extreme case the previous congruence gives no information whatsoever and any value in $\mathbb{Z}_n^*$ may be the logarithm so that searching all the possibilities would amount to a brute-force attack. Thus, when this happens, the best strategy is to restart the algorithm with different values of x_0, r_0 and s_0. On the other hand, if $0 \neq t = (s_i - s_{2i}) \bmod n$ and $u = (r_{2i} - r_i) \bmod n$ we have a congruence $t \log_g a \equiv u \pmod{n}$. Then, if $d = \gcd(t, n)$ we see that $d|u$ and, using the extended Euclidean algorithm, we may find an integer w such that $tw \equiv d \pmod{n}$. Multiplying the congruence involving $\log_g a$ by w we obtain a congruence:

$$d \log_g a \equiv uw \pmod{n},$$

in which $d|uw$. Then the linear equation $dx \equiv uw \pmod{n}$ has d solutions (see, e.g., [194, Theorem 5.7]) which are the members of the following set:

$$\left\{ \frac{uw}{d} + j\frac{n}{d} \mid j = 0, 1, \ldots, d-1 \right\}.$$

One of these solutions must be $\log_g a$ and we can find which one by trying all the possibilities. This is usually easy because d is small and, in fact, we often have that $d = 1$. Indeed, in most cryptographic applications either $n = p$ is prime or n has a large prime factor. In the first case, either $t = 0$ which, as already mentioned, is unlikely, or $0 < t < p$ which implies $d = \gcd(t, p) = 1$ giving a unique solution. In case n is not prime, it is unlikely that t will be divisible by the large prime factor of n and hence d will also be small—and, if d is too large, the algorithm can be restarted with different initial values as in the case $t \equiv 0 \pmod{n}$. The reason of choosing n in this way is precisely to make the DLP problem hard for, as we shall soon see, there is an algorithm—the Pohlig–Hellman algorithm—that essentially reduces the DLP in a group of order n (whose prime factorization is known) to the DLP in groups of prime order. Observe also that, once the Pohlig–Hellman algorithm is available, it suffices to apply the rho method to groups of prime order.

There are many possible ways to choose the subsets S_1, S_2, S_3 and, indeed, there is no reason to choose precisely three subsets. In other variants of the algorithm, many more subsets—often called *branches*—are chosen (see, e.g., [97] where additive

notation, as corresponds to an elliptic curve group, is used). In practice, one often uses a *partition function* which, in the case of three subsets, is a function $q : G \to \{1, 2, 3\}$ which is efficiently computable and defines the sets S_i as $S_j = q^{-1}(j)$ for $j = 1, 2, 3$. This means that, in the recursive formulas above, the conditions $x_i \in S_j$ are verified by checking that $q(x_i) = j$.

Since the expected number of steps is $O(\sqrt{n})$, the running time of the rho method is the same as that of the baby-step giant-step algorithm, namely $O(\sqrt{n} \cdot \text{polylog}(n))$, but the advantage of the rho method is that now storage requirements are minimal. Observe, on the other hand that, in contrast with the BSGS method which is completely rigorous, the rho method is heuristic for we do not have a proof that the method always works and finishes within the stated time bound. But, in practice, the rho method works well and is usually preferable to BSGS.

The original formulation of Pollard was for the group $\mathbb{Z}_p^*$, where p is a prime, and the partition he proposed is given by the function $q : \mathbb{Z}_p^* \to \{1, 2, 3\}$ defined by:

$$q(x) = \begin{cases} 1 & \text{if} \quad 0 < x < \frac{1}{3}\ p, \\ 2 & \text{if } \frac{1}{3}\ p < x < \frac{2}{3}\ p, \\ 3 & \text{if } \frac{2}{3}\ p < x < p. \end{cases}$$

The rho algorithm is summarized in Algorithm 6.7. Given $F : G \to G$, defined as above in terms of g and a, and $(x_i, r_i, s_i) \in G \times \mathbb{Z}_n \times \mathbb{Z}_n$ such that $h(r_i, s_i) = x_i$, we define $f(x_i, r_i, s_i) = (F(x_i), r_{i+1}, s_{i+1})$ (where r_{i+1}, s_{i+1} are the exponents of g and a when $F(x_i)$ is expressed in terms of these elements, so that $h(r_{i+1}, s_{i+1}) = F(x_i)$).

Algorithm 6.7. Pollard's rho for discrete logarithms.

Input: A finite cyclic group G of order n, a generator $g \in G$ and $a \in G$.
Output: $\log_g a$.

`Define` $f : G \times \mathbb{Z}_n \times \mathbb{Z}_n \to G \times \mathbb{Z}_n \times \mathbb{Z}_n$ so that if $(x_i, r_i, s_i) \in G \times \mathbb{Z}_n \times \mathbb{Z}_n$ then, with the previous notation:
$f(x_i, r_i, s_i) = (F(x_i), r_{i+1}, s_{i+1})$.

`Set` $b := (1, 0, 0)$ and $c := (1, 0, 0)$.

`repeat`
  $b := f(b)$;
  $c := f(f(c))$;
`until` $b[1] = c[1]$.

`if` $b[3] \not\equiv c[3] \pmod{n}$ `then`
  `Find` $log_g a$ among the solutions of the congruence:
  $(b[3] - c[3])x \equiv c[2] - b[2] \pmod{n}$;
  `return` $\log_g a$
`else`
  `Restart` computation with new values of b and c
`end if`.

Exercise 6.33 Show that each element of G has n distinct preimages by the function $h : \mathbb{Z}_n \times \mathbb{Z}_n \to G$ given by $h(r, s) = g^r a^s$, where G is a cyclic group of order n, g a generator of G, and $a \in G$. (Hint: This follows from the fact that Ker h is a subgroup of order n or it can also be checked directly).

We will briefly mention another generic method for the DLP which was also introduced by Pollard at the same time as the rho method and has similar complexity to the latter and minimal storage requirements too. It is called the *lambda method* or the *kangaroo method*. The idea is to take a walk from the group element a whose logarithm we want to compute and another one from an element whose discrete logarithm is known. If the two walks converge (which, when the paths are drawn upwards produces the 'λ' shape) then the discrete logarithm of a can be found. Pollard regarded the steps in these walks as jumps of two kangaroos, a tame one starting at an element whose discrete logarithm is known, and a wild one starting at the element a whose logarithm we search for. This algorithm is especially useful when the discrete logarithm we search is known to lie in a short interval, in which case this information can be exploited to make the algorithm run faster. We refer to [60, 194] for detailed descriptions of the method.

An interesting feature of both the rho and the lambda methods is that they can be parallelized (see [60] for details). The current record for an elliptic curve discrete logarithm, to which we will come back with more details in Example 11.20, was established in 2009 by means of a parallelized version of the rho method. This computation found a discrete logarithm in a group whose order is a 112-bit prime in a time of 3.5 months (not counting idle time) on a cluster of more than 200 PlayStation 3 consoles (see [39] for more details). We will analyze the significance of this computation when discussing the security of elliptic curve cryptography in Chap. 11.

6.5.3 The Rho Method for Discrete Logarithms in Maple

We next give a Maple implementation of the rho method for cyclic subgroups of $\mathbb{Z}_m^*$, with $m \geq 2$, using the partition defined by Pollard. We follow the algorithm structure sketched above to define the function `RhoDiscreteLog` which is given next. The function has four input parameters: `m,g,a` and `n`, with the first three required and the last optional. `m` is used to specify the modulus, i.e., to define the group $\mathbb{Z}_m^*$. `g` specifies the base g of the discrete logarithms and may (but need not) be a generator of G. `a` is used to specify the element a whose logarithm we want to compute, which should belong to the cyclic group generated by g (of course, a can be any element of $\mathbb{Z}_m^*$ in case this group is cyclic and g is a generator). Finally, `n` takes as value the order of g and, if not specified, it is computed by Maple using the function `numtheory:-order`, which supplies the default value. The function proceeds according to the summary given in Algorithm 6.7, using the function f defined by Pollard and, in case $b[3] \equiv c[3] \pmod{n}$ (with the above notation) and the order

is ≥ 1000, prompts the user asking whether the computation should be continued with new seeds (when $b[3] \equiv c[3] \pmod{n}$ and $n < 1000$, all the possible values are searched until the discrete logarithm is found). If the answer is positive then it chooses new values for b and c and restarts the computation. Observe that if a does not belong to the cyclic subgroup of $\mathbb{Z}_m^*$ generated by g, the algorithm will also ask the user whether to continue the computation, so one should make sure that this situation does not happen. The output of the function is $\log_g a$.

```
> RhoDiscreteLog := proc(m::posint, g::posint, a::posint, n:=numtheory:-order(g,m))
  local f, b, c, found, sols, i, finished, ans, u;
  if igcd(m, g) <> 1 then
    error "%1 is not a unit modulo %2", g, m
  end if;
  if igcd(m, a) <> 1 then
    error "%1 is not a unit modulo %2", a, m
  end if;
  f := proc(x, r, s)
  if x < m/3 then
    g*x mod m, r+1 mod n, s
  elif x < 2*m/3 then
    x^2 mod m, 2*r mod n, 2*s mod n
  else
    a*x mod m, r, s+1 mod n
  end if;
  end proc;
  b := 1, 0, 0;
  c := 1, 0, 0;
  finished := false;
  while not finished do
    found := false;
    while not found do
      b := f(b);
      c := f(f(c));
      found := evalb(b[1] = c[1])
    end do;
    if b[3] <> c[3] or n < 1000 then
      sols := Roots((b[3]-c[3])*x-c[2]+b[2]) mod n;
      sols := [seq(sols[i][1], i = 1 .. nops(sols))];
      for i in sols do
        if a = Power(g, i) mod m then
          return i
        end if;
      end do
    end if;
    ans := readstat("Not found, continue?, y/n");
    if ans = y then
      print("Computation proceeds with new initial values...");
      randomize();
      u := RandomTools:-Generate(integer(range = 1 .. n-1));
      b := Power(g, u) mod m, u, 0;
      c := b
    else
      finished := true;
      print("Computation interrupted")
    end if;
  end do;
  end proc:
```

Example 6.24 Let us consider the modulus $m = 2017$, which is prime, and compute a primitive root modulo m:

```
> numtheory:-primroot(2017);
```

Thus we know that 5 is a generator of $\mathbb{Z}_{2017}^*$ and hence we can compute the discrete logarithm of any integer in the range 1..2016 to the base 5. Let us compute the discrete logarithm of 74:

```
> RhoDiscreteLog(2017, 5, 74);
             "Computation proceeds with new initial values..."
                                1631
```

The function does not find the logarithm in the first attempt but if we answer 'y' the computation proceeds and the logarithm 1631 is found (it could take more than one restart).

Example 6.25 The following values were used in Example 6.22 with the BSGS method. We appreciate that this implementation of the rho method does it faster:

```
> p := 439809843839:
  g := 139046225820:
  a := 310669347635:
  RhoDiscreteLog(p, g, a);
                                  54679028452
```

An interesting feature of the rho method is the very little storage it requires (there are versions that, using some more storage, are more efficient at finding collisions). This allows the previous function to tackle larger numbers than the function `BSGS` although the running time grows similarly. We next present an example that shows that a discrete logarithm corresponding to a 17-digit modulus is easily doable with `RhoDiscreteLog`:

Example 6.26 Let us consider the modulus $m = 12345678910111223$ which, as can easily be checked, is prime. Also, 5 is a primitive root modulo m because:

```
> m = 12345678910111223:
  numtheory:-primroot(m);
                                      5
```

Then we can use `RhoDiscreteLog` to compute $\log_5 9876543210$ in $\mathbb{Z}_m^*$ (this may take a long time depending on the computer used but we will see in Example 6.28 a much faster method to compute it):

```
> RhoDiscreteLog(m, 5, 9876543210);
                              1001490480543171
```

Exercise 6.34 Modify the function `RhoDiscreteLog` so that it works with different functions which partition the group into three subsets. In order to do that, add a new input parameter that takes as value the partition function which can be previously defined. Check it with the partition functions $x \mapsto 1 + (x + 1) \bmod 3$ and $x \mapsto 1 + (x + 2) \bmod 3$.

6.5.4 The Pohlig–Hellman Algorithm and Its Maple Implementation

We have already mentioned that to make the DL problem difficult, the cyclic group in which the problem occurs is often chosen to be of prime order, as was the case in the record computation mentioned in Sect. 6.5.2. The reason for this is just the Chinese remainder theorem which, as we have already seen, reduces the problem of finding the solution of an equation modulo n, where $n = \prod_{i=1}^{k} n_i$ with the n_i pairwise relatively prime, to the problem of solving the equation modulo n_i for each $i = 1, \ldots, k$. This idea was applied by Pohlig and Hellman to the DL problem as follows:

Proposition 6.6 *Let G be a group and $g \in G$ an element of order n, where $n = \prod_{i=1}^{k} n_i$ and $\gcd(n_i, n_j) = 1$ for $i \neq j$. Let $a \in \langle g \rangle$, $g_i = g^{n/n_i}$, and $a_i = a^{n/n_i}$, for each $i = 1, 2, \ldots, k$. Then, if $x_i = \log_{g_i} a_i$, $\log_g a$ is the solution in $\mathbb{Z}_n$ of the system of congruences:*

$$\left.\begin{array}{ll} x \equiv x_1 & (\mathrm{mod}\ n_1) \\ x \equiv x_2 & (\mathrm{mod}\ n_2) \\ & \vdots \\ x \equiv x_k & (\mathrm{mod}\ n_k) \end{array}\right\} \tag{6.2}$$

Proof Each g_i has order n_i by Proposition 2.5. Let $x_i = \log_{g_i} a_i \in \mathbb{Z}_{n_i}$ and $x = \log_g a \in \mathbb{Z}_n$. Then $g_i^{x_i} = a_i = a^{n/n_i} = (g^x)^{n/n_i} = (g^{n/n_i})^x = g_i^x$. Since the order of g_i is uniquely defined as an element of $\mathbb{Z}_{n_i}$ this implies that $x \equiv x_i \pmod{n_i}$, for each $i = 1, \ldots, k$. But the n_i are pairwise relatively prime by hypothesis and so the Chinese remainder theorem tells us that x is the unique solution in $\mathbb{Z}_n$ of this system of congruences. □

We see that computation of discrete logarithms in G is reduced to the computation of discrete logarithms in the subgroups $G_i = \langle g_i \rangle$ which have order n_i, plus some additional computations which can be carried out in time $O(\mathrm{polylog}(n))$. These are the computations of each of the powers g_i and a_i and the application of the CRT. We can use either the BSGS or the rho method to compute the discrete logarithm in G_i in time $O(\sqrt{n_i} \cdot \mathrm{polylog}(n_i))$ and hence we can solve the DLP in G in time $O\left(\sum_{i=1}^{k} \sqrt{n_i} \cdot \mathrm{polylog}(n)\right)$. Since the number k of factors is $k = O(\ln n)$, we see that $O\left(\sum_{i=1}^{k} \sqrt{n_i}\right) = O\left(\max_i(\sqrt{n_i}) \cdot \ln n\right)$ and hence we obtain a running time for the entire algorithm of $O(\max_i(\sqrt{n_i}) \cdot \mathrm{polylog}(n))$. Thus we see that the time to compute discrete logarithms in G is essentially determined by the size of the largest factor of the group order n.

If the prime factorization of n is $n = \prod_{i=1}^{k} p_i^{e_i}$ then we can take $n_i = p_i^{e_i}$. In this case, there is a further step that reduces the computation of a discrete logarithm in the group G_i of order $p_i^{e_i}$ to the computation of e_i discrete logarithms in groups of

order p_i. Since $e_i = O(\ln n)$, this implies that the running time of the algorithm to compute discrete logarithms in G can be improved to $O(\max_i(\sqrt{p_i}) \cdot \text{polylog}(n))$. This is the reason why a necessary condition for the DLP to be difficult is that the order of G should have a large prime factor. We shall not go into the details of this further reduction for which we refer the interested reader to [43, 102, 193].

6.5.4.1 The Pohlig–Hellman Algorithm in Maple

We next implement the Pohlig–Hellman algorithm in Maple with the option of using either BSGS or the rho method, which will allow us to compute discrete logarithms in groups of large order whose prime factors are relatively small. We will just use the CRT reduction modulo prime powers, which is sufficient to compute discrete logarithms in $\mathbb{Z}_p^*$ for typical prime moduli p of relatively large size.

The `PohligHellman` function has the same input parameters as the previous function `RhoDiscreteLog`, plus an optional keyword parameter `method` which is used to specify which method is used to compute the discrete logarithms. The values accepted by this parameter are `method = Rho` and `method= BSGS`, with the former being the default. The output is, assuming that $a \in \langle g \rangle$, the discrete logarithm of a to the base $g \in \mathbb{Z}_m^*$.

```
> PohligHellman := proc(m::posint, g::posint, a::posint,
                        n::posint := numtheory:-order(g, m), {method::name:=Rho})
  local facts, i, exps, glist, alist, xlist;
  if igcd(m, g) <> 1 then
    error "%1 is not a unit modulo %2", g, m
  end if;
  if igcd(m, a) <> 1 then
    error "%1 is not a unit modulo %2", a, m
  end if;
  if n < 1000000 then
     return BSGS(m, g, a)
  end if;
  facts := ifactors(n)[2];
  facts := map(l -> l[1]^l[2], facts);
  exps := n/~(facts);
  glist := Power~(g, exps) mod m;
  alist := Power~(a, exps) mod m;
  if method = Rho then
    xlist := RhoDiscreteLog~(m, glist, alist, facts)
  else
    xlist := BSGS~(m, glist, alist, facts)
  end if;
  chrem(xlist, facts)
  end proc:
```

Example 6.27 We use the same elements as in some previous examples for BSGS and the rho method:

```
> p := 439809843839:
  g := 139046225820:
  a := 310669347635:
  PohligHellman(p, g, a, method = BSGS);
                                 54679028452
```

The time taken by this computation is practically the same as is taken by the BSGS method and this is because p is actually a safe prime, i.e., $p - 1 = 2q$, with q prime:

```
> ifactor(p-1);
                              (2) (219904921919)
```

so that there is not much to gain by using the CRT.

Example 6.28 Let us now consider another previous example, namely, the prime modulus $m = 12345678910111223$, modulo which 5 is a primitive root:

```
> m = 12345678910111223:
  PohligHellman(m, 5, 9876543210);
                               1001490480543171
```

The result is, of course, the same as when we used `RhoDiscreteLog` but the time taken is more than four orders of magnitude smaller. If we use Pohlig–Hellman with `BSGS` then the time taken is even less, namely, more than five orders of magnitude smaller:

```
> PohligHellman(m, 5, 9876543210, method = BSGS);
                               1001490480543171
```

The reason is that

```
> ifactor(m-1);
                      (2) (13) (607) (45971) (17016451)
```

and we see that the largest prime-power factor of $m - 1$ has only 8 decimal digits.

Example 6.29 Let us now do an example of much larger size. As we will see later on, a discrete logarithm in a group whose order is a 160-bit prime is currently regarded as infeasible with generic methods such as the ones we have seen so far, although it will probably not remain so for long.[9] Let us now consider the following number:

```
> p2 := 1514473161367983299997097529214293949732819778399;
  isprime(p2);
  ilog2(p2);
  ifactor(p2-1);
                                    true
                                     160
 (2) (3)^2  (147699571289) (254019119) (68087263) (34891393) (3593) (45763) (5741)
```

We see that `p2` is prime, it has 161 bits (since $\lfloor \log_2(p2) \rfloor + 1 = 161$) and the largest prime factor of $p2-1 = |\mathbb{Z}_{p2}^*|$ is a 12-digit prime. Hence computing a discrete logarithm in a subgroup of $\mathbb{Z}_{p2}^*$ should be well within reach of `PohligHellman`. To see that this is indeed the case, let us consider the following element of $\mathbb{Z}_{p2}^*$:

```
> g2 := 3002845391618757060472038222590765500078306106 4:
```

We see that $g2$ is a quadratic residue modulo $p2$ and hence it cannot be a generator of $\mathbb{Z}_{p2}^*$ but we can check that it is actually a generator of the subgroup of $\mathbb{Z}_{p2}^*$ consisting of all the quadratic residues. Indeed:

[9] As mentioned, the current discrete logarithm record in an elliptic curve group, where only generic methods are available, is in a group whose order is a 112-bit prime but, in the case of $\mathbb{Z}_p^*$, the index calculus methods such as NFS—see Sect. 6.5.5—are able to deal with much larger primes.

```
> iquo(p2, numtheory:-order(g2, p2));
                                    2
```

and this means that the subgroup $\langle g2\rangle$ has index 2 in $\mathbb{Z}_{p2}^*$. Since it is a subgroup of the group of quadratic residues and has the same order, the two groups are the same and the order is $(p2-1)/2$, a 160-bit number. Let us now pick an element of $\langle g2\rangle$, i.e., a quadratic residue, and find its discrete logarithm to the base $g2$. Let

```
> a2 := 14159265358979323846264338327950288419716939937 5:
```

(incidentally, the digits of $a2$ are just the first 48 digits of π after the decimal point, as in the example of [39]). We can check that $a2$ is a quadratic residue modulo $p2$ and hence it belongs to $\langle g2\rangle$ and its discrete logarithm to the base $g2$ is defined:

```
> numtheory:-legendre(a2, p2);
                                    1
```

So, let us use `PohligHellman` (with the rho method) to compute $\log_{g2} a2$:

```
> PohligHellman(p2, g2, a2);
          27749172991813506075972418485198200657987150888 1
```

The result is quickly found and another possibility is to use the combination `PohligHellman/BSGS`:

```
> PohligHellman(p2, g2, a2, method = BSGS);
          27749172991813506075972418485198200657987150888 1
```

In this case BSGS is slower than rho but it is a matter of seconds anyway. We check that the result is correct:

```
> evalb(Power(g2, 27749172991813506075972418485198200657987150888 1) mod p2 = a2);
                                  true
```

6.5.5 The Index Calculus Method for Discrete Logarithms

The generic algorithms for the DL problem discussed so far work in any finite cyclic group whose elements can be represented in such a way that we have efficient algorithms to perform the group multiplication. As we have seen, the number of steps or group operations required by these algorithms is of the order of the square root of the group order, and hence they run in exponential time. However, we have also seen an example of a class of groups for which there is an efficient specific method to compute discrete logarithms, namely, the (additive) groups $\mathbb{Z}_n$, for which there is an algorithm with running time $O(\text{len}(n)^2)$. The specific feature that underlies this algorithm is that these groups have an additional multiplicative structure which makes them (commutative) rings. This suggests the possibility that additional features of the group might also be exploited in other cases, such as the groups $\mathbb{Z}_p^*$ with p prime or, more generally, the multiplicative groups of nonzero elements of finite fields. In the case of $\mathbb{Z}_p^*$, its operation is closely related to the multiplication of $\mathbb{Z}$ and hence the idea arose of using smooth numbers and factor bases in a similar way to the subexponential

factorization methods we have seen. This idea turned out to be fruitful and led to a class of algorithms for the DL problem that are generically known as *index calculus algorithms* (we recall here that 'index' is just another name for discrete logarithm). As we will indicate, the similarities with the smoothness-based factorization methods extend to the running times, which are also subexponential in this case.

We will give a description of the basic index calculus method for the groups $\mathbb{Z}_p^*$ but we shall not go into much detail for a couple of reasons. The first is that, from a theoretical point of view, these algorithms are similar to factorization algorithms such as the quadratic sieve which we have already discussed in some detail. More importantly, the very fact that these algorithms exist has discouraged the use of cryptographic schemes based on the DL problem in the multiplicative groups of finite fields. The reason is that there are some important classes of groups—in particular the elliptic curve groups that we will study in Chap. 11—for which the only known methods to solve the DL problem are the generic ones, which are exponential. This makes these groups very appealing from a cryptographic point of view because the additional difficulty of the problem allows the use of smaller key lengths, which is obviously beneficial for key management, efficiency and so on. In addition, we also remark that some of the more advanced index calculus algorithms, such as the ones inspired by NFS and those that use function fields, require a mathematical background that is beyond the scope of this book.

We next describe the basic index calculus algorithm for $\mathbb{Z}_p^*$. Suppose that $g \in \mathbb{Z}_p^*$ is a primitive root modulo p, i.e., a generator of the group, and that $a \in \mathbb{Z}_p^*$. The problem is then to compute $\log_g a \in \mathbb{Z}_{p-1}$. In a similar way to the quadratic sieve factorization algorithm, the basic idea is to choose a factor base $FB = \{p_1, p_2, \ldots, p_k\}$, consisting of small primes (usually all the primes $\leq B$ for a smoothness bound B), and obtain factorizations over the factor base of group elements of the form $g^t \mod p$. Here the integers t will be randomly chosen in the interval $[1, p-2]$ and so the resulting algorithm is probabilistic. To factorize these residues, a method such as trial division, the rho method, or the more powerful Elliptic Curve Method, is used and a series of *relations* is obtained:

$$\begin{aligned} g^{t_1} \bmod p &= \prod_{j=1}^{k} p_j^{e_{1j}}, \\ g^{t_2} \bmod p &= \prod_{j=1}^{k} p_j^{e_{2j}}, \\ &\vdots \\ g^{t_l} \bmod p &= \prod_{j=1}^{k} p_j^{e_{lj}}. \end{aligned} \tag{6.3}$$

Observe that smoothness is, again, the concept at work here and observe also that, when using these factorizations, we are in some conceptual sense stepping outside the group $\mathbb{Z}_p^*$: indeed, the primes are not defined by properties of $\mathbb{Z}_p^*$ but by those of $\mathbb{Z}$. Now, the idea is to obtain sufficient equations like these to be able to compute the discrete logarithms $\log_g p_j$ for $j = 1, \ldots, k$. This problem can be reduced to solving a system of linear equations by using the discrete logarithm isomorphism $\mathtt{dlog}_{(p,g)} : \mathbb{Z}_p^* \to \mathbb{Z}_{p-1}$. Applying this isomorphism to both terms of the preceding equations (or, in other words, taking logarithms in these equations) we obtain equations in $\mathbb{Z}_{p-1}$:

$$t_i \equiv \sum_{j=1}^{k} e_{ij} \log_g p_j \pmod{p-1}, \quad i = 1, \ldots, l.$$

In more explicit terms, we obtain the following system of linear equations over $\mathbb{Z}_{p-1}$ where, as customary, we write the 'independent term' on the right hand side and, for brevity, we write $x_j = \log_g p_j$ (so that the discrete logarithms we are trying to find become the variables of the linear system):

$$\left.\begin{aligned} e_{11}x_1 + e_{12}x_2 + \cdots + e_{1k}x_k &\equiv t_1 \pmod{p-1} \\ e_{21}x_1 + e_{22}x_2 + \cdots + e_{2k}x_k &\equiv t_2 \pmod{p-1} \\ &\vdots \\ e_{l1}x_1 + e_{l2}x_2 + \cdots + e_{lk}x_k &\equiv t_l \pmod{p-1} \end{aligned}\right\} \tag{6.4}$$

This system can be solved using linear algebra algorithms and if the solution is unique then it will consist of the discrete logarithms of the factor base primes. Here there is the additional difficulty that the modulus $p-1$ is not prime and hence $\mathbb{Z}_{p-1}$ is not a field but just a ring. The solution we are looking for is an element of the 'free module' (the analogue of vector space in this more general setting) $\mathbb{Z}_{p-1}^k$. The method we can use to solve the system is an extension of the usual methods (such as Gaussian elimination) for systems defined over fields. Indeed, these methods can be used to solve the system modulo the prime factors of $p-1$ and there are also ways to lift these solutions to solutions modulo higher prime powers if necessary.[10] Once the system is solved modulo the prime powers dividing $p-1$, the final solution is obtained just by applying the Chinese remainder theorem. Indeed, the 'Chinese remainder isomorphism' $\mathbb{Z}_{p-1} \cong \mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_n}$ from Theorem 2.14 allows the problem to be moved from $\mathbb{Z}_{p-1}$ to the $\mathbb{Z}_{m_i}$, where $m_1, m_1, \ldots, m_n$ are the prime powers dividing $p-1$.

Observe that the logarithms we are looking for always define a solution of this system but the system may be 'under-defined' or, in other words, there may be other solutions. Thus it is important to collect a sufficient number of relations and usually a few more than k should be sufficient so we can take $l = k + 10$ (or a higher number if we want more assurance).

[10] We will not explain these methods here but note that they are not necessary if p is a safe prime because then $p-1$ is square-free.

Note also that the computation of the discrete logarithms of the factor base primes is a precomputation that does not require the element a as input, since it depends only on g and p. Because of this, its output can later be used to compute the discrete logarithm of any $a \in \mathbb{Z}_p^*$, without having to redo this precomputation each time.

Once this first step is finished, the next one looks, again for r randomly chosen, for a special relation of the form:

$$g^r a \mod p = \prod_{j=1}^{k} p_j^{c_j},$$

which, in turn, gives a linear equation modulo $p-1$:

$$\log_g a \equiv -r + c_1 \log_g p_1 + c_2 \log_g p_1 + \cdots + c_k \log_g p_k \pmod{p-1}.$$

Plugging into this equation the values of the discrete logarithms of the primes, which were obtained in the previous step, the discrete logarithm of a is found.

These steps admit many variants and refinements. For example, -1 can be added to the factor base and this allows us to factor the least absolute residue of g^{t_i} (i.e., the integer closest to 0 which is congruent to g^{t_i} modulo p) instead of the least non-negative residue, with a noticeable efficiency gain due to the fact that the residues obtained are smaller and hence more likely to factor over the factor base. Note that this variant is possible because the discrete logarithm of -1 is known in advance, namely, $\log_g(-1) = (p-1)/2$.

The algorithm thus sketched is summarized in Algorithm 6.8:

Algorithm 6.8. Index calculus method for $\mathbb{Z}_p^*$.

Input: A prime p, a primitive root modulo p, g, and an element $a \in \mathbb{Z}_p^*$.
Output: $\log_g a \in \mathbb{Z}_{p-1}$.
Precomputation:

1. `Choose` a smoothness bound B and the factor base $FB = \{p_1, \ldots, p_k\}$ formed by the primes $\leq B$.
2. `Choose` random integers $t \in [1, p-2]$ until $k+10$ (or other similar target number) B-smooth residues $g^t \mod p$ are found and save the corresponding relations.
3. `Solve` the linear system given by the relations and find $\log_g p_1, \ldots, \log_g p_k$.

Computation of $\log_g a$:

1. `Choose` random integers in $[1, p-2]$ until r is found such that $g^r a \mod p = \prod_{j=1}^{k} p_j^{c_j}$ is B-smooth.
2. `return` $\log_g a = (-r + \sum_{j=1}^{k} c_j \log_g p_j) \bmod (p-1)$.

There are a few things that have not been specified in the preceding description of the algorithm. Perhaps the most important is how to choose the smoothness bound B which, as happens with the quadratic sieve, has a direct impact on the complexity

of the algorithm. In a way similar to the case of QS, it can be shown that an optimal choice for B (in case the Elliptic Curve Method is used to obtain the relations) is $B = L(p)^{1/\sqrt{2}}$, where $L(p) = \exp(\sqrt{(\ln p)(\ln \ln p)})$, leading to a subexponential complexity $L(p)^{\sqrt{2}+o(1)}$.

6.5.6 The Index Calculus Method in Maple

We next give an implementation of the basic index calculus algorithm. For the factorization of the residues we will use trial division and so we will adapt the previously defined function `FBTrialDivision`—which was used for the Factor Base and QS factorization methods—for this purpose. The function `ICTrialDivision` below has four required input parameters which are used, in order, to specify a prime modulus p, a primitive root g, an integer in the range $1..p-2$ (which will be pseudo-randomly chosen by the main `IndexCalculus` function below) and the factor base *fb*. There are two optional parameters, the first of which is `special`, which tells the function whether an ordinary relation or a special one (i.e., the final relation involving the element whose discrete logarithm is being computed) is sought. The default value for this parameter is `false` (it will be set to `true` only when looking for the special relation at the end). The second optional parameter `a` will take the value 1 in the 'nonspecial' case and the integer whose discrete logarithm we are searching for in the 'special' case. The default value is 1. Note that this function is intended to be called from the other functions in this implementation and not directly by the user.

The output is a relation in the appropriate format: the exponent vector corresponding to the factor base in the 'special' case, the exponent vector plus the exponent of g in the 'nonspecial' case, with a modification of this exponent in order to exploit the known logarithm of -1 to speed up the process by replacing the integer to be factored, in case it is greater than $(p-1)/2$, by its opposite in $\mathbb{Z}_p$.

```
> ICTrialDivision := proc(p, g, x, fb, special:=false, a:=1)
  local r, z, t, k, e, q, i;
  r := 1;
  z := modp(a*Power(g, x), p);
  t := iquo(p-1, 2);
  if not special and t < z then
    q := p-z
  else
    q := z
  end if;
  k := nops(fb);
  e := Vector[row](k, datatype = integer[1]);
  for i to k while r <> 0 do
    r := modp(q, fb[i]);
    while r = 0 do
      q := iquo(q, fb[i]);
      e[i] := 1+e[i];
      r := modp(q, fb[i]);
      if q = 1 then
        if special then
```

```
        return Vector[row](e)
      elif t < z then
        return Vector[row]([e, (x-t) mod (p-1)])
      else
        return Vector[row]([e, x])
      end if;
    end if;
  end do;
end do;
end proc:
```

The next function, `IndexCalculusPrecomp`, embodies the precomputation stage of the index calculus algorithm, which depends only on p and g and not on the element whose logarithm is being computed. The smoothness bound and the factor base are obtained first and then the ordinary relations are computed by generating pseudo-random integers x and calling `ICTrialDivision` to try to factor $g^x \mod p$ for each of these x. This way, the matrix M whose solution gives the logarithms of the factor base is also computed. Then the linear system defined by this matrix is solved by calling `LinearAlgebra:-Modular:-LinearSolve`, which works also for composite moduli such as $p-1$. The output is a list whose first term is the factor base and whose second term is the vector of the discrete logarithms of the factor base primes.

The required inputs are the prime p and the primitive root g. Moreover, there are two optional keyword parameters. One of them, `verbose`, tells the function whether or not to print additional information on screen and has `false` as default. The other, `addrels`, specifies how many relations are to be computed, their number being equal to the size k of the factor base plus the value of `addrels`. The default value for this parameter is 10, as it seems that $k+10$ ordinary relations are enough in most cases. If the relations found are not sufficient to determine the discrete logarithms of the factor base primes as their only solution, then either Maple's function `LinearSolve` will return an error or an error message will be printed by the main function below indicating that the number of relations is insufficient (and hence that a higher value should be passed to `addrels`).

This function can either be called from the `IndexCalculus` function below or it can be used in an autonomous way. The latter has the advantage that it separates the precomputation from the final stage of the algorithm and hence allows the computation of several discrete logarithms in the same group without having to recompute the factor base primes.

```
> IndexCalculusPrecomp := proc(p::posint, g::posint, {verbose:=false, addrels:=10})
  local B, k, fb, i, rels, x, s, M, fblogs;
  if `not`(isprime(p)) then
    error "%1 is not prime", p
  elif numtheory:-order(g, p) <> p-1 then
    error "%1 is not a primitive root modulo %2", g, p
  end if;
  RandomTools:-MersenneTwister:-SetState();
  B := ceil(evalf(sqrt(exp(sqrt(ln(p)*ln(ln(p)))))/sqrt(2)));
  k := numtheory:-pi(B);
  fb := [seq(ithprime(i), i = 1 .. k)];
  rels := [];
```

```
  i := 0;
  while i < k + addrels do
    x := RandomTools:-MersenneTwister:-GenerateInteger(range = 1 .. p-1);
    rels := [op(rels), ICTrialDivision(p, g, x, fb)];
    i := nops(rels)
  end do;
  M := LinearAlgebra:-Modular:-Mod(p-1, rels, integer);
  if verbose then
    s := interface(rtablesize = k + addrels + 1);
    printf("Solving the linear system modulo %d of augmented matrix:\n", p-1);
    print(M)
  end if;
  LinearAlgebra:-Modular:-LinearSolve(p-1, M, 1);
  fblogs := LinearAlgebra:-Modular:-Copy(p-1, M, 1 .. k, k+1);
  if verbose then
    printf("Discrete logs of the factor base primes:\n");
    print(LinearAlgebra:-Modular:-Copy(p-1, fblogs, 'transpose'));
    interface(rtablesize = s)
  end if;
  [fb, fblogs]
  end proc:
```

Next we give the main function of the algorithm, namely the function `IndexCalculus` below. This function has three required input parameters that serve to specify the prime p, the primitive root g, and the element $a \in \mathbb{Z}_p^*$ whose discrete logarithm is to be computed. The optional input parameter, `add` is used to specify the number of relations to be computed, which will be equal to the size of the factor base plus the value of `add` (this value will be subsequently passed to the function `IndexCalculusPrecomp`). The remaining optional input parameters are the keyword parameters `verbose` and `pre`. The purpose of the first is to tell the function whether to print some additional information or not (with default value `false`). Finally, `pre` is used to feed the function with the result of the precomputation. The default value calls `IndexCalculusPrecomp` but, if the result of calling this function is already known, it can be passed directly to `pre` without having to redo the precomputation. The output is $\log_g a \in \mathbb{Z}_{p-1}$.

```
> IndexCalculus := proc(p::posint, g::posint, a::posint, add:=10,
                    {verbose:=false, pre:=IndexCalculusPrecomp(p, g, addrels=add)})
  uses LinearAlgebra:-Modular;
  local fb, fblogs, k, rels, s, r, m;
  RandomTools:-MersenneTwister:-SetState();
  fb := pre[1];
  fblogs := pre[2];
  k := nops(fb);
  if fblogs[k] = 0 then
    error "Insufficient number of relations"
  end if;
  rels := [];
  while nops(rels) = 0 do
    r := RandomTools:-MersenneTwister:-GenerateInteger(range = 2 .. p-2);
    rels := [op(rels), ICTrialDivision(p, g, r, fb, true, a)]
  end do;
  if verbose then
    printf("(%d^%d)*%d mod %d has exponent vector over the factor base:\n",g,r,a,p);
    s := interface(rtablesize = k + 1);
    print(op(rels));
    interface(rtablesize = s)
  end if;
  m := Multiply(p-1, Mod(p-1, op(rels), integer), fblogs);
```

```
  if verbose then
    printf("The discrete logarithm is:\n")
  end if;
  (m-r) mod (p-1)
  end proc:
```

The functions `IndexCalculus` and `IndexCalculusPrecomp` require that p is a prime and g is a generator of $\mathbb{Z}_p^*$, otherwise they will return an error. The method can be adapted to work in cyclic subgroups of $\mathbb{Z}_p^*$ which are distinct from $\mathbb{Z}_p^*$ itself but this requires use of a factor base formed by elements of this cyclic subgroup and the resulting algorithm is less efficient. There is a variant of the algorithm in which, instead of computing first the discrete logarithms of the primes and then $\log_g a$, they are all computed at once by solving the linear system obtained by collecting together the ordinary relations and the special relation. This variant may allow $\log_g a$ to be found even if not all discrete logs of primes in the factor base are found, since only those corresponding to primes in the final relation are needed.

Example 6.30 We present a detailed example to show how `IndexCalculus` computes a discrete logarithm. We define a prime p and a primitive root g:

```
> p := 17279730901583:
  isprime(p);
  ifactor(p-1);
  g := numtheory:-primroot(1850226918917, p);
                                    true
                              (2) (8639865450791)
                                 1850226918918
```

Note that p is actually a safe prime, although this is not necessary for the method to work. We can separately display the smoothness bound and the factor base for p:

```
> ceil(evalf(sqrt(exp(sqrt(ln(p)*ln(ln(p)))))/sqrt(2)));
  [seq(ithprime(i), i = 1 .. numtheory:-pi(%))];
                                     117
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
  59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113]
```

We see that the factor base consists of the primes ≤ 117, i.e., the first 30 primes. Suppose that we want to compute the discrete logarithm to the base g of $9876543210 \in \mathbb{Z}_p^*$. We will separate the precomputation step from the computation of this discrete logarithm, so we first compute:

```
> pr := IndexCalculusPrecomp(p, g, verbose = true):

Solving the linear system modulo 17279730901582 of augmented matrix:
```

$$\begin{bmatrix}
1 & 4 & 3 & 0 & 2 & 2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 11908534083343 \\
1 & 0 & 4 & 0 & 1 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8634527510415 \\
3 & 12 & 1 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2074798658290 \\
3 & 2 & 2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 6985303737832 \\
1 & 3 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 8812005166391 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 7292473383297 \\
1 & 6 & 0 & 0 & 2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8473728166773 \\
0 & 2 & 1 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8251324123939 \\
7 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 3321518619199 \\
2 & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 16344515442851 \\
1 & 0 & 0 & 2 & 0 & 2 & 0 & 1 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 13111148356376 \\
4 & 1 & 3 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 16601326113336 \\
7 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1456434246950 \\
5 & 4 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 404027629520 \\
3 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 13065668932751 \\
0 & 5 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 7525531865225 \\
5 & 3 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 4001162293292 \\
0 & 2 & 0 & 0 & 0 & 1 & 0 & 2 & 0 & 3 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3842569352563 \\
0 & 8 & 0 & 3 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 964914952665 \\
10 & 2 & 0 & 4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 17035585297150 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14458766030584 \\
0 & 2 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 14476141439409 \\
7 & 8 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 11647353909720 \\
2 & 3 & 0 & 2 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 9788123357331 \\
1 & 0 & 0 & 4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 11629589763307 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 & 13159446123783 \\
4 & 6 & 0 & 1 & 0 & 0 & 2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 182566550689 \\
1 & 1 & 0 & 0 & 0 & 1 & 2 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3247086793907 \\
5 & 2 & 3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 15800049899526 \\
2 & 0 & 2 & 1 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 9854413260328 \\
8 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 4381398209006 \\
0 & 5 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1390250884705 \\
5 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 15629049208190 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 6147525811383 \\
1 & 0 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16120430259376 \\
2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 5390342243078 \\
0 & 6 & 4 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 290365350528 \\
1 & 0 & 0 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 11847489523301 \\
1 & 5 & 2 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 15363122732369 \\
7 & 1 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16451950295874
\end{bmatrix}$$

```
Discrete logs of the factor base primes:

[7426253006290, 4492127725870, 11192451597521, 17081013481576, 15896098548408,
8053973676115, 14246651143093, 4166997212594, 8724619318636,4 469420019618,
5773327701243, 11127950744169, 7948673165404, 8646993757009, 1650773234392,
12735188425689, 10375910229785, 7580738887351, 8423060253370, 12616673387636,
4883524291858, 16066730563529, 15730446834125, 6016972182391, 10817894383613,
15892873589102, 2696216508825, 9069388948579, 14060808948759, 1891579519393]
```

These data may be interpreted as follows. The rows of the matrix contain the relations that will serve to compute the discrete logarithms of the 30 primes in the factor base. For example, the third row just means that:

$$g^{2074798658290} \bmod p = 2^3 \cdot 3^{12} \cdot 5 \cdot 7^4 \cdot 113$$

We may use Maple's `ifactor` to check that this is indeed the case:

```
> ifactor(Power(g, 2074798658290) mod p);
                          (2)^3 (3)^12 (5) (7)^4 (113)
```

This relation, in turn, gives the following equation that the discrete logarithms of the involved primes must satisfy:

$$3\log_g 2 + 12\log_g 3 + \log_g 5 + 4\log_g 7 + \log_g 113 \equiv 2074798658290 \pmod{p-1},$$

and each row of the matrix defines one such equation, where the first 30 columns of the matrix correspond to the discrete logarithms of the primes in the factor base (in their natural order) and the last column gives the 'independent terms'. Thus this matrix defines a linear system of 40 equations in 30 unknowns over $\mathbb{Z}_{p-1}$ whose solution is the vector of $\mathbb{Z}_{p-1}^{30}$ whose components are the discrete logarithms of the primes in the factor base. The system is solved by Maple's function `LinearAlgebra:-Modular:-LinearSolve` and the result obtained in this case is the vector printed above, just after the matrix (the solution of the system is given by Maple in the form of a column vector but we display it as a row vector for convenience). For example, since the last component of the vector is 1891579519393 and the last prime in the factor base is 113, we know that $\log_g 113 = 1891579519393$ and we may use Maple to check that this is indeed the case:

```
> Power(g, 1891579519393) mod p;
                                    113
```

We are now ready to compute $\log_g 9876543210$, which was our original goal. The output of the function `IndexCalculusPrecomp` was stored in a global variable called `pr` and we feed it to `IndexCalculus` through the keyword parameter `pre`. The result is the following:

```
> IndexCalculus(p, g, 9876543210, verbose = true, pre = pr);
(1850226918918^10951180283336)*9876543210 mod 17279730901583 has exponent vector
over the factor base:
          [4,1,1,1,1,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0]
The discrete logarithm is:
                                 11628971657383
```

The special relation that produced the discrete logarithm is displayed in the first two printed lines after the command. These lines inform us that

$$g^{10951180283336} \cdot 9876543210 \bmod p = 2^4 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 29 \cdot 37 \cdot 43 \cdot 61 \cdot 79$$

and, although not necessary, we can again check that this is indeed the case:

```
> ifactor(modp(9876543210*Power(g, 10951180283336), p));
               (2)^4 (3) (5) (7) (11) (29) (37) (43) (61) (79)
```

Once this final relation is obtained, the discrete logarithm 11628971657383 follows easily and it was given as the output of the `IndexCalculus` function. The result can also be obtained from this relation by direct manipulation and we explicitly show how it is obtained. Taking logarithms, the final relation translates into the following equation in $\mathbb{Z}_{p-1}$:

$$\log_g 9876543210 = (-10951180283336 + 4\log_g 2 + \log_g 3 + \log_g 5 + \log_g 7 + \log_g 11 + \log_g 29 + \log_g 37 + \log_g 43 + \log_g 61 + \log_g 79) \bmod (p-1).$$

Since the discrete logarithms of the factor base primes are now known, this identity can be used to find $\log_g 9876543210$ as follows:

```
> (-10951180283336+4*7426253006290+4492127725870+11192451597521+17081013481576+
15896098548408+4469420019618+11127950744169+8646993757009+7580738887351+
16066730563529) mod (p-1);
                                11628971657383
```

We see that the result obtained is the same as the one provided by the function `IndexCalculus`. As a final check, let us use binary exponentiation to see that the result is correct:

```
> Power(g, 11628971657383) mod p;
                                  9876543210
```

Of course, this discrete logarithm may also be computed in one pass by calling the function `IndexCalculus` with default values as follows:

```
> IndexCalculus(p, g, 9876543210);
                                11628971657383
```

Exercise 6.35 Write variants of the function `IndexCalculus` (and the associated auxiliary functions) with each one of the following changes:

(i) Replace trial division by the elliptic curve method (implemented in `ifactor` with the option 'lenstra').
(ii) Solve the system involving both the discrete logarithms of the primes and the discrete logarithm being computed, after the final relation is found.

Exercise 6.36 Do an experimental study by timing the function `IndexCalculus` on primes of different sizes in order to check how well the running time agrees with the expected subexponential time.

6.5.7 Extensions of the Index Calculus Method

We have discussed the index calculus method for the group $\mathbb{Z}_p^*$, where p is prime, but this method can be extended to other classes of cyclic groups as well. The most immediate generalization is for the multiplicative group $\mathbb{F}_{p^n}^*$ of the nonzero elements of a finite field $\mathbb{F}_{p^n}$, of which $\mathbb{Z}_p^* = \mathbb{F}_p^*$ is the particular case corresponding to $n = 1$. It should be noted, however, that $\mathbb{F}_{p^n}^*$ is quite different from $\mathbb{F}_p^*$ when $n > 1$. In this case there is no direct relation between $\mathbb{F}_{p^n}$ and $\mathbb{Z}_{p^n}$ and, although we have seen a way to represent the elements of the former as integers, the field operations between

these elements are not directly related to the operations in $\mathbb{Z}$, so that it is not possible to use a factor base consisting of primes in $\mathbb{Z}$ as we did in the case of $\mathbb{Z}_p^*$. However, as we saw in Sect. 2.8, the finite field $\mathbb{F}_{p^n}$ is constructed from the polynomial ring $\mathbb{Z}_p[x]$ in much the same way as $\mathbb{Z}_p$ is constructed from $\mathbb{Z}$, as it is again a quotient ring modulo an irreducible element. On the other hand, we also remarked that the ring $\mathbb{Z}_p[x]$ shares many properties with the ring $\mathbb{Z}$ and, for example, both are 'unique factorization domains' in the sense that any nonzero non-unit of the ring factors, in an essentially unique way, as a product of irreducible elements (the prime numbers in the case of $\mathbb{Z}$ and the irreducible polynomials in the case of $\mathbb{Z}_p[x]$).[11] Moreover, the elements of $\mathbb{F}_{p^n}$ can be represented by elements of $\mathbb{Z}_p[x]$, namely, by the polynomials of degree $< n$. Thus the ideas behind the index calculus algorithm for $\mathbb{Z}_p^*$ generalize to this new setting just by looking at the elements of $\mathbb{F}_{p^n}$ as polynomials in $\mathbb{Z}_p[x]$ and by trying to factor them as products of irreducible polynomials.

An important requirement for this idea to work is that there should be a suitable concept of 'smoothness' also in this more general framework, and indeed there is. Although the ring $\mathbb{Z}_p[x]$ is not ordered like $\mathbb{Z}$ is, there is a concept of 'size' given by the degree of a polynomial and so we may say that a polynomial is d-smooth when each of its irreducible factors has degree $\leq d$. This concept of smoothness is not useful when n is small, for then the possible choices of d are severely restricted, but it works well when n is large, in which case the same procedure as in the prime case gives a subexponential index calculus algorithm also for $\mathbb{F}_{p^n}$. When $n > 1$ but n is small, the method can also be successfully implemented using another way of representing the elements of $\mathbb{F}_{p^n}$ based on the theory of algebraic number fields as in the case of the NFS factorization algorithm (see [60, 6.4.2] for more details).

The basic index calculus method we have described and implemented is inspired by the quadratic sieve factoring algorithm. But there are other variants of the index calculus method that are inspired by the NFS factoring algorithm (and also called NFS) and, as in the factorization case, are more powerful than the integer-based methods. NFS for discrete logarithms can be used in any finite field and has a heuristic subexponential running time similar to the NFS factoring algorithm. For an introduction to these and other related methods (such as those based on function fields), we refer to [106, Chap. 15].

The index calculus methods and, in particular, the NFS variant, are much faster than the generic DL algorithms we have mentioned. In order to give an idea of how far this algorithm has gone in practice, we shall briefly mention the current record for a discrete logarithm in $\mathbb{Z}_p^*$, which was established in 2007 using NFS [112]. In this case, p is a 530-bit, 160-digit safe prime, in contrast with the already mentioned record for the generic methods which was established in a group whose order is a 112-bit prime. This clearly shows the power advantage of NFS over the generic methods but then, of course, NFS was not applicable in the elliptic curve group where the generic record was established. . .

[11] In fact, both rings share an even stronger property, namely both are 'Euclidean domains' since they have a 'Euclidean function' which provides a division algorithm and a Euclidean algorithm.

6.5.8 Final Remarks on the Discrete Logarithm Problem

We have given an overview of the most important algorithms to solve the discrete logarithm problem. This problem appears to be difficult and hence is a good source for building cryptographic algorithms. Some conclusions from the preceding discussion can be summarized as follows:

1. There are two broad classes of algorithms for solving the DL problem: the generic algorithms which work in any cyclic group and the algorithms for specific groups. The former are exponential while the index calculus algorithms for the cyclic (sub)groups of finite fields are subexponential.
2. The index calculus algorithms have running times similar to the subexponential factoring algorithms and hence the cryptographic schemes based on the DL problem over finite fields do not have any substantial advantage over schemes based on the difficulty of the integer factorization problem. This fact led to the search for other groups where only generic algorithms were applicable.
3. The elliptic curve groups that we study in Chap. 11 are the best example of groups where only generic algorithms are known for the DL problem. There are specific classes of elliptic curves where the index calculus method can be applied but this is not the case in general and the greater difficulty of the DL problem in this setting makes it possible to use smaller keys.
4. Similarly to the schemes based on the difficulty of factoring integers, all the DL-based cryptographic schemes, including those that use elliptic curve groups, will be broken if functional quantum computers are built, since they can solve the DL problem in polynomial time. We refer to [22] for a detailed examination of the impact of quantum computers in cryptography.

Chapter 7
Introduction to Public-Key Cryptography: The Diffie–Hellman Protocol

In this chapter, we present an introduction to the ideas underlying public-key cryptography and we illustrate some of them with the very first public-key protocol. This protocol is the Diffie–Hellman key agreement, which was proposed in [67] and heralded the birth of public-key cryptography.

7.1 From Private-Key to Public-Key Cryptography

The private-key encryption schemes we have studied in preceding chapters have some drawbacks when used in the context of modern communication networks. The main inconvenience is that the communicating parties, Alice and Bob, must share a secret key and for this they need a secure channel. This channel may adopt different formats and it might be, for example, a physical meeting between Alice and Bob or a trusted courier. When cryptography was the domain of government agencies and the military—as happened until the last quarter of the twentieth century—this was not a big problem because these organizations had the capacity to use these channels. But for average users of modern communication systems, secure channels are not readily available—in fact, it may be said that these channels are notoriously scarce. This is the problem Alice and Bob face if, for example, they live in different continents and want to communicate by e-mail. This key exchange problem becomes even more important if we consider a large group of people who want to communicate through an open network like the Internet. If the number of users of the communication network is n and any two of them exchange a secret key for private communication, then $n(n-1)/2$ keys must be exchanged and securely stored, and each user must keep $n-1$ keys. If one takes into account that, for security reasons, the keys should be renewed periodically, it is clear that both the key distribution and key management problems soon become intractable.

These problems may be alleviated by means of a key center that acts as a sort of intermediary between users. In its simplest incarnation, the key center works as

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_7, © Springer-Verlag Berlin Heidelberg 2013

follows. Each user exchanges a secret key with the key center. Then, if Alice wants to send a message to Bob, she encrypts it with her secret key and sends it to the key center, together with a message that tells the center that the encrypted message is addressed to Bob. The center decrypts the message using Alice's key, encrypts it with Bob's key and sends the resulting ciphertext to Bob. This way the number of key exchanges for n users is reduced to n and each user needs to store only one secret key. While this approach makes private-key cryptography viable in a single organization where there is one identity trusted by everyone, it also has some disadvantages. The most obvious one is that all the users must trust the key center, something that is not attainable in many situations. Another drawback is that secure channels are still necessary for each user to exchange its key with the center and, even if these problems can be solved, there are still other concerns such as, for example, the fact that a successful attack against the center would completely break the system.

Another disadvantage of private-key cryptography is related to the fact that, as we have seen, authentication is one of the most important goals of modern cryptography and it can be argued that it is even more so in the context of open communication networks. We have seen that private-key cryptography has ways to ensure the authenticity and the integrity of messages through the use of MACs but it cannot provide *non-repudiation*, which means that if Alice sent a message she cannot later deny having sent it. To realize the importance of this feature consider, for example, a situation where Alice agreed to pay Bob a large amount of money in exchange for some service. If the message was encrypted and authenticated by means of a symmetric encryption scheme and a MAC, using a secret key shared by Alice and Bob, there is no way in which Bob could convince a third party (for example, a judge) that the message really came from Alice. Since Bob also knows the secret key, the message could have been forged by him and falsely attributed to Alice. What is lacking here is the digital analogue of a written *signature*.

These disadvantages of private-key cryptography were clearly perceived by Whitfield Diffie and Martin Hellman who, in 1976, published a paper entitled *"New Directions in Cryptography"*, [67], which was to have a tremendous influence on the future development of the discipline. The very first sentence in this paper is premonitory:

> *We stand today on the brink of a revolution in cryptography.*

It was precisely this work that started the revolution and in the second paragraph of the paper we can see why: in it, Diffie and Hellman point out the need to address the main disadvantages of public-key cryptography:

> *... such applications create a need for new types of cryptographic systems which minimize the necessity of secure key distribution channels and supply the equivalent of a written signature. At the same time, theoretical developments in information theory and computer science show promise of providing provably secure cryptosystems, changing this ancient art into a science.*

These were visionary words indeed! They not only clearly identified the goals of disposing with the need for private channels and providing digital signatures but they also realized the importance of obtaining cryptosystems that can be proven secure relative to hard computational problems, something that at the time was not

in the cryptographer's mindset. For all these reasons this work was revolutionary but there is another aspect that would be revolutionized as well: up to that time all relevant cryptography was in the hands of government agencies and the work of their cryptographers was classified as secret. With the birth of public-key cryptography this changed and since then cryptography has been openly discussed in the academic world.

There is a historical episode that illustrates the importance of this last point and, at the same time, completes this sketch of the origins of public-key cryptography. As was revealed in 1997, some cryptographers working for the British GCHQ (Government Communications Headquarters) in the 1970s discovered the idea of public-key cryptography and some of its applications before the publication of the Diffie–Hellman paper. In particular, James Ellis put forward the idea of "non-secret encryption" (i.e., public-key encryption), Clifford Cocks discovered a public-key encryption scheme that was essentially a variant of RSA (which was introduced a few years later) and Malcolm Williamson discovered a method to exchange secret keys through open channels. Had they been made public at the time, these discoveries would have changed the history of cryptography. But it seems that GCHQ did not make much use of them and, since this work remained secret until 1997, it did not have any impact on the development of cryptography as a science, so that this episode remains only a historical anecdote.

The crucial idea underlying the concept of a public-key encryption scheme is that in such a scheme there are two keys instead of one. One of these keys is public and is used by senders to encrypt messages, the other is private and is used by receivers to decrypt the ciphertexts. More concretely, each user of the scheme generates a pair of keys (pk, sk), where pk is called the *public key* and sk is the *private key*. The public keys of all users are listed in a public directory and are used to encrypt messages, the private keys are used to decrypt the ciphertexts and only each user knows her private key. Thus if Bob wants to send a message to Alice, he gets Alice's public key from the directory and uses it to encrypt the message, obtaining a ciphertext which he sends to Alice. Then Alice uses her private key to recover the message from the ciphertext.

For a public-key encryption scheme to be secure, it is essential that the private key of a user cannot be recovered by an adversary that knows the public key and this leads to the requirement that it should be computationally infeasible to find the private key from the public key; because of the inherent asymmetry between the capability to encrypt and the capability to decrypt, these schemes are also called asymmetric encryption schemes. We will formally define these schemes and their security properties later on but for now we remark that public-key encryption schemes permit private communication through public channels and, as Diffie and Hellman already noticed, solve the problem of key distribution since secret key exchange is no longer necessary. On the other hand, key storage needs are also drastically reduced as each party only needs to securely store her own private key. The directories containing the public keys must be publicly readable but protected from unauthorized writing by active adversaries. As we will see later, it is necessary to use *authenticated*

distribution channels for public keys in order to prevent an adversary from replacing the public key of a user.

Besides the concept of public-key encryption, Diffie and Hellman also introduced two other public-key primitives. One of them is *digital signatures*, which are the public-key analogues of MACs but, as we already mentioned, have important advantages and make it possible, for example, to electronically sign contracts. The other primitive is *key agreement* through public channels, which was also discovered independently by Ralph Merkle and is a protocol that enables two parties who do not share any information to generate a shared secret key by communicating over an insecure authenticated channel. This was a powerful idea as nothing even remotely similar was publicly known at the time and it was the only one of these three primitives for which Diffie and Hellman offered a concrete construction called the *Diffie–Hellman key agreement protocol*, which we study in the next section.

A *key agreement protocol* Π between two parties, say, Alice and Bob, proceeds by Alice and Bob taking turns in sending messages to each other and terminates with both parties obtaining a common value k which will be their shared secret key. Abstractly, Π may be modeled as a pair of probabilistic machines A, B that, starting with a security parameter 1^n run the protocol as indicated. The *protocol transcript* t_Π is the sequence of messages exchanged by A and B which, since they implement PPT algorithms, will be different each time the protocol is run. The object of the protocol is to prevent an adversary $\mathcal{A}$ that sees the transcript from learning information about the secret key k. Thus the security of a key agreement protocol Π may be defined through the following experiment:

Definition 7.1 The *key agreement experiment* $\mathbf{KA}^{eav}_{\mathcal{A},\Pi}(n)$, where Π is a key agreement protocol, $\mathcal{A}$ a PPT adversary, and n any value of the security parameter, is the following:

1. The parties A, B, execute the protocol on input 1^n and they output a key k.
2. A random bit $b \leftarrow \{0, 1\}$ is chosen. A key k' is defined by setting $k' := k$ if $b = 0$ and choosing it at random, $k' \leftarrow \{0, 1\}^n$, if $b = 1$.
3. $\mathcal{A}$ is given t_Π and k' and outputs a bit b'.
4. The output of the experiment is defined to be 1 if $b' = b$ and 0 otherwise. If $\mathbf{KA}^{eav}_{\mathcal{A},\Pi}(n) = 1$ then we say that $\mathcal{A}$ *succeeded*.

Definition 7.2 The key agreement protocol Π is said to be *secure in the presence of an eavesdropper* if, for every PPT adversary $\mathcal{A}$, there exists a negligible function $\mathtt{negl}$ such that $\Pr(\mathbf{KA}^{eav}_{\mathcal{A},\Pi}(n) = 1) \leq 1/2 + \mathtt{negl}(n)$. The advantage of $\mathcal{A}$ in the experiment is defined as $|\Pr(\mathbf{KA}^{eav}_{\mathcal{A},\Pi}(n) = 1) - 1/2|$.

The definition says that an adversary eavesdropping the key agreement protocol cannot distinguish the key k from a randomly generated key. This is not a very strong security concept because it does not provide assurance of the identities of the parties, so that a protocol which is secure in this sense may be vulnerable, as we shall soon see, to certain active attacks.

7.2 The Diffie–Hellman Key Agreement

In this section we study the Diffie–Hellman key agreement or Diffie–Hellman protocol (we will often write *DH protocol* for short) and its security properties.

7.2.1 The DH Protocol and the DH Problems

The DH protocol may be described as follows. We assume that the parties, Alice and Bob, have no prior shared information and have access to a channel that provides authentication and integrity but may otherwise be insecure.

Definition 7.3 *Diffie–Hellman key agreement.*
As a first step, Alice and Bob agree on a cyclic group G of order t, equipped with an efficiently computable group law, and also agree on a generator g of G. These parameters are public and may be generated by one of the parties and sent to the other through the public channel. Then the protocol runs as follows:

1. Alice chooses $x \leftarrow \mathbb{Z}_t$ uniformly at random, computes $u = g^x \in G$ and sends u to Bob.
2. Bob chooses $y \leftarrow \mathbb{Z}_t$ uniformly at random, computes $v = g^y \in G$ and sends v to Alice.
3. Alice computes $k = v^x \in G$.
4. Bob computes $k = u^y \in G$.

After the protocol is run, Alice and Bob agree on using the common key k. The protocol is correct because $v^x = (g^y)^x = g^{xy} = (g^x)^y = u^y$ and this common value is the key k.

The most important question now is whether this protocol is secure. The goal, according to Definition 7.2 is that an eavesdropper, Eve, cannot distinguish the key from a random string of the same length. The natural attack that Eve may use to try to recover k is to find x (or, alternatively, y) for then, since v is known, she just computes $k = v^x$. But computing x given u is just an instance of the discrete logarithm problem on the group G, so this leads to the requirement that computing discrete logarithms (see Example 3.2 and the discussion of the algorithms for the DL problem in Sect. 6.5) should be hard for the family of groups considered. In other words, the family of discrete exponential functions corresponding to these groups should be one-way in the sense of Definition 3.10 and Example 3.2. Solving the discrete logarithm problem in G will break the protocol but this is by no means the only attack conceivable and it is possible that there may be other ways of obtaining information about the key, or even of recovering k, without finding first x or y. Thus hardness of the discrete logarithm problem is only a necessary condition for the security of the protocol.

The possibility of recovering the key without computing discrete logarithms leads by abstraction to the following problem:

The computational Diffie–Hellman problem (CDH).
Given a cyclic group G and a generator $g \in G$ define, for each pair of elements $u, v \in G$:

$$\mathrm{DH}_g(u, v) = g^{(\log_g u)(\log_g v)}.$$

The problem is then to compute $\mathrm{DH}_g(u, v)$ for randomly chosen u and v.

Exercise 7.1 Give a precise definition of the hardness of the CDH problem by defining an experiment in which an adversary can only succeed with negligible probability.

It is clear that being able to solve CDH is equivalent to being able to recover the key in the Diffie–Hellman protocol. It is clear also that the discrete logarithm problem is at least as difficult as the CDH problem or, in other words, if the discrete logarithm problem is easy so is CDH. However, it is not known whether the converse holds, i.e., whether hardness of the discrete logarithm problem implies that CDH is hard. It seems possible that there is some way of solving CDH without computing discrete logarithms but so far no such method has been found and hence it seems plausible that, whether or not equivalent to the discrete logarithm problem, the CDH problem is hard (in appropriate groups). This hypothesis is known as the *computational Diffie–Hellman assumption* and, as we have mentioned, it entails that a computationally bounded adversary cannot recover the key k shared by the parties in the Diffie–Hellman protocol.

But this is, of course, still insufficient to guarantee that the DH protocol is secure in the sense of Definition 7.2. In practical situations, we have to bear in mind the possibility of attacks that do not recover the key but only partial information about it, in which case the protocol is also insecure (in Example 7.1 below, this situation arises even though the computational Diffie–Hellman assumption is believed to hold). In an extreme case, it might happen that the partial information leaked might be enough to reduce the possible keys to a tiny fraction of elements of G, leading to a possible recovery by means of a brute-force attack.

If we look at the DH problem from the point of view of Definition 7.2, we are naturally led to the following problem:

The decisional Diffie–Hellman problem (DDH).
DDH is, roughly speaking, to distinguish $\mathrm{DH}_g(u, v)$ from a random element of G for randomly chosen u and v or, a little more explicitly, to distinguish between the two distributions $(u, v, \mathrm{DH}_g(u, v))$ and (u, v, w), when u, v, w are randomly chosen in G or, equivalently, to distinguish between the distributions (g^x, g^y, g^{xy}) and (g^x, g^y, g^z) when x, y, z are randomly chosen in $\mathbb{Z}_t$, where t is the order of G. Thus we see that what is required for the DH protocol to be secure is that the DDH problem is hard (also known as the *decisional Diffie–Hellman assumption*). If the CDH problem is easy then certainly so is the DDH problem but the converse is not believed to hold.

The DDH problem can be more formally defined relative to a group generating algorithm $\mathbf{Gen}_G$ that, on input 1^n, outputs a (specification of a) cyclic group G

of order t with $\text{len}(t) = n$ and a generator $g \in G$, by considering the following experiment:

Definition 7.4 The *DDH experiment* $\mathbf{DDH}_{\mathcal{A},\mathbf{Gen}_G}(n)$ is the following:

1. $\mathbf{Gen}_G$ is run on input 1^n to obtain (G, g) where G is a cyclic group of order t, with $\text{len}(t) = n$ and $G = \langle g \rangle$.
2. A random bit $b \leftarrow \{0, 1\}$ and random elements $x, y \leftarrow \mathbb{Z}_t$ are chosen and an element $h \in G$ is defined by $h := g^{xy}$ if $b = 0$ and $h \leftarrow G$ (h is a random element of G) if $b = 1$.
3. $\mathcal{A}$ is given (G, g, g^x, g^y, h) and outputs a bit b'.
4. The output of the experiment is defined to be 1 if $b' = b$ and 0 otherwise. If $\mathbf{DDH}_{\mathcal{A},\mathbf{Gen}_G}(n) = 1$ then we say that $\mathcal{A}$ succeeded.

This leads to the following hardness definition:

Definition 7.5 We say that *the DDH problem is hard relative to* $\mathbf{Gen}_G$ if, for every PPT algorithm $\mathcal{A}$, there exists a negligible function $\texttt{negl}$ such that

$$\Pr(\mathbf{DDH}_{\mathcal{A},\mathbf{Gen}_G}(n) = 1) \leq 1/2 + \texttt{negl}(n).$$

In Definition 7.2, security of a key agreement protocol was defined by requiring that an eavesdropper has only negligible probability of distinguishing the key k obtained from the protocol from a random string of the same length. This definition can be modified in the setting of the DH protocol by requiring only that the key output after the parties run the protocol (which is an element of the cyclic group G used in the DH protocol) be indistinguishable to an eavesdropper from a random element from G (except with negligible probability). Then the following is easily proved and we leave the details as an exercise:

Exercise 7.2 Prove that if the DDH problem is hard relative to $\mathbf{Gen}_G$, then the DH protocol is secure in the sense that the shared key obtained is indistinguishable from a random element of G.

Example 7.1 We show that the DDH problem may be easy even if the CDH problem is supposed to be hard, showing that the DH protocol may be insecure even if recovering the key is hard. Let p be a prime number, $G = \mathbb{Z}_p^*$ and g a primitive root modulo p. Then the DDH problem is not hard (more precisely, we can say that the DDH problem is not hard relative to the generating algorithm that, on input 1^n, picks a random n-bit prime and outputs the pair $(\mathbb{Z}_p^*, g)$, where g is a primitive root modulo p). Indeed, the generator g is, as we have seen, a quadratic non-residue, so that its Legendre symbol is $\left(\frac{g}{p}\right) = -1$. Moreover, given $u = g^{\log_g u}$ and $v = g^{\log_g v}$ we can efficiently compute $\left(\frac{u}{p}\right)$ and $\left(\frac{v}{p}\right)$ using Algorithm 2.7. Then we know that $\left(\frac{u}{p}\right) = \left(\frac{g^{\log_g u}}{p}\right) = \left(\frac{g}{p}\right)^{\log_g u} = (-1)^{\log_g u}$ and, similarly, $\left(\frac{v}{p}\right) = (-1)^{\log_g v}$. This gives the parity (or, equivalently, the least significant bit) of both $\log_g u$ and $\log_g v$ and hence the

parity of their product $(\log_g u)(\log_g v)$ allowing the easy computation of the Legendre symbol $\left(\frac{\mathrm{DH}_g(u,v)}{p}\right) = \left(\frac{g^{(\log_g u)(\log_g v)}}{p}\right) = (-1)^{(\log_g u)(\log_g v)}$. The knowledge of its Legendre symbol serves to distinguish $\mathrm{DH}_g(u, v)$ from a random element of G because the adversary, once given (G, g, u, v, h), computes $\left(\frac{\mathrm{DH}_g(u,v)}{p}\right) = \left(\frac{u}{p}\right)\left(\frac{v}{p}\right)$ and $\left(\frac{h}{p}\right)$. If both Legendre symbols have the same value, the adversary guesses that $h = \mathrm{DH}_g(u, v)$, otherwise it guesses that h is random. Its success probability is, by Proposition 2.1 (i), $\Pr(\mathbf{DDH}_{\mathcal{A},\mathbf{Gen}_G}(n) = 1) = \Pr(\mathbf{DDH}_{\mathcal{A},\mathbf{Gen}_G}(n) = 1 | h \text{ is random}) \cdot \frac{1}{2} + \Pr(\mathbf{DDH}_{\mathcal{A},\mathbf{Gen}_G}(n) = 1 | h = \mathrm{DH}_g(u, v)) \cdot \frac{1}{2} = \frac{1}{2} \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = \frac{3}{4}$ and hence its advantage is $\frac{1}{4}$, which is non-negligible.

Despite the DDH problem being easy relative to the group generating algorithm considered here, both the discrete logarithm and the CDH problems are thought to be hard for the groups $\mathbb{Z}_p^*$ (we have seen in Chap. 6 that the best algorithms for the DL problem in these groups are subexponential). Thus, as stated above, the hardness of the CDH problem is insufficient to guarantee the security of the DH protocol.

The DL problem and its derivatives, the CDH and DDH problems, are especially useful for cryptography because of a characteristic that we now briefly describe. First we give an informal definition:

Definition 7.6 A computational problem is *random self-reducible* if there exists a polynomial-time algorithm that transforms any given instance of the problem into a uniformly distributed random instance such that the solution of the original instance can be obtained in polynomial time from the solution of the new instance.

Random self-reducibility means that the existence of an efficient algorithm to solve the problem in the average case implies the existence of an efficient algorithm to solve it in the worst case. In other words, if the problem is hard in the worst case then it is also hard on average, a feature which makes these problems especially useful for cryptography. The lack of this property is what makes, for example, the subset sum problem (given a set of positive integers and a target integer s, is there a subset whose sum is precisely equal to s?) inadequate for cryptographic use. This problem is $\mathcal{NP}$-complete and hence it is thought to be hard in the worst case but there are efficient algorithms to solve many instances of the problem. Thus all cryptographic schemes based on this problem, such as the original Merkle–Hellman knapsack cryptosystem, were eventually broken (a survey of these schemes is [150]).

Exercise 7.3 Prove that the DL problem on a finite cyclic group G is random self-reducible. (Hint: given an element $a \in G = \langle g \rangle$ whose logarithm to the base g is to be computed, choose $r \leftarrow \mathbb{Z}_t$, where $t = |G|$, and consider the element $ag^r \in G$.)

The CDH problem and the DDH problem are both well suited for cryptographic use in the sense just mentioned because both can be shown to be random self-reducible. For example, in the case of the former problem we have:

Proposition 7.1 *The CDH problem on a finite cyclic group is random self-reducible.*

Proof Suppose that we have a CDH instance on $G = \langle g \rangle$, where $|G| = t$. This instance is given by two elements $u, v \in G$ and to solve it we have to compute $g^{(\log_g u)(\log_g v)}$. To obtain the reduction, we choose uniformly at random and independently two elements $r, s \leftarrow \mathbb{Z}_t$ and consider the instance of the CDH problem corresponding to the pair of elements $ug^r, vg^s \in G$. Observe that the pairs (ug^r, vg^s) obtained this way are uniformly distributed in $G \times G$, so that we have obtained a random instance of the problem in polynomial time. Now, if the solution for this new instance is the element $a \in G$, the solution to the original instance is

$$u^{-s} v^{-r} g^{-rs} a \in G.$$

This can easily be checked because, if we set $x = \log_g u$ and $y = \log_g v$, we have that $a = g^{(\log_g ug^r)(\log_g vg^s)} = g^{(x+r)(y+s)} = g^{xy} g^{(xs+ry+rs)} = g^{xy} u^s v^r g^{rs}$. Then we have that $u^{-s} v^{-r} g^{-rs} a = g^{xy}$ which gives the solution to the original instance of the problem. It is also clear that this solution can be obtained from a in polynomial time (assuming, as usual, that multiplication and inverses in G may be efficiently computed). □

7.2.2 Man-in-the-Middle Attacks

The preceding discussion of the security of the DH protocol focuses on the case of an eavesdropping adversary but one should also bear in mind the possibility of active attacks, where the adversary sends messages to one or both of the parties. To prevent active attacks, the messages exchanged between Alice and Bob should be authenticated using, for example, digital signatures as we will see later on. If this is not the case, then the DH protocol is vulnerable to a *man-in-the-middle-attack* which can be described as follows:

- Eve intercepts the message g^x that Alice sends to Bob and the message g^y that Bob sends to Alice.
- Then Eve chooses elements $x', y' \in \mathbb{Z}_t$ (where $t = |G|$) and sends $g^{y'}$ to Alice and $g^{x'}$ to Bob.
- At this point Alice believes that $g^{y'}$ came from Bob and Bob believes that $g^{x'}$ came from Alice. So they both follow the DH protocol and Alice computes $k_A = (g^{y'})^x$ while Bob computes $k_B = (g^{x'})^y$.
- Eve knows g^x, g^y, x', y' and so she also computes $k_A = (g^x)^{y'}$ and $k_B = (g^y)^{x'}$.
- Now Eve shares the key k_A with Alice and the key k_B with Bob, while Alice and Bob think that they share a key with each other.
- When Alice sends a message to Bob, Eve intercepts it, decrypts it and then re-encrypts it (or encrypts a different message) before sending it to Bob. This way Eve can read all messages exchanged between Alice and Bob and even alter them if she so chooses, without Alice and Bob noticing anything.

There is a subtler variant of the man-in-the-middle (or, rather, woman-in-the-middle in this example) attack that allows Eve to learn the common key that Alice and Bob share after the protocol and hence to passively eavesdrop all the messages exchanged once the key is established. It is based on the fact that, as seen in Sect. 6.5.4, computing discrete logarithms in a group whose order t is known to be a product of powers of small primes is easy because of the Pohlig–Hellman algorithm.

Suppose then that the order of G can be written in the form $t = t'r$, where r has only small prime factors (a trivial way to do this is to take $r = 1$). When Eve intercepts the group elements $u = g^x$ and $v = g^y$, she replaces them by $u' = u^{t'}$ and $v' = v^{t'}$, and sends these values to Bob and Alice, respectively. Following the DH protocol, Alice computes $k = (v')^x = g^{yt'x}$ and Bob computes $k = (u')^y = g^{xt'y}$, so that they share this key k. But Eve knows u' and $g^{t'}$ and hence she only has to solve the equation $u' = (g^{t'})^x$ to find x, which will allow her to find k as $k = (v')^x$. Now $g^{t'}$ is, by Proposition 2.5, an element of G of order r and hence the cyclic group $\langle g^{t'} \rangle$ has order r. The equation $u' = (g^{t'})^x$ lives in this group, whose order is a product of small prime factors, and the Pohlig–Hellman algorithm will allow Eve to compute the discrete logarithm of u' to the base $g^{t'}$. Observe, however, that this discrete logarithm will not, in general, be equal to x. Indeed, the discrete logarithm $l = \log_{g^{t'}} u'$ is defined as the smallest (positive) integer solution of the above equation, and is an integer $< r$ while x is, in general, much larger since we only know that $x < t$. However, it is clear that x is also a solution of the equation and hence $l = x \bmod r$. From this it follows that knowing l is sufficient for Eve to get hold of the key k because $k = (v')^x$ and, since the order of v' is, by Proposition 2.4, a divisor of r, we have that $(v')^l = (v')^x = k$. Thus, finding k essentially reduces to just solving an instance of the discrete logarithm problem in a group of order r and the Pohlig–Hellman algorithm allows Eve to do it. We use Maple to give a concrete demonstration of this attack in Sect. 7.2.4.

Note that if $r = 1$ then both u' and v' are equal to 1 and the key that Alice and Bob will share is $k = 1$. Even if Alice and Bob do not detect anything wrong when running the protocol, this is easy to prevent by just checking whether the element received is 1. More generally, Alice and Bob may prevent this attack by checking that the elements they receive do not have an order that is a product of small primes or, to eliminate this possibility altogether, they can use a group G whose order does not have small prime factors, for example, a group of prime order. This precaution is helpful not only against an active adversary but also against a passive one who tries to find the key by solving a discrete logarithm; next we will indicate how to find adequate groups for the protocol. We also remark that if the channel used for the DH protocol is authenticated, then active attacks like these would not be successful because Eve's manipulations would be detected.

7.2.3 Groups for the DH Protocol

In order to implement the DH protocol one must be careful when choosing the group G; here we will present some of the most common solutions. As we have seen, a

minimal requirement is that the discrete logarithm problem should be hard in G and, taking into account the Pohlig–Hellman algorithm, this excludes cyclic groups whose orders are products of powers of small primes. Hence an obvious possibility is to use groups of (suitably large) prime order. These groups have the additional advantage that finding a generator is very easy because any element $\neq 1$ has this property. Elliptic curve groups of prime order, which we study in Chap. 11, are very interesting in this regard because the discrete logarithm in these groups is thought to be especially difficult.

Another class of groups where the discrete logarithm problem is believed to be hard is given by the groups $\mathbb{Z}_p^*$, where p is a prime; more specifically, it is conjectured that the discrete logarithm problem is hard relative to an algorithm that, on input 1^n, chooses a random n-bit prime p and a generator g of $\mathbb{Z}_p^*$. But this does not mean that the groups $\mathbb{Z}_p^*$ are suitable for implementation of the DH protocol. One minor disadvantage is the fact that the order of these groups $p-1$ is not prime and, in addition to the factor 2, it may have other small prime factors and hence subgroups in which the discrete logarithm problem is not hard. But there is a much more serious problem: as we have seen in Example 7.1, the DDH problem is easy in these groups and hence the DH protocol is automatically insecure. As we shall see later, this also implies that these groups are also inadequate for other cryptographic applications whose security relies on the hardness of the DDH problem.

The DDH problem is easy in $\mathbb{Z}_p^*$ because some of the elements of this group are squares and some are non-squares, and they can easily be distinguished by computing Legendre symbols. Thus a natural idea is to consider only the squares, i.e., the quadratic residues modulo p which, as can easily be checked, form a subgroup of $\mathbb{Z}_p^*$. Moreover, since exactly one-half of the elements of $\mathbb{Z}_p^*$ are quadratic residues by Proposition 2.11, we see that this subgroup has order $(p-1)/2$. But this group may still have elements of small order that, as we have seen, can be troublesome when implementing the DH protocol. Thus one of the preferred solutions to implement the protocol is to ensure that this group has prime order, something that happens if we take p to be a *safe prime* which, as seen in Sect. 6.3, is a prime of the form $p = 2q + 1$, where q is also prime (q is then a Sophie Germain prime). Then the group of quadratic residues modulo p has prime order equal to q and so it has the advantages we have just mentioned. Note that this group is cyclic (in general, every subgroup of a cyclic group is cyclic but here this also follows in a straightforward way from the fact that the group has prime order, using Lagrange's theorem) and every element distinct from the identity is a generator. Thus finding a generator is very easy: just pick a random element $c \in \mathbb{Z}_p^*$ different from 1 and -1, and take the element $g = c^2 \in \mathcal{QR}_p$. Now g is an element of order q in $\mathbb{Z}_p^*$ and the protocol proceeds as indicated above picking random elements $x, y \in \mathbb{Z}_q$.

A variant of the "safe prime method" consists of taking a prime p of the form $p = rq + 1$ where q is also prime with, say, $q > p^{1/10}$, and finding an element g of order q in $\mathbb{Z}_p^*$, to work in the subgroup $\langle g \rangle$ of $\mathbb{Z}_p^*$. Since now q is smaller relative to the size of p, this approach is more efficient because it requires smaller exponents (as $x, y < q$), although p will usually be noticeably larger than in the safe prime case.

We are going to give an example of the DH protocol with Maple, using a safe prime of realistic size. Current estimates that take into account the effectiveness of the DLP algorithms studied in Chap. 6 suggest that the safe prime should have at least 1024 bits.

Example 7.2 Alice and Bob want to use the DH protocol to agree on a key. We will show how they can share a secret integer of size close to 1024 bits from which a secret key such as, for example, an AES key, can be derived.

First, Alice and Bob have to agree on a safe prime p and a generator g of the group $\mathcal{QR}_p$ of quadratic residues modulo p. They want a 1024-bit prime and they will use the one in Example 6.7 (we do not print it to save space):

```
> p := PseudoRandomSafePrime(29477983376442558187316294782580797080 8, 1024):
```

A check that p is indeed (very probably) a safe prime of the form $p = 2q + 1$ with q also prime and that p has 1024 bits is the following:

```
> q := (p-1)/2:
  isprime~([p, q]);
  ilog2(p)+1;
                                [true, true]
                                    1024
```

To complete the previous setup for the DH protocol, Alice and Bob must also agree on a generator of $\mathcal{QR}_p$, i.e., an element of order q of $\mathbb{Z}_p^*$, which is found by taking a pseudo-random element of the latter group and squaring it. They choose an integer in the range $0..2^{1024}$ and, after checking that it is less than $p-1$, they square it modulo p to obtain the generator g:

```
> with(RandomTools:-BlumBlumShub):
  BG := NewBitGenerator(279804548445444154496400548857816873066,
                        numbits = 1024, output = integer):
> h := BG():
  evalb(h < p-1);
                                    true
> g := h^2 mod p;
145017393555582210670936971933521206971919100525482708774150046600318249217515534 5\
  291586192540112759248861951366399283546117564217509211129155137292274523969665 79\
  119737011539218568736320700336207176945702045426008601024694258226892027212348 06\
  842822776070773143024073604117843001886701377466464472040934538369 6
```

An alternative way to find a generator of $\mathcal{QR}_p$ would be to just pick elements at random in $\mathbb{Z}_p^*$ until one is found which is different from 1 and has Legendre symbol equal to 1. We already know that this is the case for g (because it is a square):

```
> numtheory:-legendre(g, p);
                                    1
```

Alice and Bob agree on using p (which determines the group) and g, and this information can be exchanged through the public channel. Now Alice chooses at random an element of $\mathbb{Z}_q$ that she will use as exponent. In fact, she will settle by choosing it pseudo-randomly using Blum–Blum–Shub and a 128-bit random seed. She will choose an integer x of 1023 bits or less (since q has 1023 bits) and she will check that $x < q - 1$; if this turns out not to be the case she chooses another integer until one is found that satisfies this condition:

```
> BA := NewBitGenerator(19866860674210209345000602224585126090 5,
                        numbits = 1023, output = integer):
> x := BA();
  364305782800853632161805978904935195410152639629670149788563016746242600398350100 5\
    263440872306188483122968202731715832789866204742909826935354079411229632569916 00\
    146745679850976189132002850427098931025855986030509267891418132736354823632181 67\
    913800888478068684904318274980204118455515566599741812763978841 3412

> evalb(x < q-1);
                                        true
```

Then Alice computes $u = g^x \in \mathcal{QR}_p$ and sends it to Bob through the public channel. This element is obtained below but we do not print it:

```
> u := Power(g, x) mod p:
```

Similarly, Bob chooses a random (or, rather, pseudo-random in this case) $y \in \mathbb{Z}_q$, computes $v = g^y \in \mathcal{QR}_p$ and sends this element to Alice (again, we do not print v in order to save space):

```
> BB := NewBitGenerator(30216150765456976350476742100961740276 3,
                        numbits = 1023, output = integer):
> y := BB();
  271957257309228539472614584396070674849819970373549149291932877082184688801296053 5\
    405214051416746412539193279240905697901707839230319627792517345464471139293215 74\
    759142332622936945211078579705330658646708839144902414285222872923241414833460 46\
    068297947540353250235715294437776422169804829292077951399566979773

> evalb(y < q-1);
                                        true

> v := Power(g, y) mod p:
```

Now both Alice and Bob have the information necessary to compute the key k they are going to share. Bob computes:

```
> k := Power(u, y) mod p;
  449048773130508505243044109951494391312226346639420577846407574358894134634382782 2\
    282720579618419821110407070324524460342568727389301656914355722892425661746591 33\
    710687879191992664182781260959414973222863703917573343342683187868738575042383 24\
    103742048047270263709831203146386081644955581637071123001523188989
```

Alice computes:

```
> k := Power(v, x) mod p;
  449048773130508505243044109951494391312226346639420577846407574358894134634382782 2\
    282720579618419821110407070324524460342568727389301656914355722892425661746591 33\
    710687879191992664182781260959414973222863703917573343342683187868738575042383 24\
    103742048047270263709831203146386081644955581637071123001523188989
```

Alice and Bob now share the same element $k \in \mathcal{QR}_p$, whose length is 1022 bits:

```
> ilog2(k)+1;
                                        1022
```

Even knowing u and v, k appears to be hard to distinguish from a random element of $\mathcal{QR}_p$ but what Alice and Bob require is, say, a 128-bit key. To obtain it from k they should apply a *Key Derivation Function* (KDF) as discussed, for example, in [10] to which we refer for details.

Now, as mentioned before, if Eve observes the entire exchange and knows p, g, u, and v, the best passive attack she has to recover k seems to be to solve the discrete logarithm in $\mathcal{QR}_p$ and to compute x from the relation $g^x \bmod p = u$ (or, similarly, to compute y). This is thought to be infeasible with our present knowledge since, as mentioned in Chap. 6, the current record for a discrete logarithm in $\mathbb{Z}_p^*$

corresponds to a 530-bit safe prime. Moreover, the DDH problem is also thought to be hard in the groups $\mathcal{QR}_p$ so Eve will likely not have any information that allows her to distinguish k from a random element of $\mathcal{QR}_p$.

Exercise 7.4 Maple's built-in function `numtheory:-safeprime` takes as input a natural number i and generates (with high probability) the smallest safe prime greater than i. Write a Maple procedure that uses this function to generate a safe prime of specified length by choosing a random integer and then the safe prime `numtheory:-safeprime`(i). Perform an experiment that highlights the fact that this way of choosing safe primes is not uniform.

7.2.4 Attacking the Diffie–Hellman Protocol with Maple

We have seen that the Diffie–Hellman protocol is vulnerable to "man-in-the-middle" attacks and, in particular, to an attack of this type that allows the attacker to learn the key that the parties share and to passively eavesdrop the messages exchanged afterwards. A central component of this attack is the Pohlig–Hellman algorithm and, using it, we are now going to demonstrate the attack in detail with Maple.

So, suppose that Eve is going to mount a "woman-in-the-middle" attack against the DH protocol that Alice and Bob are using to establish their shared key. Alice and Bob will choose a large prime p and work within the group $\mathbb{Z}_p^*$. Thus they agree to use as modulus a 1121-bit prime p such that $p-1$ has a 1024-bit prime factor. This seems to provide sufficient security because it is estimated that the discrete logarithm problem is currently infeasible in a subgroup of $\mathbb{Z}_p^*$ whose order is a 1024-bit prime, even using powerful specific methods for $\mathbb{Z}_p^*$, such as the index calculus algorithms mentioned in Sect. 6.5.5. In fact, Alice and Bob, being aware that the use of the full group $\mathbb{Z}_p^*$ is not secure because the DDH problem is not hard in it, will use the subgroup $\mathcal{QR}_p$ of quadratic residues instead. This subgroup has index 2 in $\mathbb{Z}_p^*$ (in other words, exactly half of the elements of $\mathbb{Z}_p^*$ are quadratic residues) so its order is equal to $(p-1)/2$, which is a 1120-bit integer with a 1024-bit prime factor. Hence Alice and Bob feel secure against a combination of Pohlig–Hellman with an index calculus algorithm such as NFS. But Eve thinks otherwise ...

The prime that Alice and Bob are going to use is the following:

```
> p := 22084971198600004861550078572243353504451270023543010378998044625497413302 1\
    46147729248901772581705578563747416230011543484985865472529635048849685503969 1\
    32295691590560012089830390615407783608599996198770138402972674762868611393512 5\
    73365570049696055813450386176108631191771287723696773325972972201523680642816 8\
    52105468022493285647627940227:
```

This is a 1121-bit (probable) prime:

```
> isprime(p);
  ilog2(p)+1;
                                        true
                                        1121
```

In addition, Alice and Bob choose a generator of the group $\mathcal{QR}_p$. In order to find this generator one can apply a method similar to that used in the function

`findgenerator` given in Chap. 2. This function finds a primitive root modulo p and a generator of $\mathcal{QR}_p$ may be obtained by squaring such a primitive root modulo p, but Alice (or Bob) can also use the following Maple function that chooses elements of $\mathcal{QR}_p$ pseudo-randomly and checks whether they are generators by means of Proposition 2.6 (in contrast with the case when p is a safe prime, this is necessary here since the order of $\mathcal{QR}_p$ is not prime).

```
> findqrgenerator := proc(p)
  uses RandomTools:-MersenneTwister;
  local n, primefactors, found, g;
  SetState();
  n := iquo(p-1, 2);
  primefactors := [seq(x[1], `in`(x, ifactors(n, 'pollard')[2]))];
  found := false;
  while not found do
    g := GenerateInteger(range = 2 .. p-1)^2 mod p;
    found := andmap(x -> x<>1, [seq(Power(g, iquo(n,i)) mod p, i in primefactors)])
  end do;
  g;
  end proc:
```

The generator of $\mathcal{QR}_p$ shared by Alice and Bob is then:

```
> g := findqrgenerator(p);
  40061392678330193142632810871966551486004987278890174215381735300517445889082526 26\
    56934581865202674817386049580900176924684939078071816502463941738188627254776808\
    26253412965225290392357111896467686644247208709573914070598773738658180540949332\
    88322290369402817649171337284783695401285795169124110784836378348660138163557353\
    50984814997013
```

Eve observes p and g and suspects that $p-1$ has a large prime factor and hence it should be easy to factor with the rho method, because all its prime factors except the largest one should be small (note that, if p is chosen at random subject to the condition that $p-1$ has a large prime factor q, it is likely that the cofactor of q is a product of relatively small primes). Thus she uses the rho method embodied in Maple's `ifactor` with the option `'pollard'` (a good alternative would be the Elliptic Curve Method, which is the option `'lenstra'` of `ifactor`). The factorization of $p-1$ also tells Eve that p is not a safe prime:

```
> ifactor(p-1, 'pollard');
   (2)(3)(13)(10740895172706974336835790232145029570183055562678907163274057642810\
  10474564285224138802516233522861623662835287092605143143520618045567592051213003\
  02317619055051753810948058681230600944442783772186964499655858154770852745807823\
  36154463865135778340978000462384573554579349222720520095416182644216485243538671\
  9) (56526273169) (71729243) (650152379)
```

Eve sees that $p-1=2qr$, where

```
 > q:= 10740895172706974336835790232145029570183055562678907163274057642810 1047456\
    42852241388025162335228616236628352870926051431435206180455675920512130030 2317\
    61905505175381094805868123060094444278377218696449965585815477085274580782 3361\
    54463865135778340978000462384573554579349222720520095416182644216485243538 6719:

> r := 3*13*71729243*650152379*56526273169:

> ilog2(q)+1;
                                    1024
```

q is a 1024-bit prime and r is a product of small primes, the largest of which has 11 decimal digits. Thus $|\mathcal{QR}_p| = (p-1)/2 = qr$ and, by Corollary 2.1, the order of an element of $\mathcal{QR}_p$ is a divisor of qr and hence either a multiple of q or a divisor of r. Eve checks that g has order $qr = |\mathcal{QR}_p|$ (so that g is really a generator of $\mathcal{QR}_p$)

and realizes that when Alice and Bob exchange through the public channel elements $u \in \mathcal{QR}_p$, $v \in \mathcal{QR}_p$ such that $u = g^x$ and $v = g^y$ (where x and y are unknown to Eve),[1] she can raise both equations to the qth power and convert them to equations in the subgroup of $\mathcal{QR}_p$ generated by g^q which, according to Proposition 2.5, has order $qr/q = r$. Since r is a product of small primes, the resulting discrete logarithm problems will be easily vulnerable to a Pohlig–Hellman attack.

Next, with the notation used in Sect. 7.3 Alice chooses the integer x pseudo-randomly in the range $2..qr - 1$ and computes $u = g^x$:

```
> x := 47810246253398334400355967545641040299316949347488772222845719898179931003 1\
    87942076111028521822403387829988564379947121847253258276025217731311899707916 6\
    75824950029984253177484488155479329417165984888490396764130473227743766117038 9\
    29789875226984623912842044801091422868574356052414111276722178507328792942820 4\
    10967443173637957484578009 98:

> u := Power(g, x)  mod p:
```

Before sending u to Bob over the public channel, Alice makes sure that u does not have low order for, otherwise, an eavesdropper could solve the DLP problem in a small group. The probability that the order of u is not a multiple of q (and hence $\leq r$) is very small, but Alice checks it out nonetheless. By Proposition 2.5, the order of u is equal to $qr/\gcd(x, qr)$ and she checks that this value is, in fact, equal to qr (or that $\gcd(x, qr) = 1$):

```
> evalb(q*r/igcd(q*r, x) = q*r);
                                    true
```

Alice then sends u to Bob over the public channel. Similarly, Bob chooses y and sends $v = g^y$ to Alice after checking that the order of v is also qr:

```
> y := 11020757594294304385271307911459859710882465530758174472500881785865904743 5\
    90145878824802697714346899554882789237315299447002685514778329575454702585586 5\
    20288981199481452989926385156426849287336373644412677893121186149471172674287 3\
    35668819942347920581659914392729668175879433725874932745917783527035412407032 0\
    41987493033050828314340624458:

> v := Power(g, y) mod p:

> igcd(q*r, y);
                                    1
```

Eve knows that computing $\log_g u$ or $\log_g v$ is infeasible as the group generated by g has order qr. But she knows the factorization of $p - 1$ and she also knows that by raising u to the qth power, the problem of computing $\log_g u$ is in some sense reduced to the problem of computing $\log_{g^q} u^q$. As already mentioned, g^q has order r, which is a product of small primes, and hence computing $\log_{g^q} u^q$ is feasible using Pohlig–Hellman combined with a generic method such as BSGS. Thus Eve intercepts both u and v and replaces them by $u_2 = u^q$ and $v_2 = v^q$, respectively, sending the first to Bob, who thinks it came from Alice, and the second to Alice, who thinks it came from Bob. The group elements sent by Eve are:

[1] Here and in what follows we use the generic notation to indicate an exponentiation in the group $\mathbb{Z}_p^*$ but if we want to be specific and the elements of the group are viewed as integers, then this is really $u = g^x \bmod p$.

```
> u_2 := Power(u, q) mod p;
554654063981012007942423963504791491019661003509657650817066932931194925801554167O\
  9495801585663800542893966944946943391174518577525133372869687917078013891515699 3\
  0872269237881770917008945321960392067582439045672749129225923002431312804807599 5\
  8579301648043464476103706338573714802400245761318076526002172352934546326702216 2\
  935580911245812

> v_2 := Power(v, q) mod p;
342723678563553583455407140372380641058101990128793007129289609906417385161234482 1\
  9988545309742696217664656524998265433477673168509294437991678344600972284355040 3\
  0588105895599295252255091707571967198423552701457441152953046616538857481292264 3\
  6558884824139105842108981651179949877143235245439233169944110331036217818128262 2\
  112641462805445
```

Now, Alice and Bob don't notice anything wrong and perform the computations established in the DH protocol to obtain the shared number k. Alice obtains k by computing:

```
> k := Power(v_2, x) mod p;
109659374293791814589289836207167057905423426550983089906300165525380263355931242 9\
  7098198279216046588805629521343965569845587822127179332834962202366724004495392 9\
  4651708345531584234210903213091913240195721465452601204925412003567634502371819 2\
  7984537276155447311731539805490017231004252411142374807796729620506192088497315 9\
  8614284230480561
```

Similarly, Bob obtains k by computing

```
> k := Power(u_2, y) mod p:
```

and, from this point on, Alice and Bob use the key k to exchange encrypted messages with their favorite private-key encryption scheme. Meanwhile, in order to carry out her attack, Eve first computes $g_2 = g^q$, namely the group element:

```
> g_2 := Power(g, q) mod p;
871347062891857658540272764389590932275603347174123431445133806375777298092309499 1\
  7979364819032793236480585593885857550493174578232789452933948221561596889656788 1\
  4499142175596201345180625450086888820826443446761898754670234893935431732903785 7\
  5820678109794489805037600190203182325965962962895277796189503298301171316824964 3\
  658418696952190
```

Now Eve knows p, $g_2 = g^q$ and $u_2 = u^q$ and she will use this information to compute $\log_{g_2} u_2$. She uses the function `PohligHellman` given in Sect. 6.5.4 and, in order to discharge Maple from the burden of computing the order of g_2 in $\mathbb{Z}_p^*$, she passes the order r to the function directly. She uses the option `BSGS` (with the rho method, Maple may give an error because numbers that are too large for the function `Roots` are being used):

```
> l := PohligHellman(p, g_2, u_2, r, method = BSGS);
                              64413781753853186629011731261
```

Observe that what Eve has obtained is not quite x, because the logarithm of an element to the base g_2 is only defined modulo the order r of g_2, whereas x is an integer in the range $2..qr-1$. Since $u^q = (g_2)^x = (g_2)^l$ we have that $x \equiv l \pmod{r}$ and since $l < r$ we actually have that $l = x \bmod r$. Trying to find x from l would still be pretty much infeasible but Eve does not need to know x to find k. Since $k = (v_2)^x \bmod p$ and the order of v_2 divides r (it is actually equal to r), she knows that also $k = (v_2)^l \bmod p$, so Eve finds k this way. We will not print k again, but we show that this holds:

```
> evalb(Power(v_2,l) mod p = k);
                                        true
```

The result is that Eve was successful and now Alice and Bob share a key k which is also known to Eve. Therefore, Eve can passively eavesdrop on all encrypted messages subsequently exchanged between Alice and Bob. What should Alice and Bob do to prevent this attack? As we have mentioned, a good solution is to choose the modulus p to be a safe prime. Then $(p-1)/2$ would be prime itself and trying to mount the same attack as above would not work for then q would be the order of g and so $u^{(p-1)/2} = v^{(p-1)/2} = 1$ and, when receiving these values from each other, Alice and Bob would easily notice that something was going on. Indeed, in the next step they would have to send x and y through the channel allowing an eavesdropper to recover the key. But even with the prime p chosen by Alice and Bob, they could have thwarted the attack by checking the order of the elements u_2 and v_2 they received from each other. Alice had checked that the element u she sent to Bob had order qr and similarly Bob had checked that v had order qr so they should also check that the elements they receive from each other have this property. Thus, upon receiving u_2 supposedly from Alice, Bob should check that its order is not a divisor of r which is equivalent to $(u_2)^r \neq 1$ by Proposition 2.4. To do that, he would compute:

```
> Power(u_2, r) mod p;
                                    1
```

This would tell Bob that the order of u_2 is a divisor of r and hence that u_2 fails this test. Thus Bob should reject u_2 and, similarly, Alice should reject v_2. These precautions are also effective against passive attackers who can find the key if the elements exchanged have low order, but, of course, the most general and effective method to counter active attacks like this one is, as already pointed out, to use an authenticated channel.

Exercise 7.5 Looking at the code of the function `findqrgenerator` above, explain in detail the reasons why g is indeed a generator of $\mathcal{QR}_p$.

Exercise 7.6 In the previous example, suppose that an integer z is chosen uniformly at random in the range $2..qr-1$ and calculate the probability that the order o of g^z in $\mathbb{Z}_p^*$ satisfies:

(i) $o = qr$.
(ii) $o \geq q$.

7.2.5 Concluding Remarks on the Diffie–Hellman Protocol

Despite being the very first public-key protocol defined, the Diffie–Hellman key agreement remains widely used today as an important component of several protocols such as, for example, the IPsec protocol suite for network communications. Thus the DH protocol is not merely of historical importance but is also practically relevant. Since the basic protocol is vulnerable to man-in-the-middle attacks, the versions that are actually used rely on variants that are resistant to these attacks and whose nucleus is the original DH protocol. The most important attacks against the protocol are based

on solving discrete logarithms and, as we shall see, there are other cryptographic systems that also rely on the hardness of the discrete logarithm problem for their security. Therefore, an evaluation of the hardness of this problem taking into account the algorithms and the computations mentioned in Chap. 6 is necessary to establish parameters which provide reasonable security when implementing these systems.

Chapter 8
Public-Key Encryption

The preceding chapter offered a brief glimpse at the birth of public-key cryptography and some of its basic features. In the present chapter we continue the study of public-key cryptography by discussing one of its main themes: public-key encryption.

8.1 Public-Key Encryption Schemes

As we have seen in Chap. 7, public-key cryptography was born with the purpose of enabling secure communication between two parties that do not have to share a common secret key. That this should be possible was the revolutionary idea of Diffie and Hellman, who realized the possibility of designing an encryption scheme in which it is computationally infeasible to find the decryption algorithm from the encryption one. This, in turn, entails that the same key cannot be used for both encryption and decryption as happens in private-key cryptography and, as explained in Chap. 7, leads to each user having two keys: a public key pk which is used for encryption and a private key sk which is used for decryption.

This idea was suggested by the fact that in the real world there are actions that are easily performed but not easily reversed. A simple example is provided by a padlock that can easily be locked without a key but is very difficult to unlock. A telephone book provides a less physical analogy. Using it, it is easy to check the telephone number corresponding to a user of the system but it is much more difficult to find the name of the user with a given number. In the first case, one only has to go through the letters in the name using the fact that names are alphabetically ordered. Thus if the book has n telephone numbers and the names are constructed without redundancy from an alphabet with b symbols, then strings of $\log_b n$ symbols are sufficient to write down all the names and so to locate a name in the book one only needs to go through at most $\log_b n$ symbols. On the other hand, since the telephone numbers are not sorted, for the reverse process one has to go through $n/2$ numbers on average

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_8, © Springer-Verlag Berlin Heidelberg 2013

before finding a given one, and the relation between $n/2$ and $\log_b n$ is, of course, exponential.

More relevant examples can be found in algorithmic number theory: as we have seen when studying the integer factorization problem, the function $(x, y) \mapsto x \cdot y$, where x and y are primes of the same length, is easy to compute but the primes are difficult to recover from their product using known algorithms. Similarly, while it is usually easy to compute $y = g^a$ given an element g of a group G and a positive integer a, computing a given g and y is believed to be hard in general. These difficult computational problems lead to the concept of one-way function which, as we have already seen, is also crucial for private-key cryptography (a fact that was not known in the mid-1970s) and one of the basic ideas of Diffie and Hellman was to construct encryption schemes for which the encryption functions were one-way functions, so that encrypting would be "easy" but decrypting would be "difficult".

It is readily apparent, however, that one-way functions are not really sufficient for this purpose, because then decryption would be difficult even for legitimate users of the scheme, which is clearly undesirable. To overcome this difficulty, Diffie and Hellman realized that a stronger concept was necessary, namely, a one-way function which has a *trapdoor*, i.e., secret information whose knowledge permits efficient inversion of the function. The idea was then to use as encryption functions a family of trapdoor permutations, i.e., a family of one-way permutations (see Definition 3.10) for which the parameter-generation algorithm outputs a trapdoor enabling efficient inversion of the functions in the family:

Definition 8.1 A family of permutations $\{f_i : D_i \to D_i\}_{i \in I}$, where I is a set and D_i a finite set, is *trapdoor one-way* (or simply *trapdoor* as the one-way property is understood) if the following conditions hold:

1. There exists a PPT algorithm **Gen** which takes as input a security parameter 1^n and outputs a pair (i, t_i) such that $i \in I$ satisfies $\text{len}(i) \geq n$ and t_i is the *trapdoor* of i.
2. There exists a PPT algorithm **Samp** which, on input $i \in I$, outputs (with uniform distribution) $x \leftarrow D_i$ (except possibly with probability negligible in $\text{len}(i)$).
3. There exists a deterministic polynomial-time evaluation algorithm **Eval** which, on input $i \in I$ and $x \in D_i$, outputs $f_i(x)$.
4. There exists a deterministic polynomial-time inverting algorithm **Inv** that, on input i, t_i and $f_i(x)$, outputs $x := \mathbf{Inv}(i, t_i, f_i(x))$, for all $i \in I$ and $x \in D_i$ (that is, f_i is easy to invert once the trapdoor information t_i is known).
5. For every PPT algorithm $\mathcal{A}$, there exists a negligible function `negl` such that, for i obtained from $(i, t_i) \leftarrow \mathbf{Gen}(1^n)$, $x \leftarrow \mathbf{Samp}(i)$, and the random coins of $\mathcal{A}$:

$$\mathbf{pr}(\mathcal{A}(f_i(x)) = x) \leq \texttt{negl(n)}.$$

The idea then was to use a family of trapdoor permutations to build an encryption scheme more or less as follows:

1. Each user U of the system runs the parameter generation algorithm on security parameter 1^n, obtaining as output a pair (f_U, t_U), where f_U is a trapdoor permutation and t_U is the associated trapdoor information. The user makes f_U public but keeps t_U private.
2. A user wishing to send a message m to user U computes $c = f_U(m)$ and sends it to U.
3. Upon receiving the ciphertext c, U uses the trapdoor information t_U to invert f_U on c and computes $f_U^{-1}(c) = f_U^{-1}(f_U(m)) = m$.

This setup gives a first approximation to the idea of a public-key encryption scheme but it is far from providing a reasonable measure of security. One reason is that it may happen that the trapdoor permutation f_U is easy to invert on some special messages. For example, it could happen that, when m is taken from the set of sentences in the English language, then it is easy to recover m from $f_U(m)$. Moreover, since the encryption algorithm is deterministic, it will not have indistinguishable encryptions which, as we shall soon see, can be defined for public-key encryption schemes in a way similar to the private-key case. Even worse, suppose that only two messages, say 0 and 1 (meaning, for example, "Yes" and "No", are being encrypted). Since f_U is public, an attacker can easily recover the plaintext given such an encrypted message simply by computing $f_U(0)$ and $f_U(1)$ and comparing them with the ciphertext. Yet another shortcoming of modeling encryption schemes this way is that, even if a trapdoor permutation prevents the computation of m from $f_U(m)$ without knowing the trapdoor, it may well allow the leakage of partial information about m. For example, the parity of $f_U(m)$, which is easily computable, gives partial information about m. All these problems derive from the fact that, in this model, encryption algorithms are deterministic. In order to obtain secure public-key encryption, it will be necessary to use probabilistic encryption which, together with trapdoor permutations and their hard-core predicates, will give public-key encryption schemes with security reductions under reasonable assumptions.

Despite these shortcomings, the concept of public-key encryption scheme can be approximately modeled by the preceding description, making sure that the encryption algorithms used are probabilistic and adding the requirement of compliance with a precise definition of security which will be given later.

Definition 8.2 A *public-key encryption scheme*, also called an *asymmetric encryption scheme*, is a 3-tuple $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ of polynomial-time algorithms such that **Gen** and **Enc** are probabilistic and **Dec** is deterministic, and satisfy the following conditions:

- The *key generation algorithm* **Gen** takes as input a *security parameter* 1^n (whose purpose is to determine the size of the key) and outputs a key (pk, sk). We denote this by $(pk, sk) \leftarrow \mathbf{Gen}(1^n)$, where pk is the *public key* and sk is the *private key*.
- The *encryption algorithm* **Enc** takes as input a public key pk and a plaintext m (from a message space which may be associated with pk) and outputs a ciphertext c, written $c \leftarrow \mathbf{Enc}(pk, m)$.

- The *decryption algorithm* **Dec** takes as input a private key sk and a ciphertext c and outputs either a plaintext m, or a symbol $\perp$ denoting failure. We write $m := \mathbf{Dec}(sk, c)$.
- There exists a negligible function $\mathtt{negl}$ such that, for every n, every $(pk, sk) \leftarrow \mathbf{Gen}(1^n)$ and every message m:

$$\Pr(\mathbf{Dec}(sk, \mathbf{Enc}(pk, m)) \neq m) \leq \mathtt{negl}(\mathtt{n}).$$

Remarks 8.1

1. Observe that the definition of public-key encryption scheme is very similar to that of private-key encryption scheme, with the important difference that there are different keys for encryption (the public key) and for decryption (the private key).
2. For the reasons already explained, it is important for security that the encryption algorithm be probabilistic which, in practice, means that there can be many different ciphertexts corresponding to the same message. However, for illustrative purposes, we will also consider some schemes with deterministic encryption such as plain RSA discussed in 8.3.2.
3. There is a more general definition that includes the possibility of a probabilistic decryption algorithm but we will not need it. On the other hand, although a negligible decryption error is allowed by the definition, we will henceforth tacitly assume that no such error occurs in practice, so that we always have $\mathbf{Dec}(sk, \mathbf{Enc}(pk, m)) = m$.
4. We have not specified what the plaintext space is. In practical applications it is desirable to make it independent of the key but using a key-dependent message space is also possible provided there are efficient ways to encode and decode bit strings as elements of the message space.

8.2 Security for Public-Key Encryption

The definition of public-key encryption scheme must be completed with a suitable definition of security. In the public-key setting there is an important additional concern that did not arise in private-key cryptography, namely how to ensure the secure distribution of public keys. Indeed, an active adversary tampering with communication between the honest parties might replace the public key of some user, say Alice, with her own public key, allowing her later to recover encrypted messages addressed to Alice. Thus, if Alice and Bob want to prevent such attacks, they will have either to share some information in advance or to rely on some trusted third party. The latter solution is clearly more convenient and is the one usually adopted; we will discuss it when studying digital signatures and for now we will just assume that the users of a public-key encryption scheme have legitimate copies of the public keys of the remaining users (or, at least, of the receivers of the messages they send).

8.2.1 Definitions of Security

The strongest form of security for private-key encryption schemes is perfect security and we could try to define the analogous concept for public-key encryption. However, it is easy to see that a perfectly secret public-key encryption scheme cannot exist, because a computationally unbounded adversary can recover the plaintext encrypted as ciphertext c just by encrypting all plaintexts in the message space and comparing the resulting ciphertexts with c. This will work even if encryption is probabilistic because the adversary can try all possible choices of random bits, of which there will be only a finite number.

Recall that, when defining security in the private-key setting, there exists an important distinction between security notions based on the assumption that the adversary is a passive eavesdropper and the stronger notion in which the adversary is able to mount a chosen plaintext attack (IND-CPA). This distinction no longer makes sense in the public-key setting because the adversary knows the public key pk and hence has access to an encryption oracle that will return the ciphertext corresponding to any message. Thus it is readily apparent that in this situation any reasonable concept of security against an eavesdropper will actually be equivalent to the corresponding notion of security against a chosen plaintext attack. Therefore we will deal only with the latter.

We will make a first attempt at defining security for a public-key encryption scheme taking as a starting point the idea that the adversary should be unable to recover the message from a given ciphertext. In this case the adversary goal is to invert the encryption function and hence this kind of security should be attained if the encryption functions are a family of trapdoor permutations. We have already mentioned the shortcomings of modeling the encryption scheme this way and the fact that the security level thus attained is very low. However, this kind of security is a first step that can be proved under reasonable assumptions for some simple schemes and is also a necessary condition for stronger security goals which we shall subsequently define, so we start by formalizing it. First of all, we introduce the probabilistic experiment that we will use for the definition:

Definition 8.3 The *public-key CPA one-wayness experiment* $\mathbf{PubK}^{ow\text{-}cpa}_{\mathcal{A},\mathcal{E}}(n)$, where $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ is a public-key encryption scheme, $\mathcal{A}$ a PPT adversary, and n any value of the security parameter, is the following:

1. **Gen** is run to obtain keys $(pk, sk) \leftarrow \mathbf{Gen}(1^n)$.
2. A message m is randomly chosen in the plaintext space and the corresponding ciphertext $c \leftarrow \mathbf{Enc}(pk, m)$ is computed.
3. Adversary $\mathcal{A}$ is given pk and the *challenge ciphertext* c.
4. $\mathcal{A}$ outputs a message m'.
5. The output of the experiment is defined to be 1 if $m' = m$ and 0 otherwise. If $\mathbf{PubK}^{ow\text{-}cpa}_{\mathcal{A},\mathcal{E}}(n) = 1$ then we say that $\mathcal{A}$ *succeeded*.

Based on this experiment, we define one-way security:

Definition 8.4 A public-key encryption scheme $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ is *one-way secure against chosen plaintext attacks* (OW-CPA secure, for short) if, for every PPT adversary $\mathcal{A}$, there exists a negligible function $\mathtt{negl}$ such that

$$\Pr(\mathbf{PubK}^{ow\text{-}cpa}_{\mathcal{A},\mathcal{E}}(n) = 1) \leq \mathtt{negl(n)},$$

where the probability is taken over the random bits used in the experiment.

We stress again the main weaknesses of this definition. First of all, it only requires that the ciphertext of a plaintext which is randomly chosen in the message space is hard to decrypt, but this is not meaningful if the challenge ciphertext comes from a message chosen from a smaller subset of the message space that might be known to the adversary. Also, it is only required that the recovery of the entire message is hard but this leaves open the possibility that the adversary might learn a lot of partial information about the message. We will mention this security concept again when discussing the RSA encryption scheme in Section 8.3.

Since one-wayness is insufficient for providing a reasonable level of security, we will now look at indistinguishability which, as we have remarked in our discussion of security in the private-key setting, provides an adequate tool to formalize the intuitive idea that the ciphertext does not leak any partial information about the plaintext.

Definition 8.5 The *public-key CPA indistinguishability experiment* $\mathbf{PubK}^{ind\text{-}cpa}_{\mathcal{A},\mathcal{E}}(n)$, where $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ is a public-key encryption scheme, $\mathcal{A}$ a PPT adversary, and n any value of the security parameter, is the following:

1. **Gen** is run to obtain keys $(pk, sk) \leftarrow \mathbf{Gen}(1^n)$.
2. Adversary $\mathcal{A}$ is given pk. The adversary outputs two messages of the same length: m_0, m_1.
3. A random bit b is chosen and the challenge ciphertext $c \leftarrow \mathbf{Enc}(pk, m_b)$ is computed and given to $\mathcal{A}$.
4. $\mathcal{A}$ outputs a bit b'.
5. The output of the experiment is defined to be 1 if $b' = b$ and 0 otherwise. If $\mathbf{PubK}^{ind\text{-}cpa}_{\mathcal{A},\mathcal{E}}(n) = 1$ then we say that $\mathcal{A}$ succeeded.

This leads to the following security definition:

Definition 8.6 A public-key encryption scheme $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ has *indistinguishable encryptions under a chosen plaintext attack* (or is IND-CPA secure or, simply, CPA secure) if, for every PPT adversary $\mathcal{A}$, there exists a negligible function $\mathtt{negl}$ such that

$$\Pr(\mathbf{PubK}^{ind\text{-}cpa}_{\mathcal{A},\mathcal{E}}(n) = 1) \leq 1/2 + \mathtt{negl(n)}.$$

Remarks 8.2

1. The difference between OW-CPA security and IND-CPA security rests only on the stronger goal that the adversary has to achieve in the latter. We have not

explicitly included oracle access to encryption for $\mathcal{A}$ in Definitions 8.3 and 8.5 because this access is automatically granted by the knowledge of the public key pk. Thus Definition 8.5 serves also to define the concept of indistinguishability in the presence of an eavesdropper in the public-key setting which is, in this case, equivalent to being indistinguishable under a chosen plaintext attack.

2. We have seen in Remarks 3.8 that in the private-key setting an encryption scheme may have indistinguishable encryptions in the presence of an eavesdropper without having *multiple* indistinguishable encryptions in the presence of an eavesdropper. This is essentially due to the fact that a deterministic private-key encryption scheme may have the former property but cannot have the latter one. However, an IND-CPA secure private-key encryption scheme also has multiple encryptions indistinguishable to an eavesdropper. We did not give a formal proof of this fact but we remarked that multiple indistinguishability comes from the fact that IND-CPA secure schemes must have a probabilistic encryption algorithm (in other words, no deterministic encryption scheme can be IND-CPA secure). In the public-key setting indistinguishable to eavesdroppers is the same as IND-CPA secure and, similarly to the private-key case, it can be proved that the concept of IND-CPA secure for a single encryption is equivalent to the concept of IND-CPA secure for multiple encryptions (see, for example, [109, Theorem 10.10] for a proof). Therefore, in the public-key setting there is no difference between security for a single encryption or for multiple encryptions.
3. The just-observed fact that security for single encryption implies security for multiple encryptions entails that if we have a public-key encryption scheme for fixed-length messages that is CPA secure, then a public-key encryption scheme for arbitrary-length messages can be constructed from it which is also CPA secure. This is because a longer message can be regarded as the concatenation of shorter messages and the resulting ciphertext will be the result of concatenating the ciphertexts corresponding to the shorter messages.

Another important notion of security is obtained by considering, like in the private-key setting, chosen ciphertext attacks. These attacks are, if anything, even more dangerous in public-key cryptography because receivers may get ciphertexts from many senders, who are possibly unknown to them. For example, we mention in Sect. 8.3.5 an attack against an implementation of RSA called PKCS #1 v1.5, based on sending to a server many variants of a ciphertext and looking at the server responses. The server returned an error message if the ciphertext was not well formatted and by observing these messages it was possible to recover the plaintext.

The definition of CCA security is very similar to the corresponding one in the private-key setting, except that in this case no explicit mention of an encryption oracle—which is always available in the public-key setting—is made:

Definition 8.7 The *public-key CCA indistinguishability experiment* $\mathbf{PubK}^{ind\text{-}cca}_{\mathcal{A},\mathcal{E}}(n)$, where $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ is a public-key encryption scheme, $\mathcal{A}$ a PPT adversary, and n any value of the security parameter, is the following:

1. Keys $(pk, sk) \leftarrow \mathbf{Gen}(1^n)$ are generated.

2. $\mathcal{A}$ is given pk and oracle access to $\mathbf{Dec}(sk, -)$ and it outputs a pair of messages of the same length, m_0, m_1, belonging to the plaintext space.
3. A random bit b is chosen and the challenge ciphertext $c \leftarrow \mathbf{Enc}(pk, m_b)$ is computed and given to $\mathcal{A}$.
4. $\mathcal{A}$ continues having oracle access to $\mathbf{Dec}(sk, -)$ but is not allowed to query it on the challenge ciphertext. Then $\mathcal{A}$ outputs a bit b'.
5. The output of the experiment is defined to be 1 if $b' = b$ and 0 otherwise. In the first case we say that $\mathcal{A}$ succeeded.

The corresponding security definition is then the following:

Definition 8.8 The public-key encryption scheme $\mathcal{E}$ is said to have *indistinguishable encryptions under a chosen ciphertext attack* (or to be CCA secure or IND-CCA secure) if, for every PPT adversary $\mathcal{A}$, there exists a negligible function `negl` such that

$$\Pr(\mathbf{PubK}^{ind\text{-}cca}_{\mathcal{A},\mathcal{E}}(n) = 1) \leq 1/2 + \mathtt{negl(n)}.$$

Remarks 8.3 Chosen ciphertext attacks are often subdivided into two classes: CCA1 and CCA2. In a CCA1 attack the adversary can obtain decryptions of arbitrary ciphertexts of its choice before it gets the challenge ciphertext. This resembles a scenario in which the adversary obtains access to the decryption device (as a black box) while the legitimate user is away at lunch. Afterwards, the legitimate user returns and the adversary is given the challenge ciphertext which it has to decrypt without further access to the decryption box. Because of this analogy CCA1 attacks are often called 'lunchtime attacks'. Since the queries to the oracle cannot depend on the challenge ciphertext, CCA1 attacks are also called *non-adaptive* or *indifferent* chosen ciphertext attacks, and another term used for them is *a priori* chosen ciphertext attacks ([87]).

In a CCA2 attack the adversary has the additional capability of continued access to the decryption device after it has received the challenge ciphertext. Thus it can query the decryption box about ciphertexts that might be related to or in some way suggested by the challenge ciphertext (with the restriction that it cannot ask for decryption of the challenge ciphertext itself). Therefore CCA2 attacks (also called *adaptive* or *a posteriori* chosen ciphertext attacks) form a much stronger class of attacks than CCA1.

Observe that in the definition of $\mathbf{PubK}^{ind\text{-}cca}_{\mathcal{A},\mathcal{E}}(n)$ it is stated that the adversary $\mathcal{A}$ continues having oracle access to $\mathbf{Dec}(k, -)$ after being given the challenge ciphertext, so IND-CCA security is the same as what is called IND-CCA2 security in the finer classification. This is the strongest notion of security we consider and is the one at which the design of public-key encryption schemes should aim.

The construction of CCA secure public-key schemes is an important problem. There are some constructions that are rather complex and inefficient and thus they are not used in practice. As for efficient schemes, the first one to be proved CCA secure (under the DDH assumption according to which the DDH problem is hard) in the standard security model is due to Cramer and Shoup [59] and we study it in Sect. 8.6.

However, the most-used public-key scheme that can be proved CCA secure in some (weaker) sense is a variant of RSA called RSA-OAEP that we study in Sect. 8.3.6. The security proof for this scheme is a reduction to the RSA assumption (see 8.3.1 below) in the so-called *random oracle model* [18], in which it is assumed that some underlying hash functions are ideal in the sense that they are modeled as random oracles.[1] Of course, the hash functions used in practice are not random oracles but it is expected that their behavior is sufficiently close for security to carry over to the real world and hence this proof, while not providing a guarantee of security, increases our confidence in the scheme. For a thorough discussion of the random oracle model and its implications we refer to [109, Chap. 13].

8.2.2 Hybrid Encryption and Its Security

Public-key encryption has the drawback of being significantly less efficient than private-key encryption and, in fact, it can be orders of magnitude slower depending on the implementations (as we will see, the encryption operation for public-key schemes is usually modular exponentiation which, although efficient from the complexity-theoretic point of view, is much slower than, say, AES encryption). Thus, although public-key encryption schemes can be used to encrypt arbitrarily long messages, they are seldom used when the amount of data to be encrypted is large. To be more precise, a public-key encryption scheme is often combined with a private-key encryption scheme in such a way that one gets the advantages of both: on the one hand, secure communication can be established between the parties without having to share previous information (assuming, as usual, secure key distribution for public keys) and, on the other, efficiency is high, approaching that of the private-key encryption scheme for large amounts of data. This combination of private-key with public-key produces a *hybrid encryption scheme* and the basic idea is that the sender chooses a random symmetric private key (sometimes called a *session key*) and encrypts it with the public key of the receiver. The sender also encrypts the message with the symmetric key he generated and sends both ciphertexts to the receiver. The receiver first uses her private key to recover the symmetric key and then uses this key to recover the message. The hybrid scheme can then be defined as follows, assuming that the keys of the private-key scheme $\mathcal{E}_2$ are contained in the plaintext space of the public-key scheme $\mathcal{E}_1$:

Definition 8.9 Let $\mathcal{E}_1 = (\mathbf{Gen}_1, \mathbf{Enc}_1, \mathbf{Dec}_1)$ be a public-key encryption scheme and $\mathcal{E}_2 = (\mathbf{Gen}_2, \mathbf{Enc}_2, \mathbf{Dec}_2)$ a private-key encryption scheme. A hybrid encryption scheme $\mathcal{E} = (\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ is constructed as follows:

[1] A random oracle is an ideal hash function which, on each new input query, will pick uniformly at random some response from its output domain and will always return the same response if asked the same query again.

- $\mathbf{Gen} = \mathbf{Gen}_1$.
- **Enc** acts on pairs (pk, m) where pk is a public key of $\mathcal{E}_1$ and m a message in the plaintext space of $\mathcal{E}_2$. Thus, on input (pk, m), a random key k for $\mathcal{E}_2$ is chosen and $c \leftarrow \mathbf{Enc}(pk, m)$ is defined as the pair $c = (\mathbf{Enc}_1(pk, k), \mathbf{Enc}_2(k, m))$.
- **Dec** acts on pairs (sk, c) where $c = (c_1, c_2)$, with c_1 a ciphertext corresponding to $\mathcal{E}_1$ and c_2 a ciphertext corresponding to $\mathcal{E}_2$, by computing first $k := \mathbf{Dec}_1(sk, c_1)$ and then outputting $m := \mathbf{Dec}_2(k, c_2)$.

Remarks 8.4

1. A hybrid encryption scheme as defined above is actually a special kind of public-key encryption scheme because no previous secret information needs to be exchanged between the parties.
2. The plaintext space for the hybrid scheme $\mathcal{E}$ is the same as the one of $\mathcal{E}_2$, so we may assume that arbitrary length messages can be encrypted as we saw when discussing private-key cryptography.
3. The combination of $\mathcal{E}_1$ and $\mathcal{E}_2$ to produce a hybrid encryption scheme is often referred to as a KEM/DEM scheme in which $\mathcal{E}_1$ provides the *Key Encapsulation Mechanism* (KEM) and $\mathcal{E}_2$ provides the *Data Encapsulation Mechanism* (DEM). There are also more involved KEM/DEM constructions, with these mechanisms usually subject to specific security requirements.

Regarding the security of hybrid encryption schemes, we have the following important result:

Theorem 8.1 *If $\mathcal{E}_1$ is a CPA secure public-key encryption scheme and $\mathcal{E}_2$ is a private-key encryption scheme indistinguishable to eavesdroppers, then the hybrid encryption scheme $\mathcal{E}$ constructed from them as in Definition 8.9 is a CPA secure public-key encryption scheme.*

Sketch of proof. We give an informal description of the proof, and we refer to [109, Theorem 10.13] for a detailed proof. Assuming that m_0 and m_1 are two messages, we argue that a PPT adversary that knows the public key pk cannot distinguish between $(\mathbf{Enc}_1(pk, k), \mathbf{Enc}_2(k, m_0))$ and $(\mathbf{Enc}_1(pk, k), \mathbf{Enc}_2(k, m_1))$. A first intuition is that this should be true because $\mathbf{Enc}_1(pk, k)$ does not reveal information about k (because $\mathcal{E}_1$ is CPA secure) and hence $\mathbf{Enc}_2(k, m_0)$ is indistinguishable from $\mathbf{Enc}_2(k, m_1)$ taking into account that $\mathcal{E}_2$ is indistinguishable to eavesdroppers. But the rigorous version of '$\mathbf{Enc}_1(pk, k)$ does not reveal information about k' that we have is in terms of indistinguishability of two encryptions and hence we have to formalize the argument in these terms. Since indistinguishability is transitive (due essentially to the fact that the sum of two negligible functions is negligible) we can go from the first ciphertext pair to the second in three 'indistinguishability steps' as follows:

- $(\mathbf{Enc}_1(pk, k), \mathbf{Enc}_2(k, m_0))$ is indistinguishable from $(\mathbf{Enc}_1(pk, 0^n), \mathbf{Enc}_2(k, m_0))$. This follows from the fact that $\mathcal{E}_1$ is CPA secure (we are assuming here that $pk \leftarrow \mathbf{Gen}(1^n)$ and $k \leftarrow \{0, 1\}^n$).

- $(\mathbf{Enc}_1(pk, 0^n), \mathbf{Enc}_2(k, m_0))$ is indistinguishable from $(\mathbf{Enc}_1(pk, 0^n), \mathbf{Enc}_2(k, m_1))$. This is because $\mathcal{E}_2$ is indistinguishable to eavesdroppers and k is a key chosen uniformly at random about which $\mathbf{Enc}_1(pk, 0^n)$ does not give any information (this was the reason why the 0-string, which is unrelated to k, was introduced). Observe that we are using for $\mathcal{E}_2$ the IND-EAV hypothesis which, for private-key schemes such as this one is weaker than IND-CPA. But this is enough to ensure the indistinguishability of these ciphertext pairs under a CPA attack because each symmetric key k used by $\mathcal{E}$ is randomly generated for each new encryption and hence there are no multiple encryptions with the same k.
- $(\mathbf{Enc}_1(pk, 0^n), \mathbf{Enc}_2(k, m_1))$ is indistinguishable from $(\mathbf{Enc}_1(pk, k), \mathbf{Enc}_2(k, m_1))$. This follows, again, from the CPA security of $\mathcal{E}_1$. Composing the three steps we see that $(\mathbf{Enc}_1(pk, k), \mathbf{Enc}_2(k, m_0))$ is indistinguishable from $(\mathbf{Enc}_1(pk, k), \mathbf{Enc}_2(k, m_1))$ and hence that $\mathcal{E}$ is CPA secure.

Remark 8.1 Because of this theorem, the relative inefficiency of public-key encryption schemes is not a big concern. Indeed, we do not have to worry about encrypting long messages, as it is always possible to use for them an efficient hybrid scheme which is CPA secure if so is the underlying public-key scheme and the underlying private-key scheme satisfies the even milder requirement of being indistinguishable to eavesdroppers.

8.3 RSA

The first public-key encryption scheme was publicly introduced in [164], published in 1978, just two years after the publication of the seminal Diffie–Hellman paper. It is called RSA after the initials of its inventors, R. Rivest, A. Shamir and L. Adleman and, with many variants included in several standards, it is still today the most important and widely used public-key encryption algorithm. We next discuss this scheme and its security properties.

8.3.1 The RSA Assumption

From the very beginning of public-key cryptography, i.e., since the publication of [67], it was clear that the search for a public-key encryption scheme should be centered on finding an appropriate hard computational problem. One number-theoretic problem that was considered hard—and still is today—is the integer factorization problem that we have discussed in Chap. 6. However, it was not clear how to use this problem directly for an encryption scheme and Rivest, Shamir, and Adleman found another closely related problem that was useful for this purpose: the RSA problem.

The idea is that if $n = pq$, where p and q are two distinct large primes and $e \in \mathbb{Z}^*_{\phi(n)}$, then the function

$$\mathrm{RSA}_{(n,e)} : \mathbb{Z}_n \longrightarrow \mathbb{Z}_n$$

defined by $\mathrm{RSA}_{(n,e)}(x) = x^e \bmod n$, is trapdoor one-way, where the trapdoor is the prime factorization of n.[2] The RSA problem is essentially to invert this function (without knowing the trapdoor), i.e., to find for a randomly chosen $y \in \mathbb{Z}_n$, an $x \in \mathbb{Z}_n$ such that $\mathrm{RSA}_{(n,e)}(x) = y$, so saying that the function is one-way is saying that the RSA problem is hard. Let us analyze this idea more closely.

Observe first that the RSA function is a permutation of $\mathbb{Z}_n$:

Proposition 8.1 *Let $n = pq$ and $0 < e < (p-1)(q-1)$ an integer such that $\gcd(e, (p-1)(q-1)) = 1$. Then $\mathrm{RSA}_{(n,e)}$ is a permutation of $\mathbb{Z}_n$ whose inverse is given by $\mathrm{RSA}_{(n,d)}$, where d is the unique integer such that $0 < d < (p-1)(q-1)$ and $ed \equiv 1 \pmod{(p-1)(q-1)}$.*

Proof Note that $\phi(n) = (p-1)(q-1)$ and, since by hypothesis $\gcd(e, \phi(n)) = 1$, e is actually an element of $\mathbb{Z}^*_{\phi(n)}$. Let $d \in \mathbb{Z}^*_{\phi(n)}$ be the inverse of e (d can be found by the extended Euclidean algorithm) which satisfies $ed \equiv 1 \pmod{\phi(n)}$. Thus $ed = r\phi(n) + 1$ for some positive integer r and hence if $x \in \mathbb{Z}^*_n$, we have that $(\mathrm{RSA}_{(n,e)} \circ \mathrm{RSA}_{(n,d)})(x) = (\mathrm{RSA}_{(n,d)} \circ \mathrm{RSA}_{(n,e)})(x) = x^{ed} \bmod n = x^{r\phi(n)+1} \bmod n = (x^{\phi(n)})^r x \bmod n = x$ (where we have used that $x^{\phi(n)} \bmod n = 1$ by Euler's theorem (Theorem 2.15)). The proof that this equality still holds when $x \in \mathbb{Z}_n - \mathbb{Z}^*_n$ is left as an exercise (Exercise 8.1). □

Exercise 8.1 Prove that, in the hypotheses of Proposition 8.1 and with the notations therein, we have that if $x \in \mathbb{Z}_n$ then $x^{ed} \equiv x \pmod{p}$ and $x^{ed} \equiv x \pmod{q}$. Conclude that $x^{ed} \equiv x \pmod{n}$ and hence that $\mathrm{RSA}_{(n,e)}$ and $\mathrm{RSA}_{(n,d)}$ are mutually inverse.

The RSA assumption can then be informally described as stating that, in the hypotheses of Proposition 8.1 and given a randomly chosen $y \in \mathbb{Z}_n$, it is hard to compute an $x \in \mathbb{Z}_n$ such that $x^e \equiv y \pmod{n}$. Such an x exists by Proposition 8.1 and we can call it the eth root of y modulo n. In order to formally state this assumption, we describe the RSA instance generator $\mathbf{Gen}_{\mathrm{RSA}}$, a polynomial-time algorithm that generates a modulus n and two distinct primes p, q, satisfying suitable conditions.

Algorithm 8.1. $\mathbf{Gen}_{\mathrm{RSA}}$.

Input: Security parameter 1^k.
Output: (n, e, d) such that $n = pq$, with p and q primes and $\mathrm{len}(p) = \mathrm{len}(q) = k$, and $e, d \in \mathbb{Z}^*_{\phi(n)}$ such that $ed \equiv 1 \pmod{\phi(n)}$.

1. `Choose` primes p and q at random with $\mathrm{len}(p) = \mathrm{len}(q) = k$.
2. `Set` $n := pq$, $\phi(n) := (p-1)(q-1)$.
3. `Find` $e \in \mathbb{Z}^*_{\phi(n)}$, randomly or otherwise.
4. `Compute` $d = e^{-1} \in \mathbb{Z}^*_{\phi(n)}$.
5. `return` (n, e, d).

[2] More precisely, the family of functions indexed by the pairs (n, e) generated by Algorithm 8.3.1 and with trapdoor information given by the corresponding private keys is thought to be a family of trapdoor permutations.

Next we define the RSA experiment about the hardness of inverting the RSA function:

Definition 8.10 The *RSA experiment* $\mathbf{RSA}_{\mathcal{A},\mathbf{Gen}_{\mathrm{RSA}}}(k)$, where $\mathcal{A}$ is a PPT adversary and k any value of the security parameter, is the following:

1. $(n, e, d) \leftarrow \mathbf{Gen}_{\mathrm{RSA}}(1^k)$.
2. $x \leftarrow \mathbb{Z}_n$ is randomly chosen and $y := RSA_{(n,e)}(x) \in \mathbb{Z}_n$ is computed.
3. $\mathcal{A}$ is given n, e, y, and outputs $x' \in \mathbb{Z}_n$.
4. The output of the experiment is 1 if $x = x'$ and 0 otherwise.

This experiment allows us to formally define the hardness of the 'RSA problem':

Definition 8.11 The RSA problem is *said to be hard relative to* $\mathbf{Gen}_{\mathrm{RSA}}$ if, for every PPT adversary $\mathcal{A}$, there exists a negligible function $\mathtt{negl}$ such that

$$\Pr(\mathbf{RSA}_{\mathcal{A},\mathbf{Gen}_{\mathrm{RSA}}}(k) = 1) \leq \mathtt{negl(k)}.$$

The *RSA assumption* is the assumption that the RSA problem is hard.

Remarks 8.5

1. In the description of $\mathbf{Gen}_{\mathrm{RSA}}$ given in Algorithm 8.1 we required that, on input 1^k, $\mathbf{Gen}_{\mathrm{RSA}}$ should return a modulus n which is the product of two random primes of length k. The reason for taking primes of the same length is to make the modulus as difficult to factor as possible relative to its size since, as we have mentioned, there are factoring algorithms like ECM whose running time is a function of the size of the smallest factor of the number being factored. It is not strictly necessary that the two primes have exactly the same length, so this condition is often relaxed by requiring that both primes have *approximately* the same length.
2. $\mathbf{Gen}_{\mathrm{RSA}}$ may be modified by computing d as the inverse of e in $\mathbb{Z}^*_{\lambda(n)}$, where λ is the *Carmichael lambda function*, with $\lambda(n)$ defined as the smallest positive integer such that $a^{\lambda(n)} \equiv 1 \pmod{n}$ for each $a \in \mathbb{Z}^*_n$. It is straightforward to see that Proposition 8.1 also holds if this value of d is used instead of the one in Algorithm 8.1 and this implies that the modified version of $\mathbf{Gen}_{\mathrm{RSA}}$ may be used in all the RSA-based schemes we will introduce in the sequel. This variant has the advantage that this way d may be a little smaller (see Exercise 8.4 below).
3. The definition of hardness of the RSA problem is relative to the instance generator algorithm $\mathbf{Gen}_{\mathrm{RSA}}$, as we may consider different variants of this algorithm (for example, there is the variant mentioned in the preceding item and, on the other hand, the way in which e should be chosen was not specified). However, the most important known requirement that $\mathbf{Gen}_{\mathrm{RSA}}$ must fulfill for the RSA problem to be hard is that n should be the product of two randomly chosen primes of (approximately) the same length. This condition is already included in our definition and hence, in what follows, we shall not make explicit mention of $\mathbf{Gen}_{\mathrm{RSA}}$ when referring to the RSA problem.

Exercise 8.2 Prove that the RSA problem is random self-reducible.

We will return to the RSA problem when discussing the RSA encryption scheme but for now we just remark that, as already mentioned, a necessary condition for the RSA assumption to hold is that factoring the modulus n be hard. Indeed, if the factorization of n is known, then one may proceed as in the $\mathbf{Gen}_{\mathrm{RSA}}$ algorithm to solve the RSA problem: first $\phi(n) = (p-1)(q-1)$ is computed, then the inverse $d = e^{-1} \bmod \phi(n) \in \mathbb{Z}^*_{\phi(n)}$ is computed by means of the extended Euclidean algorithm and, finally, $y^d \bmod n$ is computed by using the binary exponentiation method. All these computations are efficient and hence the RSA assumption implies that factoring the modulus n is hard. Thus, recalling the factoring assumption introduced in Definition 3.15, we have proved the following:

Proposition 8.2 *The RSA assumption implies the factoring assumption.*

Since the RSA problem is no harder than factoring, it is natural to ask whether the converse is also true, i.e., whether both assumptions are equivalent. In order to disprove it one may try to find some condition that implies that the RSA problem is easy but does not imply that factoring is easy. For example, it is clear that if one is able to efficiently compute $\phi(n)$ on input n, then the RSA problem becomes easy for we may take advantage of the knowledge of $\phi(n)$ to compute $d = e^{-1} \in \mathbb{Z}^*_{\phi(n)}$. However, this leads nowhere because we can show that *computing $\phi(n)$ is no easier than factoring the modulus n* (and hence factoring $n = pq$ and computing $\phi(n) = (p-1)(q-1)$ are equally hard). Indeed, suppose that both n and $\phi(n)$ are known. Then $\phi(n) = (p-1)(q-1) = pq - (p+q) + 1$ and hence we have equations:

$$\left.\begin{array}{rl} p+q &= n - \phi(n) + 1 \\ pq &= n \end{array}\right\}$$

where p and q are the unknowns and the remaining terms are known. The zeros of the polynomial $(x-p)(x-q) = x^2 - (p+q)x + pq = x^2 - (n - \phi(n) + 1)x + n$ are precisely p and q and hence we only have to solve the quadratic equation:

$$x^2 - (n - \phi(n) + 1)x + n = 0$$

whose coefficients are known, to find the factorization of n.

Example 8.1 Consider the modulus $n = p \cdot q$, where p and q are (unknown) 1024-bit primes, and the values of n and $\phi(n)$ are given by:

```
> n :=
  23538372498408898020904820516194882190870515954482550253711433294824967383591284\
  46844042074690277603497784339917211555362254378971172185266564969960989190738145\
  81559762081612973350421474680153315264623295830664463318924022191113778607540232\
  32012364526054251283789295120489864731342782075372824073611435656184645877272042\
  24922319990089478607013995416230247080039877670515529684753026668533969494606154\
  97754388878053194737328927330425629335766593509172159059739618216167237695957287\
  65481579907273086935430737472301974772107008657018808368405599605158523746168029\
  49734669519358827496916433670057537612654700305155410180l:
```

```
> phi_n :=
  2353837249840889802090482051619488219087051595448255025371143329482496738359128 4\
  4684404207469027760349778433991721155536225437897117218526656496996098919073814 5\
  8155976208161297335042147468015331526462329583066446331892402219111377860754023 2\
  3201236452605425128378929512048986473134278207537282407361143565618433786656195 1\
  9261331118463689635012752956371403778201241350687266994092956944502837600238380 3\
  4290967708058568574526474760518166739450997152281119431648693894593766963981916 7\
  4730693085018222708358554123878815255109719163122444459186925949535904818927258 0\
  6691459655516240351837156557497474820323349783513702581 32:
```

These data are enough to factor n quickly. We compute the coefficients of the equation $x^2 - bx + c = 0$ that we must solve:

```
> b := n-phi_n+1:
  c := n:
```

Finally we use Maple to solve this equation, which is done almost instantaneously:

```
> facts := solve(x^2-b*x+c = 0, x);
167394786854516508663732462011077391957828910691941526274174945601964177118761218 2\
  3858794718328852484931324838399956251741823939266691338780827140415144958916665 3\
  6280891723558139376678516196892172924079358312821734345262547670171143194608597 1\
  7651759465021160602915439915134864901607483156505786734310651214703, 14061592323\
  5806581424322063811491472700696251401038748367461482995474057453473837696334276 3\
  3219594633043350251520685545510615680145874881082782507017381833785630317752463 9\
  6623607113892392401626279038255779909388363824995395967594220206489397579051359 8\
  3807040126726524324650583779343344737588425290389532628967
```

We check that n is indeed the product of these integers (it may be also checked that they are prime):

```
> evalb(n = mul(i, i = facts));
                                    true
```

We stress that, while factoring this 2048-bit modulus n was very easy knowing $\phi(n)$, it is currently infeasible to factor it on input n alone, even with the most powerful resources available. To put it in perspective we recall that the current record factorization of an RSA modulus like this one corresponds to a 768-bit number factored by NFS in 2009 (described in [113]) and that, as discussed in Sect. 6.4.10, an extrapolation using the complexity of NFS indicates that the factorization of a 2048-bit modulus with the same resources would take more than 10^{12} years.

Exercise 8.3 Use the function `PseudoRandomPrime` to generate two 2048-bit primes p and q. Compute the modulus $n = pq$ and $\phi(n) = (p-1)(q-1)$ and discard p and q. Use the values of n and $\phi(n)$ to factor n.

The previous attempt to break the RSA assumption without being able to factor the modulus n was not successful because, although we can easily invert the RSA function once we know $\phi(n)$, this knowledge also allows us to factor n. But we do not really need to know $\phi(n)$ to invert the RSA function for it is clear that knowing the inverse d of e in $\mathbb{Z}^*_{\phi(n)}$ is enough. So we may also ask: is there any way of computing d that is easier than factoring n?

The answer is, again, no; if we can efficiently compute d from n and e, then we can also efficiently factor the modulus n. Indeed, there is a PPT Las Vegas algorithm due to Miller that, on input (n, e, d) and a random $x \in \mathbb{Z}^*_n$ produces the factorization of n with probability at least $1/2$ (see [140, 8.2.2]). In fact, this problem has also been solved deterministically by Coron and May [55], who found a deterministic

polynomial-time algorithm that on input (n, e, d) gives as output the prime factors of n. Summing up, all the attempts to break the RSA hypothesis so far led to algorithms that are also able to efficiently factor the modulus and so it is still possible that the RSA problem is as hard as factoring, although in [38] some results are given that suggest that the RSA problem might be easier than factoring when the exponent e is small (such as, for example, $e = 3$ or $e = 17$).

8.3.2 Plain RSA

We are now ready to introduce the RSA encryption scheme. We will first describe *'plain RSA'* (often also called 'textbook RSA') which is exactly the system introduced by its inventors in 1978. This scheme is deterministic and hence it is not CPA secure which, nowadays, is a minimal security requirement. However, the analysis of its security properties will serve as a starting point to consider more secure variants of the scheme.

Definition 8.12 *Plain RSA* is the public-key encryption scheme RSA = (**Gen**, **Enc**, **Dec**) defined by the following algorithms:

- **Gen**: On input 1^k, run the RSA instance generator to obtain $(n, e, d) \leftarrow \mathbf{Gen}_{\mathrm{RSA}}$. Then set $pk := (n, e)$ and $sk := (n, d)$. The output of the algorithm is then $(pk, sk) \leftarrow \mathbf{Gen}(1^k)$, where pk is the public key and sk is the private key.
- **Enc**: On input a public key $pk = (n, e)$ and a message $m \in \mathbb{Z}_n$ ($\mathbb{Z}_n$ is the plaintext space associated with the modulus n), compute the ciphertext:

$$c := \mathbf{Enc}(pk, m) = \mathrm{RSA}_{(n,e)}(m) = m^e \bmod n \in \mathbb{Z}_n.$$

- **Dec**: On input a private key $sk = (n, d)$ and a ciphertext $c \in \mathbb{Z}_n$, compute the message:

$$m := \mathbf{Dec}(sk, c) = \mathrm{RSA}_{(n,d)}(c) = c^d \bmod n \in \mathbb{Z}_n.$$

Remarks 8.6

1. As in other public-key encryption schemes, each user should run **Gen** to obtain her public and private keys or, alternatively, these keys may be obtained from a trusted third party. As already mentioned, n is called the RSA *modulus* and, similarly, e is the *encryption exponent* and d is the *decryption exponent*.
2. Sometimes the primes p, q used by $\mathbf{Gen}_{\mathrm{RSA}}$ to build the modulus $n = pq$ are considered as part of the private key because, as we will see, they can be used to speed up decryption. This is a convenience issue but the primes are not necessary for decryption so they can also be discarded after running **Gen**. On the other hand, the only part of the private key (n, d) that must remain secret is the decryption exponent d, as the modulus is also part of the public key. Because of this it can also be considered that the private key is just d.

3. As we are going to see, a minimal security requirement for RSA is that the modulus should be hard to factor. This is the reason why $\mathbf{Gen}_{\mathrm{RSA}}$ is required to return a modulus which is the product of two primes of (approximately) the same length.
4. Observe that plain RSA satisfies the correctness condition (iv) in the definition of public-key encryption scheme (Definition 8.2) as a consequence of Proposition 8.1, which shows that in plain RSA, decryption always recovers the encrypted message.

Exercise 8.4 For a positive integer n, let $\lambda(n) = \min\{m \in \mathbb{Z} | m > 0 \text{ and } a^m \equiv 1 \pmod{n} \text{ for all } a \in \mathbb{Z}_n^*\}$ be the Carmichael function we mentioned in Remarks 8.5. Prove the following facts:

(i) $\lambda(n)$ is the maximum of all orders of elements of the group $\mathbb{Z}_n^*$ and $\lambda(n)|\phi(n)$.
(ii) If $n = pq$ is an RSA modulus, then $\lambda(n) = \mathrm{lcm}(p-1, q-1)$ (where lcm denotes the least common multiple) so that in this case we have that $\lambda(n) < \phi(n)$.
(iii) If (n, e) is an RSA public key, then any positive integer d such that $ed \equiv 1 \pmod{\lambda(n)}$ may be used as the exponent for RSA decryption. In particular, one may choose d equal to the inverse of e in $\mathbb{Z}_{\lambda(n)}^*$. Give an example where this inverse is smaller than the inverse of e in $\mathbb{Z}_{\phi(n)}^*$.

Let us now analyze the security properties of plain RSA. We have:

Proposition 8.3 *Under the RSA assumption, plain RSA is OW-CPA secure.*

Proof Suppose that plain RSA is not OW-CPA secure. Then, according to Definition 8.4, there exists a PPT adversary $\mathcal{A}$ such that $\Pr(\mathbf{PubK}^{ow\text{-}cpa}_{\mathcal{A},\mathrm{RSA}}(k) = 1)$ is non-negligible. Then we may show that there exists a PPT adversary $\mathcal{B}$ that solves the RSA problem with non-negligible probability, i.e., such that $\Pr(\mathbf{RSA}_{\mathcal{B},\mathbf{Gen}_{\mathrm{RSA}}}(k) = 1)$ is non-negligible. $\mathcal{B}$ is given n, e, and y, and it has to output x such that $\mathrm{RSA}_{(n,e)}(x) = y$. For this, $\mathcal{B}$ proceeds as follows:

- After being given its input, $\mathcal{B}$ defines the public RSA key $pk := (n, e)$ and sets $c := y$.
- Then $\mathcal{B}$ runs $\mathcal{A}$ on input the public key pk and the challenge ciphertext c and eventually $\mathcal{A}$ returns a message $m \in \mathbb{Z}_n$.
- Finally $\mathcal{B}$ returns $x := m$.

Observe that if $\mathcal{A}$ succeeds in the $\mathbf{PubK}^{ow\text{-}cpa}_{\mathcal{A},\mathrm{RSA}}(k)$ experiment, then $c = \mathrm{RSA}_{(n,e)}(m)$ and hence also $y = \mathrm{RSA}_{(n,e)}(x)$, so that $\mathcal{B}$ succeeds in the RSA experiment $\mathbf{RSA}_{\mathcal{B},\mathbf{Gen}_{\mathrm{RSA}}}(k)$. Since $\mathcal{A}$ succeeds with non-negligible probability, so does $\mathcal{B}$, breaking the RSA assumption. □

It is easily seen that the argument above is reversible and hence the RSA assumption is actually equivalent to one-wayness security of plain RSA. By Proposition 8.2, the RSA assumption implies the factoring assumption and hence the latter is a necessary condition for plain RSA to be OW-CPA secure. However, as we have

already mentioned, it is not known whether the factoring assumption implies the RSA assumption and hence whether the presumed hardness of factoring is sufficient to guarantee even this weakest form of security. But, since the factoring problem has been intensively studied, it is convenient to take it as a reference point and hence a minimal security requirement for RSA is to try to ensure that it is infeasible in practice to factor the modulus generated by **Gen**. Obtaining estimates for key sizes that provide this kind of short-term security is relatively easy but it is very difficult for long-term periods because of the many variables involved, not to mention the possibility that an efficient factoring algorithm might be found, in which case RSA—in all its variants—would be definitely broken. As mentioned when discussing the integer factorization problem, the current RSA factorization record is a 768-bit modulus and 1024-bit moduli are already regarded as insecure even for short-term use, so the recommended sizes start at 2048 bits and go up to 4096 bits and beyond. Of course, using a longer key length entails an efficiency loss and, in practical applications, there is always a trade-off between security and performance. Some estimates about the possible evolution of factoring in the future are given in [151] and also, within the more general framework of cryptographic key sizes in general, in [128].

Exercise 8.5 Consider the 2048-bit RSA modulus n obtained in Maple as follows:

```
> n := nextprime(2^1023+2^1022)*nextprime(2^1024):
```

Explain whether n can be factored in practice with Fermat's method and discuss whether n is a good RSA modulus.

Exercise 8.6

(i) Show that if (n, e) is an RSA public key and $m \in \mathbb{Z}_n$, then there exists a positive integer k such that $RSA^k_{(n,e)}(m) = m$ (where the exponent k indicates k iterations of the RSA function).

(ii) Indicate how this gives a method to recover the plaintext m from the ciphertext $c = RSA_{(n,e)}(m)$ (this is a so-called *cycling attack* which, as shown in [165], has negligible probability of success when n is a product of two sufficiently large random primes and, in fact, by [78], has expected complexity similar to that of factoring the modulus by trial division).

(iii) Use Maple to find the minimum value of k such that $RSA^k_{(143,e)}(m) = m$ for all $m \in \mathbb{Z}_{143}$ and all possible values of the encryption exponent e.

8.3.2.1 Plain RSA in Practice

We are going to implement plain RSA in Maple but before doing so we shall discuss a few points that are relevant for any implementation. In relation to the key generation algorithm, two primes of length k should be randomly chosen (or they can be chosen so that one of them has k bits and the other one $k + 1$ bits). In our implementation we will settle for choosing them pseudo-randomly using a variant of the function `PseudoRandomPrime` given in Sect. 6.3. If we choose two k-bit primes, their

product will have either $2k$ bits or $2k-1$ bits. If we want the modulus to have exactly $2k$ bits we can adopt several approaches, the simplest one being to check the length of the modulus and to discard it in case it has $2k-1$ bits. Alternatively, once the first prime p is chosen, the second prime may be chosen in the interval of the integers that produce a $2k$-bit integer when multiplied by p. We will discuss this in more detail when presenting the implementation below.

Some years ago it was usual to choose the primes in such a way that they satisfied some additional requirements and these primes were called 'strong primes'. The rationale was to choose the primes so that the resulting modulus was not vulnerable to certain factoring algorithms that are successful when the primes have some special forms. For example, there is a factoring method due to Pollard, called the $p-1$ *method*, that is able to efficiently factor an RSA modulus n if one of its prime factors p is such that $p-1$ has only 'small' prime factors. Thus the concept of 'strong prime' includes the requirement that $p-1$ has, at least, one large prime factor. However, this requirement has lost significance in recent years due, on the one hand, to the discovery of other factoring algorithms such as ECM that are more powerful than the $p-1$ method and against which taking strong primes does not protect and, on the other hand, to the realization that the probability that the randomly chosen primes for an RSA modulus turn out not to be 'strong' is negligible in practice (see [165] for a detailed discussion of these issues).

Another point that is sometimes raised in relation to RSA keys is that the two primes should not be too close in order to prevent the factorization of the modulus by Fermat's method. It is easy to make the key generation algorithm check whether the primes are close but, again, when the RSA primes are chosen at random, it is straightforward to see, taking into account the complexity of Fermat's method given in Proposition 6.3, that the probability of being able to factor the modulus by this method is negligible. Therefore, we shall be satisfied with just taking two pseudo-randomly chosen primes.

The only other aspect of the key generation algorithm that deserves discussion is how to choose the encryption exponent e. One possibility is to choose it at random in $\mathbb{Z}^*_{\phi(n)}$ but this produces very large exponents (of size close to that of $\phi(n)$ and hence also to that of n) and has a negative effect on encryption speed. Thus choosing e small seems a good choice because then encryption, which in plain RSA is just a modular exponentiation with exponent e, requires fewer multiplications. Because of this, a very popular choice for some time was $e=3$, requiring only two multiplications (in this case p and q must be $\equiv 2 \pmod 3$ in order to have $3 \in \mathbb{Z}^*_{\phi(n)}$). This choice, however, lost popularity because of an attack against plain RSA with a small encryption exponent that we shall later describe. Although this attack is not effective against more secure variants of RSA, the exponent of choice is currently $e=2^{16}+1=65537$.

The exponent $e=2^{16}+1$ has several advantages. On the one hand, it is prime (actually a *Fermat prime*, i.e., a prime of the form $2^{2^t}+1$) and hence it is very likely that it is relatively prime to both $p-1$ and $q-1$ when primes p, q are randomly chosen. More importantly, this value of e is very close to a power of 2 and so modular exponentiation with this exponent is very fast, requiring only 17 modular

multiplications because $2^{16}+1$ is a 17-bit number in which only the most significant and the least significant bits are nonzero (see the discussion done when analyzing the binary exponentiation method and its complexity). Finally, although $2^{16}+1$ is small in comparison with a typical element of $\mathbb{Z}^*_{\phi(n)}$ when n is chosen, say, in the 2048–4096-bit range, it is sufficiently large to make the attack against 'plain RSA with small exponent' very unlikely to be successful in practice. We insist, however, that the more secure variants of RSA—the only ones that should be used in real applications—are not vulnerable to this small exponent attack.

There are no more choices in the RSA key generation algorithm assuming, as we did, that $d = e^{-1} \in \mathbb{Z}^*_{\phi(n)}$, because then the decryption exponent d is uniquely determined by n and e (and, similarly, in the variant in which $d = e^{-1} \in \mathbb{Z}^*_{\lambda(n)}$). An alternative possibility is first to choose the decryption exponent d and then to find e as the inverse of d in $\mathbb{Z}^*_{\phi(n)}$ but this is not very popular because it will usually produce a large encryption exponent e. This allows the selection of a small decryption exponent but, as we shall see in Sect. 8.3.4, there is also an attack against the one-wayness of RSA that allows the efficient recovery of d when the decryption exponent is small (actually, it need not be too small, it suffices that it be roughly $< n^{1/4}$). This attack is much stronger than the one against a small encryption exponent, which only works against plain RSA. The attack against small d achieves recovery of the private key from the public key and hence breaks RSA in all its variants.

These considerations preclude the choice of a small decryption exponent but, fortunately, this feature is not as crucial from a performance point of view as having a small encryption exponent because, as we are going to see, decryption can be accelerated (to approximately one-fourth of the time taken by the modular exponentiation with exponent d) by using the Chinese remainder theorem. On the other hand, if the encryption exponent is chosen first, then the probability that the resulting key is vulnerable to the 'small decryption exponent attack' is very small (because the exponents $< n^{1/4}$ are just a tiny fraction of all possible exponents) and, moreover, it is easy to check whether d is small if one so wishes.

8.3.2.2 Decrypting with the Chinese Remainder Theorem

The Chinese remainder theorem can be used to speed up RSA decryption as follows. Suppose that (n, d) is the private key, with $n = pq$, where p and q are primes. Then we know that if $c \in \mathbb{Z}_n$ is the ciphertext corresponding to a plaintext $m \in \mathbb{Z}_n$, the decryption algorithm acts as follows:

$$m = \mathbf{Dec}((n, d), c) = c^d \bmod n.$$

For convenience, let $d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$, $m_p = m \bmod p$ and $m_q = m \bmod q$. Then we have:

$$m_p = (c^d \bmod n) \bmod p = c^d \bmod p = c^{d_p} \bmod p,$$

where the second equality comes from the fact that p divides n and the last equality follows from Fermat's little theorem (or, in other words, from the discrete logarithm isomorphism between $\mathbb{Z}_p^*$ and $\mathbb{Z}_{p-1}$; note that if p divides c then this equality is trivial). Similarly, we have that $m_q = c^{d_q} \bmod q$ and once we know m_p and m_q we can use the Chinese remainder theorem to determine m as the unique solution in $\mathbb{Z}_n$ of the system of congruences:

$$\left.\begin{aligned} x &\equiv m_p \pmod{p} \\ x &\equiv m_q \pmod{q} \end{aligned}\right\}$$

Now, if we look at the proof of Theorem 2.12, we see that the elements y_j defined therein are just the inverses of p modulo q and of q modulo p, so using the CRT in this case amounts to computing these inverses y_p, y_q, by applying the extended Euclidean algorithm to p and q to obtain

$$y_p p + y_q q = 1.$$

Once these elements are computed, the solution of the above system is given by

$$m = (m_p y_q q + m_q y_p p) \bmod n. \tag{8.1}$$

Observe also that the only values in this expression that depend on m are m_p and m_q, and that their coefficients may be precomputed and used to decrypt all ciphertexts encrypted with the public key. Thus the decryption operation essentially reduces to two exponentiations where both the exponents and the moduli have approximately half of the size of n and hence, as indicated when discussing the binary exponentiation method in Sect. 2.7, this method will make decryption almost four times faster.

8.3.2.3 The RSA Message Space

Another practical matter that must be considered when implementing plain RSA is how to encrypt binary strings or, more generally, any message over an alphabet. The natural way to do it is pretty straightforward: if the alphabet has b symbols, just regard any string of symbols as the base-b representation of a non-negative integer (through a suitable identification of the alphabet symbols with the base-b digits). If this integer is $< n$ (where n is the RSA modulus) then it is an element of $\mathbb{Z}_n$ and hence an RSA plaintext. If this is not the case then we may divide the message into blocks of suitable size so that each of these blocks can be identified with an element of $\mathbb{Z}_n$.

In more detail, we can proceed as follows. Let Σ be the alphabet and assume that $|\Sigma| = b$. Then we may identify Σ with $\mathbb{Z}_b$ by choosing a bijection between these two sets and, this way, the symbols of Σ also become the digits of the base-b number system. Then let $l := \lfloor \log_b n \rfloor$, so that n is an $(l+1)$-digit number in base b.

Then each l-symbol message represents an integer of at most l digits in base b and hence an element of $\mathbb{Z}_n$. Thus if we have a message m we may divide it into l-symbol blocks and encrypt it by encrypting the elements of $\mathbb{Z}_n$ corresponding to these blocks. Note that the ciphertext corresponding to a block is an element of $\mathbb{Z}_n$ and hence it may have $l + 1$ digits, so this produces a slight size-expansion. Also, in order to represent the ciphertext unambiguously as a string of symbols of Σ, one may use exactly $l + 1$ symbols to represent each ciphertext block, completing with 0-digits if necessary. Of course, the plaintext space may also be identified with the set of l-symbol strings in a similar manner.

For example, if Σ is the 8-bit ASCII character set and the RSA modulus n is a 2048-bit integer, then $\lfloor \log_{256} n \rfloor = 255$. Thus if we have a string of 8-bit ASCII characters, we partition it into 255-character blocks (except the last one, which can have fewer characters) and each of these blocks produces a ciphertext consisting of 256 characters (if padded with zeros to the left). This is the approach we are going to use when implementing RSA in Maple, except that ciphertexts will be represented in the hexadecimal alphabet for convenience.

As already mentioned, RSA is seldom used in practice to encrypt long messages and is typically used instead to encrypt encapsulated symmetric session keys by means of a hybrid scheme. These keys are much shorter than the currently used RSA moduli and hence they are usually encapsulated into a single element of $\mathbb{Z}_n$, so for this usage blocks are not necessary.

Exercise 8.7 A message consisting of 60 characters over the 26-letter English alphabet is numerically coded as follows. Each alphabet letter is identified with an element of $\mathbb{Z}_{26}$ in the standard way, the message is divided into six 10-letter blocks, and each block is regarded as an integer in the base-26 numeration system. Then the message blocks are encrypted with plain RSA using a 2048-bit modulus n. Explain how to recover the message from the ciphertext without factoring n.

8.3.3 Plain RSA in Maple

Let us now implement plain RSA in Maple following the guidelines just outlined. This will then allow us to perform some experiments with keys of realistic size.

We start with the key generation algorithm. As mentioned above, we will select the primes with a variant of the previously given function `PseudoRandomPrime`. This variant is given in the next function, `RSAPrime`, which uses the Blum–Blum–Shub pseudo-random bit generator to select the prime candidates in the appropriate intervals. The required parameters are the same as in `PseudoRandomPrime`, namely, one for the number k of bits that the prime should have and another for the seed that, for security reasons, should be an integer obtained from a random string of at least 128 bits.

The differences with the previous function are just two. On the one hand, we will now use Maple's `isprime` as primality test (instead of `MillerRabin`) in order to make the function a little faster. The other change is that now there is an optional input parameter `min` that, when looking for a k-bit prime, can be used to specify a 'minimum value' for the prime to be found, so that the function looks for primes in the interval of length 2^{k-1} whose initial point is this minimum value. The default value for the `min` parameter is 2^{k-1} so that, if no value is passed to this parameter, then the function just looks for a k-bit prime. If the value m of `min` is greater than 2^{k-1}, then the function looks for a prime in the interval $[m, m + 2^{k-1} - 1]$ (so that the prime obtained may actually have $k + 1$ bits in some cases). The remaining optional input parameter `bbslength` is used to set the length of the primes used by Blum–Blum–Shub as we have done on other similar occasions.

```
> RSAPrime := proc(k::posint, seed::posint,
                   min:=2^(k-1), bbslength::{512, 768, 1024}:=1024)
  local B, prime, gap, p;
  B := RandomTools:-BlumBlumShub:-NewBitGenerator(seed, primes = bbslength);
  prime := false;
  gap := min-2^(k-1);
  if gap mod 2 = 1 then
    gap := gap + 1
  end if;
  while not prime do
    p := gap + convert(cat(1, seq(B(), i = 1 .. k-2), 1), decimal, binary);
    prime := isprime(p)
  end do;
  p
  end proc:
```

We are now ready to give the RSA key generation function, `RSAKeyGen`. The required input parameters are `k`, `seed1` and `seed2`. The first is used to specify that the modulus of the key should be a $2k$-bit number and the remaining two to provide two random seeds in the form of hexadecimal strings of no less than 128 bits (or the corresponding integer) that will be used by Blum–Blum–Shub. Then the function uses `RSAPrime` to find the primes p, q such that the RSA modulus is $n = pq$. For convenience of use we have also provided the function with automatic seeding, so that it will generate a key even if no seeds are passed as arguments. But we warn that this solution produces highly insecure keys and is given only for demonstration purposes. In this case, the seeds are provided by Mersenne Twister in the form of integers in the range $0..2^{256}$ and, of course, they are not random because they have been generated from a 32-bit seed obtained from the system. Hence the keys thus generated would be vulnerable to a brute-force attack capable of doing a 2^{32}-size search.

Bearing in mind these remarks we insist that the Blum–Blum–Shub PRG that finally generates the primes should be provided with random seeds of sufficient size through the parameters `seed1` and `seed2`. The remaining optional parameters are two keyword parameters, `e` and `format`. The parameter `e` is used to specify the desired encryption exponent and takes as default the value $2^{16} + 1$ for the reasons explained in the preceding paragraphs. The parameter `format`, on the other hand, is used to specify the way in which the key will be represented and it takes as value a name that can be either `'hex'` or `'decimal'`, with the former as default.

When k is specified as security parameter, the function first searches for a k-bit prime which is pseudo-randomly chosen by the Blum–Blum–Shub bit generator. When a k-bit prime p is found, the function checks whether the condition $\gcd(p-1, e) = 1$ is satisfied for the value of the encryption exponent chosen and if this is not the case, the prime is discarded and the seed value increased in order to search for a new prime. When e is small, as happens, for example, in the case $e = 3$, this procedure is not optimal from a performance point of view and it would be preferable to discard the integers that do not satisfy the gcd condition before submitting them to the primality test but with larger exponents such as the default $2^{16} + 1$ it is unlikely that a prime chosen this way does not satisfy this condition. Once the first prime p is chosen, the second prime q is chosen in a similar way, with the difference that now the search interval is $[m, m + 2^{k-1} - 1]$, where $m = \lceil 2^{2k-1}/p \rceil$. Thus both primes are pseudo-randomly chosen from intervals of length 2^{k-1} and the modulus obtained by multiplying them is a $2k$-bit number.

The output is an RSA key in the form of a list containing two sublists. The first sublist is the public key (n, e) and the second sublist contains the private key. The first two terms of this sublist are the modulus and the decryption exponent, and they are followed in order by the primes p, q, the values $d_p = d \mod (p-1)$ and $d_q = d \mod (q-1)$ that will be used to speed up decryption by means of the Chinese remainder theorem and, finally, the values of $p^{-1} \mod q$ and $q^{-1} \mod p$. Note that there is a lot of redundant information in the private key but we include it in order to facilitate experimentation. For example, the first two components of the private key suffice to decrypt but do not allow the use of the Chinese remainder theorem. To use it for decryption the items in positions 3–6 in the private key list will be required. The last two items may also optionally be used to compute the plaintext using the Chinese remainder theorem in the form described in Eq. 8.1.

```
> RSAKeyGen := proc(k::posint, seed1::{posint,string}, seed2::{posint, string},
          {e::posint:=2^16+1, format::name:=hex, bbslength::{512,768,1024}:=1024})
  uses RandomTools:-MersenneTwister;
  local s, g, p, q, m, n, d, key;
  if _ params['seed1'] = NULL then
    SetState();
    s := GenerateInteger(':-range' = 2^127 .. 2^256)
  else
    s := stringposint(seed1)
  end if;
  g := 0;
  while g <> 1 do
    p := RSAPrime(k, s, ':-bbslength' = bbslength);
    s := s + 16;
    g := igcd(p-1, e)
  end do;
  m := ceil(2^(2*k-1)/p);
  if _ params['seed2'] = NULL then
    SetState();
    s := GenerateInteger(':-range' = 2^127 .. 2^256)
  else
    s := stringposint(seed2)
  end if;
  g := 0;
  while g <> 1 do
    q := RSAPrime(k, s, min = m, ':-bbslength' = bbslength);
    s := s + 16;
```

```
    g := igcd(q-1, e)
  end do;
  n := p*q;
  d := e^(-1) mod (p-1)*(q-1);
  igcdex(p, q, 'pinv', 'qinv');
  key := [[n, e], [n, d, p, q, d mod (p-1), d mod (q-1), pinv mod q, qinv mod p]];
  if format = hex then
    StringTools:-LowerCase~~convert~~(key, hex))
  elif format = decimal then
    key
  else
    error "Unrecognized key format"
  end if;
  end proc:
```

Example 8.2 Let us use the preceding function to generate a 2048-bit RSA key. When we speak of a t-bit RSA key, t is the bit length of the modulus, so the key we are about to generate will have a 2048-bit modulus and we will take $k = 1024$ as the value of the security parameter when calling `RSAKeyGen`. Since the output is very long, we will not print it entirely but will only print the essential parts of the public key and the private key:

```
> rsakey := RSAKeyGen(1024, 6180204295522709153838923244723480121 75,
                      4102756714646987802485435721275578031 86):

> pk := rsakey[1];
  ["d5ad715cdbbcf4d61a6cf85e7d7e20d09c5ad2683167454a0bdc67dd86b4401efbf9349b915f6e\
  2036bf9791f84df90e36f8c070fb3cd12a72c30dc04fcca208663ba7f2027af7edd6a9481d2ea84f\
  97213da2a78def50b50b059da1d3f5888b5708dcfeb7544a1526c4f334adb3db971f043d78bffc65\
  b7287ac31b5cfd810eb00f0697f8f75de30fd4ccd454ed0bc83cb06c98a6353a4f9b95b976ffb850\
  1879c77e50f121c26b6283cb10efac52d71453877948b387224522110fff795412d6cd38fc866f0f\
  36b708c16a27cb9576cd7194f1526c93414f1a89ef97417dccee2f048076c64c8397214283738764\
  8257b7f959d82cfd5bf47ed18ebc4ec245", "10001"]
```

We check the length:

```
> StringTools:-Length~(pk);
                                  [512, 5]
```

We see that the hexadecimal length of the modulus is 512—which, bearing in mind that the most significant digit is hexadecimal d, corresponds to a 2048-bit modulus. The encryption exponent is `"10001"` in hex which corresponds to the default value $2^{16} + 1$. The decryption exponent, on the other hand, is the following:

```
> sk := rsakey[2]: sk[2];
"5b8d64f5147171b60abc85ee76768438ddc54c41b8693716f2de0ce2adc4ad749921bd98aa40e93e8\
  436b30aa809409d0690ae63ba41e49cfb6ef7855eab9f015a4c55392c36e3e3d405d4102cdd90bc8\
  051984f9250e793aa954d64f91c4ad4e905b32ab0023d7bb290d063d1c92e5d21bcff59ad68c76c2\
  2bbf6bcd2ab0b902530d5c67c2e1bffa74cd00282d31e765b61e9502b96c6aa6de0c5f4897f2e3db\
  84ea8d247186f557b891827d587ac5a107615079bc151059275dc4e4c24ce15fee1fad5c2744aa8c\
  ef878117313782e08dc705c4aae4d2867f5e915b07eb5a3eebccfc5497fd9925eefaec2985de5f8e\
  26b2515a327bc99effb9bf878446441"
```

Exercise 8.8 Include in the function `RSAKeyGen` an additional check to ensure that the primes p and q that it generates are not too close, so that the modulus cannot be factored by Fermat's method. This is largely unnecessary when generating keys of realistic size but may serve to exclude extreme cases, such as the primes being equal, when generating very small 'toy examples'. Of course, these toy examples might also be banned altogether by including in the function an instruction requiring a minimum value for k.

Exercise 8.9 Modify `RSAKeyGen` in order to make it more efficient when e is small. This requires modifying `RSAPrime` to make it test whether a prime candidate x satisfies the condition $\gcd(x-1, e) = 1$ before submitting x to a primality test.

Exercise 8.10 Modify `RSAKeyGen` in order to provide the option of generating the encryption exponent in a pseudo-random way.

Exercise 8.11 Modify `RSAKeyGen` so that it chooses a $2k$-bit modulus n that is a product of two k-bit primes p, q, where p and q are pseudo-randomly chosen in the interval $[2^{(2k-1)/2}, 2^k - 1]$. To find the primes, use Blum–Blum–Shub to choose candidates in the interval $[2^{k-1}, 2^k - 1]$ and discard them when they are $< 2^{(2k-1)/2}$.

Exercise 8.12 Define a Maple function that, on input an RSA key in the format output by `RSAKeyGen`, checks compliance with the definition by making sure that $ed \equiv 1 \pmod{(p-1)(q-1)}$. Additional checks such as testing p and q for primality could also be included.

Exercise 8.13 Write a variant of `RSAKeyGen` that generates RSA keys in which the decryption exponent corresponding to the public key (n, e) is the inverse of e in $\mathbb{Z}^*_{\lambda(n)}$ instead of the inverse in $\mathbb{Z}^*_{\phi(n)}$. (Hint: Since the factorization of $n = pq$ is known, $\lambda(n)$ may be efficiently computed as $\lambda(n) = \mathrm{lcm}(p-1, q-1) = (p-1)(q-1)/\gcd(p-1, q-1)$. Maple's built-in function `numtheory:-lambda` computes $\lambda(n)$ but cannot be used for our purpose because it tries to factor n, which should be infeasible.)

We are now ready to give the plain RSA encryption and decryption functions. As we have mentioned, RSA is mainly used for short messages that can be encrypted in just one encryption operation (a modular exponentiation). However, we will also allow longer plaintexts that will be dealt with by dividing them into smaller blocks in the form explained above. Although it is easy to deal with plaintexts and ciphertexts given as files, in the same way we have seen for private-key encryption schemes, in the following we will deal exclusively with plaintexts and ciphertexts given as Maple strings and we leave to the reader the task of making the straightforward adaptations to deal with messages in file format.

We will start with a function called `bytestoblocks` that does the low-level conversion of a list of integers in the 0..255 range (bytes) of suitable size to 'blocks' that are ready to be processed by the RSA encryption function. The higher level encryption function will convert the plaintext string to a list of bytes using the 8-bit ASCII codes and then this list will be processed by this function that will partition it in suitable sublists (or blocks) each of which may be regarded as the integer in base 256 it defines, which will be close in size to but strictly smaller than the RSA modulus. This is achieved by forming the blocks in such a way that the number of bytes in each block is just one less than the number of digits (or bytes) of the modulus in base 256. The function converts a list of bytes given by the parameter `lis` to a list of blocks (lists of bytes) of length specified by the parameter `length`:

```
> bytestoblocks := proc(lis, length)
  local l, m;
  l := [];
  m := lis;
  while length <= nops(m) do
    l := [op(l), [op(1 .. length, m)]];
    m := [op(length+1 .. nops(m), m)]
  end do;
  if m <> [] then
    l := [op(l), m]
  end if;
  map(x -> ListTools:-DotProduct(x, [seq(256^i, i = 0 .. nops(x)-1)]), l)
  end proc:
```

We will also need a function that computes the integer logarithm (in other words, the floor of the logarithm) of a positive integer to a given integer base. Maple has the function `ilog` for this purpose, but this function uses real logarithms and sometimes round-off errors appear when computing the integer logarithm of a large integer which is close to a power of the base. Although it will be very unlikely that this gives an erroneous result for the logarithm of a randomly chosen RSA modulus, it is also straightforward to use the following function that computes the integer logarithm of an integer by what essentially amounts to doing a base change. To use this function to compute the integer logarithm of n in base 256 one just calls `intlog[256](n)`.

```
> intlog := proc(n::posint)
  local k, i, b;
  if type(procname, 'indexed') then
    b := op(procname)
  else
    error "logarithm base not specified"
  end if;
  if not type(b, 'posint') then
    error "the base must be a positive integer"
  elif b = 1 then
    error "the base must be greater than 1"
  end if;
  k := n;
  i := 0;
  while b <= k do
    k := iquo(k, b);
    i := i+1
  end do;
  i
  end proc:
```

Next we give the plain RSA encryption function for text strings. There are two required input parameters: `publickey`, for the RSA public key in the format given by the function `RSAKeyGen`, and `plaintext`, for the plaintext given as a text string. The final keyword parameter `outputformat` specifies in which format the ciphertext will be written. The default value is `hex`, which gives a more compact way of representing the ciphertext and, if another value is specified, the ciphertext will be given either as a list of several decimal numbers or as a unique decimal number if the plaintext is encrypted in a unique block.

```
> RSAEncrypt := proc(publickey::list, plaintext::string, outputformat::name := hex)
  local pk, blocks, blocksize, ciphertext, hexsize;
  if type(publickey[1], string) then
    pk := convert~(publickey, decimal, hex)
  else
```

```
    pk := publickey
  end if;
  blocksize := intlog[256](pk[1]);
  blocks := bytestoblocks(convert(plaintext, bytes), blocksize);
  ciphertext := Power~(blocks, pk[2]) mod pk[1];
  if outputformat = hex
    then ciphertext := StringTools:-LowerCase~(convert~(ciphertext, hex));
    hexsize := intlog[16](pk[1])+1;
    ciphertext := map(x -> cat('$'(0, hexsize-length(x)), x), ciphertext);
    cat(op(ciphertext))
  elif nops(ciphertext) = 1 then
    op(ciphertext)
  else
    ciphertext
  end if;
  end proc:
```

Example 8.3 Let us use the preceding function to encrypt a text. We are going to use the RSA key previously defined in Example 8.2 and, specifically, the public key with 2048-bit modulus that we have stored in the global variable `pk`. The text, which does not fit in a single RSA block with this key size, is the following:

```
> txt := "The great struggle in cryptology during the final quarter century of the
  millennium came over openness. A field that had for centuries been a secret
  government monopoly exploded into public consciousness under the impact of
  technology, economics, politics, and history.
  David Kahn, in 'The Codebreakers' - Chap. 27 (Cryptology goes public)."
```

We encrypt this text with the RSA public key `pk` generated above and we obtain:

```
> ctxt := RSAEncrypt(pk, txt);
"276c9dd2a30d2f6aea94b5a0a5c5928d303d993d30977ae90d49f650e10454839b08a766b7a855ed2\
  7791428cc51c4f1a5349ae563f80390989c1ea2fda47e6720895538de3c1886b53373038778691fd\
  ecc3b163dc7f9d993ecf1ed08887b06b45d64ce1f23bbf4af5c37a5b83e4e4d557cd5f29e3fbe082\
  769cd124f0e9b7f717a87cb98888a8b2514633926dab4b9e8c4f6ef16f3c35e7025cebd06c6d3449\
  2f5746844186dc3fb55f22d0728d57ed3890c75611752f6a05f7d7c613f236e24579f52bf16b83fe\
  09c50a5879d2dd1d68956a2837b9095062afb84d808556d1d2d913dd6b4787539d5a5a7588d6a2a3\
  c9f927501bfa7a09352be33cfa62164610a18a6f8428b193b7d4fc4eaffd8f7fac07d2f53599d93e\
  f9095d4bbfc19cf1f703d71f18813fb9e64a7df9a99740299f94bbb950d338f885209f55158f122e\
  5df930eef9f14e5981fd76089deba15981a9dbadca42c78aa00d07534441a44e1cdbfebb1f6c0e7d\
  7ddf4ab2e6829259a2ce84fa8b035c58e2b5ea96bb38d8ee1971633bc8241b9efe15a7c29b39cc5b\
  8fd86fc9b81556d4f8daa84a23a83cf5a2ab14cbc9e6a4a4d2b6bcaa0c04fc2d88442d1225d10b57\
  5f1d1099ddce20a67b4fdcd2bc718967abd1d67d186a270cc0553f554adf702cec28d8bc7746476f\
  ea89a827eef16702c7a5309cec92751da563f252b58c7e2c56a040c54beb458"
```

In Example 8.4 below, we show how to decrypt this ciphertext using the private key.

We now give the plain RSA decryption function. The inputs are a private key in the format produced by `RSAKeyGen` and a ciphertext in the format produced by `RSAEncrypt`. Actually, the private key need only be a list containing the first 6 items of a private key produced by the key generation function, because the last two items are included only to allow a slightly different decryption procedure (see Exercise 8.15) and are not needed by this function. The first six items in the list let the function decrypt by means of the Chinese remainder theorem in the form we have described and, in fact, the function only uses items 3 to 6 in the list but it would not work correctly if the first two were absent as they act like placeholders in this case. Of course, it is easy to modify the key generation function to omit the modulus and the decryption exponent if one so wishes and to modify the indexing in the decryption function accordingly. The output of the function is the plaintext given as a text string.

```
> RSADecrypt := proc(privatekey::list, ciphertext::list, posint, string)
  local sk, hexsize, c, plaintext;
  if type(privatekey[1], string) then
   sk := convert~(privatekey, decimal, hex)
 else
   sk := privatekey
 end if;
 if type(ciphertext, string) then
  hexsize := intlog[16](sk[1])+1;
   c := [StringTools:-LengthSplit(ciphertext, hexsize)];
   c := convert~(c, decimal, 16)
 elif type(ciphertext, posint) then
   c := [ciphertext]
 else
   c := ciphertext
 end if;
 plaintext :=
  chrem([Power~(c, sk[5]) mod sk[3], Power~(c, sk[6]) mod sk[4]],
  [sk[3], sk[4]]);
 convert(ListTools:-Flatten(convert~(plaintext, base, 256)), bytes)
 end proc:
```

Example 8.4 We decrypt the ciphertext `ctxt` obtained in Example 8.3 with the private key `sk` of Example 8.2. The result is the following:

```
> RSADecrypt(sk, ctxt);
 "The great struggle in cryptology during the final quarter century of the
    millennium came over openness. A field that had for centuries been a
    secret government monopoly exploded into public consciousness under the
    impact of technology, economics, politics, and history.
   David Kahn, in 'The Codebreakers' - Chap. 27 (Cryptology goes public)."
```

Exercise 8.14 Modify the function `RSADecrypt` in order that it uses only the (public) modulus and the decryption exponent, i.e., the two first terms of a private key in the format of `RSAKeyGen`.

Exercise 8.15 Modify the function `RSADecryt` so that it uses Eq. 8.1 to perform decryption. For this, the terms 3–8 of the private key list will be needed.

8.3.4 Security of RSA

We shall now discuss in more detail the security properties of RSA. We start with an attack that gives a total break of RSA encryption as it allows an eavesdropper to recover the private key from the public key. Fortunately, the attack requires the decryption exponent to be small, which is easily avoidable. Nonetheless, as we mentioned when discussing the practical aspects of RSA in 8.3.2, this has specific consequences as it precludes the possibility of choosing the decryption exponent small for efficiency reasons.

8.3.4.1 Wiener's Low Exponent Attack

This attack was presented by M.J. Wiener in 1990 and it uses the theory of continued fractions to compute the decryption exponent d efficiently from the public key (n, e). The attack assumes that $d < n^{1/4}/3$ and that $p < q < 2p$, where p and q are the prime factors of n (note that the latter hypothesis is commonly satisfied by RSA keys). Since $ed \equiv 1 \pmod{\phi(n)}$, there exists an integer $t < d$ such that $ed = 1 + t\phi(n)$. Moreover, the hypothesis on the prime factors of n implies that $p+q < 3p < 3n^{1/2}$. Thus, bearing in mind that $\phi(n) = n - (p + q) + 1$, we have:

$$\left|\frac{e}{n} - \frac{t}{d}\right| = \left|\frac{ed - tn}{nd}\right| = \left|\frac{1 + t(\phi(n) - n)}{nd}\right| \leq \left|\frac{t(p + q)}{nd}\right| < \left|\frac{3tn^{1/2}}{nd}\right| = \left|\frac{3t}{n^{1/2}d}\right|.$$

Now, since $d < n^{1/4}/3$ by hypothesis, we have that $3d < n^{1/4}$ and, since $t < d$, we also have that $3t < n^{1/4}$. This implies that $\left|\frac{3t}{n^{1/2}d}\right| < \left|\frac{1}{n^{1/4}d}\right|$, and combining these inequalities we obtain:

$$\left|\frac{e}{n} - \frac{t}{d}\right| < \left|\frac{1}{3d^2}\right|.$$

This inequality says that $\frac{t}{d}$ is a close approximation of $\frac{e}{n}$ and then the theory of continuous fractions allows the computation of d in polynomial time. We refer to [101,169] for more detailed explanations. There is a lattice-based extension of this attack due to Boneh and Durfee, with an efficient variant of Blömer and May that can recover the RSA decryption exponent d provided that $d < n^{0.292}$. These attacks, for which we refer again to [101], are heuristic but work well in practice and hence this bound should not be violated when choosing RSA keys. If the encryption exponent is chosen first then the probability that d turns out to be less than this bound is very small and if $e = 2^{16} + 1$ (or some other small exponent), as is often the case, then the size of d will surely be much larger than this bound because $ed > \phi(n)$ and the size of $\phi(n)$ is very close to that of n.

8.3.4.2 Some Reasons Why Plain RSA is Insecure

We will now deal with plain RSA because, although its security level is very weak, knowledge of its weaknesses provides helpful information in the quest for more secure schemes. We have seen that plain RSA is OW-CPA secure but now we explicitly show the reasons why this is a very weak security notion.

Example 8.5 Suppose that Alice sends the following ciphertext to Bob, whose public RSA key is `pk` in Example 8.2.

```
> ct :=
"04e05cec539418485e6e97bb4b1e521d39a470c2fc482998b5b18ceb32e747bd407efda65207d3ce0\
  bf70cc2355252880648217e52f3c08d4f67acc8d21021841cfddbdc996b628e7e4c8c042038835ad\
  c89b4a04386613fd2fdd6178ab898a0c0f4e9df2b74f1bdd43e18ca7b3a77a436e5b42ab154c8ceb\
  2ec2569ad722ccc695b5c2e3d412b627a2b32695df841d8203789c57ab4b851c2f3008d6ee2fe597\
  638b77430d901edb9d0d15f1f0477cb5317eafa7f57bc11b061a4f7454e7c3eebc14fbd74097765b\
  3a5b24d47f95bb5547ee7de44a95c75fe17903738fd1a41bac00635819ee3c1a658c15d328f293a0\
  287448b2db4d8077fdeeb2f4e0caa45"
```

Eve observes the ciphertext and would like to recover the plaintext. She knows (or she merely suspects) that the message contains Alice's answer to a question posed by Bob and that this answer is either "Yes" or "No". So she encrypts these two messages with Bob's public key and compares the results to the ciphertext `ct`:

```
> evalb(RSAEncrypt(pk, "Yes") = ct);
                                    false

> evalb(RSAEncrypt(pk, "No") = ct);
                                    true
```

This immediately reveals that the plaintext is indeed "No".

It might seem that in the preceding example the adversary breaks the one-wayness of plain RSA but it is really not so, and this does not contradict the fact that the scheme is OW-CPA secure under the RSA assumption. Indeed, this concept of security requires that it is hard for the adversary to decrypt a ciphertext corresponding to a plaintext which is randomly chosen in $\mathbb{Z}_n$ (where n is the RSA modulus) but in the example the plaintexts are supposed to belong to a small subset of $\mathbb{Z}_n$ so this condition is not satisfied. This attack can be extended to larger plaintext spaces provided that computing the ciphertexts corresponding to these plaintexts is feasible. In fact, there is a meet-in-the-middle attack, described in [37], which is much more efficient than a brute-force search. This attack relies on the multiplicative property of RSA which essentially says that the product of the ciphertexts corresponding to two messages in $\mathbb{Z}_n^*$ is the ciphertext corresponding to the product of these messages or, in other words, that the RSA function is a group homomorphism (actually an isomorphism) from $\mathbb{Z}_n^*$ into itself (*homomorphic* encryption schemes, for which the decryption function is a homomorphism as is the case here, are studied in more detail in Sect. 8.8). The attack may be sketched as follows. Suppose that $m \in \mathbb{Z}_n$ is an ℓ-bit plaintext and $c = m^e \bmod n$ is the corresponding ciphertext. Suppose, further, that m can be factored into two $\ell/2$-bit factors u, v (which occurs with good probability in case m is randomly chosen). We may even assume that these factors are in $\mathbb{Z}_n^*$ because otherwise we would have $\gcd(m, n) \neq 1$ and hence also that $\gcd(c, n) \neq 1$ which is easily detectable and, in fact, allows the factorization of n. Then the following holds:

$$c = m^e \bmod n = (uv)^e \bmod n = u^e v^e \bmod n.$$

Thus the algorithm tries to find u and v (and hence m) as follows. A table containing all the pairs $(r, r^e \bmod n)$, where r is an $\ell/2$-bit integer belonging to $\mathbb{Z}_n^*$, is constructed and sorted by the second component. Then, for each $\ell/2$-bit integer s, $cs^{-e} \bmod n$ is computed and searched among the second components of elements in the table. When $s = v$ we have that:

$$cv^{-e} \bmod n = u^e v^e v^{-e} \bmod n = u^e \bmod n,$$

which will be in the table as the second component of the pair $(u, u^e \bmod n)$. Thus we know that in our search we will find r and s such that $cs^{-e} \bmod n = r^e \bmod n$ and hence $c = r^e s^e \bmod n$. Once we find these elements, the RSA function corresponding to decryption, which is also an isomorphism of $\mathbb{Z}_n^*$, tells us that $m = rs \bmod n = rs$ (so that these r and s are the factors u, v we were searching for).

This attack, similarly to the birthday attacks we have seen for hash functions, reduces the number of encryptions to be performed to about the square root of the number required by a pure brute-force attack. In [37] it is shown that it is possible to recover an ℓ-bit message using $2 \cdot 2^{\ell/2}$ encryption operations (i.e., modular exponentiations) with a success probability of 18 %, which is certainly non-negligible. Since RSA is often used to encrypt symmetric session keys which may be 128-bit strings, recovering such a message would require about 2^{65} modular exponentiations and seems feasible with appropriate resources.

Both Example 8.5 and the attack just sketched offer compelling evidence that OW-CPA is not an adequate notion of security. Another reason why OW-CPA is very weak is because to break it the full plaintext must be recovered, so it allows partial information about the plaintext to be leaked without having to assume extra hypotheses such as a small message space. In other words, OW-CPA security does not imply semantic security. This can be appreciated, for example, by considering the Jacobi symbol and noticing that, since the RSA encryption exponent is odd, if m is a message and $c = m^e \bmod n$ the corresponding ciphertext, then $\left(\frac{m}{n}\right) = \left(\frac{c}{n}\right)$. Since, by Exercise 2.43, half of the elements of $\mathbb{Z}_n^*$ have Jacobi symbol 1 and the other half have Jacobi symbol -1, the knowledge of its Jacobi symbol gives about 1 bit of information about the plaintext (see, for example, [90,7.2.3] for an example of Lipton in which this feature was exploited to break an RSA-based protocol for 'mental poker').

The fact that plain RSA is not semantically secure also follows, in a more indirect way, from plain RSA having deterministic encryption. As we have already noted, a deterministic encryption scheme is not IND-CPA secure and, on the other hand, the latter property is equivalent to semantic security under chosen plaintext attacks (see [87, Theorem 5.4.11] for a proof).

8.3.4.3 Plain RSA is Not OW-CCA Secure

Since OW-CPA security is insufficient, let us analyze the possibility that plain RSA satisfies a stronger security property. One way to strengthen OW-CPA is by replacing the adversary goal, namely one-wayness, by a weaker goal like indistinguishability. This leads to IND-CPA but, as we have just remarked, plain RSA does not have this property. Another natural way to strengthen OW-CPA is by strengthening the adversary capabilities and replacing CPA by CCA to obtain what we can call OW-CCA (one-way security against (adaptive) chosen ciphertext attacks). But it is easy to show that plain RSA is not OW-CCA secure either, as the following example shows.

Example 8.6 Suppose that Eve intercepts the following ciphertext that Alice sends to Bob. The public key of Bob is, again, the value of `pk` in Example 8.2. Eve wants to recover the plaintext `m`.

```
> c := RSAEncrypt(pk, m, decimal);
90138139901950119024820914719901997304581171926799874747262337976955535868840237 87\
  47105281934160690005014031738117790204702745404894350444588369932348867238422482\
  58411418890363549042995404074652493229968514265882893474505993086560322750885651\
  21150154761398485614893692796218770764805883143898949647238924409445436033337718\
  35546149437997049833916726315768207604427628541524923907366640353192538207463607\
  39427463038749508842211134999261755810210905350018057384501908516676423051774092\
  73316470584127036980099203705226917872616502283181159585580232283138210496130198\
  31626078968020662888306170831457384237921530902272571 7
```

Eve then pseudo-randomly chooses an element $r \in \mathbb{Z}_n^*$, where n is the RSA modulus in the public key of Bob (she just chooses it in $\mathbb{Z}_n$ and then checks that it belongs to $\mathbb{Z}_n^*$, which occurs with overwhelming probability):

```
> with(RandomTools):
  n := convert(pk[1], decimal, hex):
  e := convert(pk[2], decimal, hex):
  MersenneTwister:-SetState():
  r := MersenneTwister:-GenerateInteger(':-range' = 0 .. n);
  igcd(r, n);
490824866389139260994433368622714162092083685112033955088608937180947131968989763 13\
  83829323204479697237508635335378131194154477831278267863578420194017705473129419\
  56351254618650801951111438217492636158352831931096220947698963914188698295961379\
  10792141474472001814487773268352895423211586046140653110789831640495317123204459\
  83643211823395871203738759662381824260404421504904517122258603622753476453622894\
  40324758815095419556809654161084112988630543770558992247261292457763711233144804\
  69938321488811567777140364544962519225036453910118341690883922720938999496204141\
  58463280345328248875205620411288343135098107933822 3278
                                                    1
```

Then Eve computes $f := r^e c \bmod n$ and sends it to Bob:

```
> f := Power(r, e)*c mod n;
12489642021847544778763521121552301587970076220339527856832940025599497431991873 84\
  37063802207879442544426205249128649864016817384385817965133190889363303911560209\
  24362964871814325196001079958929153805251611659880064078312932767681724146259790\
  97382105510447735012397026460172864188416313602754195924049905867079814752892199\
  79441626873204448490310154438686669727090251743048604915287270867812914813927785\
  33940704790883735732125243434465428203118683014248165699250223204491879723241207\
  27866465404097019219989416663668752041400271930624298432293263807269571726252406\
  63843982465837216741900888617000637100836578351651462 44
```

Bob decrypts f by computing `RSADecrypt`(sk, f) but obtains gibberish containing many non-printable characters. He wants to send it to Eve but, as it is non-printable, Bob computes the plaintext corresponding to f in decimal form (as an element of $\mathbb{Z}_n$) and obtains the following value x:

```
> x := Power(f, convert(sk[2], decimal, hex)) mod n;
98357907105992664070926434512400755219787394953346777078947899495396695408598104 75\
  96959780700523797996818131689990414044155033068744266833068151814460463805809784\
  10770168256986873562616093593478838099286816382554895563971343988511791821470026\
  59840101750030307401099786957066819102341901746975257929138473255689415468442999\
  49837410232051229512008883956140589987585635140991837577073880983946824608979742\
  13791031507856704231786204117678594549247575155825243681263481345469331878317051\
  52794598995507992790478933121441958912000012388805324781709636601251974621 1702\
  86807783491592484528775113734855491321705143476397 6470
```

Now, Bob sends x to Eve adding: Look at what you just sent me, it makes no sense! But Eve knows that x is the result of applying the decryption algorithm to the pair (sk, f), i.e., letting d be the decryption exponent (unknown to Eve):

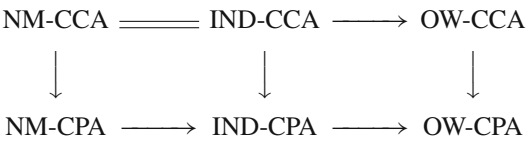
$$x = f^d \bmod n = (r^e c)^d \bmod n = rc^d \bmod n = rm \bmod n,$$

so that she can recover the plaintext m by computing $m = r^{-1}x \bmod n$. She computes it and converts it to the extended ASCII alphabet:

```
> convert(convert(x/r mod n, base, 256), bytes);
              "Eve will not be able to decrypt this message"
```

Eve finally recovers the plaintext!

Note that the fictional scenario in the preceding example faithfully models a CCA attack because the adversary is entitled to query the decryption oracle about any ciphertext distinct from the challenge ciphertext such as with the ciphertext f which is indeed distinct from c. The property that was exploited in this attack is the *malleability* of plain RSA. Roughly speaking, an encryption scheme is called *non-malleable* if, given a ciphertext, it is infeasible for an adversary to produce a distinct valid ciphertext for a related plaintext. In particular, a homomorphic encryption scheme such as plain RSA is always malleable because if $\mathbf{Enc}(pk, m) = c$, then it is always possible to pick an arbitrary plaintext a and construct the ciphertext $\mathbf{Enc}(pk, a) \cdot c$ which is also the result of encrypting the plaintext $a \cdot m$, related to the (unknown) plaintext m by multiplication with a. While it is intuitively clear that a malleable scheme will always be vulnerable to a CCA attack we remark, however, that malleability does not always imply that the full plaintext is recoverable under such an attack as in the previous example. Non-malleability requires, for example, that it should be infeasible to produce a ciphertext which corresponds to a plaintext which changes, say, a number contained in the original plaintext m, even if the adversary is not able to learn m. Thus non-malleability is a strong security property which is not a necessary condition for a public-key encryption scheme to be IND-CPA secure. In fact, for these schemes, non-malleability under chosen ciphertext attacks is equivalent to being IND-CCA secure (see [87, Proposition 5.4.34, Proposition 5.4.35] and also [14] for a detailed discussion of the relations among security properties). As we have seen, the adversary goal, OW, IND or NM (non-malleability), can be combined with the adversary capability (either CPA or CCA) and the relations between the security properties we have mentioned are summarized in the following diagram, in which an arrow means implication and an equality sign equivalence:



The previous discussion on the security of plain RSA can be summarized as follows:

Plain RSA is OW-CPA secure but is neither IND-CPA nor OW-CCA secure.

We next describe some attacks against plain RSA allowed by poor choices of some of the involved parameters and we stress that these attacks, even if they can be prevented by a more sensible parameter choice, show again the inadequacy of plain RSA.

8.3.4.4 Håstad's Broadcast Attack

One of the simplest attacks of this type arises when a small encryption exponent is used. As we have mentioned, this was quite common in the early years of RSA usage when, in order to gain efficiency, 3 was frequently used as encryption exponent.

Observe first that if $m \in \mathbb{Z}_n$ is a message such that $m^e < n$, then the ciphertext c corresponding to m is actually the eth power of m in the integers and so m can be computed from c by extracting an integer eth root, for which there are efficient algorithms. It is clear that for the condition $m^e < n$ to be satisfied, a small exponent is required (since we may assume that $m \neq 0, 1$, otherwise decrypting c is easy no matter what the exponent is). For example, when $e = 3$, any message $m \in \mathbb{Z}_n$ such that $m < n^{1/3}$ will satisfy this condition and hence will be easily recoverable. Of course, this problem is easily prevented by not allowing messages to be very small but there is a generalized attack that works even with arbitrary length messages provided that the same plaintext is encrypted with different RSA keys that use the same small encryption exponent. This attack is the following:

Proposition 8.4 (Håstad's broadcast attack) *Let* $(n_1, e), (n_2, e), \ldots, (n_t, e)$*, with* $t \geq e$*, be public RSA keys with pairwise relatively prime moduli and let* $n = \prod_{i=1}^{t} n_i$*. Then, if* m *is a plaintext such that* $m < \min(n_1, \ldots, n_t)$*, there is an algorithm that, on input the ciphertexts* $c_i = m^e \bmod n_i$ *and the keys* (n_i, e_i)*, computes* m *in polynomial time.*

Proof Since the n_i are pairwise relatively prime, the system of congruences:

$$\left.\begin{aligned} x &\equiv c_1 \pmod{n_1} \\ x &\equiv c_2 \pmod{n_2} \\ &\vdots \\ x &\equiv c_t \pmod{n_t} \end{aligned}\right\} \tag{8.2}$$

has, by the Chinese remainder theorem (Theorem 2.12) a unique solution in $\mathbb{Z}_n$, say C, and this solution may be computed in time $O(\text{len}(n)^2)$. But, by definition of the c_i, m^e is a solution of the system and, since $m < \min(n_1, \ldots, n_t)$ and $e \leq t$, $m^e \in \mathbb{Z}_n$. Thus $m^e = C$, so in order to compute m it only remains to compute the eth root of C over the integers, which can also be done in time polynomial in $\text{len}(n)$. □

Remark 8.2 Observe that the requirement that the moduli of the keys be pairwise relatively prime is not restrictive (assuming the keys are distinct). Indeed, if $\gcd(n_i, n_j) \neq 1$, then this gives the factorization of n_i and n_j.

The next function implements Håstad's broadcast attack in Maple. There are two input parameters which accept a list of public keys and a list of ciphertexts; both lists should have the same length and the ciphertexts should all be obtained from a common plaintext and should correspond, in order, to the keys in the list. Both public keys and ciphertexts should be in the decimal format provided by the functions `RSAKeyGen` and `RSAEncrypt`, respectively. It is a trivial matter to modify the function to also

accept public keys and ciphertexts in hexadecimal format but we shall not worry about this here. The function checks that the encryption exponent in all public keys is the same and that the number of key/ciphertext pairs is at least equal to the value of the encryption exponent. If these conditions are met, the function applies the Chinese remainder theorem in the form just described to obtain the eth power of the plaintext and, after extracting the eth root, the plaintext is returned as a character string.

```
> HastadBroadcastAttack := proc(publickeys::list, ciphertexts::list)
  local exps, e, moduli, blocks;
  if nops(publickeys) <> nops(ciphertexts) then
    error "There must be a ciphertext for each key"
  end if;
  exps := ListTools:-MakeUnique(op~(2, publickeys));
  if nops(exps) <> 1 then
    error "Not all public keys have the same exponent"
  elif nops(ciphertexts) < exps[1] then
    error "Insufficient number of ciphertext/key pairs"
  else
    e := exps[1]
  end if;
  moduli := ListTools:-Transpose(publickeys)[1];
  if type(ciphertexts, listlist) then
    blocks := ListTools:-Transpose(ciphertexts)
  else
    blocks := [ciphertexts]
  end if;
  blocks := root~(map(x -> chrem(x, moduli), blocks),e);
  convert(ListTools:-Flatten(convert~(blocks, base, 256)), bytes)
  end proc:
```

Example 8.7 Suppose that the message `txt` in Example 8.3 is encrypted with 17 RSA public keys all of which have 17 as encryption exponent. We are going to recover the message given the public keys and the corresponding ciphertexts. First, we generate the keys using the function `RSAKeyGen`. We will generate seventeen 2048-bit keys in decimal format as required by `HastadBroadcastAttack` (if they were in another format, it would be easy to convert them to decimal). For convenience and speed we will call `RSAKeyGen` with automatic seeding and we remark that, although this seeding method is not secure, the attack works exactly the same in case all the keys are generated with truly random seeds of adequate length. The list of public keys—which we do not print—is obtained as follows:

```
> pubkeys := [seq(RSAKeyGen(1024, 'e' = 17, format = decimal)[1], i = 1 .. 17)]:
```

Next, we encrypt the plaintext `txt` with all these public keys, again without printing the ciphertexts:

```
> ciphtexts := RSAEncrypt~(pubkeys, txt, decimal):
```

We already have all the data required to recover the plaintext without making use of the private keys. The process is very efficient in practice, and the plaintext is returned by Maple in hundredths of a second, but in order to save space we do not print the plaintext again; we merely check that it is the same as the plaintext that we keep in the global variable `txt`:

```
> evalb(HastadBroadcastAttack(pubkeys, ciphtexts) = txt);
                                    true
```

Remarks 8.7 The requirement that the same message be encrypted with several distinct RSA keys is not too difficult to meet if the encryption exponent—and hence the number of required encryptions—is small. For example, it is conceivable that a bank might send the same encrypted message to a number of clients.

On the other hand, the broadcast attack has been extended to the case in which several distinct plaintexts m_i are encrypted with the public keys, provided that these plaintexts are of the form $m_i = f_i(m)$, where the f_i are known polynomials. This generalized attack, due to Bleichenbacher, allows the recovery of m given the public keys and the ciphertexts corresponding to the m_i (see, e.g., [101] for the details).

8.3.4.5 Common Modulus Attacks

Since RSA keys are often generated by a trusted third party, a possibility considered in the early times of RSA was that all the keys shared the same modulus, each with a different encryption exponent (and hence also with different decryption exponents). However, this leads to a total break of the RSA scheme—not just the plain version but also other more powerful versions that we will discuss later on—because it allows any user of one of these keys to recover the private key of any other user by means of an already mentioned probabilistic Las Vegas algorithm due to Miller that, on input (n, e, d), produces the factorization of n. Another way to obtain a total break is based on the following result of DeLaurentis (see also [101, Theorem 3.1]):

Proposition 8.5 *Let (n, e) and (n, e') be public RSA keys such that $e \neq e'$ and let $d, d' \in \mathbb{Z}^*_{\phi(n)}$ be the decryption exponents corresponding to these keys. Let t be the greatest divisor of $ed - 1$ that is relatively prime to e', and let $d_1 = e'^{-1} \bmod t$. Then $RSA_{(n,d')} = RSA_{(n,d_1)}$, and d_1 can be efficiently computed from e, d, and e'. Therefore d_1 can be used as decryption exponent for the second key in place of d'.*

Proof Observe first that t may be efficiently computed by letting $t_0 = ed - 1$, $t_1 = \frac{t_0}{\gcd(e', t_0)}$, …, $t_{i+1} = \frac{t_i}{\gcd(e', t_i)}$, …, until t_j is found such that $\gcd(e', t_j) = 1$ and then setting $t = t_j$. Then we may compute d_1 as the inverse of e' in $\mathbb{Z}_t$ using the extended Euclidean algorithm. Next we show that $e'd_1 \equiv 1 \pmod{\phi(n)}$ for which, since by definition $e'd_1 \equiv 1 \pmod{t}$, it suffices to show that $\phi(n)|t$. Thus, if we write $t = (ed - 1)/s$, it is enough to show that $\phi(n)$ divides the numerator of this fraction and is relatively prime to its denominator. We have that $\phi(n)|ed-1$, because (n, e) is an RSA key with decryption exponent d. Moreover, all the prime factors of s are also factors of e' and so, since $\gcd(e', \phi(n)) = 1$ (because e' is the encryption exponent of an RSA key with modulus n), we also have that $\gcd(\phi(n), s) = 1$. Thus we see that $e'd_1 \equiv 1 \pmod{\phi(n)}$ and hence $e'd_1 \equiv e'd' \pmod{\phi(n)}$. Multiplying by the inverse of $e' \in \mathbb{Z}^*_{\phi(n)}$ we obtain that $d_1 \equiv d' \pmod{\phi(n)}$. Since discrete logarithms in the group $\mathbb{Z}^*_n$ are defined modulo the group order, namely $\phi(n)$, we

see that the exponents d_1 and d_1' indeed define the same exponentiation function.[3] Finally, we observe that all the computations involved are efficient. □

Remarks 8.8

1. As a consequence of the previous result, any RSA user (with public key (n, e) and decryption exponent d) can decrypt the messages addressed to any other user who shares the same RSA modulus (and hence has a public key of the form (n, e')) by computing the exponent d_1 in the above proposition.
2. Note that d_1 is not necessarily an element of $\mathbb{Z}^*_{\phi(n)}$ because t, and hence also d_1, may be greater than $\phi(n)$. However, since $d_1 \equiv d' \pmod{\phi(n)}$ and $d' \in \mathbb{Z}^*_{\phi(n)}$ by hypothesis, we have that $d' = d_1 \bmod \phi(n)$.

Exercise 8.16 Consider the RSA key obtained in Example 8.2 by setting:

```
> rsakey := RSAKeyGen(1024, 6180204295522709153838923244723480121 75,
                          4102756714646987802485435721275578031 86):
```

and, for brevity, call the modulus n, the encryption exponent e $(= 2^{16} + 1)$, and the decryption exponent d.

(i) Prove that there is another RSA key, `rsakey2`, with the same modulus as `rsakey` and with encryption exponent 257. Use Maple to find the key `rsakey2`.

(ii) Use the common modulus attack to find, given e, d and the encryption exponent 257 of `rsakey2`, an exponent d_1 that allows decryption of messages encrypted with `rsakey2`. Check that d_1 is not equal to the decryption exponent of `rsakey2` but they are congruent modulo $\phi(n)$.

The attack from Proposition 8.5 is called an *'insider attack'* because the adversary must be in possession of a private key with the same modulus as the attacked key. There is also an *'outsider attack'* against plain RSA that allows an adversary without knowledge of any private key to recover a message encrypted with two different keys that share the same modulus. For this, suppose that (n, e_1) and (n, e_2) are two public RSA keys such that $\gcd(e_1, e_2) = 1$ and let $m \in \mathbb{Z}_n$ be a message which is encrypted with both keys, producing ciphertexts $c_i = m^{e_i} \bmod n$, for $i = 1, 2$. An eavesdropper who observes these ciphertexts can recover the plaintext by using the extended Euclidean algorithm to compute integers r, s such that $re_1 + se_2 = 1$, and then computing $c_1^r c_2^s \bmod n$, since we have:

$$c_1^r c_2^s \bmod n = m^{re_1} m^{se_2} \bmod n = m^{re_1 + se_2} \bmod n = m.$$

Exercise 8.17 Exhibit a Maple demonstration of the 'outsider common modulus attack' along the following lines. Write a procedure `CommonModulusAttack` that, on input two public keys with the same modulus and two ciphertexts obtained by encrypting the same plaintext with these keys, produces as output the plaintext.

[3] Alternatively, once we have shown that $e'd_1 \equiv 1 \pmod{\phi(n)}$, the proof of Proposition 8.1 goes through to show that $\mathrm{RSA}_{(n,d_1)}$ is the inverse of $\mathrm{RSA}_{(n,e')}$ on $\mathbb{Z}_n^*$.

Then, generate two 2048-bit RSA keys with the same modulus and relatively prime encryption exponents (once the first key is generated, it is easy to modify it to obtain a second key satisfying the required conditions). Finally, encrypt a message with both keys and use `CommonModulusAttack` to compute the plaintext using as input the two ciphertexts and the public keys.

These attacks show that the use of a common modulus should be avoided in any RSA implementation. This is not difficult because if the prime factors of the modulus are randomly chosen for each user, the probability that two users get keys with the same modulus is negligible.

8.3.4.6 Side-Channel Attacks

Before studying more secure versions of RSA, we are going to briefly mention a different class of attacks that may be addressed against all versions of the scheme or, rather, against their implementations. These attacks are called *side-channel attacks* because they exploit information about the private key or the plaintext that may leak through physical channels such as power consumption, timing behavior or even the presence of faulty computations. Attacks of this type are not specific to RSA and may be also addressed against other encryption schemes.

One of the best known side-channel attacks is *Kocher's timing attack*, discovered by P. Kocher in 1995 (see [121]). Kocher showed how an adversary can find the RSA decryption exponent bit by bit (from the least significant bit to the most significant one) by carefully measuring computation times for a series of decryptions—whose ciphertexts are known to the adversary—and making a statistical analysis of these times. The attack is based on the hypothesis that decryption uses the binary exponentiation algorithm in which the operations to be performed for each bit of the exponent vary according to whether it is a 0 bit or a 1 bit. For a detailed description of the attack we refer to the original article by Kocher [121].

This attack came as a surprise but, once known, is easy to prevent. A simple way is to make the implementation perform decryptions in constant time. Alternatively, following an idea of Rivest, the attack may be prevented by using *blinding*, which consists of multiplying the ciphertext, prior to decryption, by $r^e \mod n$, where r is a randomly chosen element of $\mathbb{Z}_n^*$ which, because of the multiplicative property of RSA, can be factored out from the plaintext after decryption. This defeats the timing attack because the adversary now does not know the ciphertext which is being submitted to the binary exponentiation algorithm.

A natural prolongation of the idea behind timing attacks is to analyze how power consumption of the microprocessor performing a decryption varies over time. The binary exponentiation method proceeds sequentially by performing a squaring and a multiplication for each '1' bit of the decryption exponent d, and just a squaring for each '0' bit of d. The analysis of the power trace makes it possible to distinguish between a '1' bit, which corresponds to a longer activity period, and a '0' bit, for which the activity period is shorter. This way, the decryption exponent can be dis-

covered by the adversary, one bit at a time. There are several ways of preventing this attack, a simple one being to perform a 'dummy' multiplication after each squaring corresponding to a '0' bit. There are also more sophisticated versions of this attack that we shall not describe here.

Another important attack that may be regarded as side-channel is Bleichenbacher's attack against PKCS #1 v1.5, which is discussed below. In this attack, the side channel provides information—in the form of message errors—about the result of RSA decryptions.

8.3.5 RSA with Probabilistic Encryption

Plain RSA is, as we have seen, insecure, and this is primarily a consequence of the fact that its encryption algorithm is deterministic. Because of this, RSA is not IND-CPA secure and, if one analyzes the specific attacks we have presented, it is easy to notice that determinism plays a determinant role in most of them. Thus the basic idea to try to make RSA IND-CPA or even IND-CCA secure is to use random padding of messages before encryption in order to make the process probabilistic. The simplest way to do this is just to choose a random string of bits of appropriate length (which will usually depend on the key size), concatenate it to the message and interpret the result as an element of $\mathbb{Z}_n$ which is then encrypted with the RSA function. However, it is convenient to point out that this procedure does not guarantee that a CPA secure scheme will be obtained as there is no proof of security for such a loosely defined scheme and in fact, in some situations in which certain additional conditions are met, even a randomly padded version of RSA may be shown to be insecure.

For example, there is a variant of the attack against short encryption exponent, called *Coppersmith's short pad attack*, which works even with some random padding appended at the end of the message. Suppose that Alice sends Bob a ciphertext obtained from a properly padded message and Eve intercepts it, preventing Bob from receiving it. Since Bob does not answer, Alice re-encrypts the message and sends it again to Bob. But Eve gets also the second ciphertext and now she has two ciphertexts corresponding to two encryptions of the message with different random pads. Then Eve might be able to decrypt the message because of the following result of Coppersmith [54] (see also [33] for a detailed description which includes an attack by Franklin and Reiter on which Coppersmith's attack is based):

Theorem 8.2 *Let (n, e) be an RSA public key with $\text{len}(n) = s$ and let $t = \lfloor s/e^2 \rfloor$ and $m \in \mathbb{Z}_n^*$ a message of at most $s - t$ bits. Let $m_1 = 2^t m + r_1$ and $m_2 = 2^t m + r_2$, where r_1 and r_2 are distinct integers such that $0 \leq r_1, r_2 < 2^t$. Then m can be efficiently recovered from (n, e), and the ciphertexts c_1 and c_2 corresponding to m_1 and m_2.*

Note that this implies, in particular, that when $e = 3$, then if $r_1, r_2 < n^{1/9}$ the attack succeeds. Thus, when $e = 3$, the pad length should not be less than $1/9$ of

the total message length. Although Coppersmith's attack will not work with more sophisticated padding methods, it is one of the reasons why the value $e = 3$ is not recommended for RSA implementations and the value $e = 2^{16} + 1$ is commonly used instead. For the latter value of e this attack is unsuccessful if a modulus of reasonable size is being used.

Coppersmith's attack suggests that the padding length for RSA should not be too small in comparison with message length. Suppose that ℓ is a function such that $\ell(k) \leq 2k - 1$ and that the plain RSA encryption scheme defined in Definition 8.12 is modified as follows:

- **Gen**: As in Definition 8.12, on input 1^k it outputs a public key $pk = (n, e)$ and a private key $sk = (n, d)$ such that $\text{len}(n) = 2k$.
- **Enc**: On input a public key $pk = (n, e)$ and a message $m \in \{0,1\}^{\ell(k)}$, a random string $r \leftarrow \{0,1\}^{2k-\ell(k)-1}$ is chosen and the output is the ciphertext $c := (r||m)^e \bmod n \in \mathbb{Z}_n$ (where $r||m$ is interpreted as the element of $\mathbb{Z}_n$ given by its binary expansion).
- **Dec**: On input a private key $sk = (n, d)$ and a ciphertext $c \in \mathbb{Z}_n$, the message $m := LSB_{\ell(k)}(c^d \bmod n) \in \mathbb{Z}_n$ is computed, where LSB_x denotes the function that assigns to an integer the integer defined by its x least significant bits.

The security of this encryption scheme depends on the function ℓ, whose value should not be too close to $2k$ in order to allow a substantial amount of padding and prevent brute-force attacks in which all the possible padding values are searched. A possibility that has been used in practice is to use padding equal in length to the unpadded message, i.e., to take $\ell(k) = k$ and, while the resulting scheme has not been proved to be CPA secure, it seems reasonable to conjecture that this is indeed the case under the RSA assumption. A related construction is [87, Construction 5.3.16], which has a security reduction to an assumption that may be stronger than the RSA assumption. At any rate, even without a security reduction, the padded RSA scheme is clearly preferable to plain RSA, which is demonstrably insecure.

On the other hand, the above-defined padded RSA scheme can be proved CPA secure if the value of $\ell(k)$ is small compared with k and, more specifically, if $\ell(k) = O(\ln k)$. This follows from the main result in [98] where, building on early results about the security of the least significant bits in an RSA-encrypted message, the following fact—which we only state in an informal way—is proved:

The RSA assumption is equivalent to the impossibility of distinguishing blocks of $O(\ln \ln n)$ *bits (and hence blocks of* $O(\ln k)$ *bits in the above notation) of a message* m *from blocks of random bits, given the ciphertext corresponding to* m.

Apart from [98], we refer also to [109] for further discussion of these aspects as well as for a general construction ([109, 10.7.2]) that builds a CPA secure public-key encryption scheme from any family of one-way trapdoor permutations (such as the RSA family under the RSA assumption) by using the existence of hard-core predicates provided by the Goldreich–Levin theorem ([86, Theorem 2.5.2]).

We see that there are ways to obtain CPA secure encryption schemes from the RSA function but, as we have mentioned, the security property one should aim for is IND-CCA security. This is more difficult to attain than CPA security but it is clear

that the simple CCA attack we have seen against plain RSA no longer works if the plaintext is randomly padded before encryption, because the padding destroys the multiplicative property on which this attack was based. However, using randomized padding does not guarantee that IND-CCA security will be attained and, in fact, we are going to see an example of an RSA-based encryption scheme that is believed—although not proved—to be CPA secure but against which there is a practical CCA attack. This scheme is the one used in the PKCS #1 v1.5 standard, which belongs to a family of standards (*Public Key Cryptography Standards*) published by RSA Labs that have become part of many other standards, including the SSL/TLS protocol used by web browsers in their transactions.

8.3.5.1 Bleichenbacher's CCA Attack Against PKCS #1 v1.5

PKCS #1 v1.5 uses a version of padded RSA which pads messages before encryption in the following way. Suppose that (n, e) is an RSA public key, where the modulus n is assumed to be a t-byte integer, i.e., $2^{8(t-1)} \leq n < 2^{8t}$. Let m be a message, which is assumed to have s bytes, where $s \leq t - 11$. The message is then padded to obtain a t-byte formatted plaintext block as follows:

$$b = 00||02||\text{padding string}||00||m.$$

Here, a 00 byte (in hexadecimal notation) is followed by a 02 byte, then a random padding string consisting of $t-s-3$ nonzero bytes, then another 00 byte and, finally, the original message m (note that, since $t - s \geq 11$, the length of the padding string is ≥ 8). Then this block b is converted to an element $x \in \mathbb{Z}_n$ in the standard way and encrypted with RSA, producing the ciphertext $x^e \bmod n$. A formatted data block b is called *PKCS conforming* if it has the above format. If we let $B = 2^{8(t-2)}$, then $2B$ is represented by the following byte string:

$$2B = 00||02||\underbrace{00||00||\ldots||00}_{t-2 \text{ times}},$$

and, similarly, the byte string corresponding to $3B$ is:

$$3B = 00||03||\underbrace{00||00||\ldots||00}_{t-2 \text{ times}}.$$

The data blocks formatted by the method indicated above range from $00||02||00||00||\ldots||00$ to $00||02||ff||ff||\ldots||ff$ and hence we see that if a data block b is PKCS conforming then the corresponding integer $x \in \mathbb{Z}_n$ satisfies:

$$2B \leq x < 3B.$$

Bleichenbacher's CCA attack against PKCS #1 v1.5, presented in [29], exploits the peculiarities of the padding method used for RSA encryption in this standard, as well as the fact that a server returned an error message each time a ciphertext was received that, after decryption, did not produce a PKCS conforming block. Thus the server, in some weak sense, acted as a decryption oracle because, although it did not return the plaintext, it gave some information about it.

So, suppose Eve wants to decrypt a ciphertext c obtained by encrypting a PKCS conforming block b. She knows that this block (or, rather, the integer defined by it, but we shall not distinguish between them in what follows), belongs to the interval $[2B, 3B - 1]$ and she will proceed in a series of steps. At each of these steps, a set of subintervals of $[2B, 3B - 1]$ is determined such that b belongs to one of them, and these intervals become shorter after each step. The process ends when a set consisting of only one interval of length 1 is reached, as this uniquely determines the value of b.

The procedure used to find these sets of intervals is based on the malleability of the scheme that makes it possible to modify c to produce ciphertexts whose plaintexts are related to b in a predictable way. It proceeds by choosing integers s and computing the ciphertexts $c' = cs^e \bmod n$ which are then sent to the decryption oracle. If the oracle says that c' is PKCS conforming (i.e., if the server does not return an error message) then Eve knows that bs belongs to $[2B, 3B - 1]$ and this will allow her to narrow the interval to which b must belong by adaptively choosing the multiplier s depending on the output of the previous computations.

We refer to [29] for the details of the attack, which will typically require about 2^{20} queries to the oracle in order to be able to decrypt a message encrypted with a 1024-bit key. Although the high number of queries makes it difficult to implement in practice, this attack came as a surprise when it was discovered in 1998 and the insecurity is so serious that at the time it provoked an immediate change in web browser software. The easy way to counter this attack in practice would be to not allow the server to act as an oracle by severely limiting the number of error messages returned. But this solution is not very satisfactory because it does not address the core of the matter, which is the weakness of the padding method used. The goal then was to find a better padding method that could be proved to be CCA secure and this search led to OAEP (*Optimal Asymmetric Encryption Padding*), which is currently the algorithm of choice for obtaining secure RSA encryption.

8.3.6 RSA-OAEP

Bleichenbacher's attack was the reason that the PKCS standard was changed by adopting a more secure padding method introduced by Bellare and Rogaway in [19]. This padding method can be combined with a (candidate) one-way trapdoor permutation such as the RSA function to obtain a public key encryption scheme.

The idea that led to the RSA-OAEP algorithm was to try to reach a compromise between security and efficiency, as experience has shown that practitioners are reluctant to use more secure systems when they are less efficient. The security goal was IND-CCA in the *random oracle model* which we have already mentioned in pass-

ing. In the random oracle model, all entities, including the adversary, have access to one or several functions from $\{0,1\}^a$ to $\{0,1\}^b$ for certain specified values of a and b. The security definitions are reformulated in this model in such a way that in the attack experiment each of these functions is chosen at random from the set of all functions from $\{0,1\}^a$ to $\{0,1\}^b$ (i.e., these functions are modeled as *random oracles*). In the 'real world' these random oracles are typically instantiated as cryptographic hash functions but it is clear that the latter do not have the required randomness property. Thus a security reduction in the random oracle model does not imply that an implementation of the scheme is actually secure. In fact, in [46] an example—admittedly contrived and hard to imagine happening in practice—is given of a signature scheme which is secure in the random oracle model, but insecure under any instantiation of the random oracle.

Despite these reservations, the random oracle model is seen as a good heuristic. A reductionist security proof in this model is considered better than no security proof at all and designing systems based on such proofs when no other proofs are available—or if they are, it is only for very inefficient schemes—is regarded as a sound engineering principle (see [109, Chap. 13] for an in-depth discussion of these issues).

We next describe the RSA-OAEP algorithm. The required ingredients are the following:

- The plain RSA encryption scheme.
- Associated with a security parameter 1^k, two parameters k_0, k_1 such that $0 < k_0, k_1$, $k_0 + k_1 < k - 1$, and both 2^{-k_0}, and 2^{-k_1} are negligible as functions of k.
- The plaintext space is $\{0,1\}^l$, where $l = k - k_0 - k_1 - 1$. Thus $k - k_0 - 1 = l + k_1$.
- Two functions $G : \{0,1\}^{k_0} \to \{0,1\}^{k-k_0-1}$, $H : \{0,1\}^{k-k_0-1} \to \{0,1\}^{k_0}$. These functions are modeled as independent random oracles in the security analysis of the scheme and in practice they are derived from a hash function such as SHA-256.

Given these tools, a public-key encryption scheme may be constructed as follows:

Definition 8.13 RSA-OAEP.
RSA-OAEP is the public encryption scheme (**Gen**, **Enc**, **Dec**), where the algorithms are defined as follows:

- **Gen**: On input 1^k, with k even, run $\mathbf{Gen}_{\mathrm{RSA}}(1^{k/2})$ to obtain (n, e, d) with $\mathrm{len}(n) = k$. The public key is (n, e) and the private key is (n, d) (here, for notational convenience and without loss of generality, the bit length of the modulus is k instead of $2k$ that we used in plain RSA).
- **Enc**: Given the message $m \in \{0,1\}^l$ and the public key (n, e), the following values are successively computed:

1. $r \leftarrow \{0,1\}^{k_0}$ (so that r is a randomly chosen bit string of length k_0).
2. $s := G(r) \oplus (m||0^{k_1}) \in \{0,1\}^{k-k_0-1}$ (where $\oplus$ denotes, as usual, the bitwise Xor operation).
3. $t := H(s) \oplus r \in \{0,1\}^{k_0}$.
4. $m' := s||t \in \{0,1\}^{k-1}$.

5. $c := m'^e \bmod n \in \mathbb{Z}_n$ (Here, m' is regarded as an element of $\mathbb{Z}_n$ given by its binary representation, and encrypted with the RSA function using the public key. The resulting ciphertext $c \in \mathbb{Z}_n$ may also be regarded as a bit string of length k completing with zeros on the left if necessary).

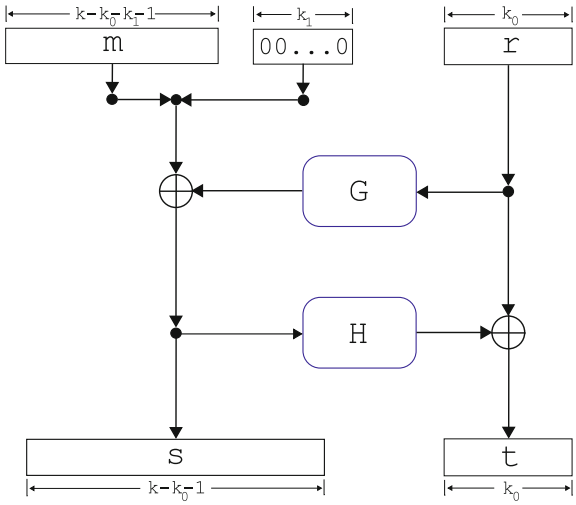
The output of **Enc** is the ciphertext c.

- **Dec**: Given the ciphertext $c \in \mathbb{Z}_n$ and the private key (n, d), the following values are successively computed:

1. $w := c^d \bmod n \in \mathbb{Z}_n$, which can be regarded, by using its binary expansion completed with leading zeros if necessary, as an element of $\{0, 1\}^k$.
2. $s := MSB_{k-k_0-1}(w)$ (recall that $MSB_i(b)$ and $LSB_i(b)$ denote, respectively, the i most significant bits and the i least significant bits of a bit string b).
3. $t := LSB_{k_0}(w)$.
4. $r := H(s) \oplus t$.
5. $z := G(r) \oplus s$.
6. $u := LSB_{k_1}(z)$.
7. $m := MSB_l(z)$.

The output of **Dec** is m if $u = 0^{k_1}$ and $\perp$ otherwise (in this case the ciphertext is rejected).

OAEP is the padding method implicitly defined above. It encodes the message m as $m' := \text{OAEP}(m, r) = s||t$, where s and t are as in Definition 8.13. A graphical depiction of OAEP padding is the following:



OAEP Encoding

Remarks 8.9

1. OAEP can be used in combination with any one-way trapdoor permutation (not necessarily RSA) to define an encryption scheme and RSA-OAEP is the result of combining RSA with OAEP.
2. The requirement that 2^{-k_0} and 2^{-k_1} are negligible functions of k means that neither k_0 nor k_1 should be 'too small' in comparison with k. For example, this rules out the possibility that $k_0 = O(\ln k)$ or $k_1 = O(\ln k)$, but the condition is satisfied whenever both k_0 and k_1 are $\Omega(\ln^2 k)$. For practical purposes, Bellare and Rogaway mentioned in [19] that the adversary's running time should be significantly smaller than 2^{k_0} steps and they suggested taking $k_0 = k_1 = 160$ (agreeing with the output length of the hash function SHA-1), which allows the encryption, in a single operation, of messages of maximum length l equal to $k - 321$.

8.3.6.1 Security of RSA-OAEP

It is easy to realize that the previously described cryptanalytic attacks against plain RSA no longer work against RSA-OAEP. Among other things, OAEP is probabilistic and it does not preserve multiplicativity. Thus, neither the attack against small message space nor the chosen ciphertext attack against plain RSA can be reproduced for this scheme. The small encryption exponent attack is also prevented because OAEP will produce a different encoding each time a given message is encrypted.

Observe that OAEP, defined by $(m, r) \mapsto s||t$, is an injective map:

$$\text{OAEP} : \{0,1\}^{k-k_0-k_1-1} \times \{0,1\}^{k_0} \to \{0,1\}^{k-1}$$

(injectivity follows from the fact that, as seen in the description of **Dec** in Definition 8.13, (m, r) can be univocally recovered from $m' = \text{OAEP}(m, r)$). Since the domain of this function is a set of 2^{k-k_1-1} elements and the codomain is a set of 2^{k-1} elements, we see that the probability that a random string of $k - 1$ bits belongs to the image of OAEP (under the above parameters) is:

$$\frac{2^{k-k_1-1}}{2^{k-1}} = 2^{-k_1},$$

which, as we have mentioned, is negligible as a function of k. In addition, the assumption that G and H are random oracles suggests that it may indeed be hard to produce a $(k - 1)$-bit string which is a valid OAEP encoding $m' = r||s$ of a message $m \in \{0,1\}^{k-k_0-k_1-1}$ without previous 'knowledge' of m. This property is called *plaintext awareness* and its precise formulation implies some subtleties whose details we are not going to discuss here.

From the results in [19] it turns out that RSA-OAEP is only *weakly plaintext aware* which, roughly speaking, means that the adversary cannot produce—without knowing the plaintext—a valid ciphertext until it has seen another valid one. As a

consequence, an encryption scheme obtained by combining OAEP with a one-way permutation is secure against indifferent chosen ciphertext attacks (IND-CCA1). This is because such a scheme is CPA secure and a CPA secure and weakly plaintext aware scheme is IND-CCA1 secure, the intuition being that the adversary cannot profit from having access to the decryption oracle 'at lunchtime' (see the remarks after Definition 8.8) since it cannot submit valid ciphertexts in its queries. However, this restriction no longer applies after the adversary is given the challenge ciphertext and hence it does not follow that such a scheme is IND-CCA2 secure. In fact, while the latter property is obtained if the stronger form of plaintext awareness is assumed—when a valid ciphertext is hard to produce even after seeing the challenge ciphertext—Shoup observed in [179] that it is very unlikely that a proof of IND-CCA2 security can be obtained for a scheme constructed from a general trapdoor permutation plus OAEP. However, in case the trapdoor permutation is the RSA function (under the RSA assumption) it was proved, first by Shoup in the case $e = 3$ and then in general, in [80], that this security property indeed holds, namely, that RSA-OAEP is IND-CCA2 secure in the random oracle model. Somewhat ironically, a crucial feature of the latter proof is that it uses the homomorphic property of RSA, which is just what makes plain RSA so vulnerable to CCA attacks. We refer to [155] for a detailed discussion of the security properties of RSA-OAEP and other public-key encryption schemes.

In addition to the different notions of security, there are other aspects that allow the establishment of finer gradings when evaluating the security level attained by a given scheme. Since all security proofs are reductions, there are two main factors that affect the security level:

- *Strength of the hypothesis*. The reduction shows that a scheme has a security property assuming that a certain problem is hard. For example, when dealing with RSA-based schemes one may assume that the RSA problem is hard but also other hypotheses which are potentially stronger than the RSA assumption may be considered. Obviously, the weaker the hypothesis, the stronger the security.
- *Tightness of the reduction*. The reduction usually proceeds by showing that if there is an efficient (i.e., PPT) algorithm to break the system, then there is also an efficient algorithm to solve the hypothetically hard problem. But it may happen that the algorithm that breaks the scheme is much more efficient, in practical terms, than the algorithm that solves the underlying hard problem, in which case we would say that the reduction is not tight (or that it is loose). In slightly more precise terms, the reduction proceeds by transforming an attacker that breaks the scheme with probability ε and within time t into a solver of the hard problem with probability ε' and time t'. Then one says that the reduction is *loose* if $\frac{t'}{\varepsilon'} \gg \frac{t}{\varepsilon}$ and that it is *tight* if $\frac{t'}{\varepsilon'} \approx \frac{t}{\varepsilon}$.

The proof that RSA-OAEP is CCA secure in the random oracle model under the RSA assumption may be evaluated taking into account these aspects. In relation to hypothesis strength, the RSA assumption used in the proof is the standard hypothesis for RSA-based schemes but there is also an additional hypothesis instantiated in the

use of the random oracle model, which entails the assumption that the functions G and H in Definition 8.13 are random oracles. We have already mentioned that the proof thus obtained is heuristic but it is now widely accepted that it provides valuable evidence for the security of the scheme. This proof shows that a successful attack on RSA-OAEP must either break the RSA assumption or exploit some particular properties of the functions G, H. Thus it can also be interpreted as providing assurance of security under the hypothesis that the adversary is generic and does not exploit any specific weaknesses of the hash functions used in practice. In addition, RSA-OAEP has for now resisted attack and, for example, it has proved to be resistant to Bleichenbacher's and similar attacks, which were not known at the time OAEP was designed.

The other factor that must be taken into account in order to evaluate the level of security provided by the proof in [80] is that, unfortunately, the reduction in this proof is not tight. This means that the proof does not give assurance that breaking RSA-OAEP with a 2048-bit modulus will require an amount of work similar to that required for breaking the RSA assumption with the same modulus. Hence, as mentioned in [155], in practical terms the proof is only meaningful for huge RSA moduli with more than 4096 bits (in contrast with the fact that the smaller moduli currently in use, such as 2048 bits, seem to be large enough to prevent breaking of the RSA assumption). We stress, however, that the fact that the proof allows the existence of a large gap between the hardness of the two problems does not mean that such a large gap must actually exist and, so far, breaking RSA-OAEP in practice seems no easier than breaking the RSA assumption (or even than factoring the RSA modulus). As a consequence, although from a strict security-theoretic viewpoint it would be perhaps advisable to use RSA moduli more than 4096 bits long, smaller moduli continue to be used for efficiency and there is a certain consensus that, even with these smaller moduli, RSA-OAEP is probably safe.

In conclusion, RSA-OAEP provides a good compromise between security and efficiency and, although its security proof is heuristic, it adds confidence in the scheme which, because of this fact, becomes preferable to other schemes for which no such proof exists, and it is currently being extensively used in practice.

8.3.6.2 RSAES-OAEP from PKCS #1 v2.1

We are now going to describe the specific implementation of RSA-OAEP contained in PKCS #1 v2.1 (called RSAES-OAEP, where the initials RSAES stand for "RSA Encryption Scheme", see [154]), with a view towards its implementation in Maple which is given below in 8.3.7. Although the scheme fits in the general framework described in Definition 8.13, there are some differences and specific requirements that we are going to make explicit. We start with the following ingredients that will be needed (here we use notation similar to that in [154]):

- An RSA public key (n, e) and its corresponding private key. The modulus n will be assumed to have byte length k. This scheme is byte-oriented, so that the length of all bit strings will always be a multiple of 8 and these strings will be regarded

as byte strings; for example, if n is a 2048-bit modulus, then we have $k = 256$. We also remark that in PKCS #1 v2.1 the decryption exponent d corresponding to an RSA public key (n, e) is only required to be a positive integer less than n satisfying $ed \equiv 1 \pmod{\lambda(n)}$. Thus, in general, d may take several possible values, among which we find the one in our definition, namely, the inverse of e in $\mathbb{Z}^*_{\phi(n)}$, and also the variant in which d is the inverse of e in $\mathbb{Z}^*_{\lambda(n)}$. As we have remarked, these two values are, in general, distinct and, when this is the case, the latter is smaller than the former.

- A hash function Hash whose outputs are *hLen* bytes in length. We will later take Hash = SHA-256 in our implementation, so that in this case $hLen = 32$ (this is the value—given in bytes—for the parameter k_0 in the previous description of RSA-OAEP).
- A *Mask Generating Function* MGF that takes as input a byte string *mgfSeed* and a desired output length $maskLen \leq 2^{32} hLen$, and produces as output MGF(*mgfSeed*, *maskLen*) = *mask*, where *mask* is a byte string of length *maskLen*. MGF will be defined from Hash according to the recommendation in [154], as follows. Consider the byte strings:

$$\text{Hash}(mgfSeed||i),\ 0 \leq i \leq \lceil maskLen/hLen \rceil - 1,$$

  where i is regarded as a byte string of length 4. Successively concatenate all of them to obtain:

$$T := \text{Hash}(mgfSeed||1)||\text{Hash}(mgfSeed||2)||\ldots$$

  and output *mask*, the string consisting of the leading *maskLength* bytes of T.
- A message M to be encrypted, given as a byte string of length *mLen*, where $mLen \leq k - 2hLen - 2$.
- An optional label L to be associated with the message, with default value the empty string.
- The ciphertext C, a byte string of length k.

We next describe the algorithms used by RSAES-OAEP. We start with the OAEP encoding (i.e., the padding algorithm), called EME-OAEP.

EME-OAEP encoding.

1. Set the optional label L and compute $lHash := \text{Hash}(L)$. The default value for L is the empty string.
2. Generate a byte string *PS* consisting of $k - mLen - 2hLen - 2$ zero bytes. Note that the *mLen* specification above allows this length to be 0, in which case *PS* is the empty string.
3. Form a data block *DB* of $k - hLen - 1$ bytes (here and in what follows we denote a single byte by two hexadecimal digits):

$$DB := lHash||PS||01||M.$$

4. Generate a random byte string *seed* of length *hLen*.
5. Compute $dbMask := \text{MGF}(seed, k - hLen - 1)$.
6. Compute $maskedDB := DB \oplus dbMask$.
7. Compute $seedMask := \text{MGF}(maskedDB, hLen)$.
8. Compute $maskedSeed := seed \oplus seedMask$.
9. Output the k-byte encoded message EM obtained as:

$$EM := 00||maskedSeed||maskedDB.$$

Once the message is encoded by means of EME-OAEP, it can be encrypted with the RSA public key:

RSA encryption.

The encryption algorithm proceeds by converting EM to an element $m \in \mathbb{Z}_n^*$ in the natural way, then encrypting m by means of the RSA function, obtaining $c := m^e \bmod n$ and, finally, converting c to a byte string C of length k, which is the output of RSAES-OAEP encryption.

Decryption begins by recovering the encoded message EM from the ciphertext:

RSA decryption.

The ciphertext C is first converted to an integer $c \in \mathbb{Z}_n^*$ in the standard way, then the RSA decryption primitive is applied to obtain $m := c^d \bmod n$ and the integer m is converted to an encoded message of k bytes, EM.

Finally, decryption is completed by using EME-OAEP decoding to recover the message M from the encoded message EM:

EME-OAEP decoding.

1. Set the optional label L and compute $lHash := \text{Hash}(L)$. The default value for L is the empty string.
2. Separate the encoded message EM into three byte strings of length 1, *hLen*, and *k–hLen*–1, respectively, obtaining:

$$EM := Y||maskedSeed||maskedDB.$$

3. Compute $seedMask := \text{MGF}(maskedDB, hLen)$.
4. Compute $seed := maskedSeed \oplus seedMask$.
5. Compute $dbMask := \text{MGF}(seed, k - hLen - 1)$.
6. Compute $DB := maskedDB \oplus dbMask$.
7. Separate DB into a byte string $lHash'$ of length *hLen*, a padding string PS consisting of ‘00’ bytes (which may be empty), and a message M, obtaining:

$$DB := lHash'||PS||01||M.$$

8. If there is no ‘01’ byte to separate PS from M, if $lHash$ is not equal to $lHash'$, or if Y is nonzero, then output ‘decryption error’ else output the message M.

8.3.7 RSAES-OAEP in Maple

Next we give a Maple implementation of RSAES-OAEP. A basic component of such an implementation is a hash function which, according to the recommendation in [154] should be either SHA-1 or SHA-256/384/512. Since SHA-1 is no longer regarded as secure, we choose SHA-256 for our implementation and hence we will use the functions defined in 5.6.3, as well as other conversion functions used by them, such as those in Appendix A.

We start with two data conversion primitives that, although not explicitly mentioned in the preceding description of the scheme, are used for conversion between two common message formats: byte strings (also called octet strings) and integers. The byte strings will be represented by even-length strings of hexadecimal digits, so that their byte length will be just one-half of their hexadecimal length (following earlier conventions we shall use lower-case letters for hexadecimal digits). We use the same notation as in [154] and the first of these functions is I2OSP ('Integer to Octet String Primitive'), which converts a non-negative integer x to a byte string of specified length *xLen*. This is the 'octet length' of the string we want to obtain (not the hexadecimal length) and hence it must be as a minimum equal to $\lfloor \log_{256} x \rfloor + 1$. Although this is not necessary for the implementation of RSA-OAEP that follows, the function uses this value as default for *xLen*, and the value is obtained through the function `intlog` that we have used in the implementation of plain RSA.

```
> I2OSP := proc(x::nonnegint, xLen::posint := intlog[256](x)+1)
  local l, s;
  if 256^xLen <= x then
    error "integer too large"
  else
    l := bytestohexstring(ListTools:-Reverse(convert(x, base, 256)));
    s := StringTools:-Repeat("00", xLen-iquo(StringTools:-Length(l), 2))
  end if;
  cat(s, l)
  end proc:
```

The next function `OS2IP` ('Octet String to Integer Primitive') inverts `I2OSP` , converting an even-length hex string (an 'octet string') to a non-negative integer:

```
> OS2IP := proc(X::string)
  convert(X, decimal, hex)
  end proc:
```

Next we give the 'mask generating function'. Following the recommendation in [154], we will use the function `MGF1` derived from the hash function (SHA-256 in our case) in the way explained above. The required inputs are, with the notation used in [154], a byte string *mgfSeed* and a positive integer *maskLen* that represents the intended length of the byte string *mask* it produces as output. In addition, there are two additional keyword parameters which are used to specify the underlying hash function and the byte length of its output. The default values are `SHA256` and 32 (since SHA-256 produces a 32-byte string as output). The output is a hex string of byte length *maskLen*.

```
> MGF1 := proc(mgfSeed, maskLen, {H := 'SHA256', hLen := 32})
  local T, i, C;
  if (2^32)*hLen < maskLen then
    error "mask too long"
  end if;
  T := "";
  for i from 0 to ceil(maskLen/hLen)-1 do
    C := I2OSP(i, 4);
    T := cat(T, H(cat(mgfSeed, C),hex))
  end do;
  StringTools:-Take(T, 2*maskLen)
  end proc:
```

Observe that `MGF1` can be seen as a family of functions parameterized by the value *maskLen* and also that *maskLen* can be either greater or smaller than the length of *mgfSeed*, so that `MGF1` is a family of functions which, on input a byte string of any length, produce as output a byte string of any desired length. Two of these instances will supply the functions G, H used in the description of RSA-OAEP above.

Example 8.8 A typical example of the use of `MGF1` below will be to produce a 223-byte string from a 32-byte seed. This corresponds to the case in which, as mentioned above, $dbMask := \text{MGF}(seed, k - hLen - 1)$ with an RSA modulus of $k = 256$ bytes and $hLen = 32$ (the output length of SHA-256), so that $223 = 256 - 32 - 1$:

```
> with(StringTools);
  Randomize();
  r := LowerCase(Random(64, xdigit));
     "b0e6fca1ba6aecf51f88fbe04a22f65b8db9d7bb2fa84f0b76ae0eec074bf6ff"

> MGF1(r, 223);
"99a91043da0ddabbc27f8d730b30e256a4e5bcc195261792d54d41f5301e88197ed7598274df4ad12\
  87cc7b5b52710e9124b359a48bd65779fd34f2ad03a3b26445d3b33f61d3e4d25c1ed9aa52fa9b93\
  31654bc35b605e75de97cacedc5c9a2f71c99cd4434b67cd018f7febe4e91ef2c1d92b3570d2c3fc\
  8ac763d9b8b0f07f2f5339524a455261b57c566da22f9c4e03a86e4e4660e9e0647d46c514c7dcce\
  69ed56465a0c92a3339259207118ac9a620f79f78199cba93dd29a87288d18c01967952707eda541\
  feb2140e78be82b9fe3ad28e0f853843a4dab6f1e6b9a"
```

The next function performs the EME-OAEP encoding operation following [154], as described in our preceding discussion. The required inputs of the function are `M`, which corresponds to the message given as an even-length hexadecimal string, `k` for the byte length of the encoded message that will be produced by the function and `seed`, a byte string of length *hLen* which is used to specify a randomly generated seed. The optional input parameters are `L`, in which a 'label' (a byte string) can be specified (with the empty string as default), `H` for the name of the hash function to be used (with `SHA256` as default) and *hLen* for the byte length of the hash function output (with 32 as default). The output is a byte string `EM` of length k.

```
> EMEOAEPEnc := proc(M::string, k::posint, seed::string,
                     L::string := "", H::name := 'SHA256', hLen := 32)
  local mLen, lHash, hhLen, PS, DB, s, dbMask, maskedDB, seedMask, maskedSeed;
  mLen := iquo(StringTools:-Length(M), 2);
  lHash := H(L, hex);
  hhLen := 2*hLen;
  PS := StringTools:-Repeat("00", k-mLen-hhLen-2);
  DB := cat(lHash, PS, "01", M);
  if _params['seed'] = NULL then
    StringTools:-Randomize();
    s := StringTools:-LowerCase(StringTools:-Random(hhLen, xdigit))
  else
```

```
  if not StringTools:-IsHexDigit(seed) or StringTools:-Length(seed) <> hhLen then
    error "the seed must be a 32-byte hex string"
  end if;
  s := seed
end if;
dbMask := MGF1(s, k-hLen-1);
maskedDB :=
  bytestohexstring(BitXor~(hexstringtobytes(DB), hexstringtobytes(dbMask)));
seedMask := MGF1(maskedDB, hLen);
maskedSeed :=
  bytestohexstring(BitXor~(hexstringtobytes(s), hexstringtobytes(seedMask)));
cat("00", maskedSeed, maskedDB)
end proc:
```

Remark 8.3 Before proceeding any further, it is convenient to point out that `EMEOAEPEnc` also works if no value is passed to the parameter `seed`. In this case, the seed will be pseudo-randomly generated by the function itself. As on other similar occasions, this possibility is included for demonstration purposes only and we stress that the security reduction for RSA-OAEP does not apply to this variant, see Exercise 8.19. In this case the seed—obtained by applying a PRG to a shorter seed taken from the system clock—is far from random because on the one hand, the PRG is deterministic and, on the other hand, the system clock seed is not random either. However, if an externally generated random seed is supplied to the function, then the security reduction given in [80] should apply.

Exercise 8.18 Write a variant of the function `EMEOAEPEnc` that uses the previous function `BBSByteGen` to generate the seed from an initial seed provided by the system. Explain the reasons why, despite the fact that `BBSByteGen` uses Blum–Blum–Shub, this method is highly insecure. Discuss the security of the variant obtained by generating the seed by means of the Blum–Blum–Shub PRG, itself seeded with a truly random 256-bit seed.

Next we give the EME-OAEP decoding function, i.e., the inverse function of the previous one. The input parameters are similar to those in `EMEOAEPEnc`, with the encoded message `EM` replacing the message and without the parameter `seed`, since the seed is not necessary for decoding. The output is the decoded message given as a byte string.

```
> EMEOAEPDec := proc(EM::string, k::posint,
                     {L::string := "", H::name := 'SHA256', hLen := 32})
  local lHash, Y, hhLen, maskedSeed, maskedDB, seedMask, seed, dbMask, DB, lHash2, dr;
  lHash := H(L, hex);
  Y := StringTools:-Take(EM, 2);
  hhLen := 2*hLen;
  maskedSeed := StringTools:-SubString(EM, 3 .. hhLen+2);
  maskedDB := StringTools:-SubString(EM, hhLen+3 .. -1);
  seedMask := MGF1(maskedDB, hLen);
  seed := bytestohexstring(BitXor~(hexstringtobytes(maskedSeed),
                                   hexstringtobytes(seedMask)));
  dbMask := MGF1(seed, k-hLen-1);
  DB :=
    bytestohexstring(BitXor~(hexstringtobytes(maskedDB), hexstringtobytes(dbMask)));
  lHash2 := StringTools:-Take(DB, hhLen);
  dr := StringTools:-Drop(DB, hhLen);
```

```
  while StringTools:-IsPrefix("00", dr) do
    dr := StringTools:-Drop(dr, 2)
  end do;
  if lHash2 <> lHash or Y <> "00" or StringTools:-Take(dr, 2) <> "01" then
    error "decryption error"
  else
    StringTools:-Drop(dr, 2)
  end if
  end proc:
```

Remark 8.4 Observe that, following a recommendation in [154], motivated by a chosen ciphertext attack of Manger [133] against the version included in PKCS#1 v2.0, in case of decoding error—which may be due to various causes—the function only returns the message "decryption error" at the end of the computation (this function is called by the RSAES-OAEP decryption function given below, hence the term "decryption" in the message). This is to prevent an attacker from distinguishing rejections at the various steps using, for example, timing analysis.

Example 8.9 Suppose that we take $k = 256$, which is the value that corresponds to a 2048-bit RSA modulus. Then the maximum message length in bytes is, according to [154], equal to $k - 2hLen - 2 = 256 - 64 - 2 = 190$, assuming that a hash function with an output length of 256 bits (32 bytes) such as SHA-256 is being used (we have not enforced this limitation in `EMEOAEPEnc` because it will be enforced in the higher level RSAES-OAEP encryption function). First we generate a message of this length, which corresponds to a message of exactly twice this length in hexadecimal, i.e., 380:

```
> with(StringTools);
  Randomize();
  M := Lowercase(Random(380, xdigit));

"8c8e4b15af772c1af9b0e8f2afd6c71bd5ff62aeaa06f3dc90afb05ec41db47c8889e7f18eae6b6e1\
  dff16060cab1fbeeff29ffc6db7f83ebfeba209292a9cffcccce5474d8c3d2a0534ddbcdaa7b2b81\
  e4cadc2ab4562dbafd8de23ebfafd4adcc1ec7d1ccc7f8e0d9cdb5ed866a0bd2a03d2ca25ed010be\
  19f9fe6ffd16b9b3f54056d2cec20aad3af6b735a9afcc250320cbfbbdf745faf8a2e79c1f2ad931\
  bcab215bd9fdcc9f08eec8a7712bbf8a76ebaafbecc6febb9c89d6986ce"
```

Next, we encode this message using $k = 256$, the seed r we generated in Example 8.8, and default values for the remaining parameters:

```
> EM := EMEOAEPEnc(M, 256, r);

"00cab3be8d020c088b1142f325585e04ae649d3d36e617b46c90a7683f959a93697a19d40142f1c6a\
  f588479bb925f5b72834bfd25f1bd84de71d8d8ee484c304c7f5bd7c961703dfd3285775d4788c62\
  e099ecaf8e61763844343e09a8efe269238d5b2d4079390264bdc128ca32302a68df9a623c9dbb21\
  f6356970ee4ece33e08d0552803793a41fa1dc3230294365d9403de1e95a6695d1303aee3b860f5f\
  ab829f27959b89959955a59bd84fa9f645d1085362e43e39f0da64bf3b7b3aca77da18161088c250\
  a99ea8af9744b10356470c593c7a243cecc72a3860b492321928db3e065c34588d61bafac6dfcfa9\
  06744c3924f469febd1f463f277ed54"
```

Next, we decode the encoded message `EM` to recover `M`; to save space we do not print `M` again but we check that `M` is recovered:

```
> evalb(EMEOAEPDec(EM, 256) = M);
                                    true
```

We are now ready to give the RSAES-OAEP encryption and decryption functions. The encryption function `RSAESOAEPEncrypt` is given next. In addition to

the input parameters already present in EMEOAEPEnc, it has a required parameter publickey where a public RSA key (in the format generated by the previous function RSAKeyGen, either decimal or hex) is specified and also an additional optional parameter messagetype with possible values text, hex, which specifies whether the message to be encrypted is to be regarded as an even-length hex string (a byte string) or as an ordinary text string (with default value hex). The output is the ciphertext, a byte string.

```
> RSAESOAEPEncrypt := proc(publickey::list, message::string, seed::string,
                           {messagetype::identical(hex, text) := hex,
                           L::string := "", H::name := SHA256, hLen::posint := 32})
  local pk, n, e, k, M, EM, m, c, C;
  if type(publickey[1], string) then
    pk := convert~(publickey, decimal, hex)
  else
    pk := publickey
  end if;
  n := pk[1];
  e := pk[2];
  k := intlog[256](n) + 1;
  M := messagetohex(message, messagetype);
  if 2*k - 4*hLen + 4 < StringTools:-Length(M) then
    error "message too long"
  end if;
  EM := EMEOAEPEnc(M, k, _params['seed'], ':-L' = L, ':-H' = H, ':-hLen' = hLen);
  m := OS2IP(EM);
  c := Power(m,e) mod n;
  C := I2OSP(c, k)
  end proc:
```

Next we give the RSAES-OAEP decryption function. The input parameters are similar to those in RSAESOAEPEncrypt, omitting the seed and replacing publickey by privatekey (for an RSA private key in the format output by RSAKeyGen) and message by C (for the ciphertext as a hexadecimal byte string). In this case, messagetype serves to indicate the format of the plaintext string.

```
> RSAESOAEPDecrypt:=proc(privatekey::list, C::string,
                         {messagetype::identical(hex,text) := hex,
                         L::string := "", H::name := 'SHA256', hLen::posint := 32})
  local sk, k, c, m, EM;
  if type(privatekey[1], string) then
    sk := `~`[convert](privatekey, decimal, hex)
  else
    sk := privatekey
  end if;
  k := intlog[256](sk[1])+1;
  if StringTools:-Length(C) <> 2*k or k < 2*hLen+2 then
    error "decryption error"
  end if;
  c := OS2IP(C);
  m := chrem([Power~(c,sk[5]) mod sk[3], Power~(c,sk[6]) mod sk[4]],
  [sk[3], sk[4]]);
  EM := I2OSP(m, k);
  hextomessage(EMEOAEPDec(EM, k, ':-L' = L, ':-H' = H, ':-hLen' = hLen), messagetype)
  end proc:
```

Remark 8.5 In this implementation of RSA-OAEP there are some minor departures from [154] that do not affect the general behavior of the scheme. For example, private keys have a slightly different format here. Also, in [154] a size limitation on the optional label L is specified. Here this restriction is not explicitly enforced because

the maximum size of Maple's lists is already smaller than the limit established in the standard.

Example 8.10 Let us now give an example of encryption/decryption with RSAES-OAEP, using the public key `pk` and the private key `sk` generated in Example 8.2. We generate this RSA key again:

```
> rsakey := RSAKeyGen(1024, 6180204295522709153838923244723480121 75,
                      4102756714646987802485435721275578031 86):
  pk := rsakey[1]:
  sk := rsakey[2]:
```

We are going to encrypt the following message:

```
> me := "The magic words are squeamish ossifrage":
```

For encryption, a randomly chosen 32-byte seed is needed (since we are using as hash function SHA-256, with a 32-byte output). We will use the following seed:

```
> s := "24deaed05ba18bc679443f35dc58dcbd63b43356b0a3d0431be35b0b82c8f1c9"
```

The ciphertext `C` is then obtained as follows:

```
> C := RSAESOAEPEncrypt(pk, me, s, messagetype = 'text');
"82af56318cab65002cbc7023ee6bccbb7ae071d8bda9c06e35b8f109f4d39e4e7079a502a7eb98bfd\
  499791a9cf0dd5772754547cf5932e9cff8eef23b81c4f58f1d2928e13dff04498ab1fa19722cac7\
  1c41970960b679c39b44c4d3384196cf5bfed8441a65e2b34d0ac3c831c712e0c02915d5b16e1077\
  7eaa0e67997638dff02c1b4b1747777e344fc7a23cbe3d4b830d6e52a067a3f4e79ef04ba066d09f\
  126b99b6c1fed19bb3381ea08f9dfd51d1edc768eea34f1dc22c32a0a4d7359a6beb7919bdeca70c\
  03f6b8e43525a2dff6752e70c39a5836d72f2664fd5143306e50e79af68c53721f27d9d4a59fb226\
  48583eeb0daaf2ea2fb480d9ebbb944"
```

The message can now be recovered as follows:

```
> RSAESOAEPDecrypt(sk, C, messagetype = 'text');
                  "The magic words are squeamish ossifrage"
```

Exercise 8.19 Suppose that a small message space such as, for example, {"Yes", "No"} is used and that messages are encrypted using the variant of `RSAESOAEPEncrypt` in which the seed is obtained by applying a PRG to a 32-bit random seed. Show that there is a CPA attack that allows the full recovery of the plaintext and hence renders this scheme completely insecure. Explain the reasons why this attack no longer works against `RSAESOAEPEncrypt` (with a random seed).

Exercise 8.20

(i) Write a Maple version of the RSAES-OAEP encryption and decryption functions that is able to encrypt/decrypt messages of arbitrary length (partitioning the message into 'blocks' of adequate size like in the implementation of plain RSA).
(ii) Write a Maple version of RSAES-OAEP encryption/decryption that is able to encrypt/decrypt messages given as files.

8.4 Rabin Encryption

We have seen that several variants of RSA enjoy security properties under the RSA assumption, i.e., under the hypothesis that the RSA problem is hard. However, the RSA problem is not as well understood as the integer factorization problem, which is presumed hard after having been studied for a longer time in different contexts not necessarily related to cryptography. Moreover, while the hardness of the factorization problem is a necessary condition for the hardness of the RSA problem, the converse is still unproven, so basing the security of RSA encryption schemes on the presumed hardness of factoring relies on two conjectures: that factoring is hard and that the RSA problem is as hard as factoring. Because of this, an interesting question is whether the second of these hypothesis may be withdrawn, i.e., whether we can obtain a public-key encryption scheme whose security relies only on the hardness of factoring. The answer is affirmative and such a scheme was first described by M. Rabin in 1979. The Rabin encryption scheme is based on the family of Rabin functions R_n, with $R_n(x) = x^2 \bmod n$ for $x \in \mathbb{Z}_n^*$ and $n = pq$ a product of two distinct primes or, in a variant due to Williams, on the family of Blum–Williams permutations that were shown to be one-way (assuming that factoring is hard) in Theorem 3.8.

8.4.1 Plain Rabin Encryption

Recall that, if n is a Blum integer, i.e., $n = pq$ where p and q are distinct odd primes which are $\equiv 3 \pmod 4$ (Blum primes), then the Blum–Williams function:

$$\begin{array}{ccc} \mathcal{QR}_n & \xrightarrow{BW_n} & \mathcal{QR}_n \\ x & \longmapsto & x^2 \bmod n \end{array}$$

is a permutation, and the family $\{BW_n\}$ is a family of one-way permutations under the factoring assumption (the PPT algorithm that outputs the modulus n is similar to the RSA key generation algorithm with the restriction that the prime factors of n are Blum primes). We can see that this family is, furthermore, trapdoor one-way if we set as trapdoor for n, $t(n) = (p, q)$. Indeed, as we have seen in Algorithm 2.8 and the subsequent discussion, computing square roots modulo a prime is easy and, as a consequence, computing modular square roots modulo $n = pq$, where p and q are primes, is also easy when the factorization of n is known. We next give an explicit description of the algorithm that computes the inverse function $BW_n^{-1} : \mathcal{QR}_n \to \mathcal{QR}_n$ when $n = pq$ is a Blum integer. This algorithm consists of first applying Algorithm 2.8 to compute the square roots of $c \in \mathcal{QR}_n$ modulo the primes p and q, which is done with just a modular exponentiation in each case (by Proposition 2.15, the square roots of c modulo p are $\pm m_p$, where $m_p = c^{(p+1)/4} \bmod p$), and then using the Chinese remainder theorem to recover the four square roots of c modulo n. Once these square roots have been found, we know from Proposition 2.14 that the

image of c under BW_n^{-1} is the unique one of these four roots that is a quadratic residue modulo both p and q, and it can be efficiently found by computing the Legendre symbols by means of Algorithm 2.7.

Thus the algorithm that computes the four square roots in $\mathbb{Z}_n^*$ of $c \in \mathcal{QR}_n$ is:

Algorithm 8.2. Computation of the four square roots of a quadratic residue modulo a Blum integer.

Input: (p, q) (where p and q are distinct Blum primes with $n = pq$) and $c \in \mathcal{QR}_n$.
Output: The four square roots of c in $\mathbb{Z}_n^*$.

1. `Compute` $n := pq$, $m_p := c^{(p+1)/4} \bmod p$ and $m_q := c^{(p+1)/4} \bmod q$.
2. `Compute`, using the extended Euclidean algorithm, y_p and y_q such that $1 = y_p p + y_q q$.
3. `Compute` $r := y_p p m_q + y_q q m_p$ and $s := y_p p m_q - y_q q m_p$.
4. `return` $\{r \bmod n, s \bmod n, -r \bmod n, -s \bmod n\}$.

Let us see that this algorithm is correct and outputs the set of the four square roots of c in $\mathcal{QR}_n$. As we have remarked, $\pm m_p$ are the two square roots of c in $\mathbb{Z}_p^*$ and, similarly, $\pm m_q$ are the two square roots of c in $\mathbb{Z}_q^*$. Thus the square roots of c in $\mathbb{Z}_n^*$ are the solutions of the four systems of congruences:

$$\left.\begin{aligned} x &\equiv \pm m_p \pmod{p} \\ x &\equiv \pm m_q \pmod{q} \end{aligned}\right\}$$

which can be found by applying the Chinese remainder theorem. Looking at the proof of Theorem 2.12 we see that the four roots are $\pm r \bmod n$ and $\pm s \bmod n$, where $r := y_p p m_q + y_q q m_p$ and $s := y_p p m_q - y_q q m_p$.

The algorithm that computes the inverse of the Blum–Williams function is the same as the preceding one, except that it selects the unique square root of c that is a quadratic residue modulo n and returns it as output:

Algorithm 8.3. Computation of the inverse of the Blum–Williams function.

Input: (p, q) (where p and q are distinct Blum primes with $n = pq$) and $c \in \mathcal{QR}_n$.
Output: $m \in \mathcal{QR}_n$ such that $m^2 \bmod n = c$.

1. `Compute` $n := pq$, $m_p := c^{(p+1)/4} \bmod p$ and $m_q := c^{(p+1)/4} \bmod q$.
2. `Compute`, using the extended Euclidean algorithm, y_p and y_q such that $1 = y_p p + y_q q$.
3. `return` $m := (y_p p m_q + y_q q m_p) \bmod n$.

The algorithm proceeds as Algorithm 8.2 until the inverse y_p of p modulo q and the inverse y_q of q modulo p are found. This gives the four square roots of c modulo n and the one that belongs to $\mathcal{QR}_n$ can be found, for example, by computing the Legendre symbols modulo p and q and checking that both are equal to $+1$. However, this computation is not really necessary for we can directly see that the root $r := (y_p p m_q +$

$y_q q m_p) \bmod n$ belongs to $\mathcal{QR}_n$ and hence we set $m := r$. Indeed, since $c \in \mathcal{QR}_n$ we have that $c \bmod p \in \mathcal{QR}_p$ and hence $m_p = c^{(p+1)/4} \bmod p \in \mathcal{QR}_p$. Similarly, m_q is a quadratic residue modulo q, i.e., $m_q \in \mathcal{QR}_q$. Now, r obviously satisfies:

$$\left.\begin{aligned} r &\equiv m_p \pmod{p} \\ r &\equiv m_q \pmod{q} \end{aligned}\right\}$$

and hence $r \in \mathcal{QR}_n$ by Proposition 2.14.

This leads to the following encryption scheme:

Definition 8.14 The *plain Rabin Encryption Scheme* is the public-key encryption scheme (**Gen**, **Enc**, **Dec**) defined by the following algorithms:

- **Gen**: On input 1^k run a PPT algorithm that generates two primes, p, q such that $\text{len}(p) = \text{len}(q) = k$ and $p \equiv q \equiv 3 \pmod 4$, compute $n = pq$ and output the public key n and the private key (p, q).
- **Enc**: On input a public key $pk = n$ and a message $m \in \mathcal{QR}_n$, compute the ciphertext:

$$c := \mathbf{Enc}(pk, m) = BW_n(m) = m^2 \bmod n \in \mathcal{QR}_n.$$

- **Dec**: On input a private key $sk = (p, q)$ and a ciphertext $c \in \mathcal{QR}_n$, compute the message:

$$m := \mathbf{Dec}(sk, c) = BW_n^{-1}(c) \in \mathcal{QR}_n.$$

Remarks 8.10 There are other variants of plain Rabin encryption. For example, the use of Blum primes is not strictly necessary and it merely simplifies things by making the squaring function a permutation of $\mathcal{QR}_n$ and also makes decryption more efficient because the computation of square roots modulo a prime p is simpler in case p is a Blum prime. However, these roots can be computed in probabilistic polynomial time even if $p \equiv 1 \pmod 4$, as we have seen in Algorithm 2.8 (when $p \equiv 1 \pmod 8$, no deterministic polynomial-time algorithm is known for computing these roots).

Another possibility consists of taking $\mathbb{Z}_n$ as the plaintext space and encrypting $m \in \mathbb{Z}_n$ as $c = m^2 \bmod n$ (as with RSA encryption, we may safely assume that $c \in \mathbb{Z}_n^*$, and hence $c \in \mathcal{QR}_n$ because otherwise if $c \neq 0$, then $\gcd(c, n)$ reveals the private key and, on the other hand, this occurs with negligible probability if m is randomly chosen in $\mathbb{Z}_n$). This has the inconvenience that, by Corollary 2.7, c has four square roots in $\mathbb{Z}_n^*$ and only one of them is equal to the plaintext m. As already mentioned, it is not a sound method to rely on semantic concepts such as 'meaningful' to determine m—after all, m might be a symmetric key which should necessarily look random. The problem can be remedied by adding some redundancy to m before encryption—such as a given bit string of sufficient length appended to the message—so that the plaintext space actually becomes a sparse subset of $\mathbb{Z}_n^*$ and the probability that more than one square root of c belongs to this subset becomes negligible.

Another approach to solve this difficulty would be to attach to the message additional information that allows it to be recovered uniquely. For example, let c be a ciphertext and $m_p := c^{(p+1)/4} \bmod p$, $m_q := c^{(p+1)/4} \bmod q$, and $\{\pm r \bmod n, \pm s \bmod n\}$ the four square roots of c with the notations used in Algorithm 8.2. Then, since m_p and m_q are quadratic residues modulo p and q, respectively, and $r \equiv m_p \pmod p$, $r \equiv m_q \pmod q$, it follows that (as we have already remarked), r is a quadratic residue modulo both p and q. Thus the Jacobi symbol of r is $\left(\frac{r}{n}\right) = 1$ and, since by Euler's criterion $\left(\frac{-1}{p}\right) = -1$ whenever p is a Blum prime, we also have that $\left(\frac{-r}{p}\right) = \left(\frac{-r}{q}\right) = -1$ and hence $\left(\frac{-r}{n}\right) = 1$.

Also, one sees that $s \equiv -m_p \pmod p$ and $s \equiv m_q \pmod q$ so that s is a quadratic non-residue modulo p and a quadratic residue modulo q, hence $\left(\frac{s}{n}\right) = -1$ and, similarly, $\left(\frac{-s}{n}\right) = -1$. Thus if we add as information the Jacobi symbol of the message, it serves to discard one of the pairs $\pm r$, $\pm s$ and, assuming for example that the discarded pair is $\pm r$, it tells us that m is equal to s or $-s$. One of these integers is $< n/2$ and the other $> n/2$ so, if we add this bit of information to the Jacobi symbol, the plaintext is uniquely determined. All this information can be easily gathered at encryption time because Algorithm 2.7 can be used to efficiently compute the Jacobi symbol of m without knowing the prime factorization of n. In contrast with this, deciding whether a given $m \in \mathbb{Z}_n^*$ is a quadratic residue is an instance of the *quadratic residuosity*[4] *problem*, which is thought to be hard in general, although it is no harder than factoring (and it might be easier) because it becomes easy once the factorization of $n = pq$ is known, as a consequence of Proposition 2.14. Of course, a necessary condition for m to be a quadratic residue modulo n is that $\left(\frac{m}{n}\right) = 1$.

The preceding discussion also suggests a concrete method to make the plaintext uniquely recoverable by restricting the message space. We may format the message prior to encryption—using appropriate padding, etc.—so that viewed as an element of $\mathbb{Z}_n^*$, m is $< n/2$ and has Jacobi symbol equal to 1. By what we have just seen, only one of the square roots of the ciphertext will satisfy these conditions.

8.4.1.1 The Security of Plain Rabin Encryption

Let us now examine the security of plain Rabin encryption. We start by noticing that it satisfies the weakest security notion:

Proposition 8.6 *Under the factoring assumption, plain Rabin encryption is OW-CPA secure.*

Proof This follows from Theorem 3.8, according to which $\{BW_n\}$ is a family of one-way trapdoor permutations under the factoring assumption. More explicitly, let n be a product of two randomly chosen k-bit Blum primes and suppose that there exists a PPT algorithm that inverts BW_n. Then a PPT algorithm that factors n proceeds by

[4] An alternative term, often preferred by number theorists, is *residuacity*.

choosing random elements $x \leftarrow \mathbb{Z}_n^*$, computing $x^2 \bmod n \in \mathcal{QR}_n$, and calling the PPT algorithm that inverts BW_n to obtain $y \in \mathcal{QR}_n$ such that $BW_n(y) = x^2 \bmod n$. This procedure is repeated until finding x, y such that $x \not\equiv \pm y \pmod{n}$ (or pick, in polynomial time, a random element $x \leftarrow \mathbb{Z}_n^*$ such that $\left(\frac{x}{n}\right) = -1$ and compute $y \in \mathcal{QR}_n$ such that $BW_n(y) = x^2 \bmod n$, which, since $\left(\frac{x}{n}\right) = -1$, $\left(\frac{y}{n}\right) = 1$, and $\left(\frac{-1}{n}\right) = 1$, satisfies $x \not\equiv \pm y \pmod{n}$). Then $\gcd(x - y, n)$ is a nontrivial factor of n. □

Remarks 8.11 The same factoring algorithm used to prove the preceding proposition serves to prove that the Rabin functions are one-way under the factoring assumption and hence that, under this hypothesis, the variant of Rabin encryption with plaintexts in $\mathbb{Z}_n^*$ is OW-CPA secure. If we compare it with plain RSA, we notice that plain Rabin encryption has the advantage that its OW-CPA security depends on the hardness of factoring (to which it is, in fact, equivalent) instead of the hardness of the RSA problem. However, as far as security is concerned, we cannot go much further with this simple encryption scheme because, as happens with plain RSA, encryption is deterministic and hence the scheme is not IND-CPA secure. On the other hand, if we consider a variant in which redundancy is added to the messages in $\mathbb{Z}_n^*$ in order to make the probability that two distinct plaintexts have the same square in $\mathbb{Z}_n^*$—and hence produce the same ciphertext—negligible, then the hardness of factoring no longer implies that recovering the plaintext is hard, because if we take, as before, a random $x \in \mathbb{Z}_n^*$, the probability that the algorithm breaking the OW-CPA security finds an element $y \in \mathbb{Z}_n^*$ (which would have to be a valid plaintext) such that $x^2 \equiv y^2 \pmod{n}$ but $x \not\equiv \pm y \pmod{n}$ would be negligible.

Since plain Rabin encryption is not CPA secure, it cannot be CCA secure either but, actually, the situation is even worse in this respect in comparison with plain RSA. Indeed, we have seen that plain RSA is vulnerable to a CCA attack that can recover the entire plaintext but no CCA attack against RSA is known that is capable of recovering the private key (in the absence of specific weaknesses such as a small decryption exponent, etc.). In contrast, the very feature that makes inverting the Blum–Williams function as hard as factoring, gives rise to a CCA attack against plain Rabin encryption that is able to recover the private key and hence all the messages subsequently encrypted with this key. This attack proceeds as the factoring algorithm outlined in the proof of Proposition 8.6 except that, instead of calling a PPT algorithm that inverts the Blum–Williams function (which now is not assumed to exist), it queries the decryption oracle to obtain the square root $y \in \mathcal{QR}_n$ of $x^2 \bmod n$, for $x \in \mathbb{Z}_n^*$ such that $\left(\frac{x}{n}\right) = -1$, and computes $\gcd(x - y, n)$ which gives a nontrivial (prime) factor of n. Similarly, if the Rabin function on $\mathbb{Z}_n^*$ is being used for encryption, a square root of $x^2 \bmod n \in \mathbb{Z}_n^*$ which is distinct from $\pm x \bmod n$ will be found with probability at least $1 - \frac{1}{2^t}$ after t trials giving, as before, a nontrivial factor of n. Of course, this attack does not work if the message space is a sparse subset of $\mathbb{Z}_n^*$ for the same reason that we cannot prove the equivalence between breaking the scheme and factoring the modulus in that case.

Exercise 8.21 Prove that the algorithm that factors the modulus of a Rabin key given oracle access to the decryption algorithm has, for a randomly chosen $x \in \mathbb{Z}_n^*$, a probability of success equal to $1/2$.

We will see that there are variants of Rabin encryption that are resistant to CCA attacks but, as it has often been remarked, the fact that plain Rabin is so completely vulnerable to CCA attacks is probably the reason why even the more secure variants of this scheme are seldom used in practice despite the fact that there is no reason to assume that they are less secure than, say, the corresponding RSA variants and, on the other hand, they may have other advantages such as a faster encryption algorithm.

8.4.2 Plain Rabin Encryption in Maple

We are going to implement in Maple the Rabin functions R_n and their inverses, for a Blum integer n. These functions are not permutations, so we cannot speak of the inverse function in the strict sense but we will see how to compute the four preimages of each quadratic residue, i.e., its four square roots modulo n, using Algorithm 8.2, and we will loosely speak of the 'inverse Rabin functions' all the same. As a particular case we will also have an implementation of BW_n and its inverse, which can be used to implement plain Rabin encryption as described above. We will leave as an exercise for the reader the implementation of this scheme, including the key generation algorithm (which can be derived from the one used to generate RSA keys) and other minor details. Later, we will use the Rabin functions as cryptographic primitives to implement a more secure version of Rabin encryption.

The function R_n is just squaring modulo n and BW_n is its restriction to the subgroup $\mathcal{QR}_n$ of $\mathbb{Z}_n^*$. As we have remarked, determining whether an $x \in \mathbb{Z}_n^*$ is a quadratic residue modulo n may be hard, but quadratic residues are easily obtained in a natural way by just squaring elements of $\mathbb{Z}_n^*$.

The Rabin functions are given by the following Maple function in which we assume that n is the product of two distinct Blum primes:

```
> Rabin := proc(m::posint, n::posint)
  m^2 mod n
  end proc:
```

The 'inverse Rabin functions'—and the inverse Blum–Williams permutations—are computed by the next Maple procedure. The input parameters are `c`, for a quadratic residue modulo n whose square roots are computed by the function, `p` and `q`, which serve to specify the primes whose product is the modulus n and, finally, `pinv` and `qinv`, where the inverse of p modulo q and the inverse of q modulo p, respectively, can be specified. These two last parameters are used to speed up the computation but they are not really necessary and, if not supplied, they will be computed inside the function by means of the extended Euclidean algorithm. In addition, there is an optional keyword parameter `BW` that serves to specify whether the function to invert is the Blum–Williams permutation. If `BW = true` then the unique square root of c which is a quadratic residue modulo n is returned, so that the function acts

as the inverse of the Blum–Williams permutation in this case. If, on the contrary, `BW = false`, which is the default value, then the four preimages of c by the Rabin function are computed and returned as a list (whose first element is the only one that is a quadratic residue). The function produces a different output (when `BW = false`) in case c is found to be a quadratic non-residue modulo n, for reasons that are explained below.

```
> InvRabin := proc(c, p, q, pinv, qinv, {BW::truefalse := false})
  local n, mp, mq, reject, yp, yq, r, s;
  reject := false;
  n := p*q;
  mp := Power(c, iquo(p+1, 4)) mod p;
  mq := Power(c, iquo(q+1, 4)) mod q;
  if mp^2 mod p <> c mod p or mq^2 mod q <> c mod q then
    reject := true
  end if;
  if BW and reject then
    error "%1 is not a quadratic residue", c
  end if;
  if _params['pinv'] = true and _params['qinv'] = true then
    yp := pinv;
    yq := qinv
  else
    igcdex(p, q, 'yp', 'yq')
  end if;
  r := (yp*p*mq+yq*q*mp) mod n;
  s := (yp*p*mq-yq*q*mp) mod n;
  if BW then
    r
  elif reject then
    [r, n-r, s, n-s, 0]
  else
    [r, n-r, s, n-s]
  end if;
  end proc:
```

An important feature of this function is the way it handles c whenever c is a quadratic non-residue modulo n. Whether this is the case can be checked by computing the Legendre symbols of c with respect to p and q and applying Proposition 2.14. However, since in the implementation of the function we have to try to compute the square roots of c modulo p and modulo q anyway (which are called `mp` and `mq` in the code of the function), it is simpler and more efficient to square `mp` modulo p and `mq` modulo q and check that these values are equal to $c \bmod p$ and $c \bmod q$ respectively. The subtle point here is that it would be a mistake to reject c with an error message at this point if it is discovered not to be a quadratic residue. Indeed, we will later use this function to implement a more secure version of Rabin encryption and, as we will see, these messages could help an adversary in a CCA attack. Because of this, the function `InvRabin` completes the computation—i.e., it tries to compute the square roots of c—even in this case and returns an augmented list of 5 terms where the last one is equal to 0. The values in the list are meaningless in this case but this makes it possible to produce the error message at the end of the computation, which will take time similar to the case when there is no error. The idea is to prevent an attacker from distinguishing rejections at the different steps of a decryption using, for example, timing analysis.

Exercise 8.22 Write a Maple function that generates public and private keys for Rabin encryption. On input a security parameter k and two random seeds, it should pseudo-randomly generate two k-bit Blum primes whose product is a $2k$-bit modulus. Make the function able to output Rabin keys in either decimal or hexadecimal format (as byte strings in the latter case).

Exercise 8.23 Modify the functions `Rabin` and `InvRabin` so that they accept as argument the public and private Rabin keys, respectively, generated by the function defined in the previous exercise, either in decimal or in hexadecimal format.

Exercise 8.24 Modify the function `InvRabin` to make it check that both p and q are Blum primes.

Example 8.11 Let us give an example of how the functions `Rabin` and `InvRabin` work. We will consider the RSA key `rsakey` generated in Example 8.10, whose 2048-bit modulus is a product of two 1024-bit Blum primes p, q. We set:

```
> drsakey := convert~~(rsakey, decimal, hex);
  rkey := [drsakey[1][1], op(drsakey[2][3 .. 4]), op(drsakey[2][7 .. 8])]:
```

This way, `rkey` is a Rabin key of the form $[n, p, q, pinv, qinv]$, where $n = pq$ and p, q are Blum primes. Let us now check the behavior of `Rabin` and `InvRabin`. Choose, for example, $m := 10^{600} \in \mathbb{Z}_n^*$ and apply `Rabin` followed by `InvRabin`:

```
> InvRabin(Rabin(10^600, rkey[1]), op(rkey[2 .. 5]));
[10000000000000000000000000000000[...540 digits...]00000000000000000000000000000000,
  269742868015351028621217638137725771855871558101556548785668605968737098677 3...,
  996108980604685847911566281350121193995648538186291411089270675101278649182 9...,
  170131969954882453830061010002713652456306704282927407676741538458609233759 0...]
```

We have recovered the four square roots of $10^{1200} \bmod n$ which is the image of 10^{600} under the map `Rabin(-,n)`. Note that the first root is precisely 10^{600}. Now, observe that 10^{600} is obviously a quadratic residue in $\mathbb{Z}_n^*$ and so we may do the analogous computation using the Blum–Williams function BW_n:

```
> InvRabin(Rabin(10^600, rkey[1]), op(rkey[2 .. 5]), BW = true);
10000000000000000000000000000000[...540 digits...]00000000000000000000000000000000
```

In this case, we used the option `BW = true` and so the result of successively applying the Blum–Williams permutation and its inverse to 10^{600} is, again, this same number.

Example 8.12 We use the key `rkey` in the previous example to illustrate how the modulus can be factored—and hence the scheme can be broken—given oracle access to the decryption function. The decryption function is given here by the inverse Blum–Williams function implemented above and, for the purpose of this example, we may assume that the function is simulating the decryption oracle and acting as a black box, so that we do not need to know the prime factors of the modulus and only need to know the result of applying the function to the elements we submit. For brevity, let us call this modulus n, that is:

```
> n := rkey[1]:
```

To factor the modulus we follow the method sketched just after Proposition 8.6. Instead of taking a random element of $\mathbb{Z}_n^*$, we will start by taking a small element

of $\mathbb{Z}_n^*$ (for notational convenience, in order to save space) and, if necessary, we will continue with the elements of $\mathbb{Z}_n^*$ following this one in the subsequent trials; it is clear that this variant of the algorithm will also work and will eventually factor n. So we start with $x = 13$ and we compute:

```
> x := 13;
  y := InvRabin(Rabin(x, n), op(rkey[2 .. 5]), BW = true);
                                    13
```

We see that $y = 13 = x$ in this case, so $\gcd(x - y, n) = n$ and we do not obtain a proper factor. Obviously, this was due to the fact that $x = 13$ is a quadratic residue modulo n; in that case we have that $BW_n^{-1}(R_n(x)) = x$. Let us now try with the next value, $x = 14$:

```
> x := 14:
  y := InvRabin(Rabin(x, n), op(rkey[2 .. 5]), BW = true):
  igcd(x-y, n);
                                    1
```

Again, we do not get a nontrivial factor of n. However, the situation is not exactly the same as in the preceding attempt. There we obtained $y = x$ because x was a quadratic residue. In this attempt we see that $y \neq x$ and hence the reason that no nontrivial factor was obtained is that $y = -x \mod n$. We can check that this is indeed the case:

```
> -y mod n;
                                    14
```

Let us make a further attempt with $x = 15$:

```
> x := 15;
  y := InvRabin(Rabin(x, n), op(rkey[2 .. 5]), BW = true);
  igcd(x-y, n);
1432104239533502582190206584922959879897681468268458233481122429999871892418707098\
  3744043526569390887082381502579554597290779090060925930683357459602592300811606 2\
  7043265512138558888240115400151990976817072194320700444588374959202833238915150 3\
  2147060303617573395827409313919186027441499054928663931343845660507
```

Now, we have factored n at last. We see at a glance that the integer obtained as output is smaller than n and we can check that it is really a (prime) factor of n:

```
> n mod %;
                                    0
```

8.4.3 CCA Secure Rabin Encryption

In the case of RSA, the plain encryption scheme is completely insecure but there are methods to turn it into a more secure encryption scheme, and we are going to see that this is also the case with Rabin encryption. The Blum–Williams family is trapdoor one-way under the assumption that factoring Blum integers is hard and, by Theorem 3.9, the least significant bit is a hard-core predicate for this family. There is a general method to build a CPA secure encryption scheme from such a family with a hard-core predicate which is discussed, for example, in [109, 10.7.2]. For the family BW_n with hard-core predicate *LSB* this construction is as follows:

- **Gen**: On input 1^k run a PPT algorithm that generates two primes, p, q such that $\text{len}(p) = \text{len}(q) = k$ and $p \equiv q \equiv 3 \pmod 4$, compute $n = pq$ and output the public key $pk = n$ and the private key $sk = (p, q)$.

- **Enc**: On input a public key n and a message $m \in \{0,1\}$, choose a random $x \leftarrow \mathcal{QR}_n$ and compute the ciphertext:

$$(c,d) := \mathbf{Enc}(pk,m) = (BW_n(x), LSB(x) \oplus m).$$

- **Dec**: On input a private key (p,q) and a ciphertext (c,d), compute the message:

$$m := \mathbf{Dec}(sk,(c,d)) = LSB(BW_n^{-1}(c)) \oplus d.$$

The fact that LSB is a hard-core predicate for the Blum–Williams family entails that it gives a pseudo-random bit generator (as we have seen when constructing the Blum–Blum–Shub PRG) when it acts on random seeds. As happens with the 'pseudo-random one-time pad', if an adversary is able to distinguish encrypted messages with non-negligible advantage given the ciphertext, then a PPT algorithm $\mathcal{B}$ can be constructed that calls this adversary and outputs, with non-negligible advantage, $LSB(x)$ for a random x, which is a contradiction and shows that the scheme is CPA secure. More concretely, $\mathcal{B}$ takes as input a public key n and $BW_n(x)$ and calls the adversary $\mathcal{A}$ that generates two different messages $m_0, m_1 \in \{0,1\}$. Then $\mathcal{B}$ chooses a random bit $b \leftarrow \{0,1\}$ and submits to $\mathcal{A}$ the ciphertext $(BW_n(x), g \oplus m_b)$, where g is another randomly chosen bit. $\mathcal{A}$ outputs a bit b' and $\mathcal{B}$ guesses that $LSB(x) = g$ precisely when $\mathcal{A}$ is successful, i.e., when $b = b'$. Thus $\mathcal{B}$ outputs $g \oplus b \oplus b'$, i.e., it outputs g if $b = b'$ and the complement of g otherwise. The advantage of this algorithm is the same as that of $\mathcal{A}$ which is non-negligible by hypothesis, so $\mathcal{B}$ outputs $LSB(x)$ with probability non-negligibly greater than $1/2$, giving the required contradiction. This is the idea of how to prove that the scheme is CPA secure and we refer to [109, Theorem 10.28] for a detailed proof.

This scheme encrypts one bit at a time and hence is very inefficient for practical use. Moreover, the scheme is not CCA secure either. Indeed, given the challenge ciphertext (c,d) whose corresponding unknown plaintext m is to be found, the adversary submits to the decryption oracle the ciphertext (c,d'), where $d' = d \oplus 1$. This ciphertext is distinct from the challenge ciphertext and the oracle returns the corresponding plaintext m'. Then the adversary knows that $c = BW_n(x)$, where $LSB(x) \oplus d' = m'$. This reveals the least significant bit of x because then $LSB(x) = m' \oplus d'$ and hence m is obtained as $m = LSB(x) \oplus d = m' \oplus d' \oplus d = m' \oplus 1$. However, as in the case of RSA, we are going to see that an efficient scheme can be constructed based on the Rabin functions and enjoying IND-CCA security in the random oracle model, again under the assumption that factoring Blum integers is hard. This scheme, introduced in 2001, is based in the OAEP construction and will be described next.

8.4.3.1 The Boneh–Rabin Encryption Scheme Rabin-SAEP$^+$

The encryption scheme Rabin-SAEP$^+$, where the initials SAEP stand for "simple OAEP" was introduced by Boneh in [34] together with a simpler scheme called Rabin-SAEP and, as starting point, requires the following ingredients:

- The family of Rabin functions R_n, where $R_n(x) = x^2 \bmod n$.
- Associated with a security parameter 1^k, two parameters s_0, s_1 such that if $l = k - s_0 - s_1$, we have that $l + s_0 \leq k/2$. Typical values for these parameters, mentioned in [34], are $k = 1024$, $s_0 = 128$ and $l = 384$, although larger values may be used for added security as in the Maple implementation we give below.
- The plaintext space is $\{0, 1\}^l$. Note that $l = 384$ is sufficient for the most common usage of a scheme like this one, which is to encrypt symmetric keys that typically have 256 bits or less.
- Two functions $G : \{0, 1\}^{l+s_1} \to \{0, 1\}^{s_0}$, $H : \{0, 1\}^{s_1} \to \{0, 1\}^{l+s_0}$. As in the case of OAEP, these functions are modeled as independent random oracles in the security analysis of the scheme and in practice they will be derived from a hash function such as SHA-256.

Definition 8.15 Rabin-SAEP$^+$ is the public encryption scheme (**Gen**, **Enc**, **Dec**), with these algorithms defined as follows:

- **Gen**: On input 1^k, with k even, a modulus $n = pq$ is computed, where p and q are $(\frac{k}{2} + 1)$-bit Blum primes and $n \in [2^{k+1}, 2^{k+1} + 2^k)$, i.e., n is a $(k + 2)$-bit integer whose two most significant bits are '10'. The output of **Gen** is the public key n and the private key (p, q).

- **Enc**: Given the message $m \in \{0, 1\}^l$ and the public key n, the following values are successively computed:

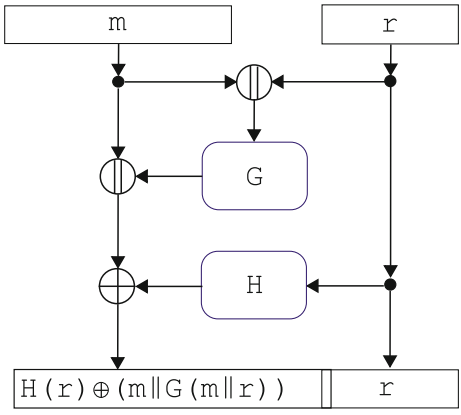
1. $r \leftarrow \{0, 1\}^{s_1}$ (r is a randomly chosen bit string of length s_1).
2. $t := G(m||r) \in \{0, 1\}^{s_0}$.
3. $v := m||t \in \{0, 1\}^{l+s_0}$.
4. $x := v \oplus H(r) \in \{0, 1\}^{l+s_0}$.
5. $y := x||r \in \{0, 1\}^k$.
6. $c := y^2 \bmod n \in \mathbb{Z}_n$ (Here, y is regarded as a k-bit element of $\mathbb{Z}_n$ given by its binary representation, and encrypted with the Rabin function using the public key. In particular, as an integer, $y < 2^k < n/2$. The resulting ciphertext $c \in \mathbb{Z}_n$ may also be regarded as a bit string of length $k + 2$ completing with zeros on the left if necessary). The output of **Enc** is the ciphertext c.

- **Dec**: Given the ciphertext $c \in \mathbb{Z}_n$ and the private key (p, q), the following steps are performed:

1. Compute $m_p := c^{\frac{(p+1)}{4}} \bmod p$ and $m_q := c^{\frac{(q+1)}{4}} \bmod q$ which, as explained in Algorithm 8.3, are square roots of c in $\mathbb{Z}_p$ and $\mathbb{Z}_q$, respectively.
2. Check that $m_p^2 \equiv c \pmod p$ and that $m_q^2 \equiv c \pmod q$. If either condition does not hold then c is not a quadratic residue modulo n and it is rejected as an invalid ciphertext.
3. Use the Chinese remainder theorem as in Algorithm 8.3 to compute the four square roots of c in $\mathbb{Z}_n$, say $\{\pm y_1, \pm y_2\}$. Two of these square roots are greater than $n/2$ and hence they are discarded (since $y < k/2$). Let y_1, y_2 be the two remaining square roots. We know that the plaintext is a k-bit integer so if none of

these roots is $< 2^k$ the ciphertext is, again, rejected as invalid. If just one of the two roots, say y_1, is $< 2^k$ then y_2 is discarded and the computations involving y_2 in the following steps are omitted (or performed as dummy computations for security reasons as explained below). So we may assume that $y_1, y_2 < 2^k$.

4. Viewing y_1, y_2 as k-bit strings, write $y_1 = x_1||r_1$ and $y_2 = x_2||r_2$, where $x_1, x_2 \in \{0,1\}^{l+s_0}$ and $r_1, r_2 \in \{0,1\}^{s_1}$.
5. Compute $v_1 := x_1 \oplus H(r_1)$ and $v_2 := x_2 \oplus H(r_2)$.
6. Write $v_1 = m_1||t_1$ and $v_2 = m_2||t_2$, where $m_1, m_2 \in \{0,1\}^l$ and $t_1, t_2 \in \{0,1\}^{s_0}$.
7. For $i = 1, 2$, test whether $t_i = G(m_i||r_i)$ and if this condition holds for either neither or both values of i, then reject c as an invalid ciphertext.
8. Let $i \in \{1, 2\}$ be the unique index for which the condition in the preceding step holds. The output of **Dec** is m_i.



SAEP$^+$ Padding

Remarks 8.12 As in the case of RSA-OAEP, the functions G, H are modeled as independent random oracles and, as we will indicate below, this allows the attainment of a strong security property. In practice, these functions can be thought of as collision resistant hash functions and, in the implementation below, we will use SHA-256 to derive them in a similar way as in RSA-OAEP.

Observe also that, in Definition 8.15, the result of encoding the message $m \in \{0,1\}^l$ by means of SAEP$^+$ is the k-bit string y obtained in step 5 of the encryption algorithm.[5] When viewed as an element of $\mathbb{Z}_n$, y is an integer of at most k bits and this is helpful in the decryption algorithm because, as already remarked, it means that $y < n/2$. This integer is one of the four square roots of c in $\mathbb{Z}_n$ obtained in step 3 of the decryption algorithm and these square roots are matched in pairs $\pm y_1 \bmod n$ and $\pm y_2 \bmod n$, where exactly one member of each pair is $< n/2$. Thus the two square roots of c in $\mathbb{Z}_n$ that are greater than $n/2$ may be automatically discarded, as is done in step 3. It may be the case that only one of the four square roots is smaller than $n/2$

[5] Once again, we see the difference between encoding and encrypting: SAEP$^+$ encodes messages into a specific format and Rabin-SAEP$^+$ encrypts them.

and then we know that this is the SAEP$^+$-encoded plaintext but a potential source of ambiguity remains as it may happen that two of these square roots are $< n/2$.

This ambiguity could be entirely removed if the encoded messages were restricted to have Jacobi symbol equal to 1 because, as shown in Remarks 8.10, only one square root of c which is less than $n/2$ satisfies this property. This would be easy to accomplish in this scheme for it would suffice to try with different random seeds r at encryption, until finding one such that the Jacobi symbol of the encoded message y is 1. However, this would make encryption less efficient and hence the preferred solution is the one applied in the decryption algorithm above: when two square roots remain after step 3, apply steps 3–7 to both of them until the correct one is found in step 7. Here is where the potential ambiguity mentioned above may surface as it is possible that a valid ciphertext is rejected because the other root also satisfies the condition tested in this step, i.e., $t_i = G(m_i, r_i)$ holds for both $i = 1$ and $i = 2$. However, it is easy to see that the probability of this occurring by chance is negligibly small for reasonable values of the parameter s_0. Indeed, if y_1 is the root that corresponds to the correct plaintext, this happens whenever $G(m_2, r_2) = t_2$ which, assuming that G behaves as a random oracle and taking into account that both terms in the equality are s_0-bit strings, occurs with probability 2^{-s_0}. Since s_0 is typically taken to be ≥ 128 (and we will take $s_0 = 256$ in our implementation), this is a low probability indeed.

We might also wonder whether a malicious encryptor can create a valid ciphertext that is rejected in step 7 but, as pointed out in [34], a similar argument shows that, under the random oracle assumption, the encryptor would have to spend expected time $O(2^{s_0})$ to complete this task and so this would be infeasible in practice for reasonable values of s_0.

It is also important to point out that one should be very careful with the messages informing of decryption errors. As mentioned when discussing the implementation of RSAES-OAEP, these messages may make possible a CCA attack based, for example, on timing analysis that may allow an attacker to distinguish whether the rejection was produced in step 2, step 3 or step 7 of the decryption algorithm. Because of this, in the implementation of Rabin-SAEP$^+$ below, we will carry out steps 4–7 of the algorithm even if the ciphertext is already detected as invalid in step 2 or step 3, and we will delay outputting any invalid ciphertext error message until step 7 of the decryption algorithm is completed.

8.4.3.2 The Security of Rabin-SAEP$^+$

Rabin-SAEP$^+$ is an especially interesting public-key encryption scheme because it fares well in both security and efficiency. From Boneh's results that we are going to explain, it seems that this scheme is preferable to RSA-OAEP from the security point of view and the only aspect in which the latter scheme is (slightly) superior is the fact that it allows the encryption of longer messages for a given modulus size.

The security properties of Rabin-SAEP$^+$ can be summed up by saying that it is a CCA secure scheme under the factoring assumption in the random oracle model.

We have seen that a similar result holds for RSA-OAEP but if we look at the results of Boneh in more detail, it is readily apparent that Rabin-SAEP$^+$ is preferable also in this aspect.

We shall not include the security reduction here and for its details we refer to [34]. We give, however, a brief explanation of the security result and of the reasons why it is especially strong. One fundamental ingredient of the security reduction is Coppersmith's efficient algorithm for finding small zeros of polynomials modulo n which, given a monic polynomial of degree d, $f(x) \in \mathbb{Z}_n[x]$, finds all $x_0 \in \mathbb{Z}$ such that $f(x_0) \equiv 0 \pmod{n}$ and $|x_0| < n^{1/d}$. This is a powerful result that was also used in the reductionist security proof for RSA-OAEP and, to put it in perspective, observe that, for n composite, finding the nontrivial roots of $x^2 \equiv 1 \pmod{n}$ (i.e., the roots distinct from ± 1) gives a nontrivial factor of n and hence is hard in general. However, this does not contradict Coppersmith's result because these nontrivial roots are not "small", as their absolute value is $\geq \sqrt{n}$.

In order to state Boneh's security result, let $T_C(n, d)$ be the running time of Coppersmith's algorithm and let's use the name '(t, q_D, q_G, q_H)-CCA attack' for an algorithm running in time t that makes at most q_D decryption queries to the decryption oracle, q_G queries to the hash function G and q_H queries to the hash function H. Then Boneh's result is the following:

Theorem 8.3 *Consider the* Rabin-SAEP$^+$ *encryption scheme with G and H modeled as random oracles and $l + s_0 \leq k/2$, where $n = pq$ is a modulus generated by the* **Gen** *algorithm in Definition 8.15 on input the security parameter 1^k. Suppose that $\mathcal{A}$ is a (t, q_D, q_G, q_H)-CCA attack which, for the modulus n, has advantage ε (i.e., with probability of success $\geq 1/2 + \varepsilon$). Then there is a PPT algorithm $\mathcal{B}$ that factors n with the following running time and the following advantage (where the advantage $adv(\mathcal{B})$ is, in this case, the probability of success):*

$$time(\mathcal{B}) = t + O(q_D + q_G + q_H T_C(n, 2)),$$
$$adv(\mathcal{B}) \geq \frac{1}{6} \cdot \varepsilon(1 - q_D/2^{s_0} - q_D/2^{s_1}).$$

This result implies the CCA security of Rabin-SAEP$^+$ under the factoring assumption because it says that if there is a PPT adversary that succeeds in the CCA indistinguishability experiment with non-negligible advantage, then there exists a PPT algorithm for factoring the modulus n. Note that, in practice, s_0 and s_1 will be ≥ 128 and the number of queries q_D will be small in comparison to 2^{128}, so the probability of success of the factoring algorithm will be close to $\frac{1}{6} \cdot \varepsilon$.

This security reduction has two clear advantages in comparison to the one for RSA-OAEP. On the one hand, it relies on the hardness of the factorization problem, which is a potentially weaker assumption than the RSA problem and, moreover, has been more extensively studied and is presumably better understood. On the other hand, this reduction is tight, which is not the case for the RSA-OAEP reduction. It has been pointed out, for example in [109] and in [120], that the reasons why the more secure versions or Rabin encryption are much less used than the RSA-based ones are

probably of a psychological nature and due to the fact that plain Rabin encryption is so completely vulnerable to a CCA attack—which, of course, is completely irrelevant for the Rabin-SAEP$^+$ scheme.

8.4.4 Rabin-SAEP$^+$ in Maple

The implementation of Rabin-SAEP$^+$ that follows makes use of many of the auxiliary functions that were used in the RSAES-OAEP implementation given in 8.3.7. We will use them without further comment and here we describe only the new functions, starting with those required by the key generation algorithm. The next function generates a Blum prime of specified bit length and is a minor variation on the function we used to generate pseudo-random primes for RSA keys. The input parameters are `length`, which serves to specify the bit length of the prime to be generated, and `seed`, which requires a random seed given as a positive integer; as usual, care must be taken that the seed is chosen uniformly at random from a set of strings of sufficient length (at least 128-bit—or 16-byte—strings). There are three additional keyword parameters. The first two of them, `min`, `max` serve to specify a minimum and a maximum value such that $2^{\texttt{length}-1} \leq min < max \leq 2^{\texttt{length}}$ and the default values for them are, respectively $2^{\texttt{length}-1}$ and $2^{\texttt{length}}$. Thus if the function is run with default values it merely picks a Blum prime of the specified `length`, but if different values for `min` and `max` are specified then it selects a prime in the subinterval `[min, max]`, which is useful if one wants to generate primes whose product is a modulus of specified length. There is another optional keyword parameter, `bbslength`, that serves to specify the bit length of the primes used by the Blum–Blum–Shub PRG, which takes care of generating the prime candidates. The output is a probable Blum prime in the interval `[min, max]`.

```
> BlumPrime := proc(length::posint, seed::posint,
          {min := 2^(length-1), max:=2^length, bbslength::{512, 768, 1024}:=1024})
  local B, prime, p, rightsize;
  B := RandomTools:-BlumBlumShub:-NewBitGenerator(seed, primes = bbslength);
  prime := false;
  while not prime do
    rightsize := false;
    while not rightsize do
      p := convert(cat(1, seq(B(), i = 1 .. length-3), 1, 1), decimal, binary);
      rightsize := evalb(min < p and p < max)
    end do;
    prime := isprime(p)
  end do;
  p
  end proc:
```

In the function `BlumPrime` we ensure that the prime candidates are $\equiv 3 \pmod 4$ by setting the two low order bits equal to 1. In a similar way, we could make the function more efficient by adequately choosing some of the high order bits whenever `min` is relatively large or `max` is relatively small. But we shall use this simple version, which is sufficient for our purposes.

The next function implements the key generation algorithm for Rabin-SAEP$^+$. We follow the description in [34] but, as usual, the implementation will be byte-oriented.

Therefore, the input parameter `k`, which corresponds to the security parameter in the algorithm, is used to specify the number of bytes (instead of bits). Thus this function generates two $(4k+1)$-bit Blum primes whose product is a $(8k+2)$-bit modulus as specified in [34]. First, a $(4k+1)$-Blum prime is generated and then the second prime is searched for in the subinterval consisting of the integers that multiplied by the first selected prime give an $(8k+2)$-bit modulus whose two most significant bits are '10'. This is done by generating $(4k+1)$-bit integers by means of Blum–Blum–Shub and discarding those that fall outside this subinterval.

The required input parameters are, in addition to `k`, `seed1` and `seed2`, which serve to specify the random seeds that the BBS PRG will use to search for each of the two primes (the seeds can be given either as positive integers or as hexadecimal strings). There is, furthermore, an optional keyword parameter `format` to specify the format in which the keys will be given. This format can be either `hex` (for hexadecimal, which is the default) or `decimal`. As on other similar occasions we allow, for demonstration purposes only, the generation of keys without explicitly supplying seeds to the function, in which case they will be pseudo-randomly generated with smaller seeds taken by Maple from the system. We stress, again, that this method is insecure and that, for proper use, externally generated random seeds of sufficient length should be supplied to the function.

The output of the function is a list whose first element is the modulus, i.e., the public key, and whose second element is the private key given as another list. Thus the output format is $[n, [p, q, pinv, qinv]]$, where p and q are $(4k+1)$-bit Blum primes, and $n = pq$ is an $(8k+2)$-bit integer. *pinv* and *qinv* are, respectively, the inverse of p modulo q and the inverse of q modulo p, which may be used to speed up decryption by means of the Chinese remainder theorem.

```
> RabinKeyGen := proc(k::posint, seed1::{posint, string}, seed2::{posint, string},
                      {format::name := hex, bbslength::{512, 768, 1024}:=1024})
  local pLen, s, p, q, min, max, key;
  pLen := 4*k+1;
  if _params['seed1'] = NULL then
    RandomTools:-MersenneTwister:-SetState();
    s := RandomTools:-MersenneTwister:-GenerateInteger(':-range' = 2^127 .. 2^256)
  else
    s:=stringposint(seed1)
  end if;
  p := BlumPrime(pLen, s, ':-bbslength' = bbslength);
  min := ceil(2^(8*k+1)/p);
  max := floor(3*2^(8*k)/p);
  if _params['seed2'] = NULL then
    RandomTools:-MersenneTwister:-SetState();
    s := RandomTools:-MersenneTwister:-GenerateInteger(':-range' = 2^127 .. 2^256)
  else
    s := stringposint(seed2)
  end if;
  q := BlumPrime(pLen, s, ':-min' = min, ':-max' = max, ':-bbslength' = bbslength);
  igcdex(p, q, 'pinv', 'qinv');
  key := [p*q, [p, q, pinv mod q, qinv mod p]];
  if format = decimal then
    key
  elif format = hex then
    StringTools:-LowerCase~~(convert~~(key, hex))
  else
    error "Unrecognized key format"
  end if
  end proc:
```

Example 8.13 We next generate a Rabin key suitable for use with Rabin-SAEP$^+$. As we have mentioned when discussing the integer factorization problem, the capability to factor a 1024-bit modulus is not far over the horizon and a minimum modulus size of 2048 bits is today regarded as necessary. This corresponds to a value of 256 for the security parameter k in the function `RabinKeyGen` and we will use 128-bit seeds (which is the minimum acceptable length). Thus we obtain a Rabin key with $k = 256$ (so that the modulus will be a 2050-bit integer which is a product of two 1025-bit primes, and whose two most significant bits are '10') as follows:

```
> seed1 := "3d53d86af89150d1437856427eb8ed78":
  seed2 := "d5d7c4472ae89a7b0b0f7580177236a6":
  rk := RabinKeyGen(256, seed1, seed2):
```

We do not print the entire key as it takes too much space but we can, for example, print the two primes in the private key:

```
> rk[2][1 .. 2];

  ["18fcfeb3d2ab10126bcc440d3771778a8feae7f147f94a82a41f0041d6510be5bd10d66ccf8cbe\
  ff825358b9571bbc933f57f14156ecd02cbe615c621e1aa7604eb165bd668c5ce8f5f4a8c95f962f\
  55ca025b67adf5393a44bd7b096ffc8020061098b8b8ccf64fa2d2f1d0644ed1134a1d2348fc63e7\
  627252e53b1ee5a44af", "149bea19772ca84ca84aff83cc3f1b5af62ae010354299205c941bb5b\
  09c43a3cc14d452aa95a31b911e276d7468fd670f9226a4e085c8a0a7ad7eed23ac66d7c80f6a4db\
  a3015a402d97cbadf861d3e1e1bf3cefd210e4353a73498e75a6f89063151a808e87d794b78388e7\
  5c64e8e0b526e5ffc6a177436bfec1a18f25f813"]
```

Exercise 8.25 Write a Maple procedure that checks whether a Rabin-SAEP$^+$ key in the format provided by `RabinKeyGen` is valid. It should accept as input a key and check that the prime factors are $\equiv 3 \pmod 4$ and have the same size, that the last two integers in the private key are the inverses of each of these primes modulo the other and, finally, that the modulus is the product of the two primes and its two most significant bits are '10'. If any of these conditions does not hold, the procedure should return an informative error message, otherwise it should declare that the key is valid. Use this procedure to check the key generated above as well as other keys generated by `RabinKeyGen`, including keys generated without passing the seed values to the function.

We are now ready to give the SAEP$^+$ encoding and decoding functions. Here the functions G, H, are required and we will implement them exactly as in RSAES-OAEP, i.e., we will obtain them from the "mask generating function" `MGF1` which, in turn, was derived from SHA-256 in our implementation. We also have to set the values of s_0 and s_1 (and that of l which follows from them and from the value of k). As we have seen in the preceding discussion on the security of the scheme, Boneh's security proof assumes that $l + s_0 \leq k/2$ and the reduction obtained makes it clear that both s_0 and s_1 should be large enough so that the number of queries to the decryption oracle in a CCA attack is small compared to 2^{s_0} and 2^{s_1}. Because of this a minimum value of $s_0 = 128$ is suggested in [34] for a 1024-bit modulus.

For the reasons already mentioned we will normally use a minimum length of 2048 bits for the modulus, and hence a minimum value $k = 256$ (recall that the value of k represents the number of bytes in this implementation). We will also take $s_0 = 256$ (i.e., a byte length of 32) as default value, which agrees with the output

length of SHA-256. This gives a suggested value of $l = k/2 - s_0$ which, for $k = 256$ produces $l = 96$ as the byte length of the messages to be encrypted but, in fact, the maximum length of messages will be one byte less than this because to recover them uniquely we will pad them with '00' bytes (possibly none) followed by a '01' byte. On the other hand, since $s_1 = k - l - s_0$ and we have taken $l + s_0 = k/2$, we see that $s_1 = k/2$. This is the length of the seed in bytes and is represented by the local variable `sLen` in the function below, which does the SAEP$^+$ encoding. Thus, in particular, for $k = 256$, a random 128-byte seed should be passed to this function by the Rabin-SAEP$^+$ encryption function to be defined later.

The encoding will be performed by the function `SAEPPlusEnc` below, where the required input parameters are the message (given as a hexadecimal string), the security parameter k, and `seed`, for a $k/2$-byte random seed (given also as a hexadecimal string). The seed will usually be supplied by the Rabin-SAEP$^+$ encryption function but the function will also work if no seed is provided; in that case the seed will be pseudo-randomly generated inside the function and, as on other similar occasions, we warn that this method is not secure. The optional keyword parameters are `H` for the name of the hash function used to build `MGF1` (with `'SHA256'` as default) and the output byte length of this function (with 32 as default).

```
> SAEPPlusEnc := proc(message::string, k::posint, seed::string,
                      {H::name := 'SHA256', hLen := 32})
  local mLen, sLen, l, pd, M, r, t, v, h, x;
  mLen := iquo(StringTools:-Length(message), 2);
  sLen := iquo(k, 2);
  l := sLen-hLen;
  if l <= mLen then
    error "message too long"
  end if;
  pd := StringTools:-Repeat("00", l-mLen-1);
  M := cat(pd, "01", message);
  if _params['seed'] = NULL then
    StringTools:-Randomize();
    r := StringTools:-LowerCase(StringTools:-Random(k, xdigit))
  else
    if not StringTools:-IsHexDigit(seed) or StringTools:-Length(seed) <> k then
      error "seed must be a %1-byte hex string", k/2
    end if;
    r := seed
  end if;
  t := MGF1(cat(M, r), hLen);
  v := cat(M, t);
  h := MGF1(r, sLen);
  x := bytestohexstring(BitXor~(hexstringtobytes(v), hexstringtobytes(h)));
  cat(x, r)
  end proc:
```

The SAEP$^+$ decoding function, `SAEPPlusDec`, is given next. The input parameters are the same as those of the encoding function, except for the seed which is no longer necessary. The function merely reverses the steps in the encoding function and only one important peculiarity is worth mentioning. As we have already remarked, one should be very careful with the error messages produced by the Rabin-SAEP$^+$ decryption function. For this reason, `SAEPPlusDec` does not return an error message if an invalid ciphertext is detected and what it does is to return the `NULL` sequence instead. This will allow the Rabin-SAEP$^+$ decryption function to output an error message only at the end in case of failure.

```
> SAEPPlusDec := proc(EM::string, k::posint, H::name := 'SHA256', hLen := 32)
  local sLen, x, r, h, v, l, M, t;
  sLen := iquo(k, 2);
  x := StringTools:-Take(EM, k);
  r := StringTools:-Drop(EM, k);
  h := MGF1(r, sLen);
  v := bytestohexstring(BitXor~(hexstringtobytes(x), hexstringtobytes(h)));
  l := sLen-hLen;
  M := StringTools:-Take(v, 2*l);
  t := StringTools:-Drop(v, 2*l);
  if t <> MGF1(cat(M, r), hLen) then
    return NULL
  end if;
  while StringTools:-IsPrefix("00", M) do
    M := StringTools:-Drop(M, 2)
  end do;
  StringTools:-Drop(M, 2)
  end proc:
```

Example 8.14 Let us encode and decode a message with default values and letting the encoding function generate the seed:

```
> SAEPPlusEnc("0123456789abcdef", 256);
"593b9e4088c813850d8453a14523d4a8c1154dc947ab7adab05eef6775565194e356c9d77791a6d91\
  b7db3b7f574472f5e8f0df784b1eeca94239a8dfb7f7781c2d03de5d57ca25181af8c40945b2b6e4\
  303d2e60daab6d7cfdd53d1ff6cbb026778d50b430cec1d9fc69b9516851a6d78d4c642667f013fb\
  696b70e862bb65fbfaae493cb92c4546ab9a707e2ee6af1feecb8addd71b9fd2cdeedc4ffecfdbef\
  d601af38bfd64eeddb59cce7b0fe946d71e6af9c42dc9c77fbdbc71ea7204d5f53aeaff43afbbfaf\
  dcbf0eb40dbeceabaef97eb310b76ac3dcfc3d5eb27c683bcc74848ae3bfe637a26cdde83b3ff78c\
  528da9705dceb2aa2d2ebcddafb9abe"

> SAEPPlusDec(%, 256);
                              "0123456789abcdef"
```

Of course, if the same commands are repeated several times, the encoded message is different each time due to the different seeds used.

We now have all the ingredients required to implement the Rabin-SAEP$^+$ scheme. We start with the encryption function whose required input parameters are `publickey`, `message` and `seed`. The first of them is used to specify the public key, i.e., the modulus which can be given either as a decimal integer or as a hexadecimal string. Similarly, `message` is for the message to be encrypted, which can be given either as a hexadecimal or an ordinary text string. The last parameter is for the seed, which should be a hexadecimal string of $k/2$ bytes, assuming that the modulus is an $(8k+2)$-bit integer generated by `RabinKeyGen` (i.e., the key has been generated with security parameter k). As happens with the function `SAEPPlusEnc`, this function will also work if no seed is provided as input (because it calls `SAEPPlusEnc`, which will pseudo-randomly generate the seed in that case). We stress once again that for proper security the seed should be randomly chosen and supplied to this function.

In addition to these required parameters, there are three input parameters that are similar to those in `SAEPPlusEnc` with the exception of `messagetype`, which serves to indicate to the function whether the message to be encrypted is to be regarded as a hexadecimal string or as an ordinary text string. The admissible values are 'hex' and 'text', with the former the default. The output is the ciphertext given as a $(k+1)$-byte string.

```
> RabinSAEPPlusEncrypt := proc(publickey::{posint, string}, message, seed::string,
    {messagetype::identical(hex,text):=hex, H::name:='SHA256', hLen::posint:=32})
  local n, e, k, M, EM, m, c, C;
  if type(publickey, string) then
    n := convert(publickey, decimal, hex)
  else
    n := publickey
  end if;
  k := iquo(intlog[2](n)-1, 8);
  M := messagetohex(message, messagetype);
  EM := SAEPPlusEnc(M, k, _params['seed'], ':-H' = H, ':-hLen' = hLen);
  m := OS2IP(EM);
  c := Power(m, 2) mod n;
  C := I2OSP(c, k+1)
  end proc:
```

Next we give the decryption function. The input parameters are similar to those in the encryption function, except that the seed is not used here. For the security reasons already stated, a local variable called `reject` is maintained inside the procedure to keep track of the different reasons why a ciphertext should be rejected as invalid. If the message is detected as invalid at the beginning, the computational steps—which are now meaningless—are completed all the same and the message declaring the ciphertext as invalid is not returned until the end of the procedure. As already mentioned, this is necessary for Boneh's security reduction to apply and, specifically, to prevent CCA attacks in which the adversary uses timing analysis. The output of the function is, assuming that all goes well, the original message.

```
> RabinSAEPPlusDecrypt := proc(privatekey::list, C::string,
    {messagetype::identical(hex,text):= hex,H::name:='SHA256', hLen::posint:=32})
  local sk, n, k, c, cand, reject, y, v;
  if type(privatekey[1], string) then
    sk := convert~(privatekey, decimal, hex)
  else
    sk := privatekey
  end if;
  n := sk[1]*sk[2];
  k := iquo(intlog[2](n)-1, 8);
  reject := false;
  if StringTools:-Length(C) <> 2*k+2 or k < 2*hLen+2 then
    reject := true
  end if;
  c := OS2IP(C);
  cand := InvRabin(c, op(sk));
  if nops(cand) = 5 then
    reject := true
  end if;
  cand := select(x -> evalb(x < 256^k), cand);
  y := I2OSP~(cand, k);
  v := SAEPPlusDec~(y, k);
  if nops(v) <> 1 or reject then
    error "invalid ciphertext"
  else
    hextomessage(op(v), messagetype)
  end if;
  end proc:
```

Example 8.15 Let us consider the previously generated Rabin-SAEP$^+$ key `rk`, and the message and seed defined below:

```
> me := "The magic words are squeamish ossifrage":

  seed3 := "d41a480c52d01a7c2faef86f94b3a35e6749a3d610fb8cebc507d7e26f206b1e6e3e18\
  54992a1c2111005b58dd11f10ba57ea81a02cdc01b78ae371228c38a233d8b2e8c9168ea2138e171\
  bb0efa1d65390e957ee931669b755a290fe46850a68bd42610f0d657bbd710e8040cae585eb1faf6\
  505a20b443899ff84f566365fd":
```

Note that `seed3` is a 128-byte string and hence, according to our implementation, has the required size to be used with a key such as `rk` which has been obtained from a security parameter $k = 256$. The result of encrypting the above message with this key and seed is then:

```
> RabinSAEPPlusEncrypt(rk[1], me, seed3, messagetype = 'text');
"01c9e281233ecfd0f9d048c99215f669245724d180ed65cb564f7262c7a962fcad55b8476d9c52b0e\
  a7d79e7ad6cd4cd29d8caaf08d96e65df0b607a7ae1f40fa06ea8f927744114bb2f0b24ca5a99ff7\
  5a2b4382f5a98825540431ec618143f2f883569be198a1337157134f333520e06b3e054384fe393c\
  ccc17875259af0a059e9bba4f5044c79458f1e45347f08d7e46042cf3c46325eafaf1e5bc98cc283\
  2c71a6d90b3f9de1e80f6b64f54af69622151092417d03e53c777839d7fe07e05c57387cadb459e6\
  21fabfa9db38c49b14aecee50530d77a24047115232a015d9786d260155078e8ef17f3f1cf87da92\
  7417ac5cac9864358635df65dd9406c92"
```

Now decryption proceeds as follows:

```
> RabinSAEPPlusDecrypt(rk[2], %, messagetype = text);
                  "The magic words are squeamish ossifrage"
```

8.5 The Elgamal Encryption Scheme

As we have seen in Chap. 7, the birth of public-key cryptography is associated with the Diffie–Hellman protocol which was motivated by the belief that the discrete logarithm problem is hard. The two most important number-theoretic problems that are thought to be hard are the factorization problem and the DL problem, and most computational problems used as assumptions to obtain cryptographic security reductions are related to one of these problems. The security of the first public-key encryption scheme, RSA, is related through the RSA assumption to the integer factorization problem but it was not clear how to define an encryption scheme similarly related to the DL problem. Finally, in 1984 T. Elgamal [71] found a way to turn the Diffie–Hellman protocol into a public-key encryption scheme.

The idea underlying the Elgamal encryption scheme is the following. Suppose that we have a finite group G and define a private-key encryption scheme as follows:

- **Gen** chooses an element $a \in G$ uniformly at random.
- **Enc** uses G as message space and, given $m \in G$, encrypts it as $c := am \in G$.
- **Dec** decrypts c by computing $m := a^{-1}c$.

This encryption scheme has perfect secrecy as a consequence of Shannon's theorem (Theorem 3.1) because the probability distribution on the key space is uniform by hypothesis and, for each $m \in G$ and each $c \in G$, there is a unique key $a \in G$—namely, $a = cm^{-1}$—such that $\mathbf{Enc}(a, m) = c$.[6] The idea to turn this into a public-key

[6] We used this argument to prove the perfect secrecy of the one-time pad, which is obtained from this construction by taking G equal to the group of binary strings of length n with the Xor operation.

encryption scheme is to let an element of G, say h, be the public key of a user and encrypt m as $h^r m$, where r is a randomly chosen integer. This way a system similar to the private-key scheme above is being used, with $a = h^r$ which, as a consequence of r being random, will "look random" to an adversary. However, this is not sufficient because then the legitimate receiver would be unable to decrypt (as she does not know r). Thus the system is modified in a way similar to the one used to build a CPA secure encryption scheme from a trapdoor one-way family of permutations with a hard-core predicate (as we have seen in 8.4.3 for the Blum–Williams family of permutations). The idea is to give the receiver additional information that will allow her to decrypt using her private key. The private key will be a positive integer x such that $h = g^x$, where g is a publicly known element of G (so that x will be hard to recover from h assuming the DL problem is hard) and the additional information that allows the receiver to recover the message is simply the element $g^r \in G$. More explicitly, this leads to the encryption scheme that we now describe.

The scheme will operate on a finite group G of order t, which is usually taken to be a cyclic group with generator g, so that $G = \langle g \rangle$. As discussed in Sect. 6.5, we assume that we have a specification of the group in which group elements are represented by strings in such a way that it is easy to check whether a string represents the identity element and whether two strings represent the same element. In addition, the specification includes efficient algorithms to compute the string representing the product of two elements or the inverse of an element, as well as the order of the group. Moreover, we will assume that there is a PPT algorithm $\mathbf{Gen}_G$ that, on input 1^k, outputs a (specification of a) cyclic group G of order t with $\text{len}(t) = k$ and a generator $g \in G$. It is often the case that G and g are the same for all users, so that they are system parameters and the key generation algorithm will take them as given inputs. The Elgamal encryption scheme is then defined as follows:

Definition 8.16 *Elgamal* is the public-key encryption scheme $(\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ defined by the following algorithms:

- **Gen**: On input 1^k run $\mathbf{Gen}_G$ to obtain (G, g), where G is a finite group of order t, with $\text{len}(t) = k$ and $G = \langle g \rangle$. Then choose $x \leftarrow \mathbb{Z}_t$ and compute $h := g^x$. The public key is the 3-tuple (G, g, h) and the private key is (G, g, x). If G and g are regarded as system parameters, then the public key is simply h and the private key is x.
- **Enc**: On input a public key $pk = (G, g, h)$ and a message $m \in G$, choose a random element $r \leftarrow \mathbb{Z}_t$ and compute the ciphertext:

$$c := \mathbf{Enc}(pk, m) = (g^r, h^r m) \in G \times G.$$

- **Dec**: On input a private key $sk = (G, g, x)$ and a ciphertext $c = (c_1, c_2) \in G$, compute the message:

$$m := \mathbf{Dec}(sk, c) = c_1^{-x} c_2.$$

Decryption works as intended: If $(c_1, c_2) = \mathbf{Enc}((G, g, h), m) = (g^r, h^r m)$ and $h = g^x$, then $\mathbf{Dec}((G, g, x), (c_1, c_2)) = c_1^{-x} c_2 = g^{-rx} h^r m = g^{-rx} g^{rx} m = m$.

Remarks 8.13 Elgamal encryption requires just two exponentiations in G, which are obtained by applying the binary exponentiation algorithm to the multiplication in G, and is, therefore, efficient. Moreover, the exponentiations are independent of the plaintext being encrypted and hence they can be precomputed and their results stored in advance, so that encryption will require only a group multiplication and will be very efficient—much more so, for example, than RSA encryption. A minor inconvenience is that the ciphertext doubles the size of the plaintext, i.e., the scheme produces *message expansion* by a factor of 2. But this is compensated by the fact that encryption is probabilistic which, as we have seen, is a necessary condition for an encryption scheme to meet even minimal security requirements. Decryption requires essentially one exponentiation in G and hence is also efficient although now we cannot use the CRT to speed it up as we did with RSA decryption.

Exercise 8.26 Show that Elgamal is a homomorphic encryption scheme by proving that, for a given key, the Elgamal decryption algorithm defines a group homomorphism $G \times G \to G$.

8.5.1 Security of Elgamal

Let us now analyze the security properties of Elgamal encryption. First, we have:

Proposition 8.7 *Elgamal is OW-CPA secure if and only if the CDH problem is hard relative to* $\mathbf{Gen}_G$.

Proof We show that if the CDH problem is hard then Elgamal is OW-CPA secure. Suppose, on the contrary, that Elgamal is not OW-CPA secure and hence, according to Definition 8.4, there exists a PPT adversary $\mathcal{A}$ such that $\Pr(\mathbf{PubK}^{ow\text{-}cpa}_{\mathcal{A},\text{Elgamal}}(k) = 1)$ is non-negligible. Then we may show that there exists a PPT algorithm $\mathcal{B}$ that solves the CDH problem with non-negligible probability. On input (G, g, g^x, g^y) where G is a cyclic group of order t, g a generator of G, and $x, y \leftarrow \mathbb{Z}_t$, $\mathcal{B}$ tries to compute $g^{xy} \in G$ by proceeding as follows:

- After being given its input, $\mathcal{B}$ defines the Elgamal public key $pk := (G, g, g^x)$ and the ciphertext $c := (g^y, 1)$.
- Then $\mathcal{B}$ runs $\mathcal{A}$ on input the public key pk and the challenge ciphertext c and eventually $\mathcal{A}$ returns a message $m \in G$.
- Finally $\mathcal{B}$ outputs $m^{-1} \in G$.

If $\mathcal{A}$ succeeds in the $\mathbf{PubK}^{ow\text{-}cpa}_{\mathcal{A},\text{Elgamal}}(k)$ experiment, then we see that $(g^y, 1) = \mathbf{Enc}((G, g, g^x), m) = (g^r, g^{xr} m)$. Hence $r = y$ and $g^{xr} m = g^{xy} m = 1$, so that $\mathcal{B}$ outputs g^{xy} and hence succeeds in solving the instance of the CDH problem. Since, by hypothesis, $\mathcal{A}$ succeeds with non-negligible probability, so does $\mathcal{B}$, and

this completes the proof of the implication. The converse implication is left as an exercise. □

Exercise 8.27 Show that if the Elgamal scheme is OW-CPA secure then the CDH problem is hard.

Remarks 8.14 We have seen in Sect. 7.3 that the CDH problem is no harder than the DL problem. Therefore, hardness of the DL problem is a necessary condition for Elgamal to be OW-CPA secure but it is not known whether the reverse implication holds. We have also seen that solving the CDH problem is equivalent to being able to recover the key in the DH protocol so we arrive at the conclusion that the latter capacity is also equivalent to breaking Elgamal (in the sense of plaintext recovery).

As we have mentioned on several occasions, OW-CPA security is a very weak security property. However, the fact that Elgamal encryption is probabilistic allows us to prove that Elgamal enjoys a much stronger security property under an assumption which is likely stronger than the CDH problem being hard, namely, that the DDH problem is hard (see Sect. 7.3 for the definition).

Theorem 8.4 *The Elgamal encryption scheme is IND-CPA secure if and only if the DDH problem is hard relative to* $\mathbf{Gen}_G$.

Proof First we show, by contradiction, that if the DDH problem is hard then Elgamal is IND-CPA secure. Let $\mathcal{A}$ be a PPT adversary against Elgamal and let $\varepsilon(k)$ be $\mathcal{A}$'s advantage in the indistinguishability experiment $\mathbf{PubK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{Elgamal}}(k)$, i.e., $\Pr(\mathbf{PubK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{Elgamal}}(k) = 1) = 1/2 + \varepsilon(k)$. Assuming that $\varepsilon(k)$ is non-negligible, we construct a PPT algorithm that succeeds on the DDH experiment of Definition 7.4 with non-negligible advantage as follows. Suppose that $\mathbf{Gen}_G$ was run on input 1^k, with output (G, g), where $|G| = t$, and random elements were chosen: $i \leftarrow \{0, 1\}$, $x, y \leftarrow \mathbb{Z}_t$. Suppose further that $h \in G$ is defined by setting $h := g^{xy}$ if $i = 0$ and $h \leftarrow G$ if $i = 1$ (so that h is chosen uniformly at random in this case). Then $\mathcal{B}$ is given (G, g, g^x, g^y, h) as input and its task is to guess the bit i. $\mathcal{B}$ proceeds as follows:

- $\mathcal{B}$ runs $\mathcal{A}$ on input the Elgamal public key (G, g, g^x) and $\mathcal{A}$ outputs two messages $m_0, m_1 \in G$.
- $\mathcal{B}$ chooses a random bit $b \leftarrow \{0, 1\}$ and sets $(c_1, c_2) = (g^y, hm_b)$.
- $\mathcal{B}$ runs $\mathcal{A}$ on the ciphertext (c_1, c_2) and $\mathcal{A}$ outputs a bit b'.
- $\mathcal{B}$ outputs the bit $b \oplus b'$.

Let us analyze the probability that $\mathcal{B}$ succeeds. Assume first that $i = 0$, in which case $h = g^{xy}$. Then $c_2 = hm_b = g^{xy}m_b$ and hence (c_1, c_2) is a valid Elgamal ciphertext for the public key (G, g, g^x). Then $\Pr(\mathcal{B} \text{ succeeds}|i = 0) = \Pr(b \oplus b' = 0|i = 0) = \Pr(b = b'|i = 0) = \Pr(\mathbf{PubK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{Elgamal}}(k) = 1) = \varepsilon(k) + 1/2$.

Suppose next that $i = 1$ and hence that h is a random element of G. Then $(c_1, c_2) = (g^y, hm_b)$ and the second component hm_b is uniformly distributed in G because $hm_b = h_0m_0 = h_1m_1$ for unique elements $h_0, h_1 \in G$ and these values are equally likely to occur since h is randomly chosen. Observe that now the pair (c_1, c_2)

may not be a valid Elgamal ciphertext but we may assume without loss of generality that $\mathcal{A}$ outputs a bit also in this case. We have just seen that the second component c_2 is a random element of G and so is $c_1 = g^y$ because y was randomly chosen in $\mathbb{Z}_t$. Thus none of them carries any information about m and hence the probability that $\mathcal{A}$ will guess the correct bit is exactly $1/2$. Thus $1/2 = \Pr(b = b'|i = 1) = \Pr(b \neq b'|i = 1) = \Pr(\mathcal{B}$ succeeds$|i = 1)$. Since we know the probability that $\mathcal{B}$ succeeds in both cases, we can compute the probability that $\mathcal{B}$ succeeds by using Proposition 2.1 and we obtain:

$$\begin{aligned}\Pr(\mathcal{B}\text{ succeeds}) &= \frac{1}{2}\Pr(\mathcal{B}\text{ succeeds}|i=0) + \frac{1}{2}\Pr(\mathcal{B}\text{ succeeds}|i=1)\\ &= \frac{1}{2}(\frac{1}{2} + \varepsilon(k) + \frac{1}{2}) = \frac{1}{2} + \frac{1}{2}\varepsilon(k).\end{aligned}$$

Since $\varepsilon(k)$ is non-negligible, so is $\frac{1}{2}\varepsilon(k)$, which is the advantage of $\mathcal{B}$ in solving the DDH problem, and this completes the proof of the implication.

For the converse implication, suppose that there exists a PPT algorithm $\mathcal{B}$ such that $\Pr(\mathbf{DDH}_{\mathcal{B},\mathbf{Gen}_G}(k) = 1) = 1/2 + \varepsilon(k)$, where $\varepsilon(k)$ is a non-negligible function (i.e., $\mathcal{B}$ solves the DDH problem with non-negligible advantage). Then we construct a PPT adversary $\mathcal{A}$ such that $\mathcal{A}$ has non-negligible advantage in the CPA indistinguishability experiment $\mathbf{PubK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{Elgamal}}(k)$ (so that $\mathcal{A}$ succeeds with probability non-negligibly better than random guessing) as follows. On input the public key (G, g, h), where G has order t, $\mathcal{A}$ generates two messages $m_0, m_1 \in G$ and is given the ciphertext $(c_1, c_2) = (g^r, h^r m_b)$, where $b \leftarrow \{0, 1\}$ and $r \leftarrow \mathbb{Z}_t$ are randomly chosen and unknown to $\mathcal{A}$. $\mathcal{A}$ runs $\mathcal{B}$ on input:

$$(G, g, c_1, h, c_2 m_0^{-1})$$

and $\mathcal{B}$ outputs a bit b'. Then $\mathcal{A}$ outputs b' too.

Let us analyze the behavior of $\mathcal{A}$. If $b = 0$, i.e., if m_0 is the encrypted message, then $(G, g, c_1, h, c_2 m_0^{-1}) = (G, g, g^r, h, h^r m_0 m_0^{-1}) = (G, g, g^r, g^x, g^{rx})$, (where (G, g, x) is the Elgamal private key—unknown to $\mathcal{A}$—, so that $h = g^x$). Thus, recalling Definition 7.4, we see that $\mathcal{A}$ succeeds in the $\mathbf{PubK}^{ind\text{-}cpa}_{\mathcal{A},\mathrm{Elgamal}}(k)$ experiment (i.e., $\mathcal{A}$ outputs 0) precisely when $\mathcal{B}$ succeeds in the DDH experiment, namely, when $\mathcal{B}$ outputs 0. On the other hand, if $b = 1$, then $c_2 m_0^{-1} = h^r m_1 m_0^{-1}$ is uniformly distributed in G because m_0 and m_1 are uniformly chosen at random and, again in this case, $\mathcal{A}$ succeeds precisely when $\mathcal{B}$ succeeds (except with negligible probability). Since the advantage of $\mathcal{B}$ is non-negligible by hypothesis, so is the advantage of $\mathcal{A}$, completing the proof. □

Remark 8.6 Observe that, while Theorem 8.4 is a much more interesting result than Proposition 8.7, there may be group generation algorithms $\mathbf{Gen}_G$ to which the latter applies but the former does not. Let us consider, for example, a PPT algorithm that on input k outputs the pair $(\mathbb{Z}_p^*, g)$, where p is a random k-bit prime and g a primitive

root modulo p. Then, as we have seen in Example 7.1, the DDH problem is easy with respect to this algorithm (or, in simpler words, the DDH problem is easy in the groups $\mathbb{Z}_p^*$) and hence the Elgamal encryption scheme over these groups is not IND-CPA secure. However, the CDH problem is thought to be hard in these groups and hence Elgamal is thought to be OW-CPA secure also in this case.

As we have seen, a desirable security property for a public-key encryption scheme is IND-CCA, which is stronger than the IND-CPA property we have just proved. However, despite having probabilistic encryption, Elgamal's behavior vis-a-vis CCA attacks is no better than that of plain RSA and the reason is that Elgamal is also malleable and the adversary can change the ciphertext in such a way that the plaintext is changed in a controlled way:

- *Elgamal is not IND-CCA secure.*

To see this, suppose that the adversary intercepts a ciphertext (c_1, c_2) that is an encryption of the plaintext $m \in G$ with the public key (G, g, h). Then the pair (c_1, c_2m') is an encryption of $mm' \in G$ for any $m' \in G$. Indeed, there exists an $r \in \mathbb{Z}_t$ (where t is the order of G) such that $(c_1, c_2) = (g^r, h^r m)$ and hence $(c_1, c_2m') = (g^r, h^r mm')$ (a particular case of the homomorphic property since $(1, m')$ is an encryption of m'). Thus an adversary can mount a successful CCA attack by submitting the ciphertext (c_1, c_2m') and multiplying the corresponding plaintext by m'^{-1}. This attack might be easy to detect since the submitted ciphertext has the same first component as the challenge ciphertext, which will occur with negligible probability assuming that the ciphertext has been obtained by applying the encryption algorithm to a plaintext and that the DDH problem is hard. However, the adversary can also disguise this fact and use the homomorphic property to construct a valid ciphertext whose first component is random. For this, the attacker picks a random $s \leftarrow \mathbb{Z}_t$ and submits the ciphertext $(c_1 g^s, c_2 h^s m')$. Now the first component is uniformly distributed in G assuming that s is uniformly chosen at random and this ciphertext is equal to $(g^{r+s}, h^{r+s} mm')$, which is, again, a valid encryption of the plaintext mm'.

In view of the preceding results, we see that Elgamal is perhaps the most efficient public-key encryption scheme that is CPA secure under reasonable conditions. This is a good thing but, of course, one needs to know that the DDH problem is hard relative to $\mathbf{Gen}_G$. While a necessary condition for this to happen is that the CDH problem is hard, we have just seen that this condition is not sufficient, although there are groups in which both conditions appear to be equivalent. On the other hand, the most studied and better understood problem related to the CDH problem is the DL problem but, again, the hardness of the latter is only known to be a necessary condition for the hardness of the former. Thus in order to implement Elgamal encryption one looks for groups in which the DL problem appears to be hard and, moreover, do not have any obvious additional weaknesses.

8.5.2 Elgamal on the Group of Quadratic Residues Modulo a Safe Prime

Apart from the security-related matters, there is another problem that must be addressed when implementing Elgamal encryption, namely the question of how to encode messages, given in the form of binary strings, as elements of a group G. This encoding depends on the specification of the groups used and we are going to see how Elgamal can be implemented on a specific family of groups in which the DDH problem is thought to be hard—and hence Elgamal is thought to be IND-CPA secure—and, moreover, binary strings can be encoded as group elements in a relatively straightforward manner.

As we have seen in 7.2.3, a class of groups suitable for the DH protocol is that of the groups $\mathcal{QR}_p \subseteq \mathbb{Z}_p^*$ (i.e., the groups of quadratic residues of the groups $\mathbb{Z}_p^*$) where p is a safe prime. The reason is that the DDH problem is thought to be hard in these groups which, therefore, seem also appropriate for Elgamal encryption. The problem of how to encode binary strings as elements of these groups is easily solved as follows. Consider a safe prime p and let $q = (p-1)/2$ and $\text{len}(q) = n+1$. The map $\{1, 2, \ldots q\} \to \mathcal{QR}_p$ given by $x \mapsto x^2 \bmod p$ is bijective—because every element of $\mathcal{QR}_p$ has two square roots in $\mathbb{Z}_p^*$ exactly one of which is $\leq q$—and its inverse can be efficiently computed by means of Algorithm 2.8. Now, an injective map $\{0,1\}^n \to \{1, 2, \ldots q\}$ may be constructed by regarding each $s \in \{0,1\}^n$ as an integer in a natural way (by identifying it with the integer whose binary expansion is s) and mapping $s \mapsto s+1$ in order to exclude 0 from the image. We will use an alternative way to encode an n-bit string as an element of $\mathcal{QR}_p$ when implementing the Cramer–Shoup scheme in 8.6.2.

In conclusion, since encoding/decoding messages in these groups can be carried out efficiently, we see that they are good candidates to implement Elgamal encryption. However, it is very likely that they are not the optimal groups for this purpose because, although the DDH problem seems hard, there are other groups in which this problem may be still harder. This has important practical consequences because the harder the problem the shorter the keys one must use in practice to attain a certain security level. The precise hardness of the DDH problem is difficult to gauge and usually one settles for an assessment of the difficulty of the DL problem instead—but one should always bear in mind that the DDH problem might be easier as actually happens in the case of the groups $\mathbb{Z}_p^*$. However, in other cases in which no obvious weaknesses are apparent, the DL problem is regarded as a good yardstick to evaluate key sizes for practical use. In Chap. 6 we have seen that there are basically two large classes of algorithms to solve the DL problem: the generic algorithms that work in any (cyclic) group and the specific ones that work only in specific classes of groups whose additional properties are exploited. The algorithms in the first class are all exponential—with complexity close to the square root of the order of the cyclic group—and hence extremely inefficient in practice, while those that exploit specific properties may be much more efficient. An extreme example is the polynomial-time algorithm that computes discrete logarithms in the additive groups $\mathbb{Z}_n$ and a more

interesting example is the class of algorithms grouped under the term 'index calculus method' for discrete logarithms in subgroups of the multiplicative groups of finite fields (including the subgroups of $\mathbb{Z}_p^*$). These algorithms are subexponential and hence, while the DL problem in the groups to which they apply may still be difficult, longer keys are required in comparison with the groups where only generic methods are available.

The best candidates for Elgamal encryption are certain elliptic curve groups because no subexponential DL algorithms are known for them and, in addition, there are arguments that suggest that finding factor base methods for these groups is unlikely. Thus these groups, which we study in more detail in Chap. 11, provide the best framework for Elgamal encryption as they allow the use of relatively small key lengths.

Example 8.16 We do a Maple example of Elgamal encryption/decryption in the group of quadratic residues modulo a safe prime. Instead of doing it by means of a couple of Maple functions we do it step by step. Let us start by generating a 1024-bit safe prime:

```
> with(RandomTools:-MersenneTwister):
  SetState():
  p := numtheory:-safeprime(GenerateInteger(range = 2^1023 .. 2^1024-1));
  q := (p-1)*(1/2);

114790821943818019585928129288248749673721993950469127804601612874474826538355007 6\
  315443837809191512800214837856113141939357003163829253508552035249423535884437 47\
  912671404888038195899825412611345277717691147490532088824750469893984359499157 06\
  781546164074339399000225773974693478954292142492766816760947723030 3
573954109719090097929640646441243748368609969752345639023008064372374132692177503 81\
  577219189045957564001074189280565709696785015819146267542760176247117679422187 39\
  563357024440190979499127063056726388588455737452660444123752349469921797495785 33\
  907730820371696995001128869873467394771460712463834083804738615151
```

Let us check that p is indeed a safe prime and what the congruence class of q modulo 4 is:

```
> isprime~(p, q);
  q mod 4;
                                  true
                                    3
```

Knowing that q is congruent to 3 modulo 4 serves to compute modular square roots modulo p more easily but this condition is not necessary and Maple will compute square roots with the command `numtheory:-msqrt` modulo any prime (or any integer that can be easily factored). Next we compute a generator of the group $\mathcal{QR}_p$:

```
> g := GenerateInteger(range = 2 .. q)^2 mod p;
408047438860197231296702362136853885871787643618678478655782004981054844789311225 2\
  829127151446843493264396310334010285756200591340245184979524672968253969869499 75\
  860850167955749628244746197876291659676845457034199159955375447267513988072092 74\
  372792408126101544138638871292617803600154515952198355600202795 37
```

We know that this element is a generator because the group of quadratic residues modulo p (to which g belongs since it is a square) has order q, which is prime, and hence any element distinct from 1 is a generator. However, we can also check it by letting Maple compute its order, which will turn out to be q:

```
> evalb(numtheory:-order(g, p) = q);
                                  true
```

Next we are going to generate a public key for the Elgamal encryption scheme that uses the group $\mathcal{QR}_p$. For this we have to choose a random $x \leftarrow \mathbb{Z}_q$ (since $|\mathcal{QR}_p| = q$) and compute $h := g^x$. For the scheme to be secure, x must be chosen uniformly at random in $\mathbb{Z}_q$ and an acceptable compromise for practical use would be to choose x pseudo-randomly using a PRG seeded with a sufficiently large random seed. Thus we could use the Blum–Blum–Shub PRG as we did, for example, in 7.2.3 but, for simplicity, we are going to use here the Mersenne Twister algorithm with automatic—and hence non-random—seeding, with the explicit warning that this method is far from secure.

```
> SetState():
  x := GenerateInteger(range = 1 .. q);
27832496460659967717400460547392704735986575559495652987585126626731674529603064 64\
  58104708039039362885098861657100793605344879664082558794193452138563885612792802\
  57529317032079657377106090678397780580037301145826758289150258633311009363206425\
  67452054354523563173167226706081488681412757808620297860173482181 5

  h := Power(g, x) mod p;
78284542243558339801635702316561362631054507079594976926125343244349222001519412 58\
  11916148010166461105690808335681809483425901923593566905839277101756901875894985\
  58602444862876541682381527958578184030274859314473498903445945999366026309981472\
  60554493251097917192502420252428918830583732348019263265938152 8567
```

The public key will then be $(\mathcal{QR}_p, g, h)$ and the private key $(\mathcal{QR}_p, g, x)$. Note that the group $\mathcal{QR}_p$ may be specified by the pair (p, q) ($\mathcal{QR}_p$ is the unique subgroup of $\mathbb{Z}_p^*$ of order q), so we may give these keys as Maple lists, with the public key equal to $[p, q, g, h]$ and the private key equal to $[p, q, g, x]$. Suppose now that we want to encrypt the message:

```
> message := "This is an example of Elgamal encryption/decryption in the group of
          quadratic residues modulo a safe prime"
```

First of all we must be sure that the message can be encrypted in one pass with the public key we have defined; if the message were too long then we would have to split it into blocks as we did, for example, when encrypting with plain RSA. We are going to convert the message to an integer by regarding its characters, as usual, as base-256 digits (through their 8-bit ASCII codes). In order to apply the encoding method explained above, this integer should be $\leq q$ and so the number of characters in the message string should be no greater than the logarithm of q to the base 256. We compute these numbers as follows:

```
> StringTools:-Length(message);
  ilog[256](q);
                                    106
                                    127
```

We see that the message is short enough and can be encrypted in one pass. We start by converting the message to a list of bytes (integers in the 0..255 range) as follows:

```
> messagebytes := convert(message, bytes);
[84, 104, 105, 115, 32, 105, 115, 32, 97, 110, 32, 101, 120, 97, 109, 112, 108, 101,
 32, 111, 102, 32, 69, 108, 103, 97, 109, 97, 108, 32, 101, 110, 99, 114, 121, 112,
 116, 105, 111, 110, 47, 100, 101, 99, 114, 121, 112, 116, 105, 111, 110, 32, 105,
 110, 32, 116, 104, 101, 32, 103, 114, 111, 117, 112, 32, 111, 102, 32, 113, 117, 97,
 100, 114, 97, 116, 105, 99, 32, 114, 101, 115, 105, 100, 117, 101, 115, 32, 109,
 111, 100, 117, 108, 111, 32, 97, 32, 115, 97, 102, 101, 32, 112, 114, 105, 109, 101]
```

Next we encode the message as an element of $\mathcal{QR}_p$. The steps to this are, first, to convert the message to an integer $\leq q$ and then to square this integer in the group $\mathbb{Z}_p^*$ to obtain an element of $\mathcal{QR}_p$. As indicated above, one may also add 1 before squaring but we shall not trouble to do that here, as the only purpose of this addition is to ensure that the integer obtained is different from 0 (and hence an element of $\mathbb{Z}_p^*$), something that is automatically satisfied by any nonempty Maple text string such as the one defining `message`.

```
> messagequadres := ListTools:-DotProduct(messagebytes,
                    [seq(256^i, i=0..nops(messagebytes)-1)])^2 mod p;
254851917004355420497637251682391067724433508334654531840075708947191589915025631\
  2
  98953273271131260701374556359259108912517343063512901734591690458500997214343750\
  1
  15405070564590002344473493385924755023527055380053960943428307605808353472823376\
  2
  24465366303590022190107012424380278959151846940740372553610311046 9
```

To encrypt the message, a random $r \leftarrow \mathbb{Z}_q$ must be used. Again, for simplicity, we use Mersenne Twister to pick this element as follows:

```
> SetState();
  r := GenerateInteger(range = 1 .. q);
40862132368476091259743814261339729299946892853950668244042564740343414177291084 2
  96514360151294401790876000690282841458447250178802879740961608266769967608503438\
  48016210282171116936369242311021439594753986426043520707243406973034217750839313\
  18266821992762475881060994848229560186990279677194480146406198102 2
```

Now, the ciphertext is computed:

```
> c := [Power(g, r) mod p, messagequadres*Power(h, r) mod p];
  [29039676808831746542677305774865287237462406031483920609320346234151332606102 4 8\
  30204752106169432656548726436503547140640509204923383906229190194576934214051981\
  36950097381873221260280334519894458266566335876877126741186170495473397366527709\
  93726052468274095705735854933459037673206405349522537010684531463191 8,
  45521190104426618827384708479377423518005279501895390122908123032649165995005811\
  95354710040543286414295407080353755429142381723164998726646563263640022912949040\
  88860259313033289385688510009832495216081367558026385029222708644345112481226741\
  82875942746792152102041286919853686326004199887303155872280327825391]
```

Let us start the decryption process in which the private key (specifically, the integer x) is used. The following computation should give us back `messagequadres`:

```
> d := (Power(c[1], -x mod q) mod p)*c[2] mod p;
254851917004355420497637251682391067724433508334654531840075708947191589915025631\
  2
  98953273271131260701374556359259108912517343063512901734591690458500997214343750\
  1
  15405070564590002344473493385924755023527055380053960943428307605808353472823376\
  2
  24465366303590022190107012424380278959151846940740372553610311046 9
```

Next we have to compute the unique square root of d in $\mathbb{Z}_p^*$ which is $\leq q$. As mentioned above, we may use Maple's command `numtheory:-msqrt` but, in this case, since q happens to be a Blum prime, it is also very easy to do it as follows:

```
> sqroot := Power(d, (p+1)/4) mod p;
114790821943818019585928129288248749673721993950469127060980676197006517885146350 8\
  83318373932646382503953272750786599072334600130436614624301781722611362912995560\
  86347521040423831204131365460839141500677740683958618340535914688652598036788992\
  22134270929119216572767779776589629182037755764078416275008946520 3
```

This is one of the two square roots of d in $\mathbb{Z}_p^*$ but we need to compute the square root that is $\leq q$, so we check whether this condition holds:

```
> evalb(q > sqroot);
                                    false
```

sqroot is greater than q so this is not the square root we are looking for. Since the two square roots are additive inverses of each other, the one we want must then be $-$sqroot mod p which is the element $p -$ sqroot $\in \mathbb{Z}_p^*$:

```
> sqroot2 := p - sqroot;
74362093667746830865320865674822600984827276877606821103482471512160110018594631 07\
  26553421802330990675448187049196194483799883858511758002953862710913740650945905\
  41939132300745837913126714560203454783147233272547976208797187133914584851982654\
  0108582578260
```

The message can now be recovered by reversing the steps we took before. First we convert this integer to base 256, which will give the list of extended ASCII codes of its characters, and then we apply Maple's convert(-,bytes) to convert the list of codes to a string of characters that will be equal to the original message:

```
> convert(convert(sqroot2, base, 256), bytes);
    "This is an example of Elgamal encryption/decryption in the group of
       quadratic residues modulo a safe prime"
```

Exercise 8.28 Write Maple functions for the Elgamal key generation, encryption, and decryption algorithms, and use them to test encryption and decryption with the data in the preceding example.

8.6 The Cramer–Shoup Encryption Scheme

Although the "plain" Elgamal encryption scheme studied in the previous section has pretty good security properties—it compares very favorably, for example, with plain RSA—it still falls short of the desirable security level for public-key encryption schemes because, as we have seen, it is not IND-CCA secure.

We have already indicated how CCA secure encryption schemes may be built from families of trapdoor permutations but these schemes are not efficient enough for practical use. We have also seen that there are efficient public-key encryption schemes—such as RSA-OAEP or Boneh's Rabin-SAEP—that are CCA secure in the random oracle model, but until 1998 no scheme was known that was both practically efficient and CCA secure without random oracles. Then the celebrated Cramer–Shoup encryption scheme was introduced in [57]. It is a modification of Elgamal encryption that is efficient—though not as much as, say, Rabin-SAEP—and achieves IND-CCA under the sole assumption that the DDH problem is hard, without random oracles. We next describe this scheme.

8.6.1 Cramer–Shoup Encryption and Its Security

The Cramer–Shoup encryption scheme works in groups of prime order q. Thus encryption and decryption will use the operations of a group G such that $|G| = q$; this group may be generated by the key generation algorithm and regarded as a component of the public key or it may be regarded as a system parameter which is

publicly known and shared by all users. Usually, G will be a subgroup of order q of $\mathbb{Z}_p^*$, where p is also prime and $q|p-1$. A good way to select such a group is to take q to be a sufficiently large Sophie Germain prime, so that $p = 2q+1$ is a safe prime and $G = \mathcal{QR}_p$ the subgroup of the quadratic residues modulo p which, as we know, has order q. In what follows we will assume that G is of this form.

Another required ingredient is a collision resistant hash function H which outputs elements of $\mathbb{Z}_q$ (so that we may assume that H outputs bit strings of length $\text{len}(q) = \lceil \log_2(q) \rceil$). Alternatively, a weaker concept may be used, namely a *universal one-way family of hash functions*, meaning that it is infeasible for an adversary to choose an input x, draw a random hash function H from the family, and then find a different input y such that $H(x) = H(y)$.

In the following description we will assume that G is generated by an algorithm $\mathbf{Gen}_G$ on input a security parameter 1^k (where $k = \text{len}(q)$) and G will be fixed as a system parameter so its specification will not be explicitly included in the public and private keys.[7] We will use generic notation for both the group operation (which in case G is as described above is just multiplication modulo p) and the operations in $\mathbb{Z}_q$ involving the exponents that appear below (operations modulo q).

Definition 8.17 The *Cramer–Shoup* public-key encryption scheme (**Gen**, **Enc**, **Dec**) is defined by the following algorithms:

- **Gen**:

1. Choose random elements $g_1, g_2 \leftarrow G$ (if they are $\neq 1$ they will be both generators since G has prime order; observe that the probability of choosing 1 will be negligible but it can be reduced to 0 by explicitly excluding it).
2. Choose, also uniformly at random, elements $x_1, x_2, y_1, y_2, z \leftarrow \mathbb{Z}_q$ (z will play a role similar to that of an Elgamal private key).
3. Compute $c = g_1^{x_1} g_2^{x_2}$, $d = g_1^{y_1} g_2^{y_2}$ and $h = g_1^z$ (h will be used like an Elgamal public key).
4. Output the public key (g_1, g_2, c, d, h, H) and the private key (x_1, x_2, y_1, y_2, z).

- **Enc**: Let $m \in G$ be a message. The following operations are performed:

1. Choose a random element $r \leftarrow \mathbb{Z}_q$.
2. Compute the following values:

$$\begin{aligned} u_1 &:= g_1^r, \\ u_2 &:= g_2^r, \\ e &:= h^r m, \\ \alpha &:= H(u_1 || u_2 || e), \\ v &:= c^r d^{r\alpha}. \end{aligned}$$

3. The ciphertext corresponding to the message m is then the 4-tuple (u_1, u_2, e, v).

[7] If the group G is generated by each user by calling $\mathbf{Gen}_G$ from the key generation algorithm, then such a specification—for example, the parameters p, q which describe the group in case $G = \mathcal{QR}_p$ and $p = 2q+1$—should be included in both the public and the private key.

- **Dec**:

1. On input the private key (x_1, x_2, y_1, y_2, z) and the ciphertext (u_1, u_2, e, v), compute:
$$\alpha := H(u_1||u_2||e).$$
2. Test whether
$$u_1^{x_1+\alpha y_1} u_2^{x_2+\alpha y_2} = v.$$
3. If the preceding condition does not hold then reject the ciphertext as invalid, otherwise output the plaintext eu_1^{-z}.

Let us now show that the scheme is correct in the sense that encryption followed by decryption recovers the original message. Suppose that (u_1, u_2, e, v) is the ciphertext corresponding to the message m, with $u_1 = g_1^r$, $u_2 = g_2^r$, $e = h^r m$ and $v = c^r d^{r\alpha}$. Then the ciphertext will pass the correctness test above because:

$$v = c^r d^{r\alpha} = (g_1^{x_1} g_2^{x_2})^r (g_1^{y_1} g_2^{y_2})^{r\alpha} = g_1^{r(x_1+y_1\alpha)} g_2^{r(x_2+y_2\alpha)} = u_1^{x_1+\alpha y_1} u_2^{x_2+\alpha y_2}.$$

Once the test is passed, the message that the decryption algorithm outputs is:

$$eu_1^{-z} = mh^r u_1^{-z} = mg_1^{rz} g_1^{-rz} = m,$$

so that the correct message is recovered.

The main result about the security of the Cramer–Shoup encryption scheme is then the following (see [57, Theorem 1] and [59, Theorem 1]):

Theorem 8.5 *If the DDH problem is hard in the group G and H is a collision resistant hash function then the Cramer–Shoup scheme is IND-CCA secure.*

Proof The proof is a reduction that shows that if there is a successful adversary against the scheme, then a successful attacker against the DDH problem in G can be constructed. We present its essential aspects following [57] and refer to [59] for additional variants of the scheme and for a more detailed proof.

Suppose that $\mathcal{A}$ is a PPT adversary that succeeds in the indistinguishability experiment $\mathbf{PubK}_{\mathcal{A},\mathcal{E}}^{ind\text{-}cca}(k)$ with non-negligible advantage, where $\mathcal{E}$ is now the Cramer–Shoup scheme and $k = \text{len}(q)$ (we will assume throughout that the function H used by the scheme is collision resistant). We construct an algorithm $\mathcal{B}$ that, using $\mathcal{A}$ as a subroutine, tries to solve the DDH problem. On input a triple (g^x, g^y, g^a), where x and y are random elements of $\mathbb{Z}_q$, and a is either random or $a = xy$, $\mathcal{B}$ will try to distinguish which of these two alternatives holds. For this $\mathcal{B}$ will generate a Cramer-Shoup public key that will be given to $\mathcal{A}$ and $\mathcal{B}$ will simulate a Cramer-Shoup decryption oracle responding to $\mathcal{A}$'s queries. $\mathcal{B}$ chooses random elements $x_1,\ x_2,\ y_1,\ y_2,\ z_1,\ z_2 \leftarrow \mathbb{Z}_q$ and, after setting $g_1 := g$, $g_2 := g^x$, computes:

$$c = g_1^{x_1} g_2^{x_2},\ d = g_1^{y_1} g_2^{y_2},\ h = g_1^{z_1} g_2^{z_2}.$$

Then the following public key is given to $\mathcal{A}$:

$$(g_1, g_2, c, d, h, H),$$

where H is the collision resistant hash function being used by the scheme. Now $\mathcal{B}$ runs the first stage of $\mathcal{A}$ in the CCA indistinguishability experiment. In this stage $\mathcal{A}$ queries for decryptions and $\mathcal{B}$ acts as a simulator of the decryption oracle: upon receiving (v_1, v_2, f, w) it computes $\beta := H(v_1||v_2||f)$ and verifies that $v_1^{x_1+\beta y_1} v_2^{x_2+\beta y_2} = w$. If this check fails $\mathcal{B}$ outputs $\perp$ (meaning "invalid ciphertext"), otherwise $\mathcal{B}$ outputs the message $f v_1^{-z_1} v_2^{-z_2}$.

Observe that the value of h generated by $\mathcal{B}$ above for the public key is computed in a different way than in the Cramer–Shoup key generation algorithm, but the adversary $\mathcal{A}$ will not be able to detect this change. Since $g_2 = g_1^x$, we have that $h = g_1^{z_1} g_2^{z_2} = g^z$, where $z = z_1 + x z_2$, and we can show that in the simulation above $\mathcal{B}$ acts the same as a legitimate decryption oracle that uses this value of z (even if $\mathcal{B}$ does not know z as it does not know x). Indeed, suppose first that the ciphertext (v_1, v_2, f, w) about which $\mathcal{A}$ queries $\mathcal{B}$ is the correct encryption of a message. In this case the ciphertext satisfies the verification condition in the Cramer–Shoup decryption algorithm and, moreover, $v_1 = g_1^r$ and $v_2 = g_2^r$ for the same $r \in \mathbb{Z}_q$ or, in other words, $\log_{g_1} v_1 = \log_{g_2} v_2$. Then this ciphertext passes $\mathcal{B}$'s check and, bearing in mind that $v_1 = g_1^r = g^r$ and $v_2 = g_2^r = g^{xr}$, the message output by $\mathcal{B}$ in the simulation is $f v_1^{-z_1} v_2^{-z_2} = f(g^r)^{-z_1}(g^{xr})^{-z_2} = f(g^r)^{-z_1 - x z_2} = f v_1^{-z}$, which is exactly the same output that would be produced by a decryption oracle.

Next, suppose that the ciphertext (v_1, v_2, f, w) submitted by $\mathcal{A}$ is invalid in the sense that $\log_{g_1} v_1 \neq \log_{g_2} v_2$. If the ciphertext does not pass the verification check then it will be rejected by $\mathcal{B}$ exactly the same as it would be by a Cramer–Shoup decryption oracle. But what about the possibility that this verification check is passed? If the simulation deviates from the decryption oracle behavior at this point, this might alter the success probability of $\mathcal{A}$ and invalidate the proof. However, we next show that this does not happen because in the first stage of $\mathcal{A}$'s attack (before the challenge ciphertext is given to $\mathcal{A}$) both $\mathcal{B}$ and the legitimate decryption oracle will reject any invalid ciphertext except with negligible probability.

The verification test is to check that $w = v_1^{x_1+\beta y_1} v_2^{x_2+\beta y_2}$ (where $\beta = H(v_1||v_2||f)$) which, taking logarithms, can be written as $\log_g w = (\log_g v_1)x_1 + (\log_g v_2)x_2 + (\log_g v_1)\beta y_1 + (\log_g v_2)\beta y_2$ or, in a more compact form, letting $v_1 = g^r$, $v_2 = g_2^{r'} = g^{xr'}$:

$$\log_g w = r x_1 + x r' x_2 + \beta r y_1 + \beta x r' y_2. \tag{8.3}$$

This can be seen as a linear equation in the variables x_1, x_2, y_1, y_2 with coefficients in $\mathbb{Z}_q$. Thus it defines a linear variety of dimension $4 - 1 = 3$ in the 4-dimensional $\mathbb{Z}_q$-vector space $\mathbb{Z}_q^4$ (the linear system has rank 1 and defines a 'hyperplane' of $\mathbb{Z}_q^4$). Now we must also take into account that $\mathcal{A}$ may have some information about the private key that it can use to guess a value of w satisfying Eq. 8.3 when submitting its decryption queries. In particular, $\mathcal{A}$ knows the public key from which, taking

logarithms in the equations $c = g_1^{x_1} g_2^{x_2}$, $d = g_1^{y_1} g_2^{y_2}$, the following information is obtained:

$$\left.\begin{aligned} \log_g c &= x_1 + x x_2 \\ \log_g d &= y_1 + x y_2 \end{aligned}\right\} \tag{8.4}$$

It is not difficult to see that this is all the information available to $\mathcal{A}$ on the values x_1, x_2, y_1, y_2, even assuming that $\mathcal{A}$ can compute discrete logarithms, which is hard in general. Thus the point (x_1, x_2, y_1, y_2) looks to $\mathcal{A}$ like a random point in the linear variety of $\mathbb{Z}_q^4$ defined by Eqs. 8.4 which, since the linear system has rank 2, is a plane $\mathcal{P}$ containing q^2 points. If $\mathcal{A}$ submits a ciphertext (v_1, v_2, f, w) such that $r = \log_{g_1} v_1 \neq \log_{g_2} v_2 = r'$ but Eq. 8.3 holds (and hence the ciphertext is not rejected), then the point (x_1, x_2, y_1, y_2) lies in the linear variety defined by the three Eqs. 8.3, 8.4. These equations define a linear system of rank 3 (it is easy to see that, except when $x = 0$ which occurs with negligible probability, the three rows of the coefficient matrix are linearly independent as vectors of $\mathbb{Z}_q^4$) and hence define a line in $\mathcal{P}$ to which (x_1, x_2, y_1, y_2) must belong. Since the line has q points, the probability that the first invalid ciphertext submitted by $\mathcal{A}$ is not rejected is $q/q^2 = 1/q$, which is negligible as a function of $k = \text{len}(q)$. After each rejection, $\mathcal{A}$ can exclude one line in $\mathcal{P}$ and so the probability that the ith invalid ciphertext submitted is not rejected is at most $1/(q - i + 1)$. Since $\mathcal{A}$ makes a polynomial number of queries, all invalid ciphertexts are rejected except with negligible probability and in this phase $\mathcal{A}$'s attack in the simulation carried out by $\mathcal{B}$ proceeds in the same way as if $\mathcal{A}$ were interacting with a legitimate decryption oracle.

After submitting its queries, at the end of this stage, $\mathcal{A}$ outputs two messages $m_0, m_1 \in G$. Next $\mathcal{B}$ chooses a random bit $b \leftarrow \{0, 1\}$ and computes the challenge ciphertext. For this, $\mathcal{B}$ uses the DDH input 3-tuple (g^x, g^y, g^a) and sets:

$$\begin{aligned} u_1 &:= g^y, \\ u_2 &:= g^a, \\ e &:= u_1^{z_1} u_2^{z_2} m_b, \\ \alpha &:= H(u_1 || u_2 || e), \\ v &:= u_1^{x_1 + \alpha y_1} u_2^{x_2 + \alpha y_2}. \end{aligned}$$

Next, $\mathcal{B}$ gives to $\mathcal{A}$ the challenge ciphertext:

$$(u_1, u_2, e, v)$$

and $\mathcal{A}$ may continue querying for decryptions of ciphertexts distinct from this one, which are answered by $\mathcal{B}$ in the form indicated above. Finally, $\mathcal{A}$ outputs a bit b' which is its guess on the value of b.

Now, algorithm $\mathcal{B}$ outputs $b \oplus b'$ which, recalling Definition 7.4 means that $\mathcal{B}$ guesses that $a = xy$ if and only $b = b'$, i.e., if and only if $\mathcal{A}$ succeeds in the CCA indistinguishability experiment. Therefore, $\mathcal{B}$ guesses that a is random in case $b \neq b'$.

The intuition behind this choice is based on the following facts:

(i) If $a = xy$ then the challenge ciphertext given to $\mathcal{A}$ in the oracle simulation is a legitimate encryption of m_b. Our hypothesis that $\mathcal{A}$ succeeds in the CCA indistinguishability experiment with non-negligible probability then implies that the probability that $b = b'$ is non-negligibly better than $1/2$ and hence so is the probability that $\mathcal{B}$ outputs 0, in which case $\mathcal{B}$ succeeds.

(ii) If a is random (and hence $a \neq xy$ except with negligible probability), then $\mathcal{A}$'s view in the CCA indistinguishability experiment is essentially independent from b and hence so is $\mathcal{A}$'s output b'. Therefore, in this case, $b = b'$ with probability negligibly different from $1/2$ and $\mathcal{B}$ succeeds in the DDH experiment with this same probability.

(i) is easy to prove as follows. If (u_1, u_2, e, v) is the challenge ciphertext, then, by construction, we have that $u_1 = g^y = g_1^y$, $u_2 = g^a = g_1^{xy} = g_2^y$. Moreover, setting $z = z_1 + xz_2$ we have that $e = u_1^{z_1} u_2^{z_2} m_b = (g_1^y)^{z_1} (g_2^y)^{z_2} m_b = g_1^{yz} m_b$. We also have that $\alpha = H(u_1||u_2||e)$ and, finally, that $v = u_1^{x_1+\alpha y_1} u_2^{x_2+\alpha y_2} = (g_1^y)^{x_1} (g_1^y)^{\alpha y_1} (g_2^y)^{x_2} (g_2^y)^{\alpha y_2} = (g_1^{x_1} g_2^{x_2})^y (g_1^{y_1} g_2^{y_2})^{y\alpha} = c^y d^{y\alpha}$. Thus we see that the challenge ciphertext is an encryption of m_b with $r = y$.

Next, in order to prove (ii) let $u_1 = g^y$, $u_2 = g^a = g^{xy'}$, where $y' \neq y$ except with negligible probability. Then (ii) is a consequence of the following claims:

Claim (1) If $\mathcal{B}$, simulating a decryption oracle, rejects all invalid ciphertexts, then the bit b is independent of $\mathcal{A}$'s view (which includes the challenge ciphertext).

Claim (2) $\mathcal{B}$ rejects all invalid ciphertexts submitted by $\mathcal{A}$ except with negligible probability.

In order to prove Claim (1), we show that if (u_1, u_2, e, v) is the challenge ciphertext given to $\mathcal{A}$ by $\mathcal{B}$, then $u_1^{z_1} u_2^{z_2}$ is uniformly distributed in G in $\mathcal{A}$'s view and hence $e = u_1^{z_1} u_2^{z_2} m_b$ looks independent from m_b (and hence from b) to $\mathcal{A}$. To see this consider the point $(z_1, z_2) \in \mathbb{Z}_q^2$. At the beginning of the attack $\mathcal{A}$ knows from the public key that $h = g_1^{z_1} g_2^{z_2}$, which, taking logarithms, gives the linear equation in the variables z_1, z_2 with coefficients in $\mathbb{Z}_q$:

$$\log_g h = z_1 + xz_2. \tag{8.5}$$

At this point, (z_1, z_2) looks to $\mathcal{A}$ like a random point in the line defined by this equation. Next, $\mathcal{A}$ may submit decryption queries that $\mathcal{B}$ will answer but, as we are assuming that $\mathcal{B}$ only answers when the submitted ciphertext (v_1, v_2, f, w) is valid, we see that then $h^{r'} = v_1^{z_1} v_2^{z_2} = g_1^{r'z_1} g_2^{r'z_2}$ for some $r' \in \mathbb{Z}_q$ and this gives an equation:

$$r' \log_g h = r'z_1 + r'xz_2, \tag{8.6}$$

which is a scalar multiple of Eq. 8.5 and hence provides no additional information to $\mathcal{A}$ about (z_1, z_2). Consider now the challenge ciphertext (u_1, u_2, e, v), where $e = \varepsilon m_b$ with $\varepsilon = u_1^{z_1} u_2^{z_2}$. This gives the equation:

$$\log_g \varepsilon = yz_1 + y'xz_2. \tag{8.7}$$

This is, again, a linear equation in the variables z_1, z_2 over $\mathbb{Z}_q$ and, except in the case $y' = y$ (which occurs with negligible probability), the equation is linearly independent from Eq. 8.5. Therefore, each element of G occurs as the value of ε for exactly one choice of the pair (z_1, z_2). This implies that from $\mathcal{A}$'s point of view, ε is uniformly distributed in G and, as in the proof of perfect security of the one-time pad based on Shannon's theorem (Theorem 3.1), $\mathcal{A}$ obtains no information about b.

Let us now show that Claim (2) is also true. As in the previous analysis of the rejection of invalid ciphertexts in the attack phase preceding the generation of the challenge ciphertext, we study the distribution of the point $P = (x_1, x_2, y_1, y_2) \in \mathbb{Z}_q^4$ conditioned on $\mathcal{A}$'s view. Now the situation is different because, in addition to Eqs. 8.4, $\mathcal{A}$ has additional information coming from the challenge ciphertext (u_1, u_2, e, v). This information comes from the fact that now $\mathcal{A}$ knows that $v = u_1^{x_1+\alpha y_1} u_2^{x_2+\alpha y_2}$, where $u_1 = g^y$, $u_2 = g^a = g^{xy'}$, with $y' \neq y$ except with negligible probability. Thus P must also lie in the hyperplane of $\mathbb{Z}_q^4$ defined by the linear equation:

$$\log_g v = yx_1 + xy'x_2 + \alpha y y_1 + \alpha x y' y_2. \tag{8.8}$$

Therefore, once the challenge ciphertext is received, P looks to $\mathcal{A}$ like a random point in the intersection of the plane defined by Eqs. 8.4 with the hyperplane defined by Eq. 8.8. The linear system formed by these three equations is easily seen to have rank 3, so it defines a line in which P must lie.

Now, in order to prove Claim (2), suppose that $\mathcal{A}$ submits to $\mathcal{B}$ an invalid ciphertext (v_1, v_2, f, w), where $\log_g v_1 = r$ and $\log_g v_2 = xr'$, with $r \neq r'$, and let $\beta = H(v_1||v_2||f)$. Note that, because of the rules of the CCA indistinguishability experiment, $(v_1, v_2, f, w) \neq (u_1, u_2, e, v)$. There are three possibilities which we consider separately:

1. $(v_1, v_2, f) = (u_1, u_2, e)$. In this case we know that $v \neq w$ but $\alpha = H(u_1||u_2||e) = H(v_1||v_2||f) = \beta$. Then, the verification test gives:

$$v_1^{x_1+\beta y_1} v_2^{x_2+\beta y_2} = u_1^{x_1+\alpha y_1} u_2^{x_2+\alpha y_2} = v \neq w,$$

   so that the ciphertext is rejected by $\mathcal{B}$.
2. $(v_1, v_2, f) \neq (u_1, u_2, e)$ and $\beta \neq \alpha$. The ciphertext will be rejected unless P lies in the hyperplane defined by Eq. 8.3 which expresses the verification condition. The intersection of this hyperplane with the line defined by Eqs. 8.4 and 8.8 is a linear variety whose dimension is $4 - rk$, where rk is the rank of the coefficient matrix of the linear system formed by these four equations, namely the matrix:

$$\begin{pmatrix} 1 & x & 0 & 0 \\ 0 & 0 & 1 & x \\ y & xy' & \alpha y & \alpha x y' \\ r & xr' & \beta r & \beta x r' \end{pmatrix}$$

To compute the rank in this case it suffices to compute the determinant d which turns out to be $\neq 0$ (and hence $rk = 4$) except with negligible probability. Indeed, by using elementary operations on columns and Laplace expansion we see that:

$$\begin{aligned} d &= \begin{vmatrix} 0 & 1 & x \\ x(y'-y) & \alpha y & \alpha x y' \\ x(r'-r) & \beta r & \beta x r' \end{vmatrix} \\ &= -\begin{vmatrix} x(y'-y) & \alpha x(y'-y) \\ x(r'-r) & \beta x(r'-r) \end{vmatrix} = x^2(\alpha-\beta)(y'-y)(r'-r) \end{aligned}$$

Now, we know that, except with negligible probability, $x \neq 0$ and $y' \neq y$, and we have also assumed that $r' \neq r$, and $\alpha \neq \beta$, so that we may conclude that $d \neq 0$ (as usual, except with negligible probability). This means that the linear variety defined by the matrix is a point Q. Therefore, the probability that an invalid ciphertext submitted by $\mathcal{A}$ is not rejected by $\mathcal{B}$ is equal to the probability that a randomly chosen point P in the line defined by Eqs. 8.4 and 8.8 is precisely Q and so is equal to $1/q$ and hence negligible.

3. $(v_1, v_2, f) \neq (u_1, u_2, e)$ and $\beta = \alpha$. Since we are assuming that H is collision resistant, the probability of this happening is, again, negligible.

In conclusion, since we have proved facts (i) and (ii) above, we know that there exists a non-negligible function $\varepsilon(k)$ such that

$$\Pr(\mathcal{B} \text{ succeeds} | a = xy) = \frac{1}{2} + \varepsilon(k)$$

and a negligible function $\texttt{negl(k)}$ such that

$$\Pr(\mathcal{B} \text{ succeeds} | a \neq xy) = \frac{1}{2} \pm \texttt{negl(k)}.$$

Since in the DDH experiment a random bit is chosen that determines whether the triple is a DDH triple (i.e., $a = xy$ in the above notation) or a random triple, both events have probability $1/2$ and so, by Proposition 2.1 we have that

$$\begin{aligned} \Pr(\mathcal{B} \text{ succeeds}) &= \frac{1}{2} \cdot \Pr(\mathcal{B} \text{ succeeds} | a = xy) + \frac{1}{2} \cdot \Pr(\mathcal{B} \text{ succeeds} | a \neq xy) \\ &= \frac{1}{2} \cdot (\frac{1}{2} + \varepsilon(k)) + \frac{1}{2} \cdot (\frac{1}{2} \pm \texttt{negl(k)}) \\ &= \frac{1}{2} + \frac{1}{2} \cdot (\varepsilon(k) \pm \texttt{negl(k)}). \end{aligned}$$

Since $\varepsilon(k)$ is non-negligible we see that so is the function $\varepsilon(k) \pm \texttt{negl(k)}$ and hence $\mathcal{B}$ succeeds with non-negligible advantage. This ends the proof. □

8.6.2 A Variant of the Cramer–Shoup Encryption Scheme in Maple

We are going to give a simple Maple implementation of a variant of Cramer–Shoup using the algorithm of [57] (algorithm CS1a in [59]) in the setting described in [59, Example 1, Example 2]. G will be a cyclic group of order q, where q is a Sophie Germain prime, which means that $p = 2q+1$ is (a safe) prime and $G = \mathcal{QR}_p \subseteq \mathbb{Z}_p^*$. Thus G is the unique subgroup of order q of $\mathbb{Z}_p^*$ and its elements are precisely the quadratic residues in $\mathbb{Z}_p^*$.

In the Cramer–Shoup scheme there are several elements that must be randomly chosen but we cannot use Maple for this purpose. These elements could be externally generated and supplied to the functions but, for convenience, we will give a variant that uses smaller random parameters that serve to pseudo-randomly generate the required elements by means of Blum–Blum–Shub. The preceding proof will not apply to this variant since the elements obtained this way will not be really random (the PRG merely "stretches" the randomness already present in the seed) and, in particular, if short seeds are used the scheme will be vulnerable to brute-force attacks. But this approach should be secure in practice if sufficiently long random seeds are used.

The implementation will make use of a series of functions previously used for handling input/output, format conversions and the like, including several functions defined in Appendix A as well as others defined and used in the implementation of RSAES-OAEP in 8.3.7. This includes the SHA-256 based function `MGF1` which implements the "mask generating function" defined in PKCS #1 v2.1 and was also used in our previous implementation of Rabin-SAEP$^+$. We will use `MGF1` as our (hopefully) collision resistant hash function.

We start with the key generation algorithm, which will be split into two functions although it can be run by executing only the second of these functions. The reason is that we will consider the group where encryption and decryption takes place, together with the two generators that, as we have seen in the previous description of the scheme, are used in the public key, as system-wide parameters which can be shared by all users. Thus we will separate the generation of the group (and the group generators) from the generation of the rest of the key, allowing the possibility of generating public and private keys for a previously given group.

The groups we will use in this implementation are entirely determined by their order, namely the Sophie Germain prime q, because once q is given we know that $G = \mathcal{QR}_p$ is the group of quadratic residues in $\mathbb{Z}_p^*$, with $p = 2q + 1$ and, as we have seen in Sect. 2.9, it is easy to determine which elements of $\mathbb{Z}_p^*$ belong to $\mathcal{QR}_p$ by computing the Legendre symbol. Since G has prime order, every element $\neq 1$ is a generator of G by Corollary 2.1. Moreover, the same efficient algorithms used for the group $\mathbb{Z}_p^*$ serve to compute the group operation and to compute inverses in G.

The next function generates the group together with two pseudo-randomly chosen generators. Since all the generated parameters will be publicly known, we do not worry about the lack of unpredictability and use Mersenne Twister with

automatic seeding to generate the safe prime. We use Maple's built-in function `numtheory:-safeprime` which, as we have observed, generates safe primes in a non-uniform way since the gaps between such primes may have large variations in size (in fact, it has not even been proved that the set of these primes is infinite although there are heuristic arguments that suggest this is the case). Again, this is not a problem because these parameters are going to be public. The function takes as input a positive integer that specifies the number of bytes of the safe prime (and also of the Sophie Germain prime) to be generated. Actually, on input k, the function will probably generate an $8k$-bit safe prime p and hence an $(8k-1)$-bit Sophie Germain prime q, as well as two generators of $\mathcal{QR}_p$. The output is a list of length 3, where the first element is the Sophie Germain prime q and the two remaining elements are the generators of the group, which are given as integers in the interval $[2, p-2]$, i.e., as elements of $\mathbb{Z}_p^*$.

```
> GroupGen := proc(k::posint)
  local bitlength, p, q, g1, g2;
  uses RandomTools:-MersenneTwister;
  SetState();
  bitlength := 8*k;
  p := numtheory:-safeprime(GenerateInteger(range=2^(bitlength-1)..2^bitlength-1));
  q := (p-1)/2;
  g1 := GenerateInteger(range = 2 .. q)^2 mod p;
  g2 := GenerateInteger(range = 2 .. q)^2 mod p;
  [q, g1, g2]
  end proc:
```

The function `CSKeyGen` below completes the implementation of the Cramer–Shoup key generation algorithm. There is only one required input parameter, `seed`, by means of which a random seed of sufficient length (say, at least 128 bits) is specified, either as a positive integer or as a hexadecimal string that will be converted to decimal form inside the function. The supplied value serves to seed the Blum–Blum–Shub PRG that will take care of generating the integers in the interval $[0, q-1]$. These integers are obtained by generating bit strings of length equal to the bit length of q and discarding them when the integer they define is $\geq q$.

The remaining input parameters are optional keyword parameters. The first of them, `k`, serves to specify the byte length of the Sophie Germain prime to be generated, in the way indicated for the preceding function. Its default value is 128 and the only effect of this parameter is to feed the next parameter, `G`, if no value is passed to it. `G` is used to specify a group of the form described above (in the format produced by the function `GroupGen`) and, if not specified, a default value for it will be generated by invoking `GroupGen` with the value of `k` specified in the previous parameter. The next parameter `bbslength` is used to specify the bit length of the primes used by Maple's Blum–Blum–Shub PRG among the three possible ones—with default value 1024—as we have already done on other similar occasions. The final keyword parameter `format` is used to specify the format in which the key will be returned and it can be either `hex` (the default) for the key components to be given as hexadecimal strings or `decimal`, in which case the key parameters are given as decimal integers.

The output is a list of length two whose elements are also lists. The first of them contains the public key in the form $[q, g1, g2, c, d, h]$, where q is the Sophie Germain prime that defines the group of order q, g_1 and g_2 are generators of the group and c, d, h are the elements mentioned—with this same notation—in the description of the key generation algorithm. Note that we do not include the hash function H in the public key because it will be regarded as a system parameter but we include q, which is also a system parameter, for convenience. The second element is the private key in the form $[q, x_1, x_2, y_1, y_2, z]$ where q is, again, the Sophie Germain prime defining the group and the remaining elements are pseudo-randomly chosen elements of $\mathbb{Z}_q$ whose role was explained in the description of the scheme.

```
> CSKeyGen:=proc(seed::{posint, string}, {k::posint:=128, G::list:=GroupGen(k),
                bbslength::{512, 768, 1024}:=1024, format::name:=hex})
  local q, g1, g2, p, qbitLen, s, B, sk, j, x, pk, c, d, h;
  q := G[1];
  g1 := G[2];
  g2 := G[3];
  p := 2*q+1;
  qbitLen := intlog[2](q)+1;
  s := stringposint(seed);
  B := RandomTools:-BlumBlumShub:-NewBitGenerator(s, primes = bbslength);
  sk := [q];
  for j to 5 do
    x := q;
    while q <= x do
      x := convert(cat(seq(B(), i = 1 .. qbitLen)), decimal, binary)
    end do;
    sk := [op(sk), x]
  end do;
  c := (Power(g1, sk[2]) mod p)*(Power(g2, sk[3]) mod p) mod p;
  d := (Power(g1, sk[4]) mod p)*(Power(g2, sk[5]) mod p) mod p;
  h := Power(g1, sk[6]) mod p;
  pk := [q, g1, g2, c, d, h];
  if format = decimal then
    [pk, sk]
  elif format = hex then
    StringTools:-LowerCase~~(convert~~([pk, sk], hex))
  else
    error "Unrecognized key format"
  end if
  end proc:
```

Example 8.17 Let us generate a Cramer–Shoup key using a 16-byte (128-bit) seed. We can do it in one step by executing, for example, a command such as `CSKeyGen(seed);` where only the seed is specified. But we can also do it in two steps by first generating the system-wide parameters q, g_1, g_2 (i.e., the prime q that specifies the group and the two generators g_1, g_2) and then calling `CSKeyGen` with an appropriate seed. Using the latter method we start by calling `GroupGen` with `k = 128`, which will likely generate a 1023-bit Sophie Germain prime q (and hence a 1024-bit safe prime); q is the first element in the output list and the remaining two elements are the generators:

```
> G := GroupGen(128);

  [71439603705278284495261866885176333759996862173770092964238687088017598748177 10\
  44516230251678357194577874211745344114799304637844601838348544484390830624759486\
  95932360634468813637697596737239838069690519627994216763581439493288047386036509\
  31299731818349244663579461851099957829807155067963272270942249863571 9,
  56600621692894720322996831833949279484205491510729070417562352113259285392750814\
  45241064072872020057929853316293678394394369121243919939352924928342168393609742\
  60392124327867520493214776038256728212989970935152898882722244842081959733688072\
  96848016347016770933697531239418456740770508272810070053458729355158,
  12834761662158188686247780916551983750441253380555481842123880828402799347346786\
  48896869300523477729603613823050704935679367796973569651545691929351791926092029\
  19328497565609358744443270379161709125820576315696572501643846814011418047708796\
  28395192084713268541485990408094255166700437921543277630971105695820]
```

Now, to generate the public key/private key pair, we invoke `CSKeyGen`. If we only use as input a random seed, then the group and its generators will be freshly generated by the function but, since we want to use the previous group `G`, we will pass it as one of the inputs of the function. We use default values for the remaining input parameters so that, in particular, the key components will be given as hexadecimal strings.

```
> key := CSKeyGen("33303080cef2f5c15da1c00c87758683", 'G' = G);
  pk := key[1]:
  sk := key[2]:
```

Note that the value of the keys is entirely determined by the seed string and the group `G`. We print only the public key; observe also that the first three elements in it are the same as those in the list `G`, except that now they are given as hexadecimal strings:

```
> pk;
  ["65bbbe4e4baff6e59d553e248230d2da1536308b2f46817b86610c8f97bdd10d0cf0a81301e3f1\
  c05445fa13260175c9f5c4272639e99ac6ed92ee4be1fd282daec872bbb2e29f9dec37bbb52cfa55\
  3cf17a0afed9b8f8425110eaae8d2960ec116aff17733a98473352d4a988511dd173d910b3ced6f0\
  30274e90d97b1bdfc7", "509a1a19df591c04e10073d2ca3e65be4a1ee85eb78828ec08d4e1593e\
  91113b0b8210a7802e302ff900beb4f5e84fb44c6a92493b688e2064d5b4b8f8329a68daa48b015b\
  2d78a375c3ab5aaa4ea29a08efc46031ee8995e2aeabcca761a1b439d1ff75420ccc9491f7864f6d\
  4191384631ddca91221b69cf31df3a5b41c796", "b6c5e81c88209d1805475d8bf0169ecb3dae3b\
  ab01316bb84d337f3f961817016621b24096a0aeeb3a600da9efe8dd8f45ef54b0a7668c5bf5d2f9\
  a8378985e8bc12c60ca87dcc03b9f7715f9fd11154c5cd48f4e4a7606c92f0a7a55ea95015bad63f\
  499becc3948490840fd62692bc984759065ae1ca22360e8113b338d84c", "a0824a1a266a8ff26e\
  9d65d30f54c9414b0750330ff764f4a599c4cdf713100a1ba10ad782a32bddc755813f0d24e00ee1\
  a682a3f0ab9abfcc6d347c97db7357c78ba8fbe99b06fa7595d8560e9dd5106ed34252f927a07967\
  e664404e536661669db5a16d9b0ecdf2882d7964d9e3d3b87eb80212956804e6ab0433d3724ef74",
  "4f523e323ac311f60f69589f0544a06d8e6a14c3f4762ffac4651c98594d84b0b4393d70c09ae3a\
  d447f3c599b47500367f160e7e06ae42c4e6d93571bddc567f726bddd9982266037e51ddcb6ae956\
  5da36c2f19875ba557e2e7456b6ecee87650e280cd72817fcc4beaeb3c5749646f75e52a771ac87d\
  4c45542ca40cc6acb", "9ca603dfc4ac264674a10af8678e07616607eda45bf462ec5372f4aca5e\
  ca841b328c99a46367e98def6ff2eb60d22bb11cff4d540cb520ea4bd06ab92c416b0c9327b26bdf\
  9316c0728a6f3c85feadf8f93fdffdfc517dc75c0fb2a7b42e0084b4fffcdc38cba3dc5156a3b944\
  6f0dbde3430454e343b279c5199abcf22814d"]
```

Exercise 8.29 Write a Maple function that tests whether a Cramer–Shoup key in the format output by `CSKeyGen` is valid. The function should accept as input either a public key pk or a public key/private key pair (given as a list $[pk, sk]$). Apart from checking that keys are given in the expected format the function should check, in case a public key is tested, that the first element in the list is a Sophie Germain prime q and that the remaining elements are all quadratic residues in $\mathbb{Z}_p^*$, where $p = 2q + 1$. If the private key is also given then the function should check that the public and private

keys match according to the formulas given in the description of the key generation algorithm.

We next implement the Cramer–Shoup encryption algorithm. Before doing so we remark that the groups chosen have the nice property that they allow the easy encoding of arbitrary bit strings of length $\text{len}(q)-1$ as elements of $G = \mathcal{QR}_p$ (where $p = 2q + 1$ as usual) as indicated in [59, Example 2]. Such a string can be uniquely identified with an integer x in the interval $[1, q]$ and then x can be mapped to $x \in G$ in case $\left(\frac{x}{p}\right) = 1$ and to $p - x \in G$ in case $\left(\frac{x}{p}\right) = -1$ (note that the safe prime p is a Blum prime and hence $\left(\frac{-1}{p}\right) = -1$, so that $p - x \equiv -x \pmod{p}$ is a quadratic residue in this case). This mapping is injective and allows for easy decoding. The Maple function that performs this encoding is given next. The function has two required input parameters. The first one, `message`, is used to specify the message to be encoded, which should be given as a hexadecimal string. The second parameter is for the Sophie Germain prime q. The function checks that the byte length of the message is less than that of q and, if this is the case, it concatenates a leading '01' byte to the message string which serves to easily recover it when decoding, even if it has leading zeros. Note that the resulting string has the same byte length as q but the integer it represents is always $\leq q$ because its seven leading bits are '0' bits; in fact, it is $< q$ if q was generated by `GroupGen` above because in that case q is an $(8k-1)$-bit prime while the string is encoded by an $(8k-7)$-bit integer. The output of the function is an element of G given as an element of $\mathbb{Z}_p^*$.

```
> QREnc := proc(message::string, q::posint)
  local mLen, qLen, p, M, md;
  mLen := iquo(StringTools:-Length(message), 2);
  qLen := intlog[256](q)+1;
  if qLen <= mLen then
    error "message too long"
  end if;
  p := 2*q+1;
  M := cat("01", message);
  md := OS2IP(M);
  if numtheory:-legendre(md, p) = 1 then
    md
  else
    p - md
  end if
  end proc:
```

We next give the decoding function `QRDec`, which reverses the encoding done by `QREnc`. On input a quadratic residue in $\mathbb{Z}_p^*$ (where $p = 2q + 1$ is a safe prime) and the Sophie Germain prime q, the function recovers the encoded string.

```
> QRDec := proc(qr::posint, q::posint)
  local p, x, s;
  p := 2*q+1;
  x := qr;
  if numtheory:-legendre(x, p) <> 1 then
    error "%1 is not a quadratic residue modulo %2", x, p
  end if;
```

```
  if q < x then
    x := p-x
  end if;
  s := I2OSP(x);
  if StringTools:-Take(s, 2) <> "01" then
    error "incorrect encoding"
  else
    StringTools:-Drop(s, 2)
  end if
end proc:
```

The encryption function is given next. The required input parameters are `publickey`, `message` and `seed`, where the first one is used to specify a public key in the format produced by `CSKeyGen`, the second to specify the message, given either as a hexadecimal string or an ordinary text string, and the last to specify a random seed, either as a string or as a positive integer.

The optional keyword parameters are `messagetype`, used to tell the function whether the message is a hexadecimal or an ordinary text string, and `bbslength` which selects the length of the primes used by Maple's Blum–Blum–Shub PRG. In addition to these, one could also set up additional parameters `H` and `hLen` (exactly as in `MGF1`) which would allow specification of the hash function on which `MGF1` is based and its output length but we are not including them as we will only use SHA-256—which is the default—for this purpose. The output is the ciphertext in hexadecimal format. A Cramer–Shoup ciphertext is a 4-tuple (u_1, u_2, e, v) of elements of G and we could give it as a Maple list containing the four elements but, in order to make it look more like other ciphertexts, we will give it as a unique hexadecimal string obtained by concatenating, in order, the hexadecimal representations of the four elements, each of which is represented by a k-byte string, where k is the byte length of the Sophie Germain prime q.

```
> CSEncrypt:=proc(publickey::list, message::string, seed::{posint, string},
     {messagetype::identical(hex, text):=hex, bbslength::{512, 768, 1024}:=1024})
  local pk, q, p, qbitLen, maskLen, k, M, EM, s, B, r, u1, u2, e, alpha, ra, v;
  if type(publickey[1],string) then
    pk := OS2IP~(publickey)
  else
    pk := publickey
  end if;
  q := pk[1];
  p := 2*q+1;
  qbitLen := intlog[2](q) + 1;
  maskLen := intlog[256](q);
  k := maskLen + 1;
  M := messagetohex(message, messagetype);
  EM := QREnc(M, q);
  s := stringposint(seed);
  B := RandomTools:-BlumBlumShub:-NewBitGenerator(s, primes = bbslength);
  r := q;
  while q <= r do
    r := convert(cat(seq(B(), i = 1 .. qbitLen)), decimal, binary)
  end do;
  u1 := Power(pk[2], r) mod p;
  u2 := Power(pk[3], r) mod p;
  e := (Power(pk[6], r) mod p)*EM mod p;
  alpha :=
        OS2IP(cat("01", MGF1(cat(I2OSP(u1,k), I2OSP(u2,k), I2OSP(e,k)), maskLen)));
  ra := r*alpha mod q;
  v := (Power(pk[4], r) mod p)*(Power(pk[5], ra) mod p) mod p;
  cat(I2OSP(u1, k), I2OSP(u2, k), I2OSP(e, k), I2OSP(v, k))
  end proc:
```

Next comes the Cramer–Shoup decryption function following the algorithm outlined in [59, Fig. 2], including the tests to ensure that the elements obtained really belong to the group G as indicated in [59, 6.1, Remark 2].

The required input parameters are the private key in the format output by `CSKeyGen` and the ciphertext string. There is also an optional keyword parameter, `messagetype` which, as in the preceding function, is used to specify whether the plaintext is a hexadecimal string or an ordinary text string. The output is either an error message if the ciphertext is incorrectly formatted (or invalid) or the original plaintext. As in the case of the encryption function, there is the possibility of adding optional parameters that will be accepted by `MGF1` and allow this function to use a hash function different from SHA-256.

```
> CSDecrypt := proc(privatekey::list, ciphertext::string,
                    {messagetype::(identical(hex, text)):= hex})
  local sk, q, p, maskLen, k, c, u1, u2, e, v, alpha, test, m;
  if type(privatekey[1], string) then
    sk := OS2IP~(privatekey)
  else
    sk := privatekey
  end if;
  q := sk[1];
  p := 2*q+1;
  maskLen := intlog[256](q);
  k := maskLen+1;
  if StringTools:-Length(ciphertext) <> 8*k then
    error "incorrectly formatted ciphertext"
  end if;
  c := StringTools:-LengthSplit(ciphertext, 2*k);
  u1 := OS2IP(c[1]);
  u2 := OS2IP(c[2]);
  e := OS2IP(c[3]);
  v := OS2IP(c[4]);
  if not evalb(numtheory:-legendre~([u1, u2, e], p) = [1$3]) then
    error "invalid ciphertext"
  end if;
  alpha := OS2IP(cat("01", MGF1(cat(c[1], c[2], c[3]), maskLen)));
  test := evalb((Power(u1, (sk[2] + alpha * sk[4]) mod q) mod p)*
         (Power(u2, (sk[3] + alpha * sk[5]) mod q) mod p) mod p = v);
  if not test then
    error "invalid ciphertext"
  else
    m := e*(Power(u1, q-sk[6]) mod p) mod p
  end if;
  hextomessage(QRDec(m, q), messagetype)
  end proc:
```

Exercise 8.30 Write modified versions of the functions `CSEncrypt` and `CSDecrypt` that are able to encrypt and decrypt messages of arbitrary size.

Exercise 8.31 Modify the functions `CSEncrypt` and `CSDecrypt` in order to allow them to encrypt and decrypt binary files.

Exercise 8.32 Write a Maple implementation of Cramer–Shoup that closely follows algorithm CS1a in [59] and uses only random parameters instead of pseudo-random ones. For this, modify the functions `CSKeyGen` and `CSEncrypt` in such a way that they do not make use of the Blum–Blum–Shub PRG and, instead, work only with random parameters that are externally generated and supplied as inputs to the functions.

Example 8.18 Let us present an example of Cramer–Shoup encryption and decryption. We will use the previously generated key pair `pk`/`sk` and the following message, which is given as an ordinary text string:

```
> ms := "This is a test of Cramer-Shoup encryption":
```

We will use the following seed for the encryption function:

```
> sd := "5d6c0ec0bb84b416a8431784052cd123":
```

We compute the ciphertext as follows; we must tell the encryption function that the message being encrypted is given as an ordinary text string:

```
> ct := CSEncrypt(pk, ms, sd, messagetype = 'text');
"9c4734b35d0584e238e28a10bbd5507ad03f4c286e2cf2c63a785c3046a36b076b0636059f54cf1a9\
  29177349e6cdfc43080423ab87f05cecd67440834b762774da4382f600bf91048baed0353775c3f2\
  9591e86a41625807f7870e5ba8c717c72e3752691d13c080d3a8e7a3d6ecd7207c6c4280192a7540\
  41ccc856137016487067d2471bef5946af06acb6e72845936fab1a4f1a56b9e0ed18404672e08d3c\
  deb4d542109cb1a1145aba81696c5f43d27a5949827f187741923469f12c33da90a1d60127826b42\
  2676a671125dfa3fc9928423bd14aaa748ba7797d9b8dad4d8ddc706ac2c7cdefd2ae5abd8400aea\
  e5d114947749a2aeb4555e1739bc05f85e1fd12d9a1cf802e5fe7b1be2bd705388390fdf079be391\
  e0538d91125eaad45b7fe26c7bd581b26ee75212e7290a46bc7697b8db3cc80c92a97c97ac6be3f7\
  4bafbe3fc04703232e0353de4e16eaeba315dcb090b2a6e11c2b804bf3c4601ad5caa488a6db812e\
  b94af6ddb8e23cacb4666e5d956f63f76bfea554f2b044f2947910282690f64f22b07390ef1df45e\
  484e5a0f96f84b2c421d92ae53f88ac35da671bec6ec07bd5fc8e138ab1c316cf6f1361a4bc46454\
  bb71a28577a230fca190eaa3c514bd2c0117306bea0c0fe28cca06559330c80f795680f3f67b0564\
  edf2e873694eddbd5fdb24d82a287b10df711188050c443a0b9a198c94ff9e0"
```

Now, the message can be recovered by using the decryption function as follows:

```
> CSDecrypt(sk, ct, messagetype = 'text');
                 "This is a test of Cramer-Shoup encryption"
```

8.7 A Speed Comparison Among Implementations of CCA Secure Schemes

We are going to examine the practical efficiency of the Maple implementations of the CCA secure public-key encryption schemes we have studied. We will make a speed comparison among these implementations but we warn that this comparison has limited value because it depends on the specific details of these implementations and, of course, different results may be obtained with other implementations and other languages. However, this can give a rough approximation of the relative speed of these schemes.

To carry out our experiment, we are going to use the following Maple function that will also serve to test the correct behavior of the encryption and decryption functions of these schemes. The first input parameter is `scheme`, which is used to specify the name of the scheme to be tested among three possibilities: `RSAOAEP`, `RabinSAEPPlus` and `CramerShoup`. The next input parameter is `key`, used to specify a key (consisting of a public key plus the corresponding private key) in the format used in our implementations. The remaining parameters are `seed`, in which a seed (given as a hexadecimal string of appropriate length) is to be specified, `iterations`, for the number of encryption/decryption operations to be performed and, finally, `bytelength`, for the byte length of the messages to be encrypted and

decrypted. The function proceeds to generate these messages pseudo-randomly and then it measures the time taken by the encryption/decryption operations performed. If any of these operations fails to recover the original plaintext, the procedure stops with an error message, otherwise it prints a description of the operations performed and the total time spent.

```
> PublicKeyTiming := proc(scheme::name, key::list, seed::string,
                          iterations::posint, bytelength::posint)
  local Encrypt, Decrypt, pk, sk, hexlength, correct, ti, i, message;
  if scheme = RSAOAEP then
    Encrypt := RSAESOAEPEncrypt;
    Decrypt := RSAESOAEPDecrypt
  elif scheme = RabinSAEPPlus then
    Encrypt := RabinSAEPPlusEncrypt;
    Decrypt := RabinSAEPPlusDecrypt
  elif scheme = CramerShoup then
    Encrypt := CSEncrypt;
    Decrypt := CSDecrypt
  else error
    "incorrect scheme %1", scheme
  end if;
  pk := key[1];
  sk := key[2];
  hexlength := 2*bytelength;
  correct := true;
  ti := time();
  for i to iterations while correct do
    message := StringTools:-LowerCase(StringTools:-Random(hexlength, xdigit));
    correct := evalb(Decrypt(sk, Encrypt(pk, message, seed)) = message)
  end do;
  if correct then
    printf("%d %s encryption/decryptions of %d-byte messages took %f seconds",
           iterations, scheme, bytelength, time()-ti)
  else
    error "an encryption/decryption error was produced"
  end if
  end proc:
```

Observe that the timing could be made more accurate by previously generating the messages to be encrypted/decrypted so that the time taken by this process would not be taken into account by the function. However, this time is small in comparison with that spent by the encryption/decryption function and so we will not worry about this.

Exercise 8.33 Write a variant of the function `PublicKeyTiming` that generates the messages to be encrypted/decrypted before starting the timing.

Let us now perform the timings for the three schemes with keys we have already used in previous examples. We will use the same machine for the three timings and we start with RSAES-OAEP with the RSA key `rsakey` of Example 8.10:

```
> rsakey := RSAKeyGen(1024, 6180204295522709153838923244723480121 75,
                      4102756714646987802485435721275578031 86):
```

This is a key with a 2048-bit modulus and we also need a 256-bit seed because this is the output length of SHA-256, which is the hash function we use in this scheme. To carry out the speed test we do not need a random seed, so we will use the following seed:

```
> sd := StringTools:-Repeat("0123456789abcdef", 4):
```

We are going to choose `bytelength = 95` because this is the maximum size that we will be able to use when testing Rabin-SAEP$^+$ with our previously generated key but we remark that substantially larger messages can be encrypted with a 2048-bit RSA key.

```
> PublicKeyTiming(RSAOAEP, rsakey, sd, 10, 95);
10 RSAOAEP encryption/decryptions of 95-byte messages took 16.282000 seconds
```

Exercise 8.34 Generate an RSA key with a 4096-bit RSA modulus and use this key in a timing experiment with the remaining parameters as above.

Next, we repeat the test for Rabin-SAEP$^+$. In this case we will use the Rabin key `rk` generated in Example 8.13:

```
> seed1 := "3d53d86af89150d1437856427eb8ed78":
  seed2 := "d5d7c4472ae89a7b0b0f7580177236a6":
  rk := RabinKeyGen(256, seed1, seed2):
```

Since this is a Rabin-SAEP$^+$ key generated with security parameter $k = 256$ (which means that it has a 2050-bit modulus) the seed required here must have $k/2 = 128$ bytes. We will use the following seed:

```
se := StringTools:-Repeat("0123456789abcdef", 16):
```

Then the timing test gives the following result:

```
> PublicKeyTiming(RabinSAEPPlus, rk, se, 10, 95);
10 RabinSAEPPlus encryption/decryptions of 95-byte messages took 32.688000 seconds
```

We observe that RSAES-OAEP was substantially faster than Rabin-SAEP$^+$ but this is not really a fair comparison because, as we have seen, the security reduction of Rabin-SAEP$^+$ is much tighter than that of RSAES-OAEP. This means that, to attain a similar security level in practice, it seems likely that much longer keys (above 4096 bits) should be used for RSAES-OAEP and this, of course, would cause a performance penalty for this scheme (although it would also have the advantage of allowing encryption of larger messages). Moreover, as we have also remarked, Rabin-SAEP$^+$ has the advantage that its security reduces to the factoring assumption, which is potentially weaker than the RSA assumption used in the RSAES-OAEP security reduction.

Let us now consider the Cramer–Shoup encryption scheme. In this case we use the Cramer–Shoup key `key` of Example 8.17 above. The seed here is used to feed the Blum–Blum–Shub PRG and we will use the same seed, `sd`, we used for RSAES-OAEP above. Then we obtain:

```
> PublicKeyTiming(CramerShoup, key, sd, 10, 95);
10 CramerShoup encryption/decryptions of 95-byte messages took 37.953000 seconds
```

We see that Cramer–Shoup is the slowest of the three implementations we are considering, although the difference with Rabin-SAEP$^+$ is not that large; it is also the scheme that has a higher message expansion factor. Once again, these aspects must be balanced against the fact that Cramer–Shoup is the only of these three schemes whose reductionist security proof does not rely on the existence of random oracles and hence it seems the strongest scheme from the security point of view.

8.8 Homomorphic Encryption

There are many public-key encryption schemes that we have not discussed so far and here we will discuss some which are of special interest and, in particular, some of the more interesting public-key *homomorphic encryption schemes*. We have already mentioned these schemes but now we are going to discuss them in more detail because their design has been gaining momentum in recent years with the goal of allowing computation on encrypted data without having to decrypt it, in such a way that the resulting data correspond to a similar computation on the plaintexts. The rough idea to achieve this is, therefore, assuming that (pk, sk) is a public key/private key pair, and $c_1, c_2 \dots c_t$ are ciphertexts such that $c_i = \mathbf{Enc}(pk, m_i)$ (for $i = 1, \dots t$ and plaintexts $m_1, m_2, \dots m_t$), to be able to compute a certain function f of the ciphertexts $c_1, \dots, c_t$ to obtain a new ciphertext $f(c_1, c_2, \dots, c_t)$ such that

$$\mathbf{Dec}(sk, f(c_1, \dots, c_t)) = f'(m_1, m_2, \dots, m_t),$$

where f' is a known function acting on t-tuples of plaintexts. This gives a more general concept of homomorphic encryption scheme, with the simplest examples being those schemes, such as plain RSA (because of the multiplicative property) and Elgamal (as we saw in Exercise 8.26), in which both the plaintext and the ciphertext spaces have a group structure and the decryption algorithm preserves the group operation and hence defines a group homomorphism. In these cases, the roles of f and f' are played by the group operations so that, if we denote these operations multiplicatively and if m_1, m_2 are messages such that $\mathbf{Enc}(pk, m_1) = c_1$ and $\mathbf{Enc}(pk, m_2) = c_2$, it holds that $c_1 \cdot c_2$ is a valid encryption of $m_1 \cdot m_2$ and hence $\mathbf{Dec}(sk, c_1 \cdot c_2) = m_1 \cdot m_2$. Thus one can multiply two ciphertexts (with the group operation in the ciphertext space) to obtain a new ciphertext that is a valid encryption of the product of the corresponding plaintexts (with the operation defined in the plaintext space). Next we briefly discuss another interesting homomorphic scheme.

8.8.1 The Goldwasser–Micali Encryption Scheme

When studying plain Rabin encryption, we mentioned the *quadratic residuosity problem (QR problem*, for short), which is the problem of deciding, for a composite positive integer n and $x \in \mathbb{Z}_n^*$, whether x is a quadratic residue modulo n or, more specifically, of distinguishing quadratic residues modulo n from quadratic non-residues modulo n whose Jacobi symbol is equal to 1.[8] This problem was the basis

[8] Recall from the definition of Legendre symbol and from Proposition 2.14 that if $n = pq$ is the product of two distinct odd primes, then an element $x \in \mathbb{Z}_n^*$ is a quadratic residue modulo n if and only if $\left(\frac{x}{p}\right) = 1$ and $\left(\frac{x}{q}\right) = 1$ which, in particular, implies that $\left(\frac{x}{n}\right) = 1$.

for the first CPA secure public-key encryption scheme, which was introduced by Goldwasser and Micali in 1984 and proved CPA secure precisely under the assumption that quadratic residuosity is hard. The Goldwasser–Micali scheme encrypts one bit at a time and hence is very inefficient and not used in practice. However, it has historical interest because this was the first time that probabilistic encryption—which is a necessary condition for CPA security—was used and the reductionist security proof for the scheme set a model that would be subsequently applied to many others.

The quadratic residuosity problem can be formulated more formally as follows, relative to a PPT modulus generation algorithm that, on input 1^k, outputs a 3-tuple (n, p, q) such that $n = pq$ and p, q are primes with $\text{len}(p) = \text{len}(q) = k$. We set $\mathcal{QN}_n^1 := \mathcal{QN}_n \cap \mathcal{J}_n = \{x \in \mathcal{QN}_n | \left(\frac{x}{n}\right) = 1\} \subseteq \mathbb{Z}_n^*$, the set of quadratic non-residues modulo n whose Jacobi symbol relative to n is equal to 1. We consider the following experiment:

Definition 8.18 The *QR experiment* $\mathbf{QR}_{\mathcal{A}}(k)$ is the following:

1. The modulus generation algorithm is run on input 1^k to obtain (n, p, q) with $n = pq$, where p and q are primes such that $\text{len}(p) = \text{len}(q) = k$.
2. A random bit $b \leftarrow \{0, 1\}$ is chosen; if $b = 0$ then a random $x \leftarrow \mathcal{QR}_n$ is chosen and if $b = 1$ then a random $x \leftarrow \mathcal{QN}_n^1$ is chosen.
3. $\mathcal{A}$ is given (n, x) and outputs a bit b'.
4. The output of the experiment is defined to be 1 if $b' = b$ and 0 otherwise. If $\mathbf{QR}_{\mathcal{A}}(k) = 1$ then we say that $\mathcal{A}$ succeeded.

We say that the *the QR problem is hard relative to the modulus generation algorithm* if, for every PPT algorithm $\mathcal{A}$, there exists a negligible function $\texttt{negl}$ such that

$$\Pr(\mathbf{QR}_{\mathcal{A}}(k) = 1) \leq 1/2 + \texttt{negl}(\texttt{k}).$$

Remark 8.7 Observe that, with the notations above, we have that $|\mathcal{QR}_n| = |\mathcal{QN}_n^1|$. Indeed, by Exercise 2.43, exactly $\phi(n)/2$ of the elements of $\mathbb{Z}_n^*$ (i.e., exactly one-half of the elements) belong to $\mathcal{J}_n$ when n is not—as happens in this case—a square. By Corollary 2.7 each quadratic residue modulo n has exactly four square roots in $\mathbb{Z}_n^*$ when $n = pq$ is a product of two distinct odd primes, so that $|\mathcal{QR}_n| = \phi(n)/4$ in this case. Since $\mathcal{J}_n$ is the disjoint union of $\mathcal{QR}_n$ and $\mathcal{QN}_n^1$, we see that each of these subsets contains exactly one-half of the elements of $\mathcal{J}_n$. Because of this, step 2 in Definition 8.18 amounts to choosing at random an element from $\mathcal{J}_n$ and then a PPT algorithm $\mathcal{A}$ which tries to decide whether the element is in $\mathcal{QR}_n$ or in $\mathcal{QN}_n^1$ has a probability of success equal to $1/2$ if it makes a random guess. Thus the QR problem is hard whenever each PPT algorithm succeeds in distinguishing whether an element of $\mathcal{J}_n$ is a quadratic residue only with probability negligibly better than $1/2$.

The idea of the Goldwasser–Micali encryption scheme is then the following. Given a modulus $n = pq$, use n as the public key and its factorization as the private key, and encrypt '0' bits as random quadratic residues modulo n and '1' bits as random quadratic non-residues modulo n with Jacobi symbol $+1$. The receiver can

decrypt messages because knowledge of the factorization of n makes it easy to decide whether an element of $\mathbb{Z}_n^*$ is a quadratic residue or not: the receiver just computes the Lagrange symbol of the ciphertext with respect to the primes p and q. If both are equal to $+1$ then the plaintext is a bit '0' by Proposition 2.14 and if both are equal to -1 then the plaintext is '1' (the mixed case in which one of the Lagrange symbols is $+1$ and the other is -1 cannot happen because, by construction, the Jacobi symbol—and hence the product of the two Lagrange symbols—is equal to $+1$).

This setup seems OK except for one thing: given only n there is no known efficient way, in general, to randomly chose an element in $\mathcal{QN}_n^1$. This problem can be solved by adding something else to the public key, namely, a randomly chosen element $x \leftarrow \mathcal{QN}_n^1$, which can easily be obtained by the key generation algorithm with the help of the factorization of n, as it suffices to ensure that the Legendre symbol with respect to both prime factors p and q is -1. Once this element $x \leftarrow \mathcal{QN}_n^1$ is added to n to form the public key, it is easy to choose random elements in both $\mathcal{QR}_n$ and $\mathcal{QN}_n^1$. In the first case one just chooses a random element of $\mathbb{Z}_n^*$ and squares it; in the second, one chooses a random quadratic residue as before and then multiplies it by x. The resulting element is a quadratic non-residue modulo p and also modulo q and its Jacobi symbol with respect to n is 1, so it is indeed an element of $\mathcal{QN}_n^1$.

Bearing in mind these remarks, the Goldwasser–Micali encryption scheme may be described as follows:

Definition 8.19 The *Goldwasser–Micali* public-key encryption scheme is defined by the following algorithms:

- **Gen**: On input 1^k, run the modulus generation algorithm to obtain (n, p, q) such that $n = pq$, where p and q are random primes of length k, and choose $x \leftarrow \mathcal{QN}_n^1$. The public key is $pk = (n, x)$ and the private key is $sk = (p, q)$.
- **Enc**: On input a public key $pk = (n, x)$ and a message $m \in \{0, 1\}$, choose a random element $r \leftarrow \mathbb{Z}_n$ and output the ciphertext $c = (x^m \cdot r^2) \bmod n \in \mathbb{Z}_n^*$.
- **Dec**: On input a private key $sk = (p, q)$ and a ciphertext $c \in \mathbb{Z}_n^*$, compute $\left(\frac{c}{p}\right)$ and $\left(\frac{c}{q}\right)$. If both values are 1 then output $m = 0$, otherwise (i.e., if both values are -1) output $m = 1$.

Exercise 8.35 Prove the correctness of the Goldwasser–Micali encryption scheme by showing that, for each public key/private key pair and each $m \in \{0, 1\}$, $\mathbf{Dec}(sk, \mathbf{Enc}(pk, m)) = m$.

Exercise 8.36 Prove that if the QR problem is hard relative to the modulus generation algorithm, then the Goldwasser–Micali encryption scheme is CPA secure.

Exercise 8.37 Write, along the lines of the preceding ones, a Maple implementation of the Goldwasser–Micali encryption scheme and use it to encrypt/decrypt (very) short messages (see remark below about message expansion).

Remark 8.8 An important drawback that makes the Goldwasser–Micali encryption scheme unsuitable for practical use is its inefficiency, since encryption requires either

one or two modular multiplications to encrypt a single bit and decryption at least one Legendre symbol computation per bit. Another inconvenience is the large message expansion it produces. In contrast with the efficient schemes we have seen, which expand messages by a small constant factor no larger than 4, the expansion factor of Goldwasser–Micali is $\log_2 n$ as each bit is encrypted as an element of $\mathbb{Z}_n^*$. Since a necessary condition for the hardness of the QR problem for the modulus n is the hardness of factoring n, in order to attain practical security the modulus should have at least 2000 bits producing a totally impractical expansion factor of at least 2000.

To prove the homomorphic property of Goldwasser–Micali, we observe that, as it is easy to see using the multiplicative property of the Legendre symbol, the following holds:

Proposition 8.8 *Let $n = pq$, where p, q are distinct odd primes, and denote by $X \cdot Y$ the subset $\{xy | x \in X, y \in Y\}$ for subsets $X, Y \subseteq \mathbb{Z}_n^*$. Then the following conditions hold:*

(i) $\mathcal{QR}_n \cdot \mathcal{QR}_n \subseteq \mathcal{QR}_n$,
(ii) $\mathcal{QR}_n \cdot \mathcal{QN}_n^1 \subseteq \mathcal{QN}_n^1$,
(iii) $\mathcal{QN}_n^1 \cdot \mathcal{QN}_n^1 \subseteq \mathcal{QR}_n$.

Exercise 8.38 Prove Proposition 8.8.

As a consequence of Proposition 8.8 it is straightforward to see that the Goldwasser–Micali encryption scheme is homomorphic because if $c_1, c_2 \in \mathbb{Z}_n^*$ are two ciphertexts, then the decryption of their product is the sum of the corresponding plaintexts $m_1, m_2 \in \mathbb{Z}_2$. Thus the decryption function is a group homomorphism but we cannot say the same about encryption because it is probabilistic and hence it does not define a map from the plaintext space $\{0, 1\}$—which can be identified with the (additive) group $\mathbb{Z}_2$—to $\mathbb{Z}_n^*$ (or to $\mathcal{J}_n$). But we can also obtain a homomorphism from the encryption algorithm as in the following exercise:

Exercise 8.39

(i) Prove that $\mathcal{J}_n$ is a subgroup of $\mathbb{Z}_n^*$ and $\mathcal{QR}_n$, in turn, is a subgroup of $\mathcal{J}_n$.
(ii) If $pk = (n, x)$ is a Goldwasser–Micali public key, consider the quotient group $\mathcal{J}_n/\mathcal{QR}_n$ and show that the map $\mathbb{Z}_2 \to \mathcal{J}_n/\mathcal{QR}_n$ defined by $m \mapsto \mathbf{Enc}(pk, m)\mathcal{QR}_n$ is a group homomomorphism (actually an isomorphism).

Exercise 8.40 Show, using the homomorphic property, that the Goldwasser–Micali encryption scheme is not CCA secure.

8.8.2 The Paillier Encryption Scheme

In order to provide an appreciation for the advantages of homomorphic encryption schemes, let us consider a situation in which they can be particularly useful. Suppose that we want to implement an electronic voting system that guarantees confidentiality,

so that no one learns anyone else's vote. Suppose also that each voter casts one vote (for a given candidate), which can be either neutral or favorable. An obvious possibility is to make each voter cast her vote by encrypting it with a public-key encryption scheme using the public key of a trusted authority which will be given the encrypted votes and will obtain the vote total after decrypting them with the private key. However, our initial goal is not attained this way since the authority learns the individual votes. Keeping the basic idea of using a public-key encryption scheme to encrypt the votes, the goal would then be to devise a protocol that allows the authority to count the total vote without having access to the individual encrypted votes, so that the votes can be counted without having to decrypt them. This is just a particular case of what a homomorphic encryption scheme can do: if the authority is given the product of all encrypted votes corresponding to a candidate, it can decrypt it to obtain the product of the corresponding (unencrypted) votes, which allows the number of favorable votes obtained to be counted. This can be achieved with plain RSA as the following exercise shows:

Exercise 8.41 Show that plain RSA can be used to efficiently obtain the vote total from the product of all encrypted votes, assuming knowledge of the private key (Hint: use two small primes to represent the neutral vote and the favorable vote, respectively.)

It might seem at first sight that plain RSA is adequate for our purpose of guaranteeing vote confidentiality but the sketched protocol fails because, as we have seen, plain RSA is completely insecure and an adversary that observes an encrypted vote c can learn the vote by just encrypting the neutral vote and the favorable vote and comparing the result with c. This total failure is due to plain RSA having a deterministic encryption algorithm and it is clear that, for the protocol to be of any use, the underlying homomorphic encryption scheme should also be CPA secure (under some reasonable hardness assumption, as usual). The variants of RSA which are CPA secure, such as RSA-OAEP, are not homomorphic because the random padding destroys the multiplicative property. We have seen, however, that there are public-key encryption schemes, such as Elgamal and Goldwasser–Micali, that are both homomorphic and CPA secure. The latter scheme is not suitable for our purpose because the plaintext space, namely $\mathbb{Z}_2$, is too small and the former could be used but we are going to discuss another scheme which also satisfies the required properties and is specially suited for counting votes because the plaintext space is $\mathbb{Z}_n$ for a large value of n: the *Paillier encryption scheme*.

Let $n = pq$ be the product of two distinct primes of the same length. We may describe the group $\mathbb{Z}_{n^2}^*$ in terms of the groups $\mathbb{Z}_n$ and $\mathbb{Z}_n^*$ and, for this purpose, we will consider the group $\mathbb{Z}_n \times \mathbb{Z}_n^*$, whose operation will be generically denoted by multiplication (notice that the first components of the elements of the product are added and the second components multiplied, in both cases modulo n). We have:

Proposition 8.9 *Let $n = pq$, where p, q are distinct odd primes of the same length and let $g \in \mathbb{Z}_{n^2}^*$ be an element of order n. Then the map*

$$\begin{array}{ccc} \mathbb{Z}_n \times \mathbb{Z}_n^* & \xrightarrow{\varepsilon_g} & \mathbb{Z}_{n^2}^* \\ (x, y) & \longmapsto & g^x y^n \bmod n^2 \end{array}$$

is a group isomorphism.

Proof We start by showing that ε_g is a group homomorphism. Given (x_1, y_1), $(x_2, y_2) \in \mathbb{Z}_n \times \mathbb{Z}_n^*$ we have to show that

$$(g^{(x_1+x_2) \bmod n}) \cdot (y_1 y_2 \bmod n)^n \bmod n^2 = (g^{x_1} y_1^n)(g^{x_2} y_2^n) \bmod n^2.$$

Note first that, since the order of g is n, we have that $g^{(x_1+x_2) \bmod n} \bmod n^2 = g^{x_1} g^{x_2} \bmod n^2$. Thus it suffices to show that $(y_1 y_2 \bmod n)^n \bmod n^2 = (y_1 y_2)^n \bmod n^2$. Setting $r = y_1 y_2 \bmod n$, we have that $y_1 y_2 = nq + r$ for some non-negative integer q and hence $(y_1 y_2)^n = \sum_{i=0}^{n} \binom{n}{i} (nq)^i r^{n-i}$. All the terms in this sum except the one corresponding to $i = 0$ are multiples of n^2, so we obtain that $(y_1 y_2)^n \bmod n^2 = r^n \bmod n^2$ and ε_g is indeed a homomorphism.

Next, to prove that ε_g is an isomorphism, i.e., that it is bijective, observe first that $|\mathbb{Z}_n \times \mathbb{Z}_n^*| = |\mathbb{Z}_n| \cdot |\mathbb{Z}_n^*| = n \cdot \phi(n) = p(p-1)q(q-1) = \phi(p^2) \cdot \phi(q^2) = \phi(n^2)$, so that both groups have the same order and hence it suffices to show that ε_g is injective. Since ε_g is a group homomorphism, it is enough to show that its kernel is the trivial group, i.e., that if $\varepsilon_g(x, y) = 1$, then $(x, y) = (0, 1)$. Assume then that $x \in \mathbb{Z}_n$ and $y \in \mathbb{Z}_n^*$ are such that $\varepsilon_g(x, y) = g^x y^n \bmod n^2 = 1$. This means that $g^x \bmod n^2$ and $y^n \bmod n^2$ are mutually inverse in $\mathbb{Z}_{n^2}^*$ and hence it follows that they have the same order. Since g has order n by hypothesis, the order of $g^x \bmod n^2$ is a divisor of n by Proposition 2.5 and, since $(y^n)^{\phi(n)} \bmod n^2 = 1$ by Corollary 2.1 we have, by Proposition 2.4, that the order of $y^n \bmod n^2$ is a divisor of $\phi(n)$. Thus the order of both elements is a divisor of both n and $\phi(n)$ and, since clearly $\gcd(n, \phi(n)) = 1$ as a consequence of p and q having the same length, we see that this order is equal to 1 and hence $g^x \bmod n^2 = 1$ and $y^n \bmod n^2 = 1$. The first equality means that $x = 0$ (since g has order n) and the second implies that the order of y in $\mathbb{Z}_{n^2}^*$, and hence also the order of y in $\mathbb{Z}_n^*$, is a divisor of n. Thus this order divides both n and $\phi(n)$ and hence is equal to 1, so that we obtain $y = 1$, completing the proof. □

Remark 8.9 Note that the hypothesis that the primes p, q have the same length has only been used to ensure that $\gcd(n, \phi(n)) = 1$ so that the former condition may be replaced by the latter if desired.

Exercise 8.42 In [152, Lemma 1] it is shown that ε_g is bijective under the more general assumption that the order of g is a nonzero multiple of n. Give an example that shows that in this case ε_g is not necessarily a homomorphism.

Corollary 8.1 *Given the same hypotheses as in the preceding proposition, let* $\pi : \mathbb{Z}_n \times \mathbb{Z}_n^* \to \mathbb{Z}_n$ *be the canonical projection. Then the composition map* $\delta_g = \pi \circ (\varepsilon_g)^{-1} : \mathbb{Z}_{n^2}^* \to \mathbb{Z}_n$ *is a group homomorphism.*

Definition 8.20 An element $w \in \mathbb{Z}_{n^2}^*$ is called an nth residue modulo n^2 if it is an nth power in $\mathbb{Z}_{n^2}^*$, i.e., if there exists $u \in \mathbb{Z}_{n^2}^*$ such that $w = u^n \bmod n^2$.

The nth residues modulo n^2 may then be characterized as follows:

Proposition 8.10 *An element $w \in \mathbb{Z}_{n^2}^*$ is an nth residue if and only if $w \in \mathrm{Ker}\delta_g$.*

Proof Suppose first that $w = u^n \bmod n^2$ is an nth residue. Then there exists $(a, b) \in \mathbb{Z}_n \times \mathbb{Z}_n^*$ such that $u = \varepsilon_g(a, b)$ and hence we have $\delta_g(w) = \delta_g((\varepsilon_g(a, b))^n) = \delta_g(\varepsilon_g((a, b)^n)) = \pi(0, b^n \bmod n) = 0$. The converse is also clear because if $\delta_g(w) = 0$ then $w = \varepsilon_g(0, b)$ for some $b \in \mathbb{Z}_n^*$ and hence $w = b^n \bmod n^2$. □

Remark 8.10 From the preceding proposition we see that the nth residues modulo n^2 form a subgroup of $\mathbb{Z}_{n^2}^*$. If we denote this subgroup by $\mathcal{R}_{n^2}^n$, then we have as a consequence of Propositions 2.7 and 8.10 that δ_g induces an isomorphism between $\mathbb{Z}_{n^2}^*/\mathcal{R}_{n^2}^n$ and $\mathbb{Z}_n$. The elements of the former group are also called nth residuosity classes and, for $w \in \mathbb{Z}_{n^2}^*$, the element $\delta_g(w) \in \mathbb{Z}_n$ determines uniquely the nth residuosity class to which w belongs and hence one also says, as in [152], that $\delta_g(w)$ is the nth residuosity class of w. Note that there are exactly n residuosity classes and that each of them contains $\phi(n)$ elements so that, in particular, $|\mathcal{R}_{n^2}^n| = \phi(n)$. From this it also follows that the number of nth roots of each nth residue w is exactly n. Indeed, from Proposition 8.10 it follows that if $w \in \mathbb{Z}_n^*$ is an nth residue, then $w = \varepsilon_g(0, u)$ for a unique $u \in \mathbb{Z}_n^*$ (observe also that ε_g induces an isomorphism between $\mathbb{Z}_n^*$ and $\mathcal{R}_{n^2}^n$ and that $\mathbb{Z}_{n^2}^*$ is isomorphic to the direct product $\langle g \rangle \times \mathcal{R}_{n^2}^n$). Thus u is the unique nth root of w which is less than n and the n elements of the set $\{g^x u | x \in \mathbb{Z}_n\}$ are nth roots of w and hence they are clearly all the nth roots.

The basic idea underlying the Paillier encryption scheme is to use ε_g for encryption and δ_g for decryption based on the hypothesis that ε_g is a one-way trapdoor permutation. The idea is then to encrypt a message $m \in \mathbb{Z}_n$ by picking a random $r \in \mathbb{Z}_n^*$ and computing the ciphertext $c := \varepsilon_g(m, r) = g^m r^n \bmod n^2 \in \mathbb{Z}_{n^2}^*$. Decryption consists of computing $m = \delta_g(c)$. Computing $\delta(w)$ for a randomly chosen $w \in \mathbb{Z}_{n^2}^*$ is conjectured to be hard and this conjecture is named the *computational composite residuosity assumption* (CCRA) in [152]. However, this computation becomes easy if the prime factorization of n is known (or, equivalently, if $\phi(n)$ is known), which is the trapdoor information that is included in the private key and allows decryption. We are going to describe the Paillier scheme taking, for simplicity, $g = 1 + n$, which is the most common choice. This value of g satisfies our requirements, for we have:

Proposition 8.11 $1 + n \in \mathbb{Z}_{n^2}^*$ *is an element of order n.*

Proof Using the binomial expansion of $(1 + n)^m$, for $1 \leq m \leq n$, we see that all summands except the first two are multiples of n^2 and hence $(1 + n)^m \bmod n^2 = 1 + mn$, so that the smallest positive integer m which makes this power equal to 1 in $\mathbb{Z}_{n^2}^*$ is $m = n$. □

Exercise 8.43 Show that the set $\{1 + xn \in \mathbb{Z}_{n^2}^* | x \in \mathbb{Z}_n\}$ is a cyclic subgroup of $\mathbb{Z}_{n^2}^*$ whose generators are the elements in the set $\{1 + xn \in \mathbb{Z}_{n^2}^* | x \in \mathbb{Z}_n^*\}$.

We next describe the Paillier scheme.

Definition 8.21 The Paillier public-key encryption scheme (**Gen**, **Enc**, **Dec**) is defined as follows:

- **Gen**: On input 1^k, generate an RSA modulus $n = pq$, where p and q are k-bit primes. The public key is n and the private key is the pair $(n, \phi(n))$, where $\phi(n) = (p-1)(q-1)$.
- **Enc**: Given the message $m \in \mathbb{Z}_n$ and the public key n, choose $r \leftarrow \mathbb{Z}_n^*$ at random and output the ciphertext:

$$c := ((1+n)^m \cdot r^n) \bmod n^2 \in \mathbb{Z}_{n^2}^*$$

- **Dec**: Given the ciphertext $c \in \mathbb{Z}_{n^2}^*$ and the private key $(n, \phi(n))$, output:

$$m := \frac{(c^{\phi(n)} \bmod n^2) - 1}{n} \cdot \phi(n)^{-1} \bmod n,$$

where the quotient $((c^{\phi(n)} \bmod n^2) - 1)/n$ is computed in $\mathbb{Z}$.

To prove the correctness of the scheme, set $g = 1 + n$ and let $d = c^{\phi(n)}$, where $c = (g^m \cdot r^n) \bmod n^2 = \varepsilon_g(m, r)$. We have that $d = c^{\phi(n)} = (\varepsilon_g(m, r))^{\phi(n)} = \varepsilon_g((m, r)^{\phi(n)}) = \varepsilon_g(m\phi(n) \bmod n, r^{\phi(n)} \bmod n) = \varepsilon_g(m\phi(n) \bmod n, 1) = (1+n)^{m\phi(n) \bmod n} \bmod n^2 = 1 + (m\phi(n) \bmod n) \cdot n$ (where we used the binomial expansion as before to obtain the last term). Thus $(d-1)/n = m\phi(n) \bmod n$ and, finally, the output of multiplying this integer by the inverse of $\phi(n)$ modulo n is precisely the plaintext m.

Now, it is clear that the Paillier encryption scheme is OW-CPA secure if and only if the CCRA holds but, as on other occasions, we are interested in obtaining a scheme that is, at least, CPA secure. We can show that the Paillier scheme enjoys this property under a suitable hardness assumption. This assumption refers to the *decisional composite residuosity problem*, which is the problem of distinguishing a random element of $\mathcal{R}_{n^2}^n$ from a random element of $\mathbb{Z}_{n^2}^*$. The hypothesis that this problem is hard, i.e., that no probabilistic polynomial-time distinguisher can distinguish these elements with non-negligible probability is called the *decisional composite residuosity assumption* (DCRA) in [152] and we are going to show that it provides the required CPA security.

Exercise 8.44 Give a formal definition of the DCRA by considering an experiment similar to the *QR experiment* in Definition 8.18, which was used to define the hardness of the quadratic residuosity problem.

Theorem 8.6 *The Paillier encryption scheme is CPA secure if and only if the DCRA holds.*

Proof Let m_0, m_1 be the two plaintexts output by the adversary in the IND-CPA experiment and let $c = g^{m_b} r^n \bmod n^2$ be the ciphertext corresponding to a randomly chosen bit b. We set $w = cg^{-m_0} \bmod n^2 \in \mathbb{Z}_{n^2}^*$ and we claim that $b = 0$ if and only if w is an nth residue. If $b = 0$ then $w = r^n \bmod n^2$ is indeed an nth residue. For the converse, assume that $b = 1$. Then $w = g^{(m_1 - m_0) \bmod n} r^n \bmod n^2$, where we may assume that m_0 and m_1 are distinct. Therefore, $\delta_g(w) = (m_1 - m_0) \bmod n \neq 0$ and hence w is not an nth residue by Proposition 8.10. As a consequence of the just-proved claim, the CPA attack is successful if and only if composite residuosity can be successfully decided. □

Remarks 8.15

1. Observe that the DCRA implies the CCRA and that the latter condition, in turn, implies that factoring the modulus is hard. It is not known, however, whether any of these implications are in fact equivalences (and it seems likely that the factoring assumption is weaker than the other two assumptions, see [152] for details).
2. Corollary 8.1 says precisely that Paillier decryption (which is given by the function δ_g) is a group homomorphism and hence we see that the Paillier encryption scheme is homomorphic. This property is especially well suited for the electronic voting problem mentioned above because the plaintext space is the additive group $\mathbb{Z}_n$. Thus it suffices to encode the neutral vote as $0 \in \mathbb{Z}_n$ and the favorable vote as $1 \in \mathbb{Z}_n$ and then, if the votes cast by t voters are $m_1, m_2, \ldots m_t \in \mathbb{Z}_n$ and the corresponding encrypted votes are $c_1, c_2, ..., c_t \in \mathbb{Z}_{n^2}^*$, the favorable votes can be counted without having to decrypt them by just giving $c := \prod_{i=1}^{t} c_i \bmod n^2 \in \mathbb{Z}_{n^2}^*$ to the authority which, after decrypting c with its private key will obtain

$$\sum_{i=1}^{t} m_i \bmod n$$

which, assuming that the number t of voters is less than n, is precisely the number of favorable votes.[9] For this, the authority need not know the individual ciphertexts c_i and hence it need not know the individual votes m_i. Observe, however, that we have only described the basic ideas about the voting system on the assumption that all parties act honestly but we have not given a complete voting protocol which would require other ingredients such as an efficient proof of validity of the vote and a threshold scheme[10] to allow an authority consisting of a set of parties that share the private key to obtain the vote total as indicated above without allowing a single party to decrypt individual votes (see, for example, [63] for a generalization of Paillier's scheme that meets these requirements).

[9] This will certainly be the case in any practical situation since, for the scheme to be secure, n should be hard to factor and hence t will be much smaller than n.

[10] A scheme for distributing a secret among several parties, each of whom is allocated a share of the secret, which can be reconstructed only when a sufficient number of shares are combined together.

Exercise 8.45 Show that the Paillier encryption scheme is not CCA secure by using the homomorphic property to exhibit a CCA attack against the voting system just mentioned that allows the adversary to recover a vote (given the encrypted vote and the public key) with just one query to the decryption oracle.

8.8.3 The Paillier Encryption Scheme in Maple

Here we give a Maple implementation of the Paillier encryption scheme and an example of its use for counting votes. For simplicity, our implementation will work with plaintexts in $\mathbb{Z}_n$ and ciphertexts in $\mathbb{Z}_n^*$ (given, like the keys, in decimal format) and we will leave as an exercise for the reader to obtain an implementation able to deal with messages of arbitrary length given as (ASCII or hex) strings or files. We start with the key generating function, which just calls the one used for (plain) RSA. The inputs are just those corresponding to the positional parameters in `RSAKeyGen` (and, also in this case, the seeds may be omitted), except for the fact that now all of them must be in decimal form and the output is a Paillier public key/private key pair, given as a Maple list:

```
> PaillierKeyGen := proc(k::posint, seed1::posint, seed2::posint)
  local key;
  key := RSAKeyGen(k, _params['seed1'], _params['seed2'], 'format' = 'decimal');
  [key[1][1], [key[1][1], (key[2][3]-1)*(key[2][4]-1)]]
  end proc:
```

The Paillier encryption function requires, in addition to the public key and the plaintext, a decimal random seed (which, for convenience, will be automatically generated if not provided). As already mentioned, the plaintexts will be the elements of $\mathbb{Z}_n$, where n is the Paillier public key.

```
> PaillierEnc := proc(publickey::posint, m::nonnegint, seed::posint)
  uses RandomTools;
  local n, l, s, B, found, r;
  n := publickey;
  l := ilog2(n)+1;
  if _params['seed'] = NULL then
    MersenneTwister:-SetState();
    s := MersenneTwister:-GenerateInteger(':-range' = 2^127 .. 2^256)
  else
    s := seed
  end if;
  B := BlumBlumShub:-NewBitGenerator(s, primes=1024, numbits=l, output=integer);
  found := false;
  while not found do
    r := B();
    found := evalb(r < n and igcd(r, n) = 1)
  end do;
  (Power(1+n, m)*Power(r, n)) mod n^2
  end proc:
```

The Paillier decryption function is then the following:

```
> PaillierDec := proc(privatekey::list, c::posint)
  local n, f, t;
  n := privatekey[1];
  f := privatekey[2];
  t := ((Power(c, f) mod n^2)-1)/n;
  t/f mod n
  end proc:
```

Example 8.19 Let us give a demonstration of how the Paillier scheme may be used to count votes without decrypting them. We first generate a key with a 2048-bit modulus, as well as the corresponding public and private keys (we do not print the keys since they occupy a lot of space):

```
> k := PaillierKeyGen(1024, 46180204295522709153839232447234801217 51,
                                  79410275671464698780248543572127557803 18):
  pk := k[1]:
  sk := k[2]:
```

Suppose now that there are 100 voters and that the votes are members of the following list which, for convenience, is pseudo-randomly generated here (but note that this list would not be compiled nor needed in the actual implementation of the voting protocol):

```
> BG := RandomTools:-BlumBlumShub:-NewBitGenerator(123456, numbits=100, primes=1024);
  votes := [BG()]:
```

The list of encrypted votes is then the following (again, we do not print it since it would take many pages):

```
> cvotes := PaillierEnc~(pk, votes):
```

The product ciphertext which is given to the authority is:

```
> pvotes := mul(i, i = cvotes) mod pk^2;
  33373495097499325915652242445390097920992216387002790039506058043211023308890366\
   619731161812823676 79[...1033 digits...]98047554429935448477063261220839485272 69\
   799100397311881066559469905000430758471854402785188619945551
```

The authority obtains the vote total by decrypting the product ciphertext:

```
> PaillierDec(sk, pvotes);
                                44
```

Now, we step outside the protocol and make use of the list of votes to check that the result obtained by the authority is correct:

```
> ListTools:-Occurrences(1, votes);
                                44
```

8.9 Final Remarks on Public-Key Encryption

We close this chapter by briefly mentioning two important areas in modern cryptography which are currently undergoing fast development: fully homomorphic encryption and lattice-based cryptography.

8.9.1 Fully Homomorphic Encryption

The examples of homomorphic encryption schemes we have mentioned are schemes that are homomorphic with respect to an operation involving plaintexts and another one involving ciphertexts. But an interesting question is to extend this property to more complex algebraic structures that involve two operations, namely to rings. The idea is to obtain an encryption scheme that preserves both operations, i.e., such that encryption behaves as a ring homomorphism (sometimes these schemes are called *algebraic homomorphic encryption schemes*). This idea was formulated in a somewhat different language already in 1978 when Rivest, Adleman and Dertouzos asked in [163] for a *fully homomorphic encryption scheme*, namely an encryption scheme endowed with an efficient algorithm **Eval** which, for any valid public key pk, any circuit C, and any ciphertexts $c_i = \mathbf{Enc}(pk, m_i)$, outputs $c \leftarrow \mathbf{Eval}(pk, C, c_1, \ldots, c_t)$, where c is a valid encryption of $C(m_1, \ldots, m_t)$ under the key pk. Without going into details, we will just mention that this way one would be able to perform arbitrary computations on encrypted data with the only restriction that they were efficiently expressed as a circuit.

An important point to take into account in the quest for fully homomorphic encryption is that, as we have seen in all our examples, a homomorphic encryption scheme cannot be CCA secure (in the IND-CCA2 sense) because it is clearly malleable and an adversary, after being given the challenge ciphertext, can operate it with another ciphertext it chooses and obtain a decryption of the new ciphertext from the oracle, from which the decryption of the challenge ciphertext can be deduced because of the known relation between the plaintexts. Thus homomorphic encryption is a two-edged sword: it cannot reach the highest security level but its additional properties make it useful in cases where CPA security is sufficient. This is, if anything, even more so in the case of a fully homomorphic scheme, so the challenge was to find a scheme that was, at the same time, fully homomorphic and CPA secure under some reasonable hardness condition.

The problem was finally solved by Gentry in 2009, and we refer to [84] for a full account and for interesting potential applications of such a scheme such as, for example, searching encrypted databases or running programs on a cluster of remote computers while the programs and the data are kept encrypted. Gentry's scheme is based on *lattices*, namely, subsets of $\mathbb{R}^n$ obtained as linear combinations with integer coefficients of a set of linearly independent vectors of $\mathbb{R}^n$ (called a basis of the lattice) and, more specifically, on *ideal lattices*, which are lattices that can be represented as ideals of a ring (i.e., as additive subgroups closed under multiplication by elements of the ring). The ring considered for this purpose is the quotient ring of $\mathbb{Z}[x]$ modulo the ideal generated by a monic irreducible polynomial f of degree n (defined similarly to the rings $\mathbb{Z}_p[x]_f$ in 2.8.3 with $\mathbb{Z}[x]$ now replacing $\mathbb{Z}_p[x]$). The elements of this ring may be identified with integer coefficient polynomials of degree at most $n-1$ and both the ring and its nonzero ideals are additively isomorphic to $\mathbb{Z}^n$ and define, in particular, lattices. Gentry's scheme is CPA secure under a lattice problem and, while not yet efficient enough for practical use, it has opened a line

of work where interesting improvements and variants are regularly being developed and much progress is expected in the coming years.

8.9.2 Lattice-Based Cryptography

As we have just mentioned, Gentry's fully homomorphic encryption scheme is lattice-based, as are some other encryption schemes that also have interesting properties. Although the study of these schemes is beyond the scope of this book, we make a brief reference to some of them for the interested reader. Lattice-based cryptography relies on the presumed hardness of lattice problems such as, for example, SVP ('Shortest Vector Problem') and its variants (see [102, Chap. 6] for a discussion of these problems and their application to cryptography). One of the reasons why lattice-based cryptography is interesting is because these problems are believed to be hard even for quantum computers and, in contrast with what happens, for example, with the integer factorization problem and the discrete logarithm problem, there are currently no known quantum algorithms for solving lattice problems that perform significantly better than the best classical algorithms.

Another interesting aspect of lattice-based cryptography is that some of its schemes enjoy very strong security reductions based on worst-case hardness instead of the average-case hardness of almost all non-lattice schemes. This means that breaking the scheme with non-negligible probability is at least as hard as solving some lattice problem in the worst case. This is a clear advantage over, say, a factorization-based scheme, where breaking the scheme would imply the ability to factor integers chosen according to a certain distribution but not all integers. An interesting scheme of this type is the Ajtai–Dwork public-key encryption scheme [4] which has the practical disadvantages of requiring keys of very large size and being inefficient; we refer also to [141] and its references for more details on this scheme and on lattice-based cryptography in general.

Yet another advantage of lattice-based cryptography is that, in addition to schemes such as Ajtai–Dwork, which are secure in a very strong sense but inefficient, it also includes schemes that lack a formal security proof but are extremely efficient; the best-known of them is probably the NTRU encryption scheme of Hoffstein, Pipher and Silverman. Although without the guarantee of a formal proof, schemes like this are regarded as good candidates to be secure even if quantum computers are built. We refer to [102] for a detailed study of the NTRU scheme.

Chapter 9
Digital Signatures

We have already mentioned that one important—perhaps the most important—goal of modern cryptography is message authentication, which protects the integrity of messages and ensures that they have not been modified. It is clear that this is of the utmost importance in fields like e-commerce and e-banking, where physical protection of exchanged data is impossible. Thus a crucial cryptographic goal in these situations is to ensure that data cannot be modified without the changes being noticed.

As we have seen, a reasonable guarantee of data authenticity in the private-key setting can be obtained by using a MAC. However, this requires that the data-exchanging parties share a secret key, which severely limits the applicability of these schemes. For example, suppose that a software publisher distributes an update to users: it would be completely impractical to manage a system in which the software company shared a secret key with each user. Thus what is required in this case is a *digital signature* that plays a similar role to a handwritten signature and convinces users of the authenticity of the update without them having to share anything with the software company. This problem presents similarities with the one solved by public-key encryption and we will see that also in this case a public and a private key can be used to obtain the desired result: the signer—the software company in this example—will 'sign' the message with its private key and any other party that knows the signer's public key will be able to use it to check authenticity.

There is another problem that makes digital signatures highly desirable and it is the fact that, even if a MAC used together with a private-key encryption scheme can provide authentication for the communication between two parties, it cannot protect them from each other. Indeed, if Alice and Bob reach an agreement this way, Bob might later repudiate it and claim that he did not send the authenticated agreement message. In this case it would be hard for Alice to convince a third party—for example, a judge—that Bob really sent the message since Bob could argue that the message apparently sent by him was in fact sent by Alice. Because of the symmetry inherent to the private-key scheme in which the same shared key is used by both users for all actions, both possibilities seem equally plausible from an outsider point of

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_9, © Springer-Verlag Berlin Heidelberg 2013

view. What is missing here is the *non-repudiation* property (also called *undeniability*) that should make it infeasible for a user such as Bob to repudiate messages previously sent by him. Traditional written signatures are supposed to have this property and so should digital signatures, which entails that the signature should be tied to a unique user, who alone has the capacity of generating it. In the public-key setting this will be possible because a signature will be generated by means of a private key which is in the sole possession of the signer and this should be sufficient to convince a third party of the authenticity of a document.

Another advantage of digital signatures with respect to MACs is implicit in the preceding comments: since verification of the signature only requires knowledge of the public key, these signatures are publicly verifiable and hence transferable. In particular, this implies that a message signed by a particular signer can be shown to a third party who can verify the signature and, in turn, can convince other parties of the message's authenticity. This feature is very important in practice to ensure that public keys can be securely distributed, which is an indispensable condition for the implementation of public-key cryptography.

9.1 Digital Signature Schemes

Digital signatures play a role in the public-key setting similar to message authentication codes in the private-key setting. Thus digital signature schemes are defined in a way similar to MACs, except that now the tag generation algorithm **Mac** is replaced by a *signing algorithm* **Sign**, which generates a *signature*. Of course, another difference with message authentication codes is that now the key generation algorithm must generate a public/private key pair instead of a single (private) key.

9.1.1 Definition of Signature Schemes

Definition 9.1 A *digital signature scheme* (or, simply, a *signature scheme*) is a 3-tuple $\Sigma = (\mathbf{Gen}, \mathbf{Sign}, \mathbf{Ver})$ of polynomial-time algorithms that satisfy the following conditions:

- The key generation algorithm **Gen** is a probabilistic algorithm that takes as input a *security parameter* 1^k and outputs a pair (pk, sk) of keys called, respectively, the public key and the private key. We write, as usual, $(pk, sk) \leftarrow \mathbf{Gen}(1^k)$.
- The signing algorithm **Sign** takes as input a private key sk and a message m (in a message space $\mathcal{M}$ which may depend on pk) and outputs a signature σ, so that we write $\sigma \leftarrow \mathbf{Sign}(sk, m)$. **Sign** is a probabilistic algorithm in most cases.
- The verification algorithm **Ver** is a deterministic algorithm that takes as input a public key pk, a message m, and a signature σ and outputs a bit $b \in \{0, 1\}$. This output $b := \mathbf{Ver}(pk, m, \sigma)$ is interpreted as meaning *valid* when $b = 1$ and *invalid* when $b = 0$.

We require that any signature properly generated by **Sign** is declared valid by **Ver**, i.e., that for any (pk, sk) output by **Gen** and any message $m \in \mathcal{M}$, the following equality holds: $\mathbf{Ver}(pk, m, \mathbf{Sign}(sk, m)) = 1$.

Signature schemes are used in a way similar to public-key encryption schemes. Each user S who wants to have the ability to sign messages runs $\mathbf{Gen}(1^k)$ to obtain a pair of keys (pk, sk). The public key pk is published and the private key remains the sole possession of S. If S wants to sign a message m, it computes $\sigma \leftarrow \mathbf{Sign}(sk, m)$. Any party wanting to verify the authenticity of m uses the public key pk and computes $\mathbf{Ver}(pk, m, \sigma)$; if it obtains the value 1 then the message is authentic. Of course, the same requirements for keys as in the case of public-key encryption are also valid here and, in particular, secure distribution of public keys must be carried out. This seems to lead to a vicious circle since these keys need to be authenticated which in turn requires digital signatures, and so on, but, as we will see when discussing public-key infrastructures in Sect. 9.7, once a single public-key has been securely distributed by whatever means, it can be used for the secure distribution of other public keys.

9.1.2 Security of Signature Schemes

A message/signature pair (m, s) which is produced by an adversary and accepted by the verification algorithm as valid is called a (valid) *forgery*. To produce forgeries is the goal of an adversary of a signature scheme and hence the adversary is often called a *forger*. The security notions for signature schemes aim at preventing adversaries from producing forgeries but there are different classes of attacks according to the adversary capabilities and the adversary goals. The attacks were classified by Goldwasser et al. [92] and, from the point of view of the capability, the weakest adversary is one that only knows the signer's public key, in which case we have a *key-only attack* (also called a *no-message attack*). But it is more reasonable to assume that the adversary has seen valid message/signature pairs and even that the adversary may have chosen these pairs adaptively. This leads to the stronger notion of *message attacks*, where the adversary is assumed to have access to some of these pairs. In [92], a hierarchy of four classes of attacks of this type is given, with the strongest being the *adaptive chosen message attack*, in which the adversary can adaptively choose messages to be signed by the signer acting as a 'signature oracle'. The security notion obtained by considering these strong attacks is the preferred one for new signature schemes. On the other hand, there is also a strength hierarchy in the types of attack obtained by considering the adversary goal; these attacks in decreasing strength order are (as always, success is required to happen with non-negligible probability):

1. *Total break*: The adversary is able to recover the signer's private key.
2. *Universal forgery*: The adversary obtains an efficient algorithm which is able to sign any message with the signer's signature.

3. *Selective forgery*: The adversary is able to forge the signature of a particular message it chooses before mounting the attack.
4. *Existential forgery*: The adversary is able to forge a signature of at least one message that it cannot choose.

The strongest attacks against digital signatures are those obtained by combining the strongest adversary capability, namely adaptive chosen message attack, with the weakest goal: existential forgery. The security notion obtained by combining these two types of attacks is the preferred one and, in order to define it, we start by defining the following experiment for a signature scheme $\Sigma = (\mathbf{Gen}, \mathbf{Sign}, \mathbf{Ver})$, an adversary $\mathcal{A}$, and a security parameter k:

Definition 9.2 The *signature unforgeability experiment under an adaptive chosen message attack*, $\mathbf{Sign}^{uf\text{-}cma}_{\mathcal{A},\Sigma}(k)$, is the following:

1. Keys (pk, sk) are generated by running $\mathbf{Gen}(1^k)$.
2. The adversary $\mathcal{A}$ is given pk and oracle access to $\mathbf{Sign}(sk, -)$. $\mathcal{A}$ asks a set of queries $\mathcal{Q}$ to the oracle and outputs a message/signature pair (m, σ).
3. The output of the experiment is defined to be 1 if and only if $m \notin \mathcal{Q}$ and $\mathbf{Ver}(pk, m, \sigma) = 1$.

Now, the security definition is obtained, as usual, by formalizing the fact that the adversary cannot succeed in the previous experiment with non-negligible probability:

Definition 9.3 A digital signature scheme $\Sigma = (\mathbf{Gen}, \mathbf{Sign}, \mathbf{Ver})$ is *existentially unforgeable under an adaptive chosen-message attack* (UF-CMA, for short) if, for every PPT adversary $\mathcal{A}$, there exists a negligible function $\mathtt{negl}$ such that

$$\Pr\left(\mathbf{Sign}^{uf\text{-}cma}_{\mathcal{A},\Sigma}(k) = 1\right) \leq \mathtt{negl}(k).$$

Remarks 9.1

1. In what follows, a signature scheme with the UF-CMA security property will often be called simply a secure signature scheme but, of course, this should not be understood as implying that the scheme is secure in some absolute sense.
2. The definition of existential unforgeability prevents the adversary from generating existential forgeries and it might seem that this requirement is too strong because the output message in such a forgery is likely to be meaningless. But we have already seen that, in the context of security notions for encryption schemes, disregarding attacks because of semantic considerations is very dangerous, and the same happens with digital signatures. As in the case of encryption, a signature scheme that allows existential forgeries cannot be used to authenticate keys, which are random-looking if they have been chosen—as should be the case—uniformly at random.
3. When the signing algorithm **Sign** is not deterministic, there will be several different signatures corresponding to a given message. Note that in the previous

UF-CMA definition, the adversary is successful only if it produces a valid signature for a message whose signature was not queried. A stronger security property is obtained when the adversary cannot generate new valid message/signature pairs (where the message may or may not be the same) and hence it cannot generate new valid signatures for one of the queried messages. A signature scheme in which an adversary may obtain, from a given message/signature pair, different signatures for the same message, is said to be malleable.

4. As in the case of message authentication codes, replay attacks in which the adversary re-sends a copy of a message previously signed by the signer are not prevented by the previous security definition. As in that case, the task of preventing such attacks is usually left to a higher-level application that may use, for example, sequence numbers or timestamps in order to make the signed messages unique.

9.2 Some Early Signature Schemes

As happens with public-key encryption, there are digital signature schemes that have security reductions to suitable assumptions about the hardness of computational problems. In fact, pointing out once more the central role that one-way functions play in cryptography, it was shown in [166] that the existence of secure (in the UF-CMA sense) signature schemes is equivalent to the existence of one-way functions. But, as in the case of encryption, there is also a trade-off between security and efficiency, although in recent years secure schemes have been found that are efficient enough for practical use. Next we are going to describe some early signature schemes that are not secure but serve to better appreciate the possible attacks and what should be done to prevent them.

9.2.1 Plain RSA Signatures

The first signature scheme appeared in the original RSA paper [164], where a way to use the RSA algorithm for this purpose was proposed. This was based on the 'trapdoor one-way permutation paradigm' which had been introduced by Diffie and Hellman in [67]. The idea is to use (a family of) trapdoor permutations in a way similar to their proposal for public-key encryption that we have described in Chap. 8, except that now the signing algorithm is the decryption algorithm and the signer would sign a message by "decrypting" it with its private key. Thus, given (f_S, t_S) where f_S is a (public) trapdoor permutation and t_S is the associated (private) trapdoor information known only to S, the signer computes the signature $s = f_S^{-1}(m)$. Afterwards, the signature is publicly verifiable by just "encrypting" it with the public function and checking that the result $f_S(s) = f_S(f_S^{-1}(m))$ is equal to m.

By applying this construction to the family of RSA permutations, plain RSA signatures were introduced in [164] as the following scheme:

Definition 9.4 The *plain RSA signature scheme* is the signature scheme (**Gen**, **Sign**, **Ver**) defined by the following algorithms:

- **Gen**: On input 1^k, run the RSA instance generator (Algorithm 8.1) to obtain $(n, e, d) \leftarrow \mathbf{Gen}_{\mathrm{RSA}}$. Then set $pk := (n, e)$ and $sk := (n, d)$. pk is the public key and sk is the secret key, so that $(pk, sk) \leftarrow \mathbf{Gen}(1^k)$.
- **Sign**: On input a private key $sk = (n, d)$ and a message $m \in \mathbb{Z}_n$, compute the signature:

$$\sigma := \mathbf{Sign}(sk, m) = \mathrm{RSA}_{(n,d)}(m) = m^d \bmod n \in \mathbb{Z}_n.$$

- **Ver**: On input a public key $pk = (n, e)$, a message $m \in \mathbb{Z}_n$, and a signature $\sigma \in \mathbb{Z}_n$, compute:

$$m' := \mathrm{RSA}_{(n,e)}(\sigma) = \sigma^e \bmod n \in \mathbb{Z}_n,$$

and set

$$\mathbf{Ver}(pk, m, \sigma) := 1 \quad \text{if and only if } m' = m.$$

The verification algorithm always accepts valid signatures because, by Proposition 8.1, the permutations $\mathrm{RSA}_{(n,d)}$ and $\mathrm{RSA}_{(n,e)}$ are inverses of each other. However, plain RSA signatures are completely insecure and are vulnerable even to key-only attacks. To see this observe that an existential forgery is obtained by picking $\sigma \leftarrow \mathbb{Z}_n$ and computing $m := \sigma^e \bmod n \in \mathbb{Z}_n$. The pair (m, σ) is then accepted as valid by the verification algorithm.

Another attack against plain RSA signatures, similar to one we have seen against plain RSA encryption, exploits the malleability of plain RSA or, more precisely, the multiplicative property. This attack is more dangerous than the preceding one because, although it is a message attack that assumes a more powerful adversary, it also achieves a stronger goal, namely, a selective forgery. Indeed, if the adversary wants to forge a signature on a message $m \in \mathbb{Z}_n$, then it simply chooses two messages $m_1, m_2 \in \mathbb{Z}_n$ that are different from m and satisfy $m_1 m_2 \bmod n = m$ (it is clear that this can be done efficiently). Then it submits m_1 and m_2 to the signing oracle, obtaining pairs (m_1, σ_1), (m_2, σ_2). Then, because of RSA's multiplicative property, it is clear that if $\sigma := \sigma_1 \sigma_2 \bmod n$, the pair (m, σ) is a forgery.

Each of these two attacks shows that plain RSA signatures are not secure in the sense of Definition 9.3. More generally, signature schemes based on the trapdoor permutation paradigm mentioned above are also insecure because the first of these two attacks also applies to them.

Exercise 9.1 Define plain Rabin signatures in a way similar to plain RSA signatures but using the Blum–Williams trapdoor permutations (introduced in Sect. 3.3.3 and used for Rabin encryption in Sect. 8.4.1) instead of the RSA permutations. Show that

a chosen message attack gives a total break of the scheme and allows an adversary to obtain the private key.

9.2.2 Elgamal Signatures

The Elgamal signature scheme is similar to the Elgamal encryption scheme. It was introduced in 1985 and it was the first signature scheme based on the discrete logarithm problem. It can be defined on more general groups but we now give the original version, which is as follows:

Definition 9.5 The *Elgamal signature scheme* (**Gen**, **Sign**, **Ver**) is defined by the following algorithms:

- **Gen**: On input 1^t generate a random prime p of length t and a primitive root modulo p, g, so that $\mathbb{Z}_p^* = \langle g \rangle$. Choose a random $x \leftarrow \mathbb{Z}_{p-1}$ and compute $y := g^x \bmod p$. The public key is the 3-tuple $pk := (p, g, y)$ and the private key is $sk := (p, g, x)$. p and g may also be regarded as system parameters, in which case the public key is simply y and the private key is x.
- **Sign**: On input a private key $sk = (p, g, x)$ and a message $m \in \mathbb{Z}_p$, choose a random element $k \leftarrow \mathbb{Z}_{p-1}^*$ and compute

$$r := g^k \bmod p, s := k^{-1}(m - rx) \bmod (p - 1).$$

  The output is then the signature $\sigma = (r, s)$.
- **Ver**: On input a public key $pk = (p, g, y)$ and a signed message (m, r, s) check whether

  – $0 < r < p$,
  – $g^m \equiv y^r r^s \pmod p$.

  Output 1 if these conditions hold and 0 otherwise.

Let us check that the verification algorithm accepts all the signatures output by the signing algorithm. We have

$$y^r r^s \equiv (g^x)^r (g^k)^s \equiv g^{rx \bmod p-1} g^{kk^{-1}(m-rx) \bmod p-1} \equiv g^m \pmod p$$

(where we can reduce all exponents modulo $p - 1$ because $p - 1$ is the order of $\mathbb{Z}_p^*$).

It is easy to see that being able to solve the DL problem in the group $\mathbb{Z}_p^*$ leads to a total break of this scheme because in that case the private key x can be computed from the public key. This, however, is secondary here because the scheme is insecure anyway. An existential forgery under a key-only attack may be constructed as follows:

- The adversary chooses two integers u, v such that $\gcd(v, p-1) = 1$.
- The adversary computes:

$$r := g^u y^v \bmod p, s := -rv^{-1} \bmod (p-1), m := su \bmod (p-1).$$

- The adversary outputs (r, s) as a signature of m.
- (m, r, s) is accepted by the verification algorithm because

$$y^r r^s \equiv y^r g^{su \bmod (p-1)} y^{sv \bmod (p-1)} \equiv y^r g^m y^{-r} \equiv g^m \pmod p.$$

9.3 The "Hash-then-Decrypt" Paradigm

The RSA and Elgamal signature schemes we have just seen are highly insecure because they are vulnerable to key-only attacks. There is a generic method to turn these and other insecure schemes into schemes in which these straightforward attacks no longer work and hence which are, in principle, more secure—although the security of many of them has not been proved. The basic idea is to use the so-called "hash-then-decrypt paradigm", namely, instead of directly applying a decryption algorithm such as RSA to the message in order to obtain the signature, the message m is first hashed by means of a hash function H and then the decryption function is applied to $H(m)$ in order to obtain the signature of m.[1]

We will return to the security aspect in a moment but let us first mention that this procedure has some other advantages that have also contributed to this methodology being adopted in practice for several standards. One of them is that messages are usually not elements of the domain of the decryption function and they may be bit strings too long to be injectively mapped into that domain. The use of a hash function that maps strings of arbitrary length to strings of a short fixed length—which can be converted to elements in the decryption function domain—is very convenient because it allows large messages to be signed in one pass. Another clear advantage is efficiency. Hash functions such as SHA-256 are much more efficient than decryption functions such as the RSA function and, once the message is hashed, the decryption function only has to be applied once, independently of message length.

[1] Observe that the term 'decrypt' must not be taken literally here and that this terminology is often used even when the signing algorithm is not a decryption algorithm like in the RSA case; an alternative terminology is "hash-then-sign" and we may also speak of "hash-then-invert" when—as happens with RSA—signing is carried out by applying the inverse of a (candidate) trapdoor permutation.

9.3.1 Hashed RSA Signatures

A hashed version of RSA signatures is obtained by adding to the public key—or as a system parameter which is publicly known—a hash function $H : \{0, 1\}^* \to \mathbb{Z}_n$, where n is the modulus of the RSA public key (alternatively, one may consider a family of hash functions indexed by the RSA moduli and such that the function corresponding to the modulus n maps $\{0, 1\}^*$ to $\mathbb{Z}_n$). The scheme is then similar to the one in Definition 9.4, except that the signing and verification algorithms now act as follows:

- **Sign**: On input a private key $sk = (n, d)$ and a message $m \in \mathbb{Z}_n$, compute the signature:
$$\sigma := \mathbf{Sign}(sk, m) = H(m)^d \bmod n \in \mathbb{Z}_n.$$
- **Ver**: On input a public key $pk = (n, e)$, a message $m \in \mathbb{Z}_n$, and a signature $\sigma \in \mathbb{Z}_n$, check whether
$$H(m) = \sigma^e \bmod n \in \mathbb{Z}_n$$
and set $\mathbf{Ver}(pk, m, \sigma) := 1$ if this condition holds and $\mathbf{Ver}(pk, m, \sigma) := 0$ otherwise.

We see that this scheme is very efficient—assuming that H is efficient—for it only has to compute a hash operation and a modular exponentiation for both signature and verification.

Let us now go back to security. A natural question is then: what have we gained, from a security standpoint, by using the hashed version of these schemes? We can get some insight by looking, for example at the two attacks against plain RSA described above. The first one consisted simply of picking $\sigma \leftarrow \mathbb{Z}_n$ and computing $m := \sigma^e \bmod n \in \mathbb{Z}_n$ to obtain a valid message/signature pair (m, σ). If an adversary tries to repeat this key-only attack for the hashed version of RSA signatures, it runs into trouble because in order to obtain a valid message/signature pair starting with $\sigma \in \mathbb{Z}_n$, it now has to find a message $m \in \mathbb{Z}_n$ such that

$$H(m) = \sigma^e \bmod n.$$

If we assume that the hash function H is preimage resistant, the forger will be unable to find such an m with non-negligible probability and the attack will be unsuccessful.

Let us now consider the chosen message attack against plain RSA signatures that allows the forger to obtain a selective forgery. The adversary chooses two messages $m_1, m_2 \in \mathbb{Z}_n$ that are different from the message m it wants to sign and satisfy $m_1 m_2 \bmod n = m$ and queries the signing oracle about these messages, obtaining two valid message/signature pairs (m_1, σ_1), (m_2, σ_2). But now it is no longer true that $\sigma := \sigma_1 \sigma_2 \bmod n$ is a valid signature for m because we have that

$$\sigma_1^e \sigma_2^e \bmod n = H(m_1) H(m_2) \bmod n,$$

and hence, for σ to be a valid signature of m it would be necessary that

$$H(m_1)H(m_2) \bmod n = H(m_1m_2) \bmod n.$$

But any 'good' hash function is expected not to be multiplicative and hence this equality does not hold except perhaps with negligible probability. Thus the chosen message attack does not apply to the modified signature scheme.

Finally, one should also bear in mind that if an adversary is able to find a collision for the hash function H, say messages m_1, m_2 such that $m_1 \neq m_2$ and $H(m_1) = H(m_2)$, then it is also able to build a forgery under a chosen message attack. Indeed, the adversary just queries the signing oracle on m_1 and, upon receiving σ as an answer, outputs the forgery (m_2, σ). Since the pair (m_1, σ) is accepted by the verification algorithm and $H(m_1) = H(m_2)$, so is the pair (m_2, σ). In order to prevent this attack, whose practical version was presented in Example 5.12, the hash function H must therefore be collision resistant.

Summing up these findings, we have that the hashed version of RSA signatures is, in some sense, more secure than plain RSA signatures if a collision resistant hash function is used (note that, except in pathological cases, a collision resistant hash function will also be preimage resistant, see Sect. 5.6). But this is only based on the fact that certain obvious attacks against plain RSA are no longer applicable and it is still a long way from allowing us to claim that these signatures are secure in the formal sense we have defined. Therefore, even if we use the above constructions with a hash function like SHA-256 which is thought to be collision resistant in the practical sense, we have absolutely no guarantee of security for the scheme.[2] However, this construction points the way to other constructions that, as we shall see, may be shown to be secure under suitable hardness assumptions.

9.3.2 Hashed Elgamal Signatures

As in the case of RSA, a hash function may be used together with the Elgamal signature scheme to obtain a "hashed version" in which messages are mapped by the hash function into $\mathbb{Z}_p$. We leave the details as an exercise.

Exercise 9.2 Write a detailed description of the Elgamal signature scheme that uses a hash function H.

Let us now consider the security of the hashed version of Elgamal signatures. As in the case of RSA signatures, it is easy to check that the simple key-only attack that constructs an existential forgery is no longer possible with this version. Thus the hashed version might be secure assuming that the DL problem is hard and that the parameters and keys are properly chosen but, as happens in the RSA case, this has not

[2] There is a theorem that ensures that if the underlying (not hashed) signature scheme is secure and H is collision resistant, then the hashed signature scheme is also secure, but this is hardly of any help in our examples because both plain RSA signatures and Elgamal signatures are insecure.

been proved. Moreover, in Elgamal signatures, another possible source of insecurity is the random element $k \leftarrow \mathbb{Z}_{p-1}^*$ which is sometimes called an 'ephemeral key'. It can be shown that if this element is not properly chosen, the scheme is insecure.

For example, if the adversary gains knowledge of the value of k used to sign the message m, then it can compute $rx \mod (p-1) = (H(m) - ks) \mod (p-1)$ and if $\gcd(r, p-1) = 1$, which holds with high probability, it can compute the secret key $x = r^{-1}(H(m) - ks) \mod (p-1)$ thus obtaining a total break of the scheme (if $\gcd(r, p-1) \neq 1$, then the equation $rx \equiv (H(m) - ks) \pmod{p-1}$ has several solutions and the adversary can still find the correct one by trying all of them in the equation $y = g^x \mod p$). Also, if the same value of k is chosen for two different messages $m \neq m_1$ (which happens with negligible probability if k is randomly chosen), then $r = r_1$ and the two signed messages will be (m, r, s), (m_1, r, s_1). Thus the adversary may detect that the same value of k is used and, again, it may compute it because $s - s_1 \equiv k^{-1}(H(m) - H(m_1)) \pmod{p-1}$ and hence, if $\gcd(s - s_1, p-1) = 1$ (otherwise the adversary may check all the solutions of this equation), the value of k is obtained as:

$$k = (s - s_1)^{-1}(H(m) - H(m_1)) \bmod (p-1).$$

Another thing that is very important for the security of Elgamal signatures is the check that $0 < r < p$. As Bleichenbacher observed in [28], if this check is not performed and a signature (r, s) corresponding to a message m such that $H(m) = h$ is accepted, then the adversary can forge a signature on an arbitrary message m_1 with $H(m_1) = h_1$, by carrying out the following computations:

- Set $u := h_1 h^{-1} \mod (p-1)$ (we assume here that h is invertible in $\mathbb{Z}_{p-1}$).
- Set $s_1 := su \mod (p-1)$.
- Use the Chinese remainder theorem to compute r_1 such that:

$$\left.\begin{array}{ll} r_1 \equiv ru & (\text{mod } p-1) \\ r_1 \equiv r & (\text{mod } p) \end{array}\right\} \tag{9.1}$$

Then (r_1, s_1) is accepted as a valid signature for m_1 because we have:

$$g^{h_1} \equiv g^{hu} \equiv g^{(ks+rx)u} \equiv (g^x)^{ru}(g^k)^{su} \equiv y^{ru} r^{su} \pmod{p}, \tag{9.2}$$

where all exponents may be regarded modulo $p-1$, which is the order of g. Then the congruences $r_1 \equiv ru \pmod{p-1}$ and $s_1 \equiv su \pmod{p-1}$ entail that we can replace the terms ru and su appearing in the exponents by r_1 and s_1, respectively. Similarly, the congruence $r_1 \equiv r \pmod{p}$ implies that we can replace the element $r^{su} \mod p = r^{s_1} \mod p$ of $\mathbb{Z}_p^*$, appearing in (9.2), by $r_1^{s_1} \mod p$. Thus the congruence (9.2) becomes:

$$g^{h_1} \equiv y^{r_1} r_1^{s_1} \pmod{p},$$

and the forged signature is accepted by the verification algorithm.

Exercise 9.3 Explain why the previous attack is very unlikely to be successful in case the verification algorithm adds the constraints $r < p$ and $r_1 < p$.

Exercise 9.4 Write a Maple implementation of Elgamal signatures using the code for SHA-256 given in Sect. 5.6.3.

Exercise 9.5 Using the Maple implementation of Elgamal signatures in the previous exercise, give an example in which the signature of a message is computed and then, assuming that the condition $0 < r < p$ is not checked, construct an Elgamal signature for a different message.

In [28], Bleichenbacher presented yet another, more dangerous, attack against Elgamal signatures in case the parameters are not carefully chosen. This attack is based on the Pohlig–Hellman algorithm that, as we have seen, gives an efficient solution to the DL problem in groups of smooth order, i.e., in groups whose order is a product of small prime factors. For this reason p should be chosen so that $p - 1$ has at least one large prime factor. However, Bleichenbacher's attack shows that this is not sufficient as it applies in case $p - 1 = bw$ with b smooth and a relation of the form $g^t \equiv cw \pmod{p}$ with $0 < c < b$ is known. We refer to the original paper for the details and here we merely point out that attacks like this one led to the introduction of signature schemes that are similar to Elgamal—and hence are also based on the DL problem—but are immune to these attacks because they work in a subgroup of (large) prime order of $\mathbb{Z}_p^*$; we will look at a couple of these schemes in the next section. Another drawback of Elgamal signatures, namely their relatively large size, is also avoided by these schemes.

Exercise 9.6 Suppose that $p \equiv 1 \pmod 4$ and the Elgamal public key of a user is (p, g, y) [140]. Suppose further that $p - 1 = gw$ and computing discrete logarithms in the subgroup of $\mathbb{Z}_p^*$ of order g is easy (for example, using the Pohlig–Hellman algorithm if g is smooth). Prove that an Elgamal signature for a message m (with $h = H(m)$ if a hash function H is used) can be forged as follows:

1. Set $t := (p - 3)/2$ and $r := w$.
2. Find z such that $g^{wz} \equiv y^w \pmod{p}$.
3. Set $s := t(h - wz) \bmod (p - 1)$ and let (r, s) be the signature of m.

Indicate how this attack may be prevented.

9.4 The Digital Signature Algorithm

The Digital Signature Algorithm has a close precedent in Schnorr's signature scheme which was introduced in 1989. This scheme was motivated by the desire to obtain a version of Elgamal which produces shorter signatures and is more efficient because the exponents in the modular exponentiations are less than q, where q is a prime divisor of $p - 1$ of much smaller size: for example, p can be a 1024-bit prime and

q a 160-bit prime. Signatures have size $2 \cdot \text{len}(q)$, which is 320 bits in the case we have mentioned, and computations are carried out in the subgroup of order q of $\mathbb{Z}_p^*$ consisting of the identity and the elements of order q; all of the latter are generators of this subgroup. Thus the scheme is much more efficient than Elgamal since modular exponentiations with a 160-bit exponent are much faster than those with a 1024-bit exponent. On top of all this, attacks like Bleichenbacher's smoothness-based one are no longer effective against this scheme.

Exercise 9.7 Obtain an estimate of the average ratio between the time required by an exponentiation modulo (a 1024-bit prime) p with a 1024-bit exponent and another one with a 160-bit exponent. Use Maple to perform an experiment that computes the average ratio for a number of exponentiations of this kind, with pseudo-randomly chosen exponents and bases, and compare the result to the previous estimate.

9.4.1 The DSA Signature Scheme

Schnorr's signatures were the precursor of the very similar *Digital Signature Algorithm* (DSA) that, after being first proposed in 1991, was adopted by NIST in the *Digital Signature Standard* [75] and, because of this, became very popular. The DSA uses as "domain parameters" (or system parameters) two primes p, q, which describe the group used by the scheme, as well as a generator of this group. Thus the domain parameters are defined by a triple (p, q, g), where p is an L-bit prime, q an N-bit prime such that $q|p-1$, and g an element of $\mathbb{Z}_p^*$ of order q (so that g is a generator of the unique subgroup of order q of $\mathbb{Z}_p^*$). The values of L and N must correspond to one of the following pairs:

$$(L, N) = \begin{cases} (1024, 160) \\ (2048, 224) \\ (2048, 256) \\ (3072, 256) \end{cases} \tag{9.3}$$

The scheme also makes use of a hash function which must be one of the NIST-approved functions specified in [74], namely, either SHA-1 or one of the SHA-2 variants, SHA-224, SHA-256, SHA-384 or SHA-512. If a pair (L, N) as above is chosen, then either SHA-1 if $N = 160$ or some SHA-xyz, where $xyz \geq N$ (often $xyz = N$) will be used (see [74] for more detail and alternatives). If $xyz > N$, then the output of H will be truncated to include only the leftmost N bits and reduced modulo q. This way, H can be regarded as a function from $\{0, 1\}^*$ to $\mathbb{Z}_q$.

In [75], methods to choose the domain parameters are also specified. We will give more details in our Maple implementation below but, for now, we just remark that the prime q is generated first and then appropriately sized pseudo-random integers p such that $q|p-1$ are generated with the help of the hash function H, until one is found which is prime. The generator g is obtained from an element of $\mathbb{Z}_p^*$ which

is raised to the $(p-1)/q$th power. Since $g^q = 1$, the order of the element g thus obtained divides q by Proposition 2.4 and one only has to ensure that $g \neq 1$ for g to have order q.

Given these parameters, the DSA algorithm may be described as follows:

Definition 9.6 The digital signature algorithm is defined as the signature scheme

$$\mathrm{DSA} = (\mathbf{Gen}_{\mathrm{DSA}}, \mathbf{Sign}_{\mathrm{DSA}}, \mathbf{Ver}_{\mathrm{DSA}}),$$

with the following algorithms:

- $\mathbf{Gen}_{\mathrm{DSA}}$: On input the system parameters (p, q, g), pick a (pseudo-) random $x \leftarrow \mathbb{Z}_q$ and compute $y := g^x \bmod p \in \mathbb{Z}_p^*$. Output (x, y), where x is the private key and y is the public key.
- $\mathbf{Sign}_{\mathrm{DSA}}$: On input a private key $x \in \mathbb{Z}_q$ and $m \in \{0,1\}^*$, proceed as follows:
  - Choose $k \leftarrow [1, q-1]$.
  - Compute $r := (g^k \bmod p) \bmod q$.
  - Compute $z := H(m)$.
  - Compute $s := k^{-1}(z + xr) \bmod q$.

  The signature of m is the pair (r, s). If either $r = 0$ or $s = 0$, a new value of k is generated and the signature is recalculated (this occurs with very small probability).
- $\mathbf{Ver}_{\mathrm{DSA}}$: On input a public key $y \in \mathbb{Z}_p^*$ and a signed message (m, r, s), proceed as follows:
  - Check that $0 < r < q$ and $0 < s < q$. If either of these conditions does not hold then output 0.
  - If the previous conditions hold, compute:
    $w := s^{-1} \bmod q$.
    $z := H(m)$.
    $u1 := wz \bmod q = s^{-1} \cdot z \bmod q$.
    $u2 := wr \bmod q$.
    $v := (g^{u1} y^{u2} \bmod p) \bmod q$.
  - If $v = r$ then output 1, otherwise output 0.

Remarks 9.2

1. In our definition of digital signature scheme (Definition 9.1) there were no domain parameters (or, rather, they were regarded as belonging to the public key). For Definition 9.6 to conform to the more general Definition 9.1, these parameters should be generated by the key generation algorithm and included in the public key.
2. In [75] the output length of the hash function H is allowed to be different from N and, if this length is greater than N, then the value z in the above definition is the one defined by the N leftmost bits of $H(m)$. However, using a hash function

H of output length greater than N requires an explicit agreement between the participating entities and, on the other hand, the use of a hash function with output length less than N reduces the security strength of the algorithm and is not recommended. Thus the definition above reflects typical usage and we shall follow it in the Maple implementation that follows.

The next proposition shows that signatures generated by $\mathbf{Sign}_{\mathrm{DSA}}$ are accepted by $\mathbf{Ver}_{\mathrm{DSA}}$:

Proposition 9.1 *Let (x, y) be a DSA private key/public key pair and (r, s) the signature output by $\mathbf{Sign}_{DSA}$ on input x and a message m. Then $\mathbf{Ver}_{DSA}(y, m, r, s) = 1$.*

Proof Letting $z = H(m)$ we have that $s = k^{-1}(z + xr) \bmod q$ and hence $k = w(z + xr) \bmod q$, where $w = s^{-1} \bmod q$. Thus we have:

$$v = (g^{wz \bmod q} y^{wr \bmod q} \bmod p) \bmod q = (g^{wz \bmod q} g^{wxr \bmod q} \bmod p) \bmod q$$
$$= (g^{w(z+xr) \bmod q} \bmod p) \bmod q = (g^k \bmod p) \bmod q = r. \qquad \square$$

9.4.2 DSA Security

Let us now briefly review the security aspects of DSA. The initial proposal was criticized, among other things, because the selection process by NIST was not public and the standard was developed in collaboration with the NSA in a too-secretive way. This is reminiscent of the criticism against DES and, also in this case, there was a serious security concern related to parameter size, specifically concerning the size of the prime p that initially was set to 512 bits and considered insufficient. This size was later revised and, as indicated above, several modulus sizes are now available, the least of them being 1024 bits. It is clear that with a 512-bit size the scheme would be easily broken today. Indeed, as we have mentioned in Chap. 6, a discrete logarithm in $\mathbb{Z}_p^*$, for a 530-bit prime p was already computed in 2007 by means of the NFS algorithm [112]. Being able to compute discrete logarithms in the subgroup of order q of $\mathbb{Z}_p^*$ gives a total break of the scheme as it allows the adversary to compute the private key x from the public key y, so it is clear that a 1024-bit size for p is the minimum required to have a reasonable security margin in relation to this attack.

When trying to evaluate the security of DSA against attacks based on computing discrete logarithms, the size of p is not the only thing that should be taken into account. Indeed, all DSA operations take place in the subgroup of order q of $\mathbb{Z}_p^*$ and so, if we want to solve a discrete logarithm in this subgroup we have, in principle, two approaches available. The first of them is to compute the discrete logarithm in $\mathbb{Z}_p^*$, a group for which the index calculus methods are available (and were used, for example, in the 530-bit record mentioned above). For a 1024-bit prime p it is estimated that solving discrete logarithms in $\mathbb{Z}_p^*$ requires a number of operations of the order of 2^{80} which is still regarded as infeasible although not by a wide margin. The second approach is to compute the discrete logarithm working directly in the subgroup of

order q of $\mathbb{Z}_p^*$ and, to evaluate the cost of this attack, the relevant parameter is the size of the prime q. Fortunately for the DSA user, the only DL algorithms that work directly in this smaller subgroup are the generic ones which, as we have seen in Chap. 6, run in fully exponential time in contrast to the subexponential time of index calculus. The number of operations required by the generic methods is of the order of the square root of q and so, for q a 160-bit prime, which is the recommended size when p is a 1024-bit prime, we obtain that this attack would also require about 2^{80} operations. Thus, in the concrete case in which $(L, N) = (1024, 160)$, both approaches present approximately the same degree of difficulty and the remaining approved values of (L, N) are also obtained by trying to balance the algorithm strength against both methods, as well as against attacks on the hash function H. When $(L, N) = (1024, 160)$ the hash function of choice is SHA-1 which, as it has a 160-bit length, also requires about 2^{80} operations to find a collision due to the birthday paradox. In fact, as we have mentioned in Chap. 5, the 'Chinese attacks' against SHA-1 require only about 2^{63} operations to find a collision and, from this point of view, the weakest link in the (1024, 160)/SHA-1 combination is, precisely, the hash function SHA-1.

From these remarks it is clear that a more reasonable choice from the security point of view is the (2048, 224)/SHA-224 combination and the remaining stronger ones. We use $(L, N) = (2048, 256)$ together with SHA-256 in the Maple implementation that follows. Unless some new attacks are discovered—which is certainly possible—this combination seems safe for years to come.

Apart from preventing DL attacks and attacks against the hash function, other practical aspects must also be taken in consideration when implementing DSA, in order to prevent certain specific attacks. For example, reuse of the 'ephemeral key' k is strictly forbidden for the same reasons explained when discussing hashed Elgamal signatures.

Summarizing the previous discussions, we can say that DSA looks, for now, secure, because no feasible attacks against it have been discovered. There is, however, no security reduction for DSA which, even if not giving an absolute guarantee of security in the practical sense, would provide additional assurance. As on other similar occasions, we must warn that the fact that no decisive attacks have been found so far does not offer any guarantee for the future.

To close these remarks we mention that Pointcheval and Stern proved that a variant of Elgamal signatures with a randomly chosen generator g is UF-CMA secure assuming that the hash function H is a random oracle (so that this is a proof in the random oracle model). This proof, however, does not extend to the original Elgamal nor to DSA, see [155] details. We will consider later on some other UF-CMA secure signature schemes.

9.4.3 DSA in Maple

We now give a Maple implementation of DSA following the guidelines in [75]. There are too many details in this standard to implement them all but we shall try to give a reasonable approximation to the algorithm. We start with a function that generates the system parameters (or 'domain parameters', as they are called in [75]). They are contained in the triple (p, q, g)—in the notation of Sect. 9.4.1—which describes the group in which the algorithm operates, so this function will play a role similar to the one played by the function `GroupGen` we used when implementing the Cramer–Shoup encryption scheme in Sect. 8.6.2. The parameters generated by this function may subsequently be shared by many users who only have to generate their keys by means of another function which is much less computing-intensive. In [75] two additional domain parameters are considered which contain the seed and the counter used to generate p and q. These parameters are only used for validation of the primes p and q, and they allow another party to check that p and q were obtained by a method specified in the standard. However, these parameters are optional and we will not include them as inputs in our functions.

The main input for the function that generates the domain parameters is the pair (L, N) described in Sect. 9.4.1, which specifies the bit lengths of the primes p and q, respectively. This pair must be one of the four approved pairs in the standard but we will only allow the last two, which provide greater security strength. Following the recommendations in [75], the hash function to be used should have an output length of at least 256 bits in our case, and we will use SHA-256—which was implemented in Sect. 5.6.3—for this purpose.

Another required input for the group generating function in [75] is the length of the seed, which is required to be $\geq N$. We shall take this length to be equal to N, so this input parameter will not be necessary. The seed itself will be pseudo-randomly generated inside the procedure as a hexadecimal string of hexadecimal length $N/4$ and we will use one of Maple's built-in pseudo-random algorithms to do the job, without caring much about its cryptographic strength because the generated parameters are going to be public. The function that generates the domain parameters is the following:

```
> DSADomainGen := proc(L::{2048, 3072}:= 2048, N::{256}:= 256, {H::name:='SHA256',
                       outlen::posint:= 256, format::identical(hex, decimal):= hex})
  local prime, found, seed, n, b, U, q, offset, counter, W, j, V, X, c, p, e, t, g;
  prime := false;
  found := false;
  while not found do
    while not prime do
      StringTools:-Randomize();
      seed := StringTools:-Random(N/4, xdigit);
      n := ceil(L/outlen)-1;
      b := L-1-n*outlen;
      U := OS2IP(H(seed, hex)) mod 2^(N-1);
      q := 2^(N-1)+U+1-(U mod 2);
      prime := isprime(q)
    end do;
    offset := 1;
    prime := false;
    for counter from 0 to 4*L-1 while not prime do
```

```
      W := 0;
      for j from 0 to n-1 do
        V := OS2IP(H(I2OSP((OS2IP(seed)+offset+j) mod 2^N), hex));
        W := W+V*2^(j*outlen)
      end do;
      V := OS2IP(H(I2OSP((OS2IP(seed)+offset+n) mod 2^N), hex)) mod 2^b;
      W := W+V*2^(n*outlen);
      X := W+2^(L-1);
      c := X mod (2*q);
      p := X-c+1;
      if 2^(L-1) < p then
        prime := isprime(p)
      end if;
      offset := offset+n+1
    end do;
    if prime then
      found := true
    end if
  end do;
  e := (p-1)/q;
  g := 1;
  RandomTools:-MersenneTwister:-SetState();
  while g = 1 do
    g := Power(RandomTools:-MersenneTwister:-GenerateInteger(range = 2.. p-2), e) mod p;
  end do;
  if format = decimal then
    [p, q, g]
  else
    StringTools:-LowerCase~(convert~([p, q, g], hex))
  end if
  end proc:
```

This function closely follows the steps in [75, A.1.1.2], where the generation of the primes is described, and [75, A.2.1], where the generation of the generator g is given, and, as usual, it uses other previous functions, including `SHA256` and its auxiliary functions from Sect. 5.6.3. We use Maple's function `isprime` which, as already mentioned, combines Miller–Rabin's test with a Lucas test, in line with the recommendations in [75, Appendix C3]. The required input parameters of the function are `L` and `N`, where the lengths of the primes are specified, with default values 2048 and 256, respectively. The optional keyword parameters are `H`, used to specify the hash function (with `SHA256` as default), `outlen` for the output length of the hash function (with 256 as default) and, finally, `format`, where the output format is specified (either `hex` or `decimal`, with the former as default). The output is a 3-element list $[p, q, g]$ containing the primes and the generator of the group.

Remark 9.1 `DSADomainGen` will automatically use the value $N = 256$ and hence the input parameter `N` is not necessary and could be replaced by setting it to 256 in the body of the function. However, we let it stand because this makes it easy to modify the function to accept other values for N by just adding them to the specifying set. For example, writing `N::{224, 256} := 256` in the parameter declarations of the function, the values 224 and 256 will be allowed, with the latter being the default. This would permit, for example, the use of $(L, N) = (2048, 224)$ together with either SHA-256 or SHA-224 (implementing the latter requires just a trivial modification of the SHA-256 code, consisting of a change of the 'initial hash values' and truncating the final hash value to the leftmost 224 bits).

Exercise 9.8 Modify the function `DSADomainGen` in order to allow it to use any one of the (L, N)-pairs (2048, 224), (2048, 256), (3072, 256).

Example 9.1 Let us build a set of DSA domain parameters with default values so that, in particular, $(L, N) = (2048, 256)$ and $H =$ SHA-256. This can be done as follows:

```
> dparams := DSADomainGen();
  ["97343aafd2a22338b609478ebf53dcd8f5c0f5846ee5b65cb2f9d38cbaa64a033ab41c012223d6\
  2cbb8880c92622087b2891ad1f12a935c38f13b3496a37342f92cd0c90c15de39811aec22ff9957b\
  e5bb612fbb0defd16dc2077de86546cfbaa7e075adbe2a745dc4fe21798bffd5b21bce2fde046bde\
  05e4ce2a89208f0511eba8b04f6eae818bd45cc4697226776a3aa7dd8f3fd19965aa548a022cc468\
  73851866bbf8fdcf6ed30d40eb0dd52bf8fe35283b0547c5afff2c0dc787023ea2f5a1671cdad08e\
  7e347582f99e4c3b4e9ad12384d79c3c5eeb02b29c016d324e4961563e61e30f4c4adaa95fdd15af\
  91e84a4a3f90704a2403d707a53237cb73",
  "9eadd47526296e2a8d4d16e8b868724469da2fba75dee074d76bda0beca22a07",
  "8d96001580b73d3877110e57d7f4cbf196e0dfe720a19433bc2f56caecaac9b42e5122d5fdc6a33\
  9e1d7f1f37afaedac86a586e158c3dfbb4b0fde6d4d798c27fc09970750cfe99e8ba2cafe37818d5\
  39f6e3211c49e7e30d79c3754dae0367cae4a2f5280a81eba0aee69a2b5bc2d084437af63337d8c5\
  43c554c2249f77637928290d34f97a4ea6a4c99968e98868a95719986eb5972e97af9a9dbc9b6a29\
  803c8d9800b7ee6f81d696aaf3577d69e160b29de1bd83f8c0fff64dbdd5d8af40881cb18f32eaa6\
  6eb72db1047f726ebb0248711c053ccfb8b2253649612249dfed7efb0218772cf4d7bb541bc03894\
  0b370db5e5f3169db5708b57b5ab43969"]
```

Next we give the key generation function. The private key is going to be pseudo-randomly generated by this function and the PRG should be seeded with an externally generated random seed. In [75] it is required that the 'security strength' of the PRG is at least equal to the one of the (L, N) pair used which, for our selected values, is 128 bits according to [11].[3] Thus our seed should have at least 128 bits. Moreover, the specification requires one of the NIST-approved PRGs but we shall simply use Maple's Blum–Blum–Shub PRG, as we have usually done when a cryptographically strong PRG is required. The input parameters are `domain`, for the list of domain parameters, `seed` for the random seed, `bbslength` to select the length of the primes to be used by Blum–Blum–Shub and `format` to select the output format. The output is a list `[x, y]`, where `x` is the private key and `y` is the public key.

```
> DSAKeyGen := proc(domain::list, seed::{posint, string},
            {bbslength::{512,768,1024}:= 1024, format::identical(decimal,hex):= hex})
  local d, p, q, g, s, B, L, N, c, x, y;
  d := stringposint~(domain);
  p := d[1];
  q := d[2];
  g := d[3];
  L := intlog[2](p)+1;
  N := intlog[2](q)+1;
  if L <> 2048 and L <> 3072 or N <> 256 then
    error "(L,N) pair not supported"
  end if;
  s := stringposint(seed);
  B := RandomTools:-BlumBlumShub:-NewBitGenerator(s, primes = bbslength);
  c := q-1;
```

[3] We are not going to elaborate on the details of these 'security strengths' and refer the reader to [11] instead.

```
  while q-2 < c do
    c := convert(cat(seq(B(), i = 1 .. N)), decimal, binary)
  end do;
  x := c+1;
  y := Power(g, x) mod p;
  if format = decimal then
    [x, y]
  else
    StringTools:-LowerCase~(convert~([x, y], hex))
  end if
  end proc:
```

Let us now generate a DSA private key/public key pair corresponding to the previously generated domain parameters contained in the global variable `dparams`, using an externally supplied 128-bit random seed:

```
> dsakey := DSAKeyGen(dparams, "dc8f0b897e0b9a235ee74551601c69c8");
["35a1df865d99c9b4ecc409601d4030c599471418067478841c62e88f10e5791e",
  "6d9b54150b12e336cc99fc49550ce9128b93730ae7a5b10664bd880e67bb9570e2bd0e513b559c5\
  b39faf8b45e3853bd14a812069e5a7ff0c181331db9e2d8b720b0fbdd61b6aea8e3bfde54a9b1630\
  822eb43840c8f510397fc68ab8f4aa074a26668508f3c3fc2724dd4d0276ea4848f653b44663f940\
  723da4893dc808a6fa6959e48a9e67bc6bb75d01528654eca9b5971eb327cfdd8182df1a7edb1ac7\
  0f395a6335658d7c9c79dfb9314414e009aa9f369914925f5f20d0fc9f70ab4f1ef666dcc4d3f4cb\
  f5941b0285f4916890870406ddc32b02a80efc4f44aa2eb6a64b271e1edcf4e1ce97949da279d7d1\
  7ce474c1fce42af07d04e88c67b3fe42d"]
```

Before proceeding further, we next briefly consider the problem of checking whether the domain parameters and the keys generated by the preceding functions are correct. For this purpose, we give a function that is simpler than the validation procedure established in [75]. The latter uses optional domain parameters—which we have not implemented—to check that the method indicated in [75] has been followed. Here we only check the (probable) primality of p and q, the length of p and q, that q divides $p-1$ and that g is a generator. We also omit all checks that (L, N) is a valid pair because `DSADomainGen` only allows $(L, N) = (2048, 256)$ or $(L, N) = (3072, 256)$, but we remark that these checks are easy to add to the function. The input parameters are `domain`, used to specify the list of domain parameters to be tested, `L` and `N`. The output is either an error message or a message stating that the parameters are valid.

```
> DSADomainTest := proc(domain::list, L::posint, N::posint)
  local d, p, q, g;
  d := stringposint~(domain);
  p := d[1];
  q := d[2];
  g := d[3];
  if not (isprime(p) and isprime(q)) then
    error "p and q must be prime"
  end if;
  if ilog2(p) <> L-1 or ilog2(q) <> N-1 then
    error "incorrect prime length"
  end if;
  if p mod q <> 1 then
    error "q does not divide p-1"
  end if;
  if g = 1 or p < g or Power(g, q) mod p <> 1 then
    error "incorrect value of g"
  end if;
  printf("Valid parameters")
  end proc:
```

For example, testing the parameters `dparams` generated above gives:

```
> DSADomainTest(dparams, 2048, 256);
Valid parameters
```

Exercise 9.9 Write a variant of the function `DSADomainGen` that additionally outputs the seed and the counter values, and a function to validate the primes generated by this variant using the method indicated in [75, A.1.1.3].

Exercise 9.10 Write a variant of `DSADomainGen` that generates the generator g by the verifiable method described in [75, A.2.3]. Write a function that validates the generators thus obtained following the method described in [75, A.2.4].

Exercise 9.11 Write a function `DSAKeyTest` which, on input a key pair $[x, y]$ and a valid list of domain parameters, checks whether the key pair is valid for the given domain (the function should check that both parameters are in the appropriate ranges and also that $g^x \equiv y \pmod{p}$). Use this function to check the previously generated key pair `dsakey`.

We are now ready to give the functions that implement the signing algorithm and the verification algorithm. For the first, it is necessary to generate an ephemeral key k (or 'per-message secret number' in the terminology of [75]). This number is, as we have seen in our description of DSA, of the utmost importance for the security of the scheme and hence it should either be randomly chosen or, at least, a strong PRG with a random seed of sufficient length should be used. We shall use the Blum–Blum–Shub PRG as usual and the function is very similar to the key-generating function, having the same input parameters with the exception of `format`, which is not necessary in this case because the function will only be used inside the signing function and its output will be in decimal format. The output is a list $[k, invk]$ containing the secret number and its inverse modulo the prime q.

```
> DSASecretGen := proc(domain::list, seed::{posint, string},
                       {bbslength::{512, 768, 1024} := 1024})
  local d, p, q, g, s, B, L, N, c, k, invk;
  d := stringposint~(domain);
  p := d[1];
  q := d[2];
  g := d[3];
  L := intlog[2](p)+1;
  N := intlog[2](q)+1;
  if L <> 2048 and L <> 3072 or N <> 256 then
    error "(L,N) pair not supported"
  end if;
  s := stringposint(seed);
  B := RandomTools:-BlumBlumShub:-NewBitGenerator(s, primes = bbslength);
  c := q-1;
  while q-2 < c do
    c := convert(cat(seq(B(), i = 1 .. N)), decimal, binary)
  end do;
  k := c+1;
  invk := k^(-1) mod q;
  [k, invk]
  end proc:
```

The next function implements the DSA signing algorithm. The required input parameters are `domain`, used to specify the list of domain parameters,

`privatekey`, for the private key given either as a positive integer or a hexadecimal string, `seed`, used to specify a random seed required for the generation of the 'per-message secret number' and, finally, `message`, for the message to be signed given either as a hexadecimal string or as an ordinary text string (i.e., an ASCII-8 string without null characters). The optional keyword parameters are `messagetype`, used to specify whether the message is hexadecimal or ordinary text and `H` for the hash function. The output is the signature given as a two-element list $[r, s]$, where r and s are hexadecimal strings.

```
> DSASign := proc(domain::list, privatekey::{posint, string}, message::string,
  seed::{posint,string}, {messagetype::identical(hex,text):= hex, H::name:='SHA256'})
  local r, d, k, p, q, g, sec, z, invk, x, s;
  d := stringposint~(domain);
  x := stringposint(privatekey);
  p := d[1];
  q := d[2];
  g := d[3];
  r := 0;
  while r = 0 or s = 0 do
    sec := DSASecretGen(domain, seed);
    k := sec[1];
    invk := sec[2];
    r := (Power(g, k) mod p) mod q;
    z := OS2IP(H(message, messagetype));
    s := invk*(z+x*r) mod q
  end do;
  I2OSP~([r, s])
  end proc:
```

Remark 9.2 The random seed is necessary because, as mentioned when discussing the security of Elgamal signatures, the value of k must remain secret in order to prevent forgeries. Moreover, it is also important that the values of k used for different messages are not repeated. Because of these requirements, a random seed of at least 128 bits should be used.

Exercise 9.12 Modify the function `DSASign` so that it does not need an external seed supplied as input. The seed can be automatically generated within the procedure by one of the methods used by Maple for this purpose but one should bear in mind that the use of automatic seeding will make the scheme much less secure.

Exercise 9.13 The function `DSASign` outputs the signature as a list $[r, s]$ where r and s are given as hexadecimal strings for convenience and brevity. Modify the function in order that it gives the choice to output the signature as a list of two decimal numbers.

We next give the function that implements the DSA verification algorithm. The required input parameters are `domain`, `publickey`, `message` (all of which are similar to those in the signing function, only replacing the private key by the public key) and `signature`, used to specify the signature pair in the format output by `DSASign`. The optional keyword parameters are the same as those in the signing function. The output of the function is `"Valid"` in case the signature is accepted and `"Invalid"` otherwise.

```
> DSAVer := proc(domain::list, publickey::{posint, string}, message::string,
       signature::list, {messagetype::identical(hex,text):= hex, H::name:='SHA256'})
  local d, y, sig, r, s, p, q, g, w, z, u1, u2, v;
  d := stringposint~(domain);
  y := stringposint(publickey);
  sig := stringposint~(signature);
  r := sig[1];
  s := sig[2];
  p := d[1];
  q := d[2];
  g := d[3];
  if r = 0 or s = 0 or q <= r or q <= s then
    error "invalid signature"
  end if;
  w := s^(-1) mod q;
  z := OS2IP(H(message, messagetype));
  u1 := z*w mod q;
  u2 := r*w mod q;
  v := (((Power(g, u1) mod p)*(Power(y, u2) mod p)) mod p) mod q;
  if r = v then
    "Valid"
  else
    "Invalid"
  end if
  end proc:
```

Exercise 9.14 Write a Maple procedure to test the functions `DSASign` and `DSAVer` defined above. The procedure should pseudo-randomly generate a specified number of either hex or ASCII-8 strings and, given valid DSA parameters and keys, compute the corresponding signatures and verify that these signatures are valid.

Example 9.2 Let us consider the following messages:

```
> m1 := "This is a message to test DSA signatures":
  m2 := "0123456789abcdef":
```

We are going to sign these messages and then to verify the corresponding signatures, using the previously generated domain parameters `dparams` and the DSA private key/public key pair `dsakey`. We will use the following 128-bit seeds to sign each of these two messages:

```
> snseed1 := "c63520000514b5634cec92c4117f0f96":
  snseed2 := "e0fa47e40a96a09a75856031c75e3a5b":
```

The signature corresponding to `m1` is the following:

```
> sig1 := DSASign(dparams, dsakey[1], m1, snseed1, messagetype = 'text');
    ["1f20ce2a492f6aab18dd84ce94f830a9e5fe9ffb1f4083d28d2f02b3d3daf940",
      "2920ab94e04ccf5a3be78c3d64736bd6660f9f5f52ffef00e53f0e8e1fba53d3"]
```

On the other hand, the signature corresponding to the hexadecimal message `m2` is:

```
> sig2 := DSASign(dparams, dsakey[1], m2, snseed2);
    ["8854608fc498c17b19ce7eb3c8143d33663daf33865a6dcc6b303ba74bad298b",
      "6a0e9e593373d071ace1bf9349b20861cea38f31e5426685cd69e67cb0f41051"]
```

Let us now verify these signatures:

```
> DSAVer(dparams, dsakey[2], m1, sig1, messagetype = 'text');
                                    "Valid"

> DSAVer(dparams, dsakey[2], m2, sig2);
                                    "Valid"
```

Verification is successful and the signatures are accepted.

We close this study of DSA by mentioning that there is also an elliptic curve version of the algorithm called ECDSA. The main difference with the DSA algorithm we have seen is that ECDSA operates on an elliptic curve group instead of on a subgroup of $\mathbb{Z}_p^*$. An introduction to elliptic curve cryptography is presented in Chap. 11 and ECDSA is studied in Sect. 11.4.2.

9.5 CMA Secure Signature Schemes

We are now going to look at signature schemes that, in contrast with the preceding ones, can be proved to be UF-CMA secure under reasonable assumptions and that, moreover, are reasonably efficient so that they can be used in practice. The most important of these schemes are RSA-based and, since plain RSA signatures are, as we have seen, extremely weak from the security viewpoint, we take the hashed version of RSA signatures as starting point in the search for such secure schemes.

9.5.1 FDH Signatures

The basic idea in the quest for RSA-based secure signatures is to obtain a security reduction which shows that a PPT algorithm that breaks such a signature scheme leads to an algorithm that inverts the RSA function for a randomly chosen point in its range. This will show that such an algorithm cannot exist assuming that the RSA problem is hard. However, such a proof seems difficult to obtain for the basic version of hashed RSA signatures described in Sect. 9.3.1 even assuming that the hash function used is collision resistant. One important obstacle for such a proof is the fact that this function may have an output length much smaller than the size of the RSA modulus used. SHA-256, for example, has a 256-bit output length and, if used together with a 2048-bit RSA-modulus n, then the set $S = \{\text{SHA-256}(m) | m \in \{0,1\}^*\}$ is a tiny subset[4] of $\mathbb{Z}_n$. Indeed, there are at most 2^{256} elements in S and 2^{2048} elements in $\mathbb{Z}_n$, so the probability that a randomly chosen element of $\mathbb{Z}_n$ happens to belong to S is $\leq \frac{2^{256}}{2^{2048}} = \frac{1}{2^{1792}}$. This probability is so tiny that, even assuming that the RSA function is one-way, it is perfectly possible that it could be easy to invert the function

[4] As is done in Sect. 9.3.1, we assume that H maps $\{0,1\}^*$ to $\mathbb{Z}_n$ by identifying a bit string with the integer it defines.

in the subset S, and hence to forge signatures computed with such a scheme. The reason is that a randomly chosen element of $\mathbb{Z}_n$ will almost surely fall outside S and hence whether inverting the RSA function on S is easy or not is irrelevant as far as its one-wayness is concerned.

The obstacle we have just mentioned can be removed if we additionally assume that the hash function H is ideal in the sense that it hashes strings of $\{0,1\}^*$ onto the full domain of the RSA function, $\mathbb{Z}_n$, uniformly at random. This is the same as saying that H behaves as a random oracle, i.e., that H returns a random element of $\mathbb{Z}_n$ each time it is invoked, except that if queried twice on the same message it returns the same result both times. For this, it is convenient to think of H as a process which maintains a table of input-output pairs. All parties involved in the signature scheme will have oracle access (as a black box) to the process H and, when a query about a message m is made, the process will check whether its table contains a pair (m, h)—which will happen if the query about m was already made before—and, in that case will return h. Otherwise, H picks a random $h \in \mathbb{Z}_n$, adds the pair (m, h) to the table, and returns h as its answer to the query.

By assuming that the hash function H is a random oracle onto the full domain of the RSA function we obtain a variant of hashed RSA which is UF-CMA secure assuming that the RSA problem is hard, thus obtaining a signature scheme with a security reduction in the random oracle model. This scheme is called *Full Domain Hash* and we will denote it simply by FDH (other notations which emphasize the fact that this is an RSA-based scheme are FDH-RSA or RSA-FDH). The scheme, introduced in [18] and analyzed in more detail in [20] can then be defined as follows:

Definition 9.7 Given a random hash function $H : \{0,1\}^* \to \mathbb{Z}_n$, the *FDH signature scheme* is the scheme (**Gen**, **Sign**, **Ver**) defined by the following algorithms:

- **Gen**: On input 1^k, run the RSA instance generator (Algorithm 8.1) to obtain $(n, e, d) \leftarrow \mathbf{Gen}_{\mathrm{RSA}}$. Then set $pk := (n, e)$ and $sk := (n, d)$. pk is the public key and sk is the private key, so that $(pk, sk) \leftarrow \mathbf{Gen}(1^k)$.
- **Sign**: On input a private key $sk = (n, d)$ and a message $m \in \mathbb{Z}_n$, proceed as follows:
  - Query the oracle H on m and obtain $H(m)$ in return.
  - Compute the signature:

$$\sigma := \mathbf{Sign}(sk, m) = H(m)^d \bmod n \in \mathbb{Z}_n.$$

- **Ver**: On input a public key $pk = (n, e)$, a message $m \in \mathbb{Z}_n$, and a signature $\sigma \in \mathbb{Z}_n$, proceed as follows:
  - Query the oracle H on m and obtain $H(m)$ in return.
  - Check whether

$$H(m) = \sigma^e \bmod n \in \mathbb{Z}_n$$

and set $\mathbf{Ver}(pk, m, \sigma) := 1$ if this condition holds and $\mathbf{Ver}(pk, m, \sigma) := 0$ otherwise.

Remark 9.3 The formal definition of the FDH scheme is very similar to our earlier definition of hashed RSA signatures in Sect. 9.3.1 but there is a fundamental difference in that now the hash function is a random oracle that must be queried by the signing and verification algorithms. This hypothesis is crucial for the security analysis but, in practice, the scheme may be implemented by defining H on top of some hash function like SHA-1 or SHA-256. The random oracle hypothesis no longer holds for such an implementation but the security analysis under this hypothesis gives a certain indication of security and, in particular, shows that such a concrete scheme may be secure against attacks that do not exploit specific properties of the hash function used.

In order to analyze the security of the FDH scheme, we slightly modify the signature UF-CMA experiment in Definition 9.2 to adapt it to the concrete case of FDH (in the random oracle model):

Definition 9.8 The *FDH signature unforgeability experiment under an adaptive chosen message attack* $\mathbf{Sign}^{\textit{uf-cma}}_{\mathcal{A},\mathrm{FDH}}(k)$ is the following:

1. A key pair (pk, sk) is generated by running $\mathbf{Gen}(1^k)$.
2. A random hash function $H : \{0,1\}^* \to \mathbb{Z}_n$ (for the RSA modulus n of pk) is chosen.
3. The adversary $\mathcal{A}$ is given pk and oracle access to both H and $\mathbf{Sign}(sk, -)$. $\mathcal{A}$ asks a set of queries to these oracles and outputs a message/signature pair (m, t).
4. The output of the experiment is defined to be 1 if m was not queried to the signing oracle and $\mathbf{Ver}(pk, m, t) = 1$, and 0 otherwise.

The security of the FDH scheme is given by the following theorem from [20]:

Theorem 9.1 *If the RSA problem is hard relative to* $\mathbf{Gen}_{RSA}$ *and the hash function H is modeled as a random oracle, then FDH is UF-CMA secure.*

Proof Let $\mathcal{A}$ be an FDH forger, i.e., a PPT adversary whose probability of success in the experiment $\mathbf{Sign}^{\textit{uf-cma}}_{\mathcal{A},\mathrm{FDH}}(k)$ is equal to $\varepsilon(k)$. $\mathcal{A}$ queries both the hash oracle and the signing oracle, so let $q = q(k)$ be an upper bound on the number of queries to the hash oracle. We construct a PPT algorithm $\mathcal{B}$ which is an adversary for the RSA experiment $\mathbf{RSA}_{\mathcal{B},\mathbf{Gen}_{\mathrm{RSA}}}(k)$ and tries to invert the RSA function as follows.

First, $\mathbf{Gen}_{\mathrm{RSA}}$ is run on input 1^k and an RSA key (n, e, d) is obtained, where $pk = (n, e)$ and $sk = (n, d)$. A random $y \leftarrow \mathbb{Z}_n$ is chosen and $\mathcal{B}$ is given (n, e, y) as input. $\mathcal{B}$ tries to compute $y^d \mod n$—without knowing d—by using $\mathcal{A}$ as a subroutine as follows:

- $\mathcal{B}$ picks a random integer $j \leftarrow \{1, \ldots, q\}$.
- $\mathcal{B}$ gives $\mathcal{A}$ the public key $pk = (n, e)$ and simulates both the hash oracle and the signing oracle that are queried by $\mathcal{A}$ in experiment $\mathbf{Sign}^{\textit{uf-cma}}_{\mathcal{A},\mathrm{FDH}}(k)$, answering all queries posed by $\mathcal{A}$. $\mathcal{B}$ initializes a counter i setting it to 0 and a table of elements of $\mathbb{Z}_n^3$, initially empty.

- If $\mathcal{A}$ makes a hash oracle query on a message m that was not already in a hash oracle query, then $\mathcal{B}$ does the following:
  - $\mathcal{B}$ increments i.
  - $\mathcal{B}$ sets $m_i := m$.
  - If $i = j$ then $\mathcal{B}$ sets $y_i := y$, $\sigma_i := \perp$ and returns y_i to $\mathcal{A}$.
  - If $i \neq j$ then $\mathcal{B}$ picks uniformly at random $\sigma_i \leftarrow \mathbb{Z}_n$, sets $y_i := \sigma_i^e \bmod n$ and returns y_i to answer $\mathcal{A}$'s query.
  - $\mathcal{B}$ adds the triple (m_i, y_i, σ_i) to the table.
- If $\mathcal{A}$ makes a hash oracle query about a message m that was already queried before, then $\mathcal{B}$ finds $m = m_t$ for $t < i$ within a table entry and returns to $\mathcal{A}$ the value y_t in the table entry.
- If $\mathcal{A}$ makes a signing oracle query on m then $\mathcal{B}$ proceeds as follows:
  - If a hash oracle query was not previously made on m, then $\mathcal{B}$ itself makes such a query and updates the table as before. Hence, in the following we may suppose that whenever $\mathcal{A}$ makes a signing query on m, it has already made a hash query on m and, therefore, m is equal to one of the m_i in the entries of the table.
  - If $i \neq j$ then there is an entry (m_i, y_i, σ_i) in the table and $\mathcal{A}$ returns the signature σ_i to $\mathcal{B}$.
  - If $i = j$, then $\mathcal{A}$ aborts the experiment.
- Finally, $\mathcal{A}$ outputs its forgery (m, σ). If a hash oracle query was not made by $\mathcal{A}$ on m, then $\mathcal{B}$ does so and updates the table. Hence we may assume that $m = m_t$ for some m_t in a table entry.
- If (m_t, σ) is the forgery output by $\mathcal{A}$ and $t = j$, then $\mathcal{B}$ outputs σ, otherwise the experiment is aborted.

Let us now analyze $\mathcal{B}$'s behavior and estimate its probability of success, i.e., the probability that $\mathcal{B}$ outputs a value σ such that $\sigma^e \equiv y \pmod{n}$. First of all we observe that $\mathcal{B}$ is indeed a PPT algorithm and that the initial choice of the index j made by $\mathcal{B}$ can be interpreted as $\mathcal{B}$'s guess at the index of the hash oracle query made by $\mathcal{A}$ about the message m whose signature will be output by $\mathcal{A}$ at the end. If this guess is not correct then $\mathcal{B}$ will fail, but if $\mathcal{B}$ guessed right then the view of $\mathcal{A}$ when run as a subroutine of $\mathcal{B}$ is identically distributed to the view of $\mathcal{A}$ in experiment $\mathbf{Sign}^{\textit{uf-cma}}_{\mathcal{A},\mathrm{FDH}}(k)$. This is because the hash queries made by $\mathcal{A}$ are answered by $\mathcal{B}$ with values chosen uniformly at random in $\mathbb{Z}_n$. Indeed, the query on m_j is answered by the randomly chosen y and the queries on messages m_i for $i \neq j$ are answered as $y_i := \mathrm{RSA}_{(n,e)}(\sigma_i)$ for randomly chosen σ_i. Since $\mathrm{RSA}_{(n,e)}$ is a permutation of $\mathbb{Z}_n$, these hash values are uniformly distributed in $\mathbb{Z}_n$, which matches the expected behavior of the random oracle H. In addition to all this, we also remark that $\mathcal{A}$ does not obtain any information about the value j that was randomly chosen by $\mathcal{B}$. Thus the probability that $\mathcal{A}$ outputs a valid forgery is the same as its probability of success in the experiment $\mathbf{Sign}^{\textit{uf-cma}}_{\mathcal{A},\mathrm{FDH}}(k)$ and hence is equal to $\varepsilon(k)$.

In this case, if the output of $\mathcal{A}$ is (m, σ) and $m = m_j$ we have that $\mathbf{Ver}(pk, m, \sigma) = 1$ and hence $\sigma^e \bmod n = y$, so that $\mathcal{B}$ succeeds in the $\mathbf{RSA}_{\mathcal{B},\mathbf{Gen}_{\mathrm{RSA}}}(k)$

experiment. The probability that $m = m_j$ is $1/q$ due to the random choice of j and hence the probability that $\mathcal{A}$ succeeds is $\varepsilon(k)/q(k)$. Since $q(k)$ is a polynomial function, $\mathcal{B}$ succeeds with non-negligible probability in case $\varepsilon(k)$ is a non-negligible function, and this completes the proof. □

The preceding security result has a drawback in that the reduction is not tight and leaves open the possibility that signatures can be forged with a probability q times higher than the probability of being able to invert the RSA function at a random point with comparable execution time. An attack with $q > 2^{60}$ queries to the hash oracle is conceivable and if, for example, the probability of success on inverting RSA for some given parameters were $\approx 2^{-70}$ within a given time, then the probability of breaking FDH in comparable time could be $\approx 2^{-10}$. It that case, the passage from 2^{-70} to 2^{-10} would entail a huge security loss.[5]

9.5.2 PSS Signatures

The lack of tightness in the FDH security reduction motivated the search for a scheme based on the same principles but with a much tighter security reduction. As a result, Bellare and Rogaway introduced such a scheme in [20]; the new scheme is called the *Probabilistic Signature Scheme* (PSS) and, while still following the hash-then-invert paradigm, it differs from FDH in that the signing algorithm uses a randomly chosen seed and is, therefore, probabilistic. Efficiency is similar to that of FDH but the tighter reduction implies that the probability of signature forgery is almost as low as the RSA inversion probability, regardless of the number of hash or sign oracle queries made by the adversary. PSS can be described as follows:

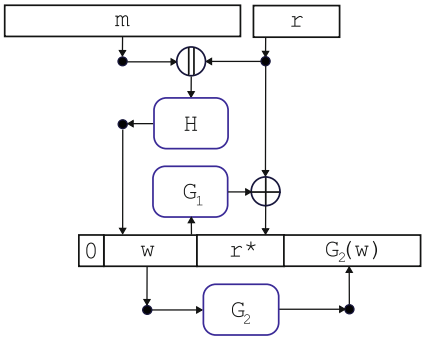
Definition 9.9 *PSS signature scheme.*

- The parameters and the **Gen** algorithm are the following:
  - PSS is parameterized by k_0 and k_1, which are integers such that $1 < k_0, k_1 < k$ and $k_0 + k_1 \leq k - 1$ where k denotes, as usual, the security parameter. Typical values for these parameters are: $k = 2048$, $k_0 = k_1 = 256$.
  - The key generation algorithm **Gen** is the same as that of FDH: $\mathbf{Gen}_{\mathrm{RSA}}$ is run to obtain $pk = (n, e)$ and $sk = (n, d)$, where $\mathrm{len}(n) = k$.
  - The signing and verification algorithms make use of two hash functions:
    $H : \{0,1\}^* \to \{0,1\}^{k_1}$ (called the *compressor*).
    $G : \{0,1\}^{k_1} \to \{0,1\}^{k-k_1-1}$ (called the *generator*). G defines functions $G_1 : \{0,1\}^{k_1} \to \{0,1\}^{k_0}$ and $G_2 : \{0,1\}^{k_1} \to \{0,1\}^{k-k_0-k_1-1}$ by setting $G(w) = G_1(w)||G_2(w)$ for $w \in \{0,1\}^{k_1}$.

[5] A tighter reduction, which depends on the number of signing oracle queries rather than on the number of hash oracle queries, was found by J.S. Coron but a still tighter reduction is desirable.

- **Sign** acts as follows on input a message $m \in \{0,1\}^*$ and a private key $sk := (n, d)$:
  - A random seed r is chosen: $r \leftarrow \{0,1\}^{k_0}$.
  - $w := H(m||r)$.
  - $r^* := G_1(w) \oplus r$.
  - $y := 0||w||r^*||G_2(w)$.
  - Output the signature of m, $\sigma := y^d \bmod n$.
- **Ver** acts as follows on input a signed message (m, σ) and a public key $pk := (n, e)$:
  - $y := \sigma^e \bmod n$.
  - y is parsed as $y = b||w||r^*||\gamma$, where $\text{len}(b) = 1$, $\text{len}(w) = k_1$ and $\text{len}(r^*) = k_0$.
  - $r := r^* \oplus G_1(w)$.
  - If $H(m||r) = w$ and $G_2(w) = \gamma$ and $b = 0$ then output $\mathbf{Ver}(pk, m, \sigma) := 1$, otherwise output $\mathbf{Ver}(pk, m, \sigma) := 0$.

A graphical depiction of PSS encoding is the following:



PSS Encoding

Remarks 9.3 We next give a brief explanation of some details in the preceding definition:

1. The PSS scheme was designed with H and G modeled as ideal hash functions, i.e., as random oracles, which makes it possible to obtain a security reduction to the RSA problem. In practical implementations, these functions are constructed from a cryptographic hash function such as SHA-256.
2. To sign a message, the signer first picks a random $r \leftarrow \{0,1\}^{k_0}$, concatenates it to the message and hashes the result with the compressor obtaining a k_1-bit string w (thus the algorithm is randomized and, except with negligible probability, a different signature will be obtained each time a given message is signed with a given key). Afterwards, r^* is obtained by applying the generator to the hashed value w, taking the first k_0 bits, and using them to 'mask' r by means of the bitwise Xor operation. Then $w||r^*$ is prepended with a 0 bit and appended with $G_2(w)$ to obtain a bit string y. The length of y is k, i.e., it is equal to $\text{len}(n)$. The fact that the leading bit of y is 0 implies that $y < n$ when regarded as an integer, so that $y \in \mathbb{Z}_n$ and it can be 'decrypted' with the RSA function to obtain the signature.

3. Given the signed message (m, σ), the verifier starts by computing y and then recovers w, r^* and, finally, the random seed r. Then these values are used to check that y was correctly constructed and only if all these checks are successful, the signature is accepted.
4. The scheme is efficient because the signing and verification algorithms each require one application of H, one application of G, and one RSA operation (decryption or encryption). The hash functions used in practice are very efficient, so the costliest part is the RSA operation.

As already mentioned, PSS is UF-CMA secure in the random oracle model assuming that the RSA problem is hard relative to $\mathbf{Gen}_{\mathrm{RSA}}$. This was proved in [20] and can be stated as follows:

Theorem 9.2 *Consider the PSS scheme with security parameters k_0 and k_1 and suppose that $\mathcal{A}$ is an adversary making q_s signing oracle queries and q_h hash oracle queries (with $q_h \geq 1 + q_s$), whose probability of success in the experiment $\mathbf{Sign}^{\textit{uf-cma}}_{\mathcal{A},PSS}(k)$ is equal to $\varepsilon(k)$. Then there exists an adversary $\mathcal{B}$ with probability of success in the $\boldsymbol{RSA}_{\mathcal{B},\mathbf{Gen}_{RSA}}(k)$ experiment equal to $\varepsilon'(k)$, where*

$$\varepsilon(k) = \varepsilon'(k) + 3(q_h - 1)^2(2^{-k_0} + 2^{-k_1})$$

and the running time of $\mathcal{B}$ is that of $\mathcal{A}$ plus $q_h \cdot k_0 \cdot O(k^3)$.

Remarks 9.4

1. We have not explicitly defined the PSS unforgeability experiment used in the theorem but suffice it to say that it is a signature experiment in the random oracle model, defined similarly to the FDH experiment.
2. The fact that $q_h \geq 1+q_s$ in the preceding theorem comes from the assumption—discussed when analyzing the security of FDH—that each signing query is preceded by a hash query and that the hash query that corresponds to the message output in the forgery is not followed by a signing query.
3. Since typically we will have $k_0 = k_1 = 256$, $1/(2^{-k_0} + 2^{-k_1})$ will dominate the factor $3(q_h - 1)^2$ and hence the value of $\varepsilon'(k)$ will be close to the value of $\varepsilon(k)$. On the other hand, we see that the time reduction (expressed by the relation between the running times of $\mathcal{A}$ and $\mathcal{B}$) is also tight.
4. It is obvious that the PSS construction (or the FDH scheme, for that matter) may also be more generally defined by using a trapdoor one-way permutation instead of the RSA function, obtaining a scheme which is secure in the random oracle model relative to the one-wayness of the permutation used.

We refer to [20] for a proof of Theorem 9.2. The crucial difference between this proof and that of Theorem 9.1 is that there the RSA inverter had to guess the index value $j \in \{1, \ldots, q\}$ for which the adversary $\mathcal{A}$ would forge a message and this guess was correct only with probability $1/q$, with q equal to the number of hash oracle queries. In the case of PSS, the proof shows that the inverter is successful, except with very small probability, no matter which is the index value for which the forgery

occurs. We refer also to [90] for very detailed explanations of the security reduction for FDH as well as of the ideas underlying the transition from FDH to PSS.

The signature schemes we have seen so far are all standard schemes in which the signer sends the unencrypted message m with the signature attached (these signatures are often called 'signatures with appendix'). There is an alternative class of signature schemes, generically called 'signatures with message recovery', in which only one data unit (an 'enhanced signature') is transmitted—with the goal of saving bandwidth—and the message is recovered from the enhanced signature and authenticity is checked at the same time. The algorithm that does this is called the *Recover* algorithm—it replaces the verification algorithm and either outputs the message in case it is authentic or declares the signature invalid.

In [20], a version of PSS with message recovery, called PSS-R, is also given. The security of this scheme is the same as for PSS.

9.5.3 RSASSA-PSS from PKCS #1 v2.1

PSS has been standardized in [154] under the name RSASSA-PSS, where the initials RSASSA stand for "RSA Signature Scheme with Appendix". This signature scheme has a structure similar to that of RSAES-OAEP—replacing the encryption and decryption algorithms by the signing and verification algorithms—and combines signature and verification primitives with an encoding method for signatures called EMSA-PSS; the specific features of PSS are contained in this encoding method.

The message to be signed is first encoded by means of EMSA-PSS and then converted to integer form to which the RSA decryption function is applied to produce the signature. Verification proceeds in reverse order: first the RSA encryption function is applied to the signature and the integer obtained is converted to an EMSA-PSS-encoded message which is then processed according to the PSS verification algorithm.

We start by describing the required components, for which we use notation similar to that in [154]. The basic ingredients are essentially the same as in RSAES-OAEP, described in Sect. 8.3.6.2, to which we refer for details. They are:

- An RSA public key (n, e) and its corresponding private key (n, d), where the bit length of the modulus n is *modBits* and its byte length is k.
- A hash function Hash whose outputs are *hLen* bytes in length.
- A *Mask Generating Function* (MGF).
- The intended byte length *sLen* of the seed (or 'salt' in the terminology of [154]) used by the signing algorithm.
- A message M to be signed, given as a byte string of arbitrary length (in practice, this length is limited by the input size limitation of the hash function used).
- The maximal bit length *emBits* of the integer corresponding to the encoded message *EM* mentioned in the following item. *emBits* is at least $8hLen + 8sLen + 9$ and it will be set equal to $modBits - 1$ in this scheme.

- The encoded message EM, a byte string of length $emLen = \lceil emBits/8 \rceil$ obtained by applying the EMSA-PSS encoding to M.
- The signature S, a byte string of length k, where k is the byte length of the RSA modulus n.

We next describe the algorithms used by RSASSA-PSS in the order they are applied, first to sign a message and then to verify the signature.

EMSA-PSS encoding.

1. Set $emBits = modBits - 1$ and compute the encoded message:

   $$EM = \text{EMSA-PSS-Encode}(M, emBits) = \text{EMSA-PSS-Encode}(M, modBits - 1)$$

   as follows.
2. If the length of M is greater than the input limitation for the hash function, output "message too long" and stop.
3. Compute $mHash = \text{Hash}(M)$, a byte string of length $hLen$.
4. If $emLen < hLen + sLen + 2$ then output "encoding error" and stop.
5. Generate a random byte string $salt$ of length $sLen$.
6. Compute $M' := 0000000000000000||mHash||salt$, a byte string of length $8 + hLen + sLen$ with eight initial zero bytes.
7. Compute $H = \text{Hash}(M')$, a byte string of length $hLen$.
8. Set PS equal to the byte string of length $emLen - sLen - hLen - 2$ consisting of zero bytes (the length of the string may be 0, in which case it is the empty string).
9. Compute $DB := PS||01||salt$, a byte string of length $emLen - hLen - 1$.
10. Compute $dbMask := \text{MGF}(H, emLen - hLen - 1)$.
11. Compute $maskedDB := DB \oplus dbMask$.
12. Set the leftmost $8emLen - emBits$ of the leftmost byte in $maskedDB$ to zero.
13. Output the encoded message EM, obtained by concatenating $maskedDB$, H and the byte given by the hexadecimal string 'bc':

    $$EM := maskedDB||H||\text{bc}.$$

Once the message is encoded by means of EMSA-PSS, it can be signed with the RSA private key. For this the encoded message EM is converted to an integer by means of the function OS2IP that we have already used in Sect. 8.3.7 (later on, the integer signature will be converted back to a byte string by means of I2OSP). EM has byte length $emLen = \lceil emBits/8 \rceil$ and the integer OS2IP(EM) will have at most $modBits - 1$ bits. Thus the signature is obtained as follows:

RSA signing.

1. Compute $m := \text{OS2IP}(EM)$.
2. Check whether $m \in [0, n - 1]$ and, if not, output an error message and stop.

3. Compute the integer signature representative $s := m^d \bmod n$.
4. Convert s to a byte string of length k and output the signature $S := \text{I2OSP}(s, k)$.

The preceding steps produce a signature S for a message M and all together make up the signing algorithm RSASSA-PSS-Sign. This can be expressed as

$$S := \text{RSASSA-PSS-Sign}((n, d), M).$$

We are now going to describe the verification algorithm for RSASSA-PSS. This is the RSASSA-PSS-Verify algorithm that, on input an RSA public key (n, e), a message M and a signature S will output either 'valid' or 'invalid'. First, 'RSA Verification' is applied to recover the encoded message EM and this is followed by EMSA-PSS-Verify, which combines EMSA-PSS decoding with the verification of the signature. The steps are the following:

RSA verification.

1. If the length of the signature S is not k bytes, then output 'invalid' and stop.
2. Convert S to an integer $s := \text{OS2IP}(S)$.
3. Check that $s \in [0, n-1]$ and compute $m := s^e \bmod n$.
4. Convert m to a byte string EM of length $emLen = \lceil emBits/8 \rceil$, $EM := \text{I2OSP}(m, emLen)$.

Verification is completed by EMSA-PSS verification that computes:

$$\text{EMSA-PSS-Verify}(M, EM, emBits) = \text{EMSA-PSS-Verify}(M, EM, modBits - 1),$$

obtaining either 'consistent' or 'inconsistent'. In the first case RSASSA-PSS-Verify will output 'valid' (thus verifying the signature) and in the second, 'invalid'. The algorithm is similar to EMSA-PSS-Encode but follows reverse steps to recover *salt* and then forward steps to recompute H and compare the value obtained with the one obtained from EM.

EMSA-PSS verification.

1. Set $emBits = modBits - 1$.
2. If the length of M is greater than the input limitation for the hash function, output 'inconsistent' and stop.
3. Compute $mHash = \text{Hash}(M)$, a byte string of length $hLen$.
4. If $emLen < hLen + sLen + 2$ then output 'inconsistent' and stop.
5. If the rightmost byte of EM does not have hexadecimal value 'bc', output 'inconsistent' and stop.
6. Set $maskedDB$ equal to the leftmost $emLen - hLen - 1$ bytes of EM and let H be the next $hLen$ bytes.
7. If the leftmost $8emLen - emBits$ bits of the leftmost byte in $maskedDB$ are not all equal to zero, output 'inconsistent' and stop.
8. Compute $dbMask := \text{MGF}(H, emLen - hLen - 1)$.

9. Compute $DB = maskedDB \oplus dbMask$.
10. Set the leftmost $8emLen - emBits$ of the leftmost byte in DB to zero.
11. If the $emLen - hLen - sLen - 2$ leftmost bytes of DB are not zero or if the byte at position $emLen - hLen - sLen - 1$ is not 01 then output 'inconsistent' and stop.
12. Set $salt$ equal to the last $sLen$ bytes of DB.
13. Compute $M' := 0000000000000000||mHash||salt$, a byte string of length $8 + hLen + sLen$ with eight initial zero bytes.
14. Compute $H' = \text{Hash}(M')$, a byte string of length $hLen$.
15. If $H = H'$ output 'consistent', otherwise output 'inconsistent'.
16. If EMSA-PSS-Verify$(M, EM, modBits - 1)$ = 'consistent' then set RSASSA-PSS-Verify$((n, e), M, S)$:= 'valid', otherwise RSASSA-PSS-Verify$((n, e), M, S)$:= 'invalid'.

Remarks 9.5

1. In the description of the RSASSA-PSS scheme we have kept the notation in [154]. For example, the parameters k_0 and k_1 in [20] are here $sLen$ and $hLen$, respectively. For simplicity, we take $sLen = hLen$ in our Maple implementation below.
2. There are some differences between the RSASSA-PSS scheme and the original PSS scheme but the security reduction for the latter is still valid for this variant. One of these differences is the fact that the value that is hashed together with the seed (or salt in [154] terminology) is the result of concatenating a string of eight bytes '00' with the hash of the message, instead of the message itself. Thus the message is hashed one more time (with the same hash function) prior to applying the PSS algorithm, i.e., the PSS algorithm is applied to the hash of the message rather than to the message itself. This initial hashing may also be performed outside of the module that does the remaining computations, which is advantageous in situations where I/O bandwidth is limited. We refer to [154] for more details on this and other specific aspects of the standard.

9.5.4 RSASSA-PSS in Maple

We are going to implement the RSASSA-PSS signature scheme in Maple. We will mostly follow the specification and the notation for variable names in [154]. We start with a couple of auxiliary functions that will be called from the main functions of the scheme. The first of them, `make0bits`, takes as input `hbyte` (a byte given as a hexadecimal string) and `numbits` (an integer in the interval [0, 7]), and returns the byte obtained by setting to 0 the leftmost `numbits` of `hbyte`. This function will be called by the EMSA-PSS encoding procedure to perform step number 12 in the above description of this algorithm, with the purpose of ensuring that the bit length of the resulting encoded message, when regarded as an integer, is one less than that

of the RSA modulus (and hence this integer is smaller than the modulus). It is also used in step 10 of the EMSA-PSS verification procedure above.

```
> make0bits := proc(hbyte::string, numbits::nonnegint)
  uses StringTools;
  local bits, b;
  if 8 <= numbits then
    error "numbits must be less than 8"
  end if;
  bits := integer2bits(op(hexstringtobytes(hbyte)), 8);
  bits := [0$numbits, op(bits[numbits+1 .. -1])];
  b := LowerCase(convert(bits2integer(bits), hex));
  if Length(b) = 1 then
    cat("0", b)
  else
    b
  end if
  end proc:
```

The next function is used by the EMSA-PSS verification algorithm in step 11 to check that some of the leftmost bits of *DB* are zero. The input parameters are as in the preceding function and the output is either `true`, in case the leftmost `numbits` bits of `hbyte` are all zero, or `false` otherwise.

```
> check0bits := proc(hbyte::string, numbits::nonnegint)
  local bits, b;
  if 8 <= numbits then
    error "numbits must be less than 8"
  end if;
  bits := integer2bits(op(hexstringtobytes(hbyte)), 8);
  evalb(bits[1 .. numbits] = [0$numbits])
  end proc:
```

Example 9.3 Let us consider the byte "ff" and compute the result of setting the leftmost i bits equal to 0, for $i = 0, 1, \ldots, 7$:

```
> make0bits~("ff", [$0 .. 7]);
              ["ff", "7f", "3f", "1f", "0f", "07", "03", "01"]

> check0bits~(%, [$0 .. 7]);
              [true, true, true, true, true, true, true, true]
```

Next we give the function that implements the EMSA-PSS-Encode algorithm. By default, we will use the hash function `SHA256` and the mask generating function `MGF1`, both of which were already used in the implementation of RSA-OAEP in Sect. 8.3.7. As we saw in the description above, another required input is *sLen*, namely the intended byte length of the salt (the seed in Bellare–Rogaway's terminology). We will use $sLen = hLen$ as suggested in PKCS #1 v2.1 and so the former will not appear as a parameter. We will not check the maximum message length—specified in step 2 of EMSA-PSS encoding above—because this maximum is, for the hash function we use, more than what Maple can handle. As far as possible we keep the notation in PKCS #1 v2.1.

The required input parameters of the EMSA-PSS encoding function are `M`, for the message given either as an even-length hexadecimal string or a text string, `emBits` for the maximum bit length of the encoded message when regarded as an integer, and `salt` for the salt given as an even-length hexadecimal string. The optional parameters

are Hash for the name of the hash function used (with SHA256 as default), hLen for the byte length of the hash function output (with 32 as default), MGF for the mask generating function (with MGF1 the default) and, finally, messagetype, used to specify whether the message is a text string or a hexadecimal string (with hex as default). The output is the encoded message EM.

```
> EMSAPSSEnc := proc(M::string, emBits::posint, salt::string, {Hash::name:='SHA256',
               hLen:=32, MGF::name:='MGF1', messagetype::identical(hex,text):=hex})
  uses StringTools;
  local mHash, emLen, hhLen, s, M1, H, PS, DB, dbMask, maskedDB;
  mHash := Hash(M, messagetype);
  emLen := ceil(emBits/8);
  if emLen < 2*hLen+2 then
    error "encoding error"
  end if;
  hhLen := 2*hLen;
  if _params['salt'] = NULL then
    Randomize();
    s := LowerCase(Random(hhLen, xdigit))
  else
    if not IsHexDigit(salt) or Length(salt) <> hhLen then
      error "salt must be a hex string of %1 bytes" ,hLen
    end if;
    s := salt
  end if;
  M1 := cat(Repeat("00", 8), mHash, s);
  H := Hash(M1, hex);
  PS := Repeat("00", emLen-2*hLen-2);
  DB := cat(PS, "01", s);
  dbMask := MGF(H, emLen-hLen-1);
  maskedDB :=
         bytestohexstring(BitXor~(hexstringtobytes(DB),hexstringtobytes(dbMask)));
  maskedDB := cat(make0bits(Take(maskedDB, 2), 8*emLen-emBits),Drop(maskedDB, 2));
  cat(maskedDB, H, "bc")
  end proc:
```

For demonstration and testing purposes, EMSAPSSEnc will also work if no argument is passed to the salt parameter, in which case the salt string will be pseudo-randomly generated inside the procedure. As on other similar occasions, we warn that the security reduction does not apply to the resulting scheme; for it to be applicable, an externally generated random salt should be supplied to the function.

Next we give the PSS signing function, which calls the previous one to encode the message and then uses the RSA private key to sign it. The required input parameters are privatekey for the private key given as a list $[n, d]$ (or in the format output by the function RSAKeyGen in Sect. 8.3.3), message for the message, and salt for the salt, where the last two parameters have the same format as in the preceding function. The optional keyword parameters are also the same as in the previous function. As before, if no value is passed to salt, the signing algorithm will use a pseudo-randomly generated one. The output of the function is the signature, given as a hexadecimal byte string.

```
> PSSSign := proc(privatekey::list,message::string,salt::string,{Hash::name := 'SHA256',
      hLen::posint := 32, MGF::name := 'MGF1', messagetype::identical(hex,text) := hex})
  local sk, n, d, modBits, EM, m, s, k;
  sk := stringposint~(privatekey);
  n := sk[1];
```

```
  d := sk[2];
  modBits := intlog[2](n)+1;
  EM := EMSAPSSEnc(message, modBits-1, _params['salt'], ':-messagetype' = messagetype,
                   ':-Hash' = Hash, ':-MGF' = MGF);
  m := OS2IP(EM);
  s := Power(m, d) mod n;
  k := intlog[256](n)+1;
  I2OSP(s, k)
  end proc:
```

Example 9.4 Let us compute the signature for a message. We will use the RSA key `rsakey` that we already used on a couple of occasions:

```
> rsakey := RSAKeyGen(1024, 618020429552270915383892324472348012175,
                      410275671464698780248543572127557803186);
  sk := rsakey[2]:
  pk := rsakey[1]:
```

We are going to sign the following ASCII-8 message:

```
> M := "This is a message to test RSASSA-PSS signatures according to PKCS #1 v2.1":
```

We need a 32-byte random salt as an input to the signing algorithm. We will use the following hexadecimal string:

```
> salt := "01f1698656b7fbccf1c94837d65a041070493da6a05863895867f8620d3ca422":
```

With these ingredients we can generate the signature of message `M` above:

```
> S := PSSSign(pk, M, salt, messagetype = 'text');
S :=
"72617c71e7114c4340d2aaedfb7e8435040c6c9242f1a0455e43fbe5f3d5dbe8c99370a6e4fb87ffb\
  d1460163a8c5e9ebf2d75350bccc3604f5175a6154c9cde35f61ca9191684966afc831233467dcfa\
  c444f60a501ea203818166bede2611e136f785620e0b61f7c19d6b7cd6a5e707d9473be1655bac12\
  f6473defa0c6eb7de86045373f150a672e4bd9b92f4ddfc82106d92cd367b4e03776b87169267afa\
  21b86d31864f55c86e68d7c2958b6ff0984b7e7e7eed127ed606e34247bcfa5b28cdb022ba09659e\
  8d615ec38a5645bc3da279384882f58791b4eaa46a776e8cd75f15f30eaf8a7b63f1f4144ab257c9\
  d22e239ac703e97ee75c4b32e2125c7"
```

As explained in the preceding section, the RSASSA-PSS verification algorithm is implemented in two parts. First there is RSA verification, which does the part that involves the application of the RSA primitive, and then EMSA-PSS verification, which deals with the specific part of the PSS-algorithm. We begin with the EMSA-PSS verification algorithm, implemented in the function `EMSAPSSVer` below, which will be called from the general verification function to be defined later on. The input parameters are similar to those in `EMSAPSSEnc` except that `salt` is not needed here and there is a new parameter `EM` for the encoded message given as a hexadecimal byte string. The output is either "consistent" if verification is successful or "inconsistent" otherwise. These outputs will be interpreted by the PSS verification algorithm to mean "valid signature" and "invalid signature", respectively.

```
> EMSAPSSVer := proc(M::string, EM::string, emBits::posint, {Hash::name := 'SHA256',
  hLen::posint := 32, MGF::name := 'MGF1',messagetype::identical(hex,text) := hex})
  uses StringTools;
  local mHash, emLen, hemLen, hhLen, em, maskedDB, H, dbMask, DB, b, salt, M1, H1;
  mHash := Hash(M, messagetype);
  emLen := ceil(emBits/8);
  hemLen := 2*emLen;
```

```
hhLen := 2*hLen;
if emLen < hhLen+2 then
  return "inconsistent"
end if;
if not IsSuffix("bc", EM) then
  return "inconsistent"
end if;
em := Take(EM, hemLen-2);
maskedDB := Take(em, hemLen-hhLen-2);
H := Drop(em, hemLen-hhLen-2);
if not check0bits(Take(maskedDB, 2), 8*emLen-emBits) then
  return "inconsistent"
end if;
dbMask := MGF(H, emLen-hLen-1);
DB :=
  bytestohexstring(BitXor~(hexstringtobytes(maskedDB), hexstringtobytes(dbMask)));
b := make0bits(Take(DB, 2), 8*emLen-emBits);
DB := cat(b, Drop(DB, 2));
if Take(DB, hemLen-2*hhLen-4) <> Repeat("0", hemLen-2*hhLen-4) or
SubString(DB, hemLen-2*hhLen-3 .. hemLen-2*hhLen-2) <> "01" then
  return "inconsistent"
end if;
salt := Drop(DB, hemLen-2*hhLen-2);
M1 := cat(Repeat("00", 8), mHash, salt);
H1 := Hash(M1, hex);
if H = H1 then
  "consistent"
else
  "inconsistent"
end if
end proc:
```

The PSS verification algorithm is implemented in the next function `PSSVer` which first performs the RSA verification and then calls `EMSAPSSVer` to complete the verification process. The required inputs are an RSA public key given as a list $[n, e]$, a hexadecimal or text string containing the message, and the signature as an even-length hexadecimal string. The optional parameters are the same as in the preceding function and the output is either "valid signature" or "invalid signature".

```
> PSSVer := proc(publickey::list,message::string,signature::string, {hLen::posint := 32,
    Hash::name := 'SHA256', MGF::name := 'MGF1',messagetype::identical(hex,text):= hex})
  local pk, n, e, modBits, s, m, EM, emLen, Result;
  pk := stringposint~(publickey);
  n := pk[1];
  e := pk[2];
  modBits := intlog[2](n)+1;
  s := OS2IP(signature);
  m := Power(s, e) mod n;
  EM := I2OSP(m, emLen);
  emLen := ceil((modBits-1)/8);
  EM := I2OSP(m, emLen);
  Result := EMSAPSSVer(message, EM, modBits-1, ':-Hash' = Hash, 'hLen' = hLen,
                       ':-MGF' = MGF, ':-messagetype' = messagetype);
  if Result = "consistent" then
    "valid signature"
  else
    "invalid signature"
  end if
  end proc:
```

Example 9.5 We give a function to test the PSS implementation. Required inputs are an RSA key and the number of iterations to be performed. The parameter `maxlength` specifies the maximum number of characters (for text messages) or

bytes (for hex messages) of the pseudo-randomly generated messages tested. The parameter `format` specifies either `text` or `hex` messages and `verbose` whether tested messages are to be printed. The test ends either by informing that it was successful or printing the message for which it failed.

```
> PSSTest := proc(key::list, iterations::posint, maxlength::posint := 100,
                {format::identical(text,hex) := hex, verbose::truefalse :=false})
  local pk, sk, gen, valid, i, r, m;
  pk := key[1];
  sk := key[2];
  if format = hex then
    gen := 'xdigit'
  else
    gen := NULL
  end if;
  valid := true;
  for i to iterations while valid do
    r := rand(0 .. maxlength)();
    if gen = 'xdigit' then
      r := 2*r
    end if;
    m := StringTools:-LowerCase(StringTools:-Random(r, gen));
    if verbose then
      printf("%s%s\n", "Message:  ", m)
    end if;
    valid := evalb(PSSVer(pk, m, PSSSign(sk, m, ':-messagetype' = format),
             ':-messagetype' = format) = "valid signature");
    if not valid then
      printf("verification failure for message:\n");
      return m
    end if
  end do;
  if valid then
    printf("%d iterations successfully completed\n", iterations)
  end if
  end proc:
```

Let us use the test function with key `rsakey`, 3 iterations, `verbose = true` and default values for the remaining parameters:

```
> PSSTest(rsakey, 3, verbose = true);
 Message  1: fcddbeb2d836fbbd97a4a1a0c5bcbf67fe1e5eb6eaacf64f06f6a7e8aeba22
 Message  2: eeddb18da6b67863e44bab09bfdad1a9a12c9dd1628ebd4a8631be67ecf1afdeacb7c\
             f2bc57cc5
 Message  3: bde5fdd4aecd5d83f0d01c1e92ffac3363a3036dced8be
3 iterations successfully completed
```

Example 9.6 Let us verify the signature generated in Example 9.4. It is done as follows:

```
> PSSVer(sk, M, S, messagetype = 'text');
                                "valid signature"
```

Exercise 9.15 Modify `PSSSign` and `PSSVer` so that both functions accept messages given as binary files and signatures are also saved to files.

9.5.5 Cramer–Shoup Signatures

If the RSA problem is hard, then the FDH and PSS signature schemes are UF-CMA secure in the random oracle model but, so far, we have not mentioned any signature scheme that is UF-CMA secure without assuming that the hash functions involved are random oracles. However, such schemes exist and the price to pay for the stronger security result is less efficiency and, sometimes, the use of a hardness hypothesis that is stronger than the RSA assumption. Perhaps the most important scheme of the latter type is the Cramer–Shoup signature scheme which we are now going to describe.

The Cramer–Shoup signature scheme was introduced in [58] and its security relies on the hardness of the following problem:

Definition 9.10 The *flexible RSA problem* is, given an RSA modulus n and $z \leftarrow \mathbb{Z}_n^*$, to find an integer $e > 1$ and $y \in \mathbb{Z}_n^*$ such that $y^e \equiv z \pmod{n}$. The *strong RSA assumption* is the assumption that this problem is hard to solve.

Remark 9.4 The difference between the strong RSA assumption and the RSA assumption is that in the latter the exponent e is chosen independently of z, whereas in the former it may be chosen in a way that depends on z. An adversary breaks the strong RSA assumption if it can find an exponent e such that it is able to extract an eth root of z, so it is clear that the strong RSA assumption is potentially stronger than the RSA one. However, the only method known so far to break either assumption is to solve the integer factorization problem.

9.5.5.1 Cramer–Shoup Signature Scheme

The scheme is parameterized by two security parameters l and l' such that $l+1 < l'-1$ and makes use of a collision resistant hash function H whose output can be interpreted as a positive integer less than 2^l. A reasonable choice for H is SHA-256 and this suggests using $l = 256$. The parameter l' specifies the length of two primes that are used to generate an RSA modulus of a special form. Thus, bearing in mind the hardness of the factorization problem, a reasonable value for this parameter is $l' = 1024$.

- **Key generation**. The algorithm proceeds as follows:

1. Two random l'-bit safe primes p, q, are chosen (so that $p = 2p' + 1$ and $q = 2q' + 1$, where p' and q' are Sophie Germain primes) and $n = pq$ is computed.
2. Two random quadratic residues $h \leftarrow \mathcal{QR}_n, x \leftarrow \mathcal{QR}_n$ and a random $(l + 1)$-bit prime e' are chosen.
3. The public key is (n, h, x, e') and the private key is (p, q).

- **Signing**. Let $m \in \{0, 1\}^*$ be a message. To sign it with the private key (p, q)—plus the public key—the following operations are performed:

1. Choose a random $(l+1)$-bit prime e such that $e \neq e'$ and a random quadratic residue $y' \leftarrow \mathcal{QR}_n$.
2. Compute the value x' defined as follows:

$$x' := (y')^{e'} h^{-H(m)} \bmod n \in \mathbb{Z}_n^*.$$

3. Compute $y \in \mathbb{Z}_n^*$ such that:

$$y^e \equiv x h^{H(x')} \pmod{n}$$

(note that this eth root modulo n is easy to compute given the private key, i.e., the factorization of n; see Remarks 9.6).
4. The signature corresponding to the message m is then the 3-tuple (e, y, y').

- **Verification**. Given a message $m \in \{0,1\}^*$ and its signature (e, y, y'), as well as the public key (n, h, x, e'), verification proceeds as follows:

1. Check that e is an odd $(l+1)$-bit integer different from e'.
2. Compute $x' = (y')^{e'} h^{-H(m)} \bmod n \in \mathbb{Z}_n^*$.
3. Check whether $x = y^e h^{-H(x')} \bmod n \in \mathbb{Z}_n^*$.
4. If both checks are affirmative then accept the signature, otherwise reject it.

Remarks 9.6

1. All computations in the signing and verification algorithms take place in the group $\mathcal{QR}_n$. Since n is a product of two distinct odd primes, each quadratic residue modulo n has exactly 4 square roots in $\mathbb{Z}_n^*$ by Corollary 2.7 and hence $|\mathcal{QR}_n| = |\mathbb{Z}_n^*|/4 = \phi(n)/4 = (p-1)(q-1)/4 = p'q'$. Thus the computation of y in the signing algorithm can be done as follows. Since $\mathcal{QR}_n$ has order $p'q'$, the exponent e is defined modulo this order and the eth root can be computed as $y = (x h^{H(x')})^{e^{-1} \bmod p'q'} \bmod n$. Indeed, since $\text{len}(e) = l + 1 < l' - 1 = \text{len}(p') = \text{len}(q')$, we have that $\gcd(e, p'q') = 1$ and hence the inverse $e^{-1} \bmod p'q' \in \mathbb{Z}_{p'q'}^*$ exists and may be efficiently computed by means of the extended Euclidean algorithm.
2. The prime e generated in the signing algorithm need not be randomly chosen. Instead, the requirement is that the probability of generating the same prime twice when signing different messages should be negligible. This will indeed be the case if these primes are randomly chosen which, on the other hand, also guarantees that $e \neq e'$ except with negligible probability. Since the quadratic residue y' is also randomly chosen, the signing algorithm is probabilistic.
3. Observe that the verification algorithm does not verify that e is prime, checking only that e is odd. This is done for efficiency, since the security reduction to which we refer below does not require checking primality.
4. As happens with the Cramer–Shoup encryption scheme, the existence of 'enough' Sophie Germain primes is required for this signature scheme to be secure. As we have mentioned, these primes are indeed conjectured to be abundant and we

will assume that this is the case; we refer to [58] for specific details and for a recommended procedure to generate these primes.

5. The signing algorithm requires the generation of an $(l+1)$-bit prime for each signature. If not carefully implemented, this process may be the costliest part of the algorithm. However, these primes need not be generated uniformly at random and in [58] an efficient algorithm is suggested to generate them in case $l = 160$.

The security of the Cramer–Shoup scheme is given by the following result:

Theorem 9.3 *The Cramer–Shoup signature scheme is UF-CMA secure under the strong RSA assumption and the assumption that H is collision resistant.*

We refer to [58, Theorem 1] for the proof, which is too long to be included here.

Exercise 9.16

(i) Show that if $n = pq$ where $p = 2p' + 1$ and $q = 2q' + 1$, with p' and q' distinct Sophie Germain primes, then $\mathcal{QR}_n$ is a cyclic group of order $p'q'$.
(ii) Write a Maple function that, given the hypotheses of the preceding question, and given $z \in \mathcal{QR}_n$ and a positive integer e which is not divisible by p' nor by q', computes the eth root of z in $\mathcal{QR}_n$ as required by Cramer–Shoup signatures. Test this function for suitable values of z and e, with p and q 1024-bit primes.

9.6 Signatures with Added Functionality

Here we briefly review some signature schemes which have added functionality.

9.6.1 Blind Signatures

Blind signatures were originally proposed by D. Chaum in the 1980s, with the purpose of creating an electronic version of money. A blind signature scheme is a two-party protocol between a sender (called also a blinder) and a signer. The sender A sends the signer B some information (a 'blinded' message m that is in some sense analogous to a sealed envelope containing m, which the signer does not see). B signs this information and returns it to A. From this signature, A computes (or 'unblinds') the signature of the message m but B knows neither m nor its signature. Thus the term 'blind signature' refers to the fact that the signer does not know what it signed. In particular, B is later unable to associate the message m with sender A.

It might seem at first sight that blind signatures are completely useless because no party would agree to sign a message without knowing its content. However, there are interesting applications of this concept, for example, in the domain of digital cash.

Suppose that the sender A wants to pay for a service by presenting a bill signed by its bank B which will then refund the service provider. However, A does not want B to know the details of the service for which the payment is being made. The solution for this is to use a blind signature scheme. The bank then signs the bill without knowing its contents and charges A accordingly. A unblinds the signature thus obtaining the signed bill that it can use to pay for the service. The service provider verifies the signature and presents the bill to the bank, which also verifies that it has indeed signed it. Thus the bank pays the service provider without being able to associate the bill with sender A.

In this example, there is the problem that the sender might cheat and try to make the bank sign a bill for a value higher than agreed. This can be dealt with in several ways—for example, the bank may only use a given key to sign bills for a fixed value—but we shall not go into the details.

In view of this, a blind signature scheme can be described as follows:

- The ingredients are a digital signature scheme Σ and a pair of functions f, g, known only to the sender, such that if $(f(m), \sigma)$ is a valid message/signature pair for Σ, then $(m, g(\sigma))$ is also a valid pair. f is called the *blinding* function and g is the *unblinding* function.
- The protocol proceeds as follows:
  - The sender blinds a message m by computing $f(m)$ and sends $f(m)$ to the signer.
  - The signer signs the blinded message, obtaining the message/signature pair $(f(m), \sigma)$ and sending it to the sender.
  - The sender unblinds the signature of the blind message by computing $g(\sigma)$. $(m, g(\sigma))$ is then a message signed by the signer.
  - The signature may be verified by using the signer's public key as usual.

Example 9.7 We present Chaum's protocol, the blind analogue of plain RSA signatures, which are used as the underlying signature scheme. Given a public RSA key $pk = (n, e)$ and the corresponding private key $sk = (n, d)$, the blinding and unblinding functions are defined as follows:

- (Blinding function) Choose an integer k such that $\gcd(k, n) = 1$ and define $f : \mathbb{Z}_n \to \mathbb{Z}_n$ by $f(m) := mk^e \bmod n$.
- (Unblinding function) $g : \mathbb{Z}_n \to \mathbb{Z}_n$, $g(\sigma) := k^{-1}\sigma \bmod n$.

Then, if $\sigma := f(m)^d \bmod n = (mk^e)^d \bmod n = km^d \bmod n$ is the signature of the blinded message $f(m)$, we have that $g(\sigma) = k^{-1}\sigma \bmod n = k^{-1}km^d \bmod n = m^d \bmod n$ is indeed the signature of m with the private key (n, d). Thus the protocol proceeds as follows:

- The sender blinds the message $m \in \mathbb{Z}_n$ using a random $k \leftarrow \mathbb{Z}_n$, obtaining $m' := mk^e \bmod n$ which is then sent to the signer.
- The signer computes the RSA signature of the blinded message m', $\sigma' := (m')^d \bmod n \in \mathbb{Z}_n$ and sends it to the sender.

- The sender computes the RSA signature σ of m, $\sigma := k^{-1}\sigma' \bmod n$.
- The signature may be publicly verified by using the signer's public key (n, e). If $\sigma^e \equiv m \pmod{n}$ then the signature is declared valid, otherwise it is declared invalid.

Observe that the signer knows the identity of the sender and may associate the signed message (m', σ') with the sender. However, if later presented with the signed message (m, σ), the signer is unable to associate it with the sender because this pair does not reveal any information about the pair (m', σ'). This is due to the fact that k is chosen uniformly at random in $\mathbb{Z}_n$, which means that, from the point of view of the signer—which does not know k—$m' = mk^e \bmod n$ is uniformly distributed in $\mathbb{Z}_n$ and the same happens to σ'.

The scheme in the previous example is not secure because of the insecurity of the underlying plain RSA signatures and, furthermore, UF-CMA security is no longer adequate for blind signatures, as the sender must be able to perform an existential forgery. A security notion adequate for this context was proposed in [156] where also some secure blind signature schemes (based on RSA or discrete logarithms) are proposed.

9.6.2 Other Signatures with Added Functionality

An interesting kind of signatures are *undeniable signatures*, which were introduced by Chaum and van Antwerpen in 1989 and whose main peculiarity is that they have to be verified interactively with the cooperation of the signer, so that verification with the public key alone is impossible. This feature protects the signer against the possibility that the signed documents might later be distributed without the signer's consent. In [140] a couple of examples where this property is useful are given, including one where a bank customer is required by the bank to sign a time and date document in order to access a safe box. If an undeniable signature is used, the bank is later unable to prove that the customer had access to the box on the specified date, because verifying the signature requires the signer's cooperation. On the other hand, a dishonest signer might take advantage of this to claim that a valid signature is a forgery or, simply, refuse to verify it. To prevent this from happening, an undeniable signature scheme must come along with a *disavowal protocol* that can be used to prove that a given signature is a forgery.

We refer to [140] and to [189] for a description of the Chaum–van Antwerpen undeniable signature scheme as well as of other kinds of signatures with added functionality such as, for example, *fail-stop signature schemes* which, essentially, are digital signatures that allow the signer to prove that a signature purportedly (but not actually) signed by itself is a forgery.

9.7 Public-Key Infrastructures

In this section we discuss methods to ensure the secure distribution of public keys.

9.7.1 Certificates

When discussing public-key encryption and public-key signatures, we mentioned that an essential requirement for public-key cryptography to work as intended is the secure distribution of public keys by means of an authenticated channel. Indeed, if public keys are not authenticated, then Bob has no way of knowing whether the key he received is authentic or perhaps an adversary, Oscar, has mounted a 'man-in-the-middle' attack and replaced Alice's public key by Oscar's own public key, allowing Oscar to decrypt all the messages that Bob subsequently sends to Alice.

Exercise 9.17 Describe in detail how digital signature schemes are vulnerable to 'man-in-the-middle' attacks if unauthenticated channels are used to distribute keys.

It is clear that secure distribution of public keys is also essential for digital signature schemes to work as intended but, on the other hand, if public keys are to be authenticated, digital signatures will be the essential tool to achieve the authentication. Thus it seems that we have reached a vicious circle: we need digital signatures to ensure that digital signatures work!

The solution to this apparent dead-end comes from the fact that it can be shown that an authenticated channel is needed *only once* to ensure the secure distribution of arbitrarily many public keys. The essential instruments to achieve this are (digital) *certificates* and *Public-Key Infrastructures*.

The idea behind certificates is to bind a particular entity to a public key by means of a digital signature that serves to attest the ownership of the public key. These are frequently called public-key certificates and, more generally, a certificate can also attest the truth of any property attributable to a certificate owner, in which case it is called an attribute certificate. Let us consider the basic structure of a public-key certificate. Suppose that C generates a public key/private key pair (pk_C, sk_C) for a secure digital signature scheme, that A's public key/private key pair for an encryption or digital signature scheme is (pk_A, sk_A) and that C knows that the public key of A is indeed pk_A. Then a public-key certificate for A is a pair:

$$\texttt{Cert}_A = ((Id_A, pk_A), \mathbf{Sign}(sk_C, (Id_A, pk_A))),$$

where Id_A is the information that uniquely identifies user A.[6]

[6] In practice, the certificate contains additional information such as, for example, the name of the entity C signing it, the names of the algorithms in which the public key is used, start and end of validity period, etc.

The certificate can now be used to check that pk_A is indeed A's public key. A user that receives this certificate can verify C's signature which protects the pair (Id_A, pk_A) from being altered. However, there are two important conditions that have to be met for this authentication method to be effective. One of them is, obviously, that the user has to trust the signer C. But this is not enough and, in addition, the user must know that pk_C, the public key used to verify the signature in the certificate, is really C's key. For example, if C is the same user as A, then the previous certificate has no value because in that case A's public key is being used to bind A's public key to A's identity, giving rise to a circular situation.

In order to overcome these difficulties, it is usual that signatures for certificates are provided by a *Certification Authority* (or CA, for short), a mutually trusted third party that issues certificates for all users. An authenticated channel is still necessary for the distribution of the CA's public key which, in turn, will authenticate the certificates that the CA emits for users' public keys but, as mentioned before, this makes it sufficient to distribute a single public key, namely, the CA's key pk_{CA}.

The system that specifies how CAs operate is called a *Public-Key Infrastructure* (PKI), which may include tools for certificate management, key management, and a series of standards for the operation of CAs, giving rise to a highly complex system. The simplest PKI consists of a unique CA trusted by all users, which issues and distributes certificates for the users' public keys. In this model every user has to obtain a legitimate copy of pk_{CA} and the important point is that this only has to be carried out once. This makes it possible to use physical channels that are easier to authenticate. For example, users might obtain pk_{CA} at the CA headquarters or, more commonly, the CA public key may be included in computer software such as, for example, web browsers. The procedure used by the CA to issue certificates to users must also be secure—for example, a certificate issued to the wrong user would produce a major security breach. Thus, obtaining a certificate may require the user to show up in person in order to provide his identification and his public key.

The following is an example—with some minor simplifications—of the data contained in an X.509 certificate[7] issued to user Jose Luis Gomez Pardo by the FNMT CA, containing the RSA public key of the user and the signature of the CA, together with other data such as validity period, algorithms used, etc.:

```
Cert. version     3
Serial number     0x3cb3bb5c
Issuer            FNMT Class 2 CA, FNMT, ES
Not valid before  25/09/2009 11:46:00 GMT
Not valid after   25/09/2012 11:46:00 GMT
Subject           GOMEZ PARDO JOSE LUIS - NIF 76...
Public key
(1024 bits)       Public key algorithm  rsaEncryption
                  Modulus:          00: BB 6E 2C 3C E7 C1 BC FD 0A FF 95 D9 D4 ...
                                    10: 97 67 AB 20 52 0F 2B B7 1B CA DD 46 87 ...
                                    ..........................................
                  Exponent:         01 00 01
```

[7] X.509 is a standard for network authentication widely used by Internet standards such as SSL/TLS and S/MIME. Secure Sockets Layer/Transport Layer Security (SSL/TLS) is a cryptographic standard for network communications and Internet, and Secure/Multipurpose Internet Mail Extensions (S/MIME) provides encryption and digital signatures for electronic mail.

```
Signature         Signature algorithm  sha1WithRSAEncryption
                  Signature:           00: 6F 69 6A AE 89 B2 C9 7F 54 14 B9 DB 45 ...
                                       10: F8 BE BF 91 38 42 FF 43 E4 EA C6 38 C5 ...
                                       ........................................

Extensions        X509v3 Private
                  Key Usage Period     Not Before: Sep 25 10:46:12 2009 GMT,
                                       Not After: Sep 25 10:46:12 2012 GMT
                  X509v3 Key Usage     Digital Signature, Key Encipherment
                  Netscape Cert Type   SSL Client, S/MIME
                  X509v3 Subject
                  Key Identifier       E5:D9:81:97:46:7E:87:B9:64:33:9E:50:EC:73: ...
                  X509v3 Authority
                  Key Identifier       keyid:40:9A:76:44:97:74:07:C4:AC:14:CB:1E: ...
```

Here, the *Subject* line gives the identity of the user to whom the certificate was issued, so that this value corresponds to the information Id_A in the description above and the *Issuer* is the CA that issues the certificate, in this case FNMT. Also, the version, the serial number and the validity period of the certificate are specified in the first lines. The validity period imposes a time limitation motivated both by security reasons and other practical reasons such as the fact that the user may cease to exist. Other important fields are the public-key scheme for which the user's public key is valid—in this case RSA encryption—and the user's public key itself. In this example, the public key is a 1024-bit RSA key which consists of a modulus and an (encryption) exponent, both given as hexadecimal strings. Note that the encryption exponent in this example is $0x010001 = 16^4 + 1 = 2^{16} + 1 = 65537$, which is currently by far the most-used exponent.

Other important data included in the certificate are the signature algorithm used to sign it, in this case RSA signatures with the hash function SHA-1 and, of course, the issuer's signature itself. This signature is computed by hashing with SHA-1 the first part of the certificate (containing the relevant data that includes the user identification and public key) and applying the issuer's RSA private key. The signature, and hence the certificate validity, may then be verified using the issuer's RSA public key.

Finally, under *Extensions*, the certificate contains some additional data, including *Key Usage*, which specifies the purpose of the public key in the certificate, and *Netscape Certificate Type*, which specifies the purposes the certificate can be used for in relation to some common standards. For example, in this case, `SSL Client` means that the certificate subject may act as an SSL client (but not, for example, as an SSL server) and `S/MIME` that the certificate may be used for S/MIME signing and encryption.

9.7.2 Multiple Certification Authorities

The model sketched above, in which a single CA issues certificates for all users, is simple but difficult to implement in practice. The world being as it is, it seems unthinkable that everyone would trust the same CA and, in fact, many countries have their own CAs and so do many private organizations. On the other hand, a hypothetic

single CA would clearly entail a great risk because if it does not perform its activity correctly—be it because of corruption, inefficiency or any other reason—then its failure would mean the failure of the entire system.

What happens in practice is that there are multiple CAs and this means that it is virtually impossible for a user to have access to the public keys of all of them. However, a user A may choose from CAs which issue a certificate and may have certificates from different CAs issued to her for the same public key. Then A can offer these certificates to other users who can choose the CA they trust among all those that emitted certificates to A. For example, it is common that web browsers come with a number of CA public keys pre-loaded in their certificate store, enabling the user to select those she is willing to trust.

There is, however, a problem inherent to the use of multiple CAs and this is the fact that two different users may trust different CAs but nevertheless want to communicate securely with each other. Suppose, for example, that A trusts CA_1 and has CA_1's public key pk_{CA_1} as well as a certificate for her public key issued by CA_1. Similarly, B has a certificate issued by CA_2 as well as CA_2 public key. A might send to B her certificate to allow B to make use of her public key but B cannot, in principle, authenticate this key because he does not have pk_{CA_1}. However, suppose that B obtains a certificate issued by CA_2 containing pk_{CA_1}. Then B is assured that A's key is authentic by making two verifications. First B verifies CA_1's public key with CA_2's public key and then he verifies A's public key with CA_1's public key. This way, a certificate chain (called a *chain of trust*) is established. Note that in this chain, a *delegation of trust* is produced when B not only verifies CA_1's public key but also trusts CA_1 as an issuer of certificates.

Situations like the one just described lead to the approach of making CAs certify each other. However, it might happen that not all CAs certify each other and so, in practice, CAs are often arranged hierarchically, so that CAs issue certificates to lower-level CAs. In a *strict hierarchy model* there is a *root* CA, which has a self-issued self-signed certificate and may issue certificates to lower-level CAs which, in turn may issue certificates to end users. Then if B wants to verify the certificate of another user A, he only needs a chain of trust from the root CA to A, for example

$$\mathrm{CA}_{\mathtt{root}} \rightarrow \mathrm{CA}_1 \rightarrow A$$

meaning that $\mathrm{CA}_{\mathtt{root}}$ has issued a certificate to CA_1 which, in turn, has issued a certificate to A. Of course, the chain of trust may contain more intermediate CAs and, on the other hand, there are more general models than that with a single root CA. These models may include multiple root CAs, mutually certifying CAs, and other configurations.

Another trust model is the one used by web browsers which, as we have already mentioned, come with a pre-configured set of CAs' public keys. There is no cross-certification between these CAs but the browser itself acts as a sort of virtual root CA for all of them because the user is trusting the browser company to include only valid CAs. This model has several security weaknesses: for example, browsers usually allow users—after being warned—to accept certificates that are not signed by one of

the CAs included in the browser. Another security issue is that of expiration dates, which leads to the possibility of the user being allowed to accept a certificate even though it has expired. Thus expiration dates often become meaningless in practice, since users tend to accept expired certificates anyway.

Yet another PKI model is the "web trust model" in which each user is his own CA. In this model, a variant of which is adopted by the PGP email encryption software, anyone can issue certificates to anyone else and each user decides whether or not to trust certificates issued by other users. Therefore, users are expected to collect public keys of other users as well as certificates of these public keys issued by other users and there is also the possibility of storing public keys and certificates in a central database. Whether the public keys and certificates come directly from users or from a database, it is up to each user to decide how much trust to place in any of these public keys.

The web of trust has the advantage of not requiring trust in a central authority but, in some sense, it is not a true PKI because it requires users to set trust levels by themselves. This requires a certain degree of technological knowledge and does not seem adequate for environments in which security is essential. Another drawback is that there is no revocation mechanism for invalid certificates and no assurance that forged certificates will be detected. However, as is the case with the PGP software, it works pretty well for encrypting email for average users.

As we have seen, certificates usually contain an expiration date after which they are not valid and should not be accepted. When users verify certificates they are supposed not only to check the validity of the signature but also the validity period. However, this does not offer a good solution for situations in which a certificate should cease to be valid despite its expiration date not having yet arrived. For example, it might happen that a certificate is stored in a smart card which is lost or stolen. Another common reason might be that an employee left an organization losing the right to use the public key that the organization had supplied. In order to deal with these situations, it must be possible for the CA to *revoke* certificates by issuing a *certificate revocation list* (CRL, for short) that contains a list of the serial numbers of the revoked certificates and signing it.

For the CRL to serve its purpose users should check, in addition to the verifications already mentioned, that the serial number of the certificate does not appear in the latest CRL issued and should also check the signature of the CA in the CRL itself. The main drawback of CRLs is that they have to be transmitted to users and, if this transmission is not done frequently, then there are time intervals in which a certificate might have been revoked without users being aware of this fact. On the other hand, contacting the CA each time one wants to use a certificate—in order to learn whether the certificate has been revoked—is not practical. Thus the best approach seems to be to update CRLs relatively often but, hopefully, not so often as to provoke network bandwidth problems.

As can be gleaned from the previous remarks, key management is a very complex process and is perhaps the weakest component in the implementation of public-key cryptography. There are important practical difficulties associated with the development of PKIs, ranging from the determination of who should be responsible for

their development and maintenance to the problem of developing adequate standards and applications compatible with these standards. Because of these difficulties, the development of large-scale PKIs has proceeded slowly. We refer to [2] for an in-depth discussion of PKIs and related issues.

Chapter 10
Identity-Based Cryptography

We have observed in Chap. 9 that the current state of PKI deployment is far from satisfactory. There are many CAs and many of the certificates currently being used are either expired or issued by CAs which are unknown to users or to the users' software. In other cases, the users themselves issue their own certificates and then trust chains are established in which friends are trusted, and then friends of friends, and so on, with trust weakening in each iteration. These usages are not in accordance with the intended one and it may be said that the purpose of PKI is defeated by them. On the other hand, the wide deployment of public-key encryption is also hindered by the fact that each time a user wants to send an encrypted message to a new person, a new public-key certificate for this person must be obtained. These issues are a big drawback for the wide deployment of public-key cryptography and hence it is very interesting that there is a nice solution for a large fraction of these problems. This solution is called *identity-based cryptography* (IBC) which, in particular, includes the somewhat more restrictive concept of *identity-based encryption* (IBE). In IBC, the schemes may be devoted to encryption, digital signatures, key exchange or other purposes, while in the case of IBE the goal of the scheme is encryption. The development of fully functional IBE schemes is quite recent—as we shall see, it may be said that it started around 2001—but identity-based cryptography holds great promise for the future and hence we will dedicate a short chapter to introduce its basic ideas and point the interested reader to the relevant literature.

10.1 Introducing Identity-Based Cryptography

The basic idea of IBC starts from the realization that there is some minimal information that a user has to learn before communicating with another person, even in an unencrypted form, namely some identity information such as, for example, the email address of this person. This was the starting point of the concept introduced by Shamir in [175], where he proposed that this basic information could replace the

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_10, © Springer-Verlag Berlin Heidelberg 2013

need for a public key or, in slightly different terms, that the public key of a user would be her identity string (or easily derivable from this identity by, for example, applying a hash function to it). This is precisely the basic definition of IBC.

It is readily apparent that such a framework has enormous practical advantages in comparison with the usual certificate-based public-key encryption. In the first place, any user could send encrypted messages to anyone without having to worry about the public key, so that the only additional requirement for the system to work properly would be that the receiver knows her private key. It is then clear that a crucial requirement for the establishment of IBC is that these private keys must be computed from the public identity in a way that makes it infeasible to compute them without additional trapdoor information. This is in some sense analogous to what happens to the relation between public and private keys in standard public-key cryptography but there is, nonetheless, an essential difference. When computing a public key/private key pair in standard public-key cryptography, the user (or the trusted party that computes the key) may start by computing some trapdoor information that, in turn, may be used to generate the public key, i.e., the private key is generated first and then the public key is computed from it. In the case of IBC, this method is not viable because the public key is already given by the public identity. Thus the private key must be obtained from the given public key and, if keys were generated by the users themselves, then each user would know how to obtain her private key from her public key and hence, using the same method, would also be able to compute the private key of any other user from that user's public identity. The system cannot take advantage of any user-specific information other than the public identity and hence the private key must be computed based on global trapdoor information which is common to all users of the system. As a consequence, the owner of this trapdoor information must be a trusted authority able to compute and distribute the keys of all users in a secure way. This owner is usually called the *Private Key Generator* (PKG) or also the *Key Generation Center*.

The nature of the PKG makes the trust issue even more crucial than in the case of standard public-key cryptography because all the security of the system relies on this single central authority that must be trusted by all users. Moreover, the PKG requires an even higher level of trust than a CA since users have to trust its ability to keep the global trapdoor information secret and to provide each of them with a private key after verifying their identity. In addition, in contrast with a CA which only issues certificates of public keys generated by users, the PKG must be trusted not to make use of the private keys it generates—which would allow it to decrypt all the messages sent by users—and not to deliver private keys to users different from the legitimate one. Thus IBC implies a strong form of key escrow and to alleviate the security risk that this entails, techniques from threshold cryptography have been proposed to allow the distribution of the master key among several "partial" PKGs, in such a way that a specified minimum number of them must cooperate in order to derive a private key.

Apart from trust issues, there are other aspects that have to be carefully considered in order to design a useful and secure IBE scheme. One of them is that a reliable method to associate an identity string with every user is needed. Another is the issue

of revocation that must also be dealt with in this context. Thus the design of IBE schemes is a highly nontrivial endeavor for which there are no automatic methods. In particular, there are no obvious ways to turn a standard public-key encryption scheme into an identity-based scheme. We shall give a more formal definition below but, for now, let us give an informal description of an IBE scheme.

An IBE scheme is a family of four algorithms, called **Setup, Der** (also called **KeyDer** or **Extract**), **Enc** and **Dec**. **Setup** is run by the PKG and generates the system parameters *mpk* (the *master public key*) and *msk* (the *master private key*); the first is public and the second is known only to the PKG. The pair $mk = (mpk, msk)$ is called the *master key*. The key derivation algorithm **Der** is also run by the PKG (on behalf of the users that request a private key) which, after checking the identity of a user, generates the private key, say, *usk*. **Enc** and **Dec** are the encryption and decryption algorithms and behave similarly to their counterparts in standard public-key cryptography, with the user identity acting as the public key in **Enc.**

In order to better appreciate the difficulties involved in trying to produce an IBE scheme from a standard public-key encryption scheme, let us examine an attempt to derive an IBE scheme from the RSA encryption scheme.

Example 10.1 A seemingly plausible idea to turn RSA into an IBE scheme would be to let all users share a common modulus, with the public key of a user being the encryption exponent and the private key the decryption exponent. Thus the scheme would be defined by the following algorithms:

- **Setup**: The PKG runs $\mathbf{Gen}_{\mathrm{RSA}}$ and generates an RSA modulus $n = pq$. The master public key is $mpk = n$ and the master private key is $msk = (p, q)$.
- The encryption exponent e (i.e., the public key) of a user is obtained by applying a publicly known hash function to the user identity string.
- **Der**: Given a user's identity the PKG uses the master key to compute the user's private key (the decryption exponent d).
- **Enc** and **Dec** are given by the RSA functions $\mathrm{RSA}_{(n,e)}$ and $\mathrm{RSA}_{(n,d)}$, respectively.

It is easy to see that this scheme is a complete failure from a security point of view. The reason is that, as we have seen, a user that knows (n, e, d), where n is an RSA modulus, e an encryption exponent and d the decryption exponent corresponding to the pair (n, e), may use either Miller's PPT algorithm or the Coron–May deterministic algorithm referred to in Sect. 8.3.1 to compute the prime factors of n and hence the master key $mk = (n, (p, q))$.[1] As a consequence, any user of the system would be able to recover the master key and hence to compute the private key of any other user, and could also issue private keys to new users. The problem of producing, from a user identity, an RSA modulus with a global trapdoor that only allows the PKG (but not users) to factor the modulus is still open.

[1] Alternatively, the result given in Proposition 8.5 may be used to mount a 'common modulus attack' which allows any user to decrypt messages addressed to another user who shares the same RSA modulus.

10.2 Identity-Based Signatures

Before studying IBE in more detail, we will have a brief look at identity-based signatures (IBS). IBS schemes are much easier to construct than IBE schemes. The reason is that in IBS schemes the first usage of a cryptographic primitive is made by the private key owner when computing the signature, allowing her to embed in the message the necessary information about her public key. In contrast, in IBE, the starting point is the encryption of a message and, in that case, the user performing it must derive all the information from the public system parameters and the identity string of the receiver. Because of this asymmetry, while full-featured IBE schemes were introduced only recently and there are not many of them, a great variety of IBS schemes are known, the first one having already been proposed by Shamir in his 1984 paper [175].

10.2.1 IBS Schemes

Before describing how to construct IBS schemes, we give the formal definition of such a scheme, which is the following:

Definition 10.1 An *identity-based signature scheme* (or an *IBS scheme*, for short) is a 4-tuple $IB\Sigma = (\mathbf{Setup}, \mathbf{Der}, \mathbf{Sign}, \mathbf{Ver})$ of polynomial-time algorithms, where the first three may be probabilistic and **Ver** is deterministic, that act as follows:

- **Setup** is run by the PKG and, on input a security parameter 1^k, outputs a master key $mk = (mpk, msk)$. The master public key mpk is published as a system parameter.
- **Der** is run by the PKG and, on input msk and the identity string $id \in \{0,1\}^*$ of a user, outputs the user's private key $usk \leftarrow \mathbf{Der}(msk, id)$, which is then securely transmitted to the user with identity id.
- The signing algorithm **Sign** takes as input a user private key usk and a message m, and outputs a signature $\sigma \leftarrow \mathbf{Sign}(usk, m)$.
- The verification algorithm **Ver** takes as input the master public key mpk, a user identity string id, a message m and a signature σ, and outputs a bit $b := \mathbf{Ver}(mpk, id, m, \sigma) \in \{0,1\}$. $b = 1$ is interpreted as meaning *valid* and $b = 0$ as meaning *invalid*.

We require that $\mathbf{Ver}(mpk, id, m, \mathbf{Sign}(usk, m)) = 1$ whenever mpk and usk have been generated as indicated above.

In order to define the concept of unforgeability under a chosen message attack for IBS schemes, we consider the following experiment:

Definition 10.2 The *IB signature unforgeability experiment under an adaptive chosen message attack* $\mathbf{IBSign}^{uf\text{-}cma}_{\mathcal{A},IB\Sigma}(k)$ is the following:

1. **Setup** is run on input 1^k, obtaining a master key $(mpk, msk) \leftarrow \mathbf{Setup}(1^k)$.

2. The adversary $\mathcal{A}$ is given the master public key mpk and oracle access to $\mathbf{Der}(msk, -)$ and to $\mathbf{Sign}(\mathbf{Der}(msk, -), -)$ which, on input an identity id and a message m, returns $\mathbf{Sign}(usk, m)$, where $usk \leftarrow \mathbf{Der}(msk, id)$. $\mathcal{A}$ asks a polynomial number of queries to the oracles and outputs a 3-tuple (id', m', σ'), consisting of an identity string, a message and a signature.
3. The experiment output is defined to be 1 if and only if $\mathbf{Ver}(mpk, id', m', \sigma') = 1$ and neither $\mathbf{Der}(msk, id')$ nor $\mathbf{Sign}(\mathbf{Der}(msk, id'), m')$ were queried to the oracles.

The security definition is then the following:

Definition 10.3 An IBS scheme $IB\Sigma = (\mathbf{Setup, Der, Sign, Ver})$ is *existentially unforgeable under an adaptive chosen-message attack* (UF-CMA, for short) if, for every PPT adversary $\mathcal{A}$, there exists a negligible function $\mathtt{negl}$ such that:

$$\Pr(\mathbf{IBSign}^{uf\text{-}cma}_{\mathcal{A},IB\Sigma}(k) = 1) \leq \mathtt{negl}(k).$$

10.2.2 From Signature Schemes to IBS Schemes

We next present a generic construction, called SS-2-IBS, that produces an IBS scheme from any standard signature scheme. This construction is made possible by the fact that, as remarked above, in a signature scheme the sender is the private key owner. Therefore, the private key may include the private key of a standard signature scheme plus a certificate for the corresponding standard public key signed by the PKG. Then the user signs the message with the standard private key and appends to the signature her standard public key and the corresponding certificate. This construction can be more formally described as follows:

SS-2-IBS construction.

Given a standard signature scheme $\Sigma = (\mathbf{SGen, SSign, SVer})$, an IBS scheme $Cert\text{-}IB\Sigma = \mathbf{SS\text{-}2\text{-}IBS}(\Sigma)$ is constructed as follows:

- $Cert\text{-}IB\Sigma = (\mathbf{Setup}, \mathbf{Der}, \mathbf{Sign}, \mathbf{Ver})$, where the algorithms are specified below.
- **Setup**: The PKG runs **SGen** on input 1^k and obtains:

$$(mpk, msk) \leftarrow \mathbf{SGen}(1^k).$$

  mpk is published as a system parameter and msk is the master private key.
- **Der: SGen** is run on input 1^k to obtain a public key/private key pair for Σ, $(pk, sk) \leftarrow \mathbf{SGen}(1^k)$. The PKG computes $cert \leftarrow \mathbf{SSign}(msk, pk||id)$, namely, the concatenation of the public key with the identity is signed with the

master private key.[2] The output of **Der** is the user private key:

$$usk := (sk, pk, cert).$$

- **Sign**: The signing algorithm takes as input a user private key *usk* and a message m, and produces a signature σ as follows:
  - *usk* is parsed into its components (*sk, pk, cert*).
  - The standard Σ-signature $s \leftarrow \mathbf{SSign}(sk, m)$ is computed.
  - The output of **Sign** is the signature $\sigma := (s, pk, cert)$.
- **Ver**: The verification algorithm takes as input the master public key *mpk*, the user identity *id*, a message m, and the signature $\sigma = (s, pk, cert)$ and outputs a bit b. $b = 1$ if and only if

$$\mathbf{SVer}(pk, m, s) = 1 \text{ and } \mathbf{SVer}(mpk, pk||id, cert) = 1.$$

  As usual, $b = 1$ is interpreted as *valid* and $b = 0$ as *invalid*.

This construction is interesting not only because it gives an automatic procedure to obtain IBS schemes from standard signatures but also because the security of the IBS schemes thus obtained may be reduced to that of the standard signature schemes used in the construction. This is discussed in detail in [17], where the following result is proved:

Theorem 10.1 *Let Σ be a UF-CMA secure standard signature scheme. Then the scheme Cert-IBΣ is a UF-CMA secure IBS scheme.*

Exercise 10.1 Write a Maple version of the IBS scheme obtained by applying the SS-2-IBS construction to the implementation of PSS signatures given in Chap. 9.

Remark 10.1 We have seen in Chap. 9 that UF-CMA secure schemes exist even without assuming random oracles—for example, the Cramer–Shoup signature scheme. From these schemes and the SS-2-IBS construction one obtains UF-CMA secure IBS schemes. We have also mentioned in Chap. 9 that, as was shown in [166], the existence of UF-CMA secure schemes is equivalent to the existence of one-way functions. From this fact and the preceding theorem, a result of considerable theoretical interest follows, namely:

- *The existence of UF-CMA secure IBS schemes is equivalent to the existence of one-way functions.*

Indeed, if one-way functions exist, then UF-CMA secure IBS schemes exist by what we have just seen and, for the converse, it suffices to observe that any secure IBS scheme defines a secure standard signature scheme by including an arbitrary

[2] This can be interpreted as the PKG creating a certificate stating that "the public key of user *id* is *pk*".

identity *id* in the public key of the standard scheme and including the user private key corresponding to *id* in the private key. For more details about the SS-2-IBS construction and related questions we refer to [17].

The SS-2-IBS construction is by no means the only generic construction of IBS schemes. This and other generic constructions are analyzed in the survey article [111], which gives a nice overview of IBS and also includes other IBS schemes that are constructed directly, without being derived from any of these constructions.

10.3 Identity-Based Encryption

We have seen that IBS schemes are easily obtained from standard signature schemes but the situation regarding IBE schemes is very different, for no generic constructions are known to transform a standard public-key encryption scheme into an IBE scheme. We have also seen that a naive approach such as that based on the use of an RSA common modulus does not work. However, fully functional IBE schemes have been constructed in recent years. In order to present some of them, we start with the formal definition of an IBE scheme.

10.3.1 IBE Definition

Definition 10.4 An *IBE scheme* is a 4-tuple $\mathcal{E} = (\mathbf{Setup}, \mathbf{Der}, \mathbf{Enc}, \mathbf{Dec})$ of polynomial-time algorithms such that the first three of them are probabilistic and **Dec** is deterministic, satisfying the following conditions:

- **Setup** is run by the PKG and, on input a security parameter 1^k, outputs a master key $mk = (mpk, msk)$. The master public key *mpk* is published as a system parameter. There may be additional system parameters such as descriptions of the message space, the ciphertext space and the identity space, all of which are public and may be used as inputs by the remaining algorithms.
- **Der** is run by the PKG and, on input *msk* and the identity string $id \in \{0,1\}^*$ of a user, outputs the user's private key $usk \leftarrow \mathbf{Der}(msk, id)$ which is then securely transmitted to the user with identity *id*.
- The *encryption algorithm* **Enc** takes as input the master public key *mpk*, an identity *id*, and a message *m*, and outputs a ciphertext *c*, written $c \leftarrow \mathbf{Enc}(mpk, id, m)$.
- The *decryption algorithm* **Dec** takes as input the master public key, a user private key *usk* and a ciphertext *c* (and, possibly, an identity *id*) and outputs either a plaintext *m*, or a symbol $\perp$ denoting failure. We write $m := \mathbf{Dec}(mpk, usk, c)$.
- There exists a negligible function $\mathtt{negl}$ such that, for every *k*, every $(mpk, msk) \leftarrow \mathbf{Setup}(1^k)$, every $usk \leftarrow \mathbf{Der}(msk, id)$ and every message *m*:

$$\Pr(\mathbf{Dec}(mpk, usk, \mathbf{Enc}(mpk, id, m)) \neq m) \leq \mathtt{negl}(k).$$

The security definition for standard encryption schemes given by IND-CCA was extended by Boneh and Franklin in [35] to identity-based encryption schemes as follows:

Definition 10.5 The *identity-based CCA indistinguishability experiment* under an adaptive chosen message attack, $\mathbf{IBE}^{ind\text{-}cca}_{\mathcal{A},\mathcal{E}}(k)$, is the following:

1. **Setup** is run on input 1^k obtaining a master key $(mpk, msk) \leftarrow \mathbf{Setup}(1^k)$.
2. The adversary $\mathcal{A}$ is given the master public key mpk and oracle access to $\mathbf{Der}(msk, -)$ and to $\mathbf{Dec}(mpk, \mathbf{Der}(msk, -), -)$ which, on input an identity id and a ciphertext c, returns $\mathbf{Dec}(mpk, usk, c)$, where $usk \leftarrow \mathbf{Der}(msk, id)$. $\mathcal{A}$ asks a polynomial number of queries to the oracle and outputs an identity id—subject to the constraint that the **Der** oracle was not queried on id—and a pair of messages of the same length, m_0, m_1, belonging to the plaintext space.
3. A random bit $b \leftarrow \{0, 1\}$ is chosen and the challenge ciphertext $c \leftarrow \mathbf{Enc}(mpk, id, m_b)$ is computed and given to $\mathcal{A}$.
4. $\mathcal{A}$ continues having access to the oracles but cannot query the **Der** oracle on the identity id nor the decryption oracle on the pair (id, c). Finally, $\mathcal{A}$ outputs a bit b'.
5. The output of the experiment is defined to be 1 if $b' = b$ and 0 otherwise. In the first case, $\mathcal{A}$ *succeeds*.

The security definition is then the following:

An IBE scheme $\mathcal{E} = (\mathbf{Setup}, \mathbf{Der}, \mathbf{Enc}, \mathbf{Dec})$ is *indistinguishable under an adaptive chosen ciphertext attack* (IND-ID-CCA, for short) if, for every PPT adversary A, there exists a negligible function `negl` such that

$$\Pr(\mathbf{IBE}^{ind\text{-}cca}_{\mathcal{A},\mathcal{E}}(k) = 1) \leq 1/2 + \mathtt{negl}(k).$$

Remark 10.2 The adversary of the IB experiment $\mathbf{IBE}^{ind\text{-}cca}_{\mathcal{A},\mathcal{E}}(k)$ is much more powerful than that of the standard IND-CCA experiment $\mathbf{PubK}^{ind\text{-}cca}_{\mathcal{A},\mathcal{E}}(k)$ because it can choose an arbitrary identity id and can query the **Der** oracle to obtain any user private key not associated with id. This is because when the adversary attacks a public key id, the system should remain secure even if the adversary knows the private keys of other users. Another, more subtle, difference between IND-CCA and IND-ID-CCA is that, while in the former the adversary is challenged on a random public key, in the latter it is challenged on an id that the adversary itself is allowed to choose.

10.3.2 Applications of IBE

IBE has many applications that take advantage of its special features to achieve things that cannot be done with standard public-key cryptography. We mention a couple of them taken from the Boneh–Franklin paper where the first fully functional IBE scheme was introduced.

10.3.2.1 Revocation of Public Keys

As we have seen in Chap. 9, public-key certificates contain an expiration date after which the public key is no longer valid. In an IBE scheme, key expiration may easily be implemented by using public keys—i.e., identity strings—of the form

"bob@company.com||time-period"

where time-period may be, for example, current-year, current-month or current-date, or any other specified time interval. If users sending messages to Bob use the identity string corresponding to the current time-period, Bob can only use his private key to decrypt messages during the specified time interval, after which he has to obtain a new private key corresponding to the new time interval. However, for Alice, who sends encrypted messages to Bob, the process is transparent: she does not need to obtain a new certificate from Bob each time his private key is refreshed. This procedure would be adequate, for example, for a corporation that maintains the PKG for its employees. This way, if Bob leaves the company, the PKG stops issuing private keys for his email address and that is all. Another interesting consequence, pointed out in [35], is that this approach enables Alice to send messages into the future, since Bob will be able to decrypt them only in the specified time interval.

10.3.2.2 Delegation of Decryption Keys

Suppose now that Bob is a manager who has several assistants, each responsible for a different task. Then Bob can act as the PKG, generating his own IBE system parameters (say, params) and his own master key. params may be regarded as Bob's public key which can be used by his assistants to send messages to him but only Bob knows his master key. Then Bob gives his assistants the private keys corresponding to their responsibilities—so that, for example, the assistant in charge of purchasing will be given the private key corresponding to the identity "purchasing". Each assistant can then decrypt the messages whose subject falls within her responsibilities but not those intended for other assistants. Of course, Bob can decrypt all these messages by using his master key.

10.3.3 The Cocks IBE Scheme

We are now going to give our first example of an IBE scheme, namely, the elegant scheme introduced by Cocks in [51], which is based on the hardness of the quadratic residuosity problem modulo an RSA modulus. This was the second fully-functional IBE scheme proposed, shortly after the Boneh–Franklin scheme that will be discussed later. As we shall see, the main drawback of the Cocks scheme is the high message expansion it produces, which is even higher than that of the Goldwasser–Micali

encryption scheme that we studied in Sect. 8.8. Indeed, the ciphertext corresponding to a single bit is essentially a pair of elements of $\mathbb{Z}_n$, where n is an RSA modulus. Thus the expansion factor is about $2\log_2 n$ and hence, if we want to encrypt, say, a 256-bit symmetric key using a 2048-bit RSA modulus, we will need $2 \cdot 256 \cdot 2048$ bits or, equivalently, 1 mebibit (the 'binary version' of 1 megabit). There is another IBE scheme, due to Boneh, Gentry and Hamburg [36] that is also based on quadratic residuosity and achieves a ciphertext size of $l + \log_2 n$ for an l-bit message. The trade-off, however, is a substantial increase of encryption time so that, for now, pairing-based IBE schemes—which will be presented later on—are preferred.

Definition 10.6 *Cocks IBE scheme*.
Like every IBE scheme, the Cocks scheme uses a set of public system parameters that specify the message space, the ciphertext space and the identities space. We will not make them explicit but these parameters may be used by any of the algorithms belonging to the scheme, as may some other public system parameters that will be specified below. The Cocks scheme is the IBE scheme (**Setup, Der, Enc, Dec**), where the algorithms are defined as follows:

- **Setup**: On input 1^k, the PKG runs the modulus generation algorithm to obtain (n, p, q), where $n = pq$ and p, q are primes such that $\text{len}(p) = \text{len}(q) = k$. The master key is then the pair $(n, (p, q))$, where n is the master public key and (p, q) the master private key. Moreover, a random $u \leftarrow \mathcal{QN}_n^1$ is chosen[3] and published as a system parameter. The remaining public system parameter is a hash function H mapping user identities to $\mathcal{J}_n$.
- **Der**: On input the master private key (p, q) and an identity string id, the PKG proceeds as follows. First, it computes $R := H(id) \in \mathcal{J}_n$. If $R \in \mathcal{QR}_n$, then it sets $r := R^{1/2} \bmod n$ (i.e. r is a square root of the quadratic residue R in $\mathbb{Z}_n^*$), otherwise it sets $r := (uR)^{1/2} \bmod n$. The output is r, the private key of the user with identity id.
- **Enc**: On input the master public key n, an identity id and a one-bit message $m \in \{0, 1\}$, compute $R := H(id)$ and $b := (-1)^m$. Choose two random elements $t_i \leftarrow \mathbb{Z}_n$ with $i \in \{0, 1\}$ and compute:

$$d_i := (t_i^2 + u^i R)t_i^{-1} \bmod n, \text{ and } c_i := b \cdot \left(\frac{t_i}{n}\right).$$

  Output the ciphertext $((d_0, c_0), (d_1, c_1))$.
- **Dec**: On input the public parameters, the user private key r, the identity id, and the ciphertext $((d_0, c_0), (d_1, c_1))$, set $i \in \{0, 1\}$ such that $r^2 \equiv u^i R \pmod n$, where $R := H(id)$. Then set $g := (d_i + 2r) \bmod n$ and compute $b := c_i \cdot \left(\frac{g}{n}\right)$. If $b = 1$ then output 0 and if $b = -1$, then output 1.

Let us now prove the correctness of the scheme:

[3] Recall here that $\mathcal{J}_n$ denotes the subset of the elements of $\mathbb{Z}_n^*$ with Jacobi symbol modulo n equal to $+1$ and $\mathcal{QN}_n^1$ is the intersection of $\mathcal{J}_n$ with the set of quadratic non-residues modulo n.

Proposition 10.1 *In the Cocks IBE scheme we have that, using the preceding notation, for any id such that r is the private key corresponding to this identity and any* $m \in \{0, 1\}$, $\mathbf{Dec}(n, r, id, \mathbf{Enc}(n, id, m)) = m$.

Proof Keeping the notations above we have to show that, if $b = (-1)^m$, $c_i = b \cdot \left(\frac{t_i}{n}\right)$, $d_i = (t_i^2 + u^i R)t_i^{-1} \bmod n$ and $g = (d_i + 2r) \bmod n$, for $i \in \{0, 1\}$ such that $r^2 \equiv u^i R \pmod{n}$, then $b = c_i \cdot \left(\frac{g}{n}\right)$. We have:

$$g = (d_i + 2r) \bmod n = ((t_i^2 + u^i R)t_i^{-1} + 2r) \bmod n = (t_i + r)^2 t_i^{-1} \bmod n. \quad (10.1)$$

Since $\mathcal{QR}_n \subset \mathcal{J}_n$ and $\left(\frac{x}{n}\right) = \left(\frac{x^{-1} \bmod n}{n}\right)$ for every $x \in \mathbb{Z}_n^*$ (because of the multiplicative property of the Jacobi symbol and the fact that $\left(\frac{1}{n}\right) = 1$), we see from Eq. 10.1 that

$$\left(\frac{g}{n}\right) = \left(\frac{t_i^{-1} \bmod n}{n}\right) = \left(\frac{t_i}{n}\right)$$

and hence

$$c_i \cdot \left(\frac{g}{n}\right) = b \cdot \left(\frac{t_i}{n}\right)\left(\frac{g}{n}\right) = b\left(\frac{g}{n}\right)^2 = b,$$

so that the correct plaintext is recovered. □

Remarks 10.1

1. The purpose of the randomly chosen element $u \leftarrow \mathcal{QN}_n^1$ is to make either R or uR a quadratic residue modulo n and hence to make it possible to extract its square root in $\mathbb{Z}_n^*$, which is easy if one knows the factorization of n (i.e., the master private key of the scheme, which is a possession of the PKG) but is presumed to be difficult without knowing this factorization; this is precisely the quadratic residuosity assumption, see Definition 8.18. Since $R = H(id) \in \mathcal{J}_n$, if $R \notin \mathcal{QR}_n$ then we have that $\left(\frac{R}{p}\right) = \left(\frac{R}{q}\right) = -1$ and, on the other hand, $u \in \mathcal{QN}_n^1$ also implies that $\left(\frac{u}{p}\right) = \left(\frac{u}{q}\right) = -1$. Thus in this case we have, by the multiplicative property of the Jacobi symbol, that $\left(\frac{uR}{p}\right) = \left(\frac{uR}{q}\right) = 1$ and so $uR \in \mathcal{QR}_n$ by Proposition 2.14. Note also that a random $u \leftarrow \mathcal{QN}_n^1$ is easy to choose if the factorization of n is known because, as discussed in relation with the Goldwasser–Micali encryption scheme, it suffices to choose an element whose Legendre symbols with respect to both p and q are equal to -1.
2. In the original version of the scheme in [51], the prime factors p, q of the modulus n are taken to be Blum primes, i.e., such that $p \equiv q \equiv 3 \pmod 4$. In this case, both $(p-1)/2$ and $(q-1)/2$ are odd and hence, as a consequence of Euler's criterion (Proposition 2.12) we have that $\left(\frac{-1}{p}\right) = \left(\frac{-1}{q}\right) = -1$. Then $\left(\frac{-1}{n}\right) = \left(\frac{-1}{p}\right)\left(\frac{-1}{q}\right) = 1$ and hence $-1 \in \mathcal{QN}_n^1$. Thus we can choose

$u = -1 \bmod n$ and, although this value is not randomly chosen, this does not affect the security of the scheme.

3. The reason why the ciphertext consists of the two pairs (d_0, c_0) and (d_1, c_1) instead of just the one that will be used for decryption is that the sender does not know which one of the values R, uR is a quadratic residue modulo n. Therefore, sufficient information to cover both possibilities must be sent, which effectively doubles the size of the ciphertext.
4. The hash function H must apply identity strings to elements of $\mathcal{J}_n$ and hence it may be constructed from a cryptographic hash function such as SHA-256 by means of a public deterministic algorithm, so that anyone knowing the public parameters and id can compute $H(id) = R$. For example, the cryptographic hash function can be applied to obtain a value in $\mathbb{Z}_n$ which is then incremented until reaching the first value whose Jacobi symbol is $+1$.
5. The elements t_0, t_1, chosen by the encryption algorithm, are supposed to belong to $\mathbb{Z}_n^*$ (where all computations take place) and so one might choose them in $\mathbb{Z}_n$ as indicated and check that they indeed belong to $\mathbb{Z}_n^*$, discarding them if this is not the case. However, this check can be dispensed with because such an element does not belong to $\mathbb{Z}_n^*$ only with negligible probability. In fact, if we find an element of $\mathbb{Z}_n$ that does not belong to $\mathbb{Z}_n^*$ then, by computing its gcd with n, we obtain the factorization of n.
6. As usual with public-key encryption schemes, Cocks IBE will be used mostly in hybrid schemes to encrypt session keys, i.e., symmetric keys which, in turn, are used to encrypt data. The scheme is efficient enough for this purpose because, if the session key is l bits long, then the cost of encryption is dominated by the computation of $2l$ Jacobi symbols—which can be efficiently computed using Algorithm 2.7—and $2l$ divisions modulo n. Similarly, the cost of decryption is dominated by the computation of l Jacobi symbols. As mentioned above, the main drawback of the scheme is the space requirement which is high because it uses, for each bit, two numbers of size close to the size of n.

Exercise 10.2 Suppose that $n = pq$ with $p \equiv q \equiv 3 \pmod 4$ and let $R \in \mathcal{QR}_n$ be a quadratic residue modulo n. Prove that a square root of R in $\mathbb{Z}_n^*$ is given by $r = R^{\frac{n-(p+q)+5}{8}} \bmod n$.

Exercise 10.3 Explain the reasons why if, in the encryption and decryption algorithms of Definition 10.6 one encodes the bit m as $b = m$ (i.e., $b \in \{0, 1\}$, instead of $b \in \{-1, 1\}$), then one obtains a scheme which is completely insecure.

Let us say a few words about the security of the Cocks IBE scheme. It is clear that the scheme can be broken if the modulus n is factored and so, for its practical implementation the size of the modulus should be similar to the sizes we have discussed for RSA moduli. It can be shown that Cocks IBE is IND-ID-CPA secure[4] in

[4] The definition of IND-ID-CPA is obtained from the definition of IND-ID-CCA above by dropping the decryption queries.

the random oracle model (where the hash function H is modeled as a full domain random oracle) under the assumption that quadratic residuosity is hard. We refer to the original paper by Cocks as well as [36], where a quadratic residuosity-based IBE scheme which is space-efficient is presented and the security properties are analyzed in more detail. We remark, however, that this scheme is not IND-ID-CCA secure because, as it encrypts one bit at a time, in a CCA attack the ciphertext corresponding to, say, a single plaintext bit, could be inserted by the adversary into a longer ciphertext submitted to the oracle, thus recovering the plaintext bit by bit.

On the other hand, there is a method to construct an IND-ID-CCA secure scheme from an IND-ID-CPA secure scheme (or even from an OW-CPA secure scheme). This is accomplished by a generic transformation due to Fujisaki and Okamoto which applies not just to IBE schemes but also to standard ones, see [79]. This transformation uses random oracles, so that the security of the schemes thus obtained is relative to the random oracle model. Until 2004, no IBE schemes that could be proved to be IND-ID-CCA secure without random oracles were known but, recently, several such schemes have been introduced. We refer to [36] and to several chapters in [107] for more details on IBE schemes and their security properties.

10.3.4 The Cocks IBE Scheme in Maple

Here we give a Maple implementation of the Cocks IBE scheme. As usual, we make use of several previously defined functions. For starters, we need a function to generate the modulus, which may be built on top of the `RSAKeyGen` function.

Exercise 10.4 Write a Maple function that calls `RSAKeyGen` and, on input parameters similar to those in the latter function, returns an RSA modulus and its prime factorization.

Since the encryption and decryption exponents are not needed here, we are going to proceed differently and we will construct the function `GenModulus` as a simplified form of `RSAKeyGen` which calls `RSAPrime` to generate the modulus. The inputs are the same as in `RSAKeyGen`, except the (optional) encryption exponent that does not appear here:

```
> GenModulus := proc(k::posint, seed1::{posint, string}, seed2::{posint, string},
                     {format::name := hex})
  uses RandomTools:-MersenneTwister;
  local s, p, q, m, n, key;
  if _params['seed1'] = NULL then
    SetState();
    s := GenerateInteger(':-range' = 2^(127) .. 2^(256))
  else
    s := stringposint(seed1)
  end if;
  p := RSAPrime(k, s);
  m := ceil(2^(2*k-1)/p);
  if _params['seed2'] = NULL then
     SetState();
     s := GenerateInteger(':-range' = 2^(127) .. 2^(256))
```

```
  else
    s := stringposint(seed2)
  end if;
  q := RSAPrime(k, s, min = m);
  n := p*q;
  key := [n, [p, q]];
  if format = decimal then
    key
  elif format = hex then
    LowerCase~~(convert~~(key, hex))
  else
    error "Unrecognized key format"
  end if
  end proc:
```

We also need a function to generate the system parameter $u \leftarrow \mathcal{QN}_n^1$ that will be used to turn the hash value of a user identity into a quadratic residue modulo n in case it is not already one. This value will be generated by the function `GenPar` below on input the prime factors of the modulus plus an optional parameter to tell the function whether the output will be decimal or hexadecimal.

```
> GenPar := proc(p::posint, q::posint, {format::name := hex})
  uses RandomTools:-MersenneTwister;
  local n, legp, legq, u;
  n := p*q;
  SetState();
  legp := 1;
  legq := 1;
  while legp = 1 or legq = 1 do
    u := GenerateInteger(':-range' = 1 .. n-1);
    legp := numtheory:-legendre(u, p);
    legq := numtheory:-legendre(u, q)
  end do;
  if format = decimal then
    u
  elif format = hex then
    StringTools:-LowerCase~~(convert~~(u, hex))
  else
    error "Unrecognized key format"
  end if
  end proc:
```

We now have in place all the ingredients we need to implement the Cocks IBE scheme. We start with the function that implements the `Setup` algorithm. This is the function `CocksIBESetup` below, which collects the data computed by the previous two functions and outputs the master key and the u parameter (see Definition 10.6 above). The master key, in turn, consists of the master public key—namely, the RSA modulus n—and the master private key, i.e., the factorization of the modulus. The input parameters of the function are the same as in `GenModulus` and the output is a 3-element list. The first item in the list is the modulus, the second a list with the two prime factors of the modulus, and the third element is the u-parameter. These values may be returned by the function in either decimal or hexadecimal format, with the latter being the default.

```
> CocksIBESetup := proc(k::posint, seed1::{posint, string}, seed2::{posint, string},
                        {format::name := hex})
  local m, p, q, u;
  m := GenModulus(args);
  p := stringposint(m[2][1]);
  q := stringposint(m[2][2]);
  u := GenPar(p, q, ':-format' = format);
```

```
    [op(m), u]
    end proc:
```

Example 10.2 As an example of the use of the previous function, we generate a master key with a 2048-bit modulus and the corresponding u-parameter. As in the case of RSA keys, the prime factors of the modulus are pseudo-randomly generated and in order to do this random seeds of 128 bits or more should be provided.

```
> params := CocksIBESetup(1024, "c9a71d49bf75130f37188769ceb0ed782a",
                          "e9e5d7742eb98a6c0c0f5710877326a6bb"):
```

Let us now have a look at these parameters. The master public key is:

```
> mpk := params[1];
"d5e9e97cc764bb430f34e5aa1f21d26f09a179a8860b6525ced8cc9eb876233cdf4b82397494f66ec\
  ed9ab9e1a300e4c443254b866047bb16c5d56013c59aa42e78cfa4d860fcd6617e6dc8e21dd72c09\
  31ab519d87a15416f3c02c264eb8fc3912870f02371d39810cf751bb28be7f645108817dde0151b6\
  de006551457387eed2f57f6ebf26134290b6a2f06256429ee7b66b121719e4759cbcdeb558679da1\
  e8e4d9eea6b5f4e9c059df512c7c86742525c4dff075cd35a007b37e242e3744fe5767eb819c14b3\
  514601259986f1710253c080530f61208f5b73786b95eff694b5ea2c60e9e51a8e2b80ef43277892\
  3dcd269b6a9a7417812cf3a0e01e4fd"
```

The master private key is:

```
> msk := params[2];
  ["d1ddbc669e789567c2f186b51c4fff1b4c597b54e7d79db7b329a2067434548ee7178efbf212af\
  65ef194361d40af3acccb4fc90caf25c168650bee96c326146e5e5f6dcbd914aa6f4eab20a7d5c40\
  e831430ba655854b732964536138f924157ca0c0d456454043c79d5013b6f769632836602e9a5e6b\
  b16a4af5d4c0b9e60b", "104eff4201c7d4836bcac04a8168c59d4a5c1bdfd0512cf67d22d18260\
  efecfb1ec5c810781f45425920d5c5cbc1112a5699e2533a40987471779a16455c22962f78ffcdbd\
  a6405c22bca5b1d65ddfa91166cae2c6b0b8546d8e29bbe019c2c27c80b3be5c086d88f447c5e2b0\
  2db77a99e9c81ebb1aa274b74285449d250ee17"]
```

Finally, the u-parameter is:

```
> u := params[3];
"507c72a1c4ab6a9368afc217aff88f8dbbcb8318ba3add8482c10fbb61e890b52bc5de159a4722b9b\
  0f3d8ea3ed6006da724d90f72363a78a399cbb618c097e2b6ebe510983a94e2c3f7ef31f32a4c8c2\
  58e3b67aae3a85a577d6d75517cfc40669bba065c299ae4c1f3b00acbbe406d4d47c389bfb250340\
  95be619b635464c873f3646ff43dcbcea11734080e471c73ddc03ad88cb3263927b477b1df3bddc5\
  d31a1877897ba816c9e96b4399e8cc5e93af1e26dcbea94e2d69a6aa32f63bfede9073fa894c7d17\
  cff5a56430c7c86f10dd29f1dd796aeb65f441ad73e3b1468d123e464eabc8f27aa8364b217a4f97\
  db5b512097e05f60343b570f78f86f2"
```

We may check, for example, that u has Jacobi symbol 1 but is not a quadratic residue modulo mpk. This is equivalent to u having Legendre symbol -1 modulo both prime factors of the modulus:

```
> numtheory:-legendre~(stringposint(u),stringposint~(msk));
                                  [-1, -1]
```

Proceeding with the implementation of the Cocks IBE scheme, it is now the turn of the key derivation algorithm. This algorithm makes use of the hash function that maps identity strings to elements of $\mathcal{QN}_n^1$. We construct this hash function based on the already used mask generating function `MGF1` which, in turn, was based on SHA-256. The hash function we will use is then the following, where the inputs are

the modulus n (or, in other words, the public master key) and the identity string given as an ordinary 8-bit ASCII string:

```
> CocksIBEH := proc(n::posint, id::string)
  local length, hid, R, jac;
  length := intlog[256](n);
  hid := messagetohex(id, text);
  R := OS2IP(MGF1(hid, length));
  jac := -1;
  while jac = -1 do
    R := R+1;
    jac := numtheory:-jacobi(R, n)
  end do;
  R
  end proc:
```

The function that implements the key derivation algorithm is `CocksIBEDer` below. The required input parameters are the master private key given as a list, the u-parameter and the identity string; there is also an optional keyword parameter for the output format which can be either `hex` (the default) or `decimal`. The output is the private key of the user with identity `id`.

```
> CocksIBEDer := proc(msk::list, u::{posint,string},id::string, {format::name:=hex})
  local p, q, n, R, v, r;
  p := stringposint(msk[1]);
  q := stringposint(msk[2]);
  n := p*q;
  R := CocksIBEH(n, id);
  v := stringposint(u);
  if numtheory:-legendre(R, p) = -1 then
    R := v*R mod n
  end if;
  r := chrem([numtheory:-msqrt(R, p), numtheory:-msqrt(R, q)], [p, q]);
  if format = decimal then
    r
  elif format = hex then
    StringTools:-LowerCase~(convert~(r, hex))
  else
    error "Unrecognized key format"
  end if
  end proc:
```

Example 10.3 Let us give an example of how the PKG would use the key derivation algorithm to compute the private key of a user with a given identity. In our example the identity of the user will be given by the (fictitious) email address "gomez.pardo@nonexisting.com" and the master key and the u-parameter are those computed in Example 10.2.

```
> usk := CocksIBEDer(msk, u, "gomez.pardo@nonexisting.com");
"597699e3c2140a4bfac17a066f2d92addfb9b686ceddf9674b3160253a35de5b4a96973f276e06f9d\
  bbb71e242177559c6ebeaee149afa4a710ec53e61b3d67c7579c79404c7b882faf5027ecc2bf01cd\
  cc3f36c0f803fb47deaf67c35686649ec36f80f47bd0212df39678d46ea01f1c9575506b3c64b3f2\
  b28bc2bd41fdb5888b4a29eae5050419f14b94c9cd4d0dcf3fb4c92907767ec14c388c32b5253fa1\
  0a73021348b7f089b9517f409c0c5831cd0da99c8697c211279f6e71b6a2e5ff464edd1b6efd6e91\
  921bda57e6d1f1bb2213293cc488cae1c389e6c2b616688b6078e4ef995e856dcf67ad2ca8a9b846\
  774e303ca6e38cd530e05470fecd1ce"
```

Exercise 10.5 Check which of the conditions $r^2 \equiv R \pmod{n}$ or $r^2 \equiv uR \pmod{n}$ holds, when $r =$ `usk` is the user private key of Example 10.3, $n =$ `mpk` is the master public key of Example 10.2, u is the parameter obtained in this same example, and $R =$ `CocksIBEH(n,"gomez.pardo@nonexisting.com")`.

We next give the Cocks IBE encryption function, `CocksIBEEncrypt`. The input parameters are the master public key, the u-parameter, the identity string, the one-bit message given as an element of {0, 1} and, finally, `t0` and `t1` for two externally generated random values less than the modulus n. For demonstration purposes, we have also included the possibility of letting Mersenne Twister generate these values, which will happen if no arguments are passed to the function for the last two parameters. In this case, the Mersenne Twister generator should be initialized by means of `RandomTools:-MersenneTwister:-SetState()` prior to the first use of the encryption function but, as mentioned on other occasions, this method is not secure anyway. The output of `CocksIBEEncrypt` is the ciphertext in the form of a list `[[d0, c0], [d1, c1]]`, corresponding to the format explained in Definition 10.6, where the component numbers are given as decimal integers.

```
> CocksIBEEncrypt := proc(mpk::{posint, string}, u::{posint, string}, id::string,
                          message::(integer[0 .. 1]), t0::posint, t1::posint)
  uses RandomTools:-MersenneTwister;
  local n, v, R, b, x0, x1, d0, c0, d1, c1;
  n := stringposint(mpk);
  v := stringposint(u);
  R := CocksIBEH(n, id);
  b := (-1)^message;
  if _params['t0'] = NULL then
    x0 := GenerateInteger(':-range' = 1 .. n-1)
  else
    x0 := t0
  end if;
  if _params['t1'] = NULL then
    x1 := GenerateInteger(':-range' = 1 .. n-1)
  else
    x1 := t1
  end if;
  d0 := (x0+R*(x0^(-1) mod n)) mod n;
  c0 := b*numtheory:-jacobi(x0, n);
  d1 := (x1+v*R*(x1^(-1) mod n)) mod n;
  c1 := b*numtheory:-jacobi(x1, n);
  [[d0, c0], [d1, c1]]
  end proc:
```

It only remains to give the decryption function to complete this implementation of Cocks IBE. This is the function `CocksIBEDecrypt` below, whose input parameters—all required—are the master public key, the user private key `usk` corresponding to the user with identity string `id`, the identity string `id` itself, and the ciphertext, given as a list in the format output by the preceding function `CocksIBEEncrypt`. The output is one bit, given as an element of {0, 1}.

```
> CocksIBEDecrypt := proc(mpk::{posint, string}, usk::{posint, string},
                          id::string, ciphertext::list)
  local n, r, R, i, g, b;
  n := stringposint(mpk);
  r := stringposint(usk);
  R := CocksIBEH(n, id);
  if (r^2 mod n) = (R mod n) then
    i := 0
```

```
else
  i := 1
end if;
g := (ciphertext[i+1][1]+2*r) mod n;
b := ciphertext[i+1][2]*numtheory:-jacobi(g, n);
if b = 1 then
  0
else
  1
end if
end proc:
```

Exercise 10.6 Modify the functions `CocksIBEEncrypt` and `CocksIBE Decrypt` above so that they are able to encrypt and decrypt messages of arbitrary length instead of just one-bit messages.

Example 10.4 In this example, we use the parameters and the user private key constructed in Examples 10.2 and 10.3 to encrypt and decrypt a one-bit message. Let us encrypt, for example, the '1' bit:

```
> cr1 := CocksIBEEncrypt(mpk, u, "gomez.pardo@nonexisting.com", 1);
 [[193065871685672448692912295877351962302556989347994560217284024722222477352862 9\
 5522194587916240375662481874263128005728980391303715299658071722400717856919208 01\
 6183739626146890227587406719040339766302069092438068609824757719026217125244473 88\
 2895990396889070906127438503593412957079540970517864931122087102531460047163593 62\
 8384527952645327101762120613703547303293773538770223552961388473870606442991097 82\
 9481656211040957600318561954654197240647387123172237595603705987652504073532882 07\
 6196713573042792704980165893762850575230108866226912621882358260855769131078008 45\
 7180021562153351139570120140250714669891057970174114, -1],
  [215354100371821072582761698124667964548368513560593096960693422517100570974458 5\
 3017179582914377793027129944354476991210577972009287092168777777084029494087827 38\
 6860609327556793813707998729246321117684475204537384483602256360360131903264715 99\
 9927096876836012610539062889552469410839445659052526366887109273498303569276741 19\
 6474845318951917197902598571089036520814922409848868834593298168316953333207085 71\
 6241911473172963714998231411525042267203255084549647957489049980108831717472293 47\
 7725984935456797870906667028735328559934179538216992643970993393173409390063710 46\
 7669538969460060226198624380406134182167886955457447, 1]]
```

Now, we decrypt the ciphertext we have just generated:

```
> CocksIBEDecrypt(mpk, usk,"gomez.pardo@nonexisting.com", cr1);
                                    1
```

We see that the plaintext is correctly recovered.

Exercise 10.7 Write a Maple function that tests the behavior of the Cocks IBE encryption and decryption functions. The function should pseudo-randomly generate a specified number of one-bit messages, encrypt each one of them, decrypt the corresponding ciphertext, and compare it with the original message. If some encryption/decryption operation fails the function should print an informative message, otherwise the function should print a message stating that the test was successfully completed.

10.4 The Boneh–Franklin IBE Scheme

Next we present the first fully functional IBE scheme, which was introduced by Boneh and Franklin in 2001 and is explained in detail in [35]. As we will see, this scheme is pairing-based, so we start by introducing pairings defined on groups.

10.4.1 Pairings

We are going to define a *pairing*, also called a (*nondegenerate* or *admissible*) *bilinear map*, or a *bilinear pairing*. This definition involves two groups G_1, G_2, the first of which will be denoted additively while the second will be denoted multiplicatively. The identity element of G_1 will be denoted by $\mathcal{O}$ and that of G_2 by 1. The reason why these differing notations are used for the groups is historical and is due to the fact that, in the only cryptographically useful pairings known so far, G_1 is a subgroup of an elliptic curve (traditionally written additively),[5] while G_2 is a subgroup of the multiplicative group of a finite field. In addition, we will assume that G_1 and G_2 are of order p, where p is (a large) prime. In particular, G_1 and G_2 are cyclic groups and we will assume that P is a generator of G_1, so that we may write $G_1 = \langle P \rangle$.

Definition 10.7 A *pairing* on (G_1, G_2) is a map

$$\hat{e} : G_1 \times G_1 \to G_2$$

that satisfies the following conditions:

1. (Bilinearity) For all $R, S, T \in G_1$,

$$\hat{e}(R + S, T) = \hat{e}(R, T)\hat{e}(S, T) \text{ and } \hat{e}(R, S + T) = \hat{e}(R, S)\hat{e}(R, T).$$

2. (Non-degeneracy) $\hat{e}(P, P) \neq 1$.
3. (Computability) There is an efficient algorithm to compute $\hat{e}$.

Remarks 10.2 Some straightforward consequences of the definition of pairing are, for all $R, S \in G_1$:

1. $\hat{e}(R, \mathcal{O}) = \hat{e}(\mathcal{O}, R) = 1$.
2. $\hat{e}(R, -S) = \hat{e}(-R, S) = \hat{e}(R, S)^{-1}$.
3. $\hat{e}(aR, bS) = \hat{e}(R, S)^{ab}$ for all $a, b \in \mathbb{Z}$.
4. (Symmetry) $\hat{e}(R, S) = \hat{e}(S, R)$. This follows from the fact that $R = aP$ and $S = bP$, together with the previous item.

[5] Capital letters—often starting with the letter P—are commonly used to denote the elements of G_1, which comes from the fact that, when G_1 is an elliptic curve group, its elements are points of the curve. Elliptic curve groups are defined in Chap. 11.

5. $\hat{e}(R, S) = 1$ for all $S \in G_1$ implies that $R = \mathcal{O}$. Note that this just a reformulation of the non-degeneracy property.

The existence of a pairing on (G_1, G_2) establishes a strong connection between properties of G_1 related to discrete logarithms and the corresponding properties of G_2. For example, we have the following reduction which was proved by Menezes et al. in [139]:

Proposition 10.2 (MOV reduction)
If there is a pairing on (G_1, G_2), then the DLP on G_1 is no harder than in G_2.

Proof Let $R, S \in G_1$. We want to find $a \in \mathbb{Z}^+$ such that $S = aR$. Setting $g = \hat{e}(R, R)$ and $h = \hat{e}(S, R)$ we have, because of the bilinearity of $\hat{e}$, that $h = \hat{e}(S, R) = \hat{e}(aR, R) = \hat{e}(R, R)^a = g^a$. Thus we have that $\log_R S = \log_g h$ and we can compute the former logarithm by computing the latter. □

Observe that, as a consequence of the MOV reduction, in order for the discrete logarithm problem to be hard in G_1, it must be also hard in G_2 and this must always be taken into account when selecting the security parameter for a DLP-based scheme on G_1.

Another important consequence of the existence of a pairing on (G_1, G_2) is the following:

Proposition 10.3 *If there is a pairing on (G_1, G_2), then the decisional Diffie–Hellman problem on G_1 can be efficiently solved.*

Proof Recall that the DDH problem on G_1 is to distinguish between the distributions (aP, bP, abP) and (aP, bP, cP) when a, b, c are randomly chosen in $\mathbb{Z}_p$. Then, on input (P, A, B, C), where $G_1 = \langle P \rangle$ and $A, B, C \in G_1$, the distinguisher computes $g_1 = \hat{e}(P, C)$ and $g_2 = \hat{e}(A, B)$ and it guesses that (A, B, C) is of the form (aP, bP, abP) if and only if $g_1 = g_2$. To see that the distinguisher succeeds except with negligible probability, let $A = aP, B = bP, c = cP$, where $a, b, c \in \mathbb{Z}_p$ and either $c = ab \mod p$ or c is random. Then $g_1 = \hat{e}(P, cP) = \hat{e}(P, P)^c$ and $g_2 = \hat{e}(aP, bP) = \hat{e}(P, P)^{ab}$. Since P is a generator of G_1, we have that $cP = abP$ if and only if $c = ab \mod p$ and, since $\hat{e}(P, P)$ is a generator of G_2, the latter condition is equivalent to $g_1 = g_2$. Hence $g_1 = g_2$ if and only if the triple is of the form (aP, bP, abP). In this case the distinguisher succeeds and, if c is random, it also succeeds except when $c = ab \mod p$, which only occurs with negligible probability. Thus the problem can be efficiently solved with two pairing evaluations and one comparison. □

Remark 10.3 The result of the preceding proposition does not seem to extend to the computational Diffie–Hellman problem (recall that the CDH problem in G_1 is to compute abP given aP, bP, for randomly chosen $a, b \in \mathbb{Z}_p$). Joux and Nguyen have given examples of pairings on (G_1, G_2) where the CDH problem in G_1 is believed to be hard. In [139] it is shown that the MOV reduction may be applied to a pairing defined on a special class of elliptic curves, giving a successful cryptanalytic attack

(the MOV attack) against DLP-based schemes (such as, for example, Elgamal) on this class of curves (see Sect. 11.3.3 for more details).

In order to define a pairing-based IBE scheme, we need to associate a hard problem with a pairing on (G_1, G_2). For this purpose, Boneh and Franklin introduced in [35] the *bilinear Diffie–Hellman problem* (BDH problem) which we now define:

Definition 10.8 Let $\hat{e}$ be a pairing on (G_1, G_2), where G_1 and G_2 are groups of prime order p and P is a generator of G_1. The *BDH problem* in $(G_1, G_2, \hat{e})$ is the following: Given (P, aP, bP, cP) with $a, b, c \in \mathbb{Z}_p^*$, compute $\hat{e}(P, P)^{abc} \in G_2$. An algorithm $\mathcal{A}$ *has advantage* ε *in solving BDH* if

$$\Pr(\mathcal{A}(P, aP, bP, cP) = \hat{e}(P, P)^{abc}) \geq \varepsilon.$$

If we consider a BDH generator $\mathcal{G}$, i.e., a PPT algorithm that, on input a security parameter 1^k, outputs a set of BDH parameters (i.e., a description of a pairing between two groups G_1, G_2, of prime order p, where the security parameter k is used to determine the size of p), then we say that *the BDH problem is hard relative to* $\mathcal{G}$ (or that $\mathcal{G}$ *satisfies the BDH assumption*) if the advantage of any PPT algorithm $\mathcal{A}$ in solving the BDH for $\mathcal{G}$ is negligible as a function of k.

The relevant question now is: how hard is the BDH problem? The natural way to look at this question is to consider the relationship between BDH and other problems that are presumed hard. It is obvious that the BDH problem is no harder than the DLP in G_1 and the following result is also straightforward:

Proposition 10.4 *The BDH problem in* $(G_1, G_2, \hat{e})$ *is no harder than the CDH problem in* G_1.

Proof Given (P, aP, bP, cP) we may use the CDH algorithm to compute abP and then we may use the pairing to obtain $\hat{e}(abP, cP) = \hat{e}(P, P)^{abc}$. □

Exercise 10.8 Show that the BDH problem in $(G_1, G_2, \hat{e})$ is no harder than the CDH problem in G_2.

The converse of these results is an open problem, i.e., we do not know whether an algorithm for solving the BDH problem is sufficient to solve the CDH problem in either G_1 or G_2. Apart from the Boneh–Franklin scheme that we are going to describe, the BDH assumption has been used in other cryptographic schemes such as one of Joux for tripartite Diffie–Hellman key agreement.

10.4.2 The Boneh–Franklin Scheme

We next define the Boneh–Franklin IBE scheme, starting from the so-called basic version `BasicIdent` which will subsequently be extended to the more secure `FullIdent` version by means of a generic transformation process.

Definition 10.9 Boneh–Franklin `BasicIdent` scheme.
The Boneh–Franklin `BasicIdent` scheme is the scheme (**Setup, Der, Enc, Dec**), where the algorithms are defined below. We let 1^k be a security parameter which is the input of the algorithm $\mathcal{G}$ that generates the BDH parameters (a description of the groups and the pairing involved).

- **Setup**: The algorithm runs in several steps, as follows:

1. Run $\mathcal{G}$ on input 1^k to generate a prime p, two groups G_1, G_2 of order p, and a pairing $\hat{e}$ on (G_1, G_2). Choose a random generator P of G_1 (i.e., a random element $P \leftarrow G_1^*$, where $G_1^* = G_1 - \{\mathcal{O}\}$).
2. Pick a random $s \leftarrow \mathbb{Z}_p^*$ and set $P_{pub} := sP$.
3. Choose cryptographic hash functions $H_1 : \{0,1\}^* \rightarrow G_1^*$ and $H_2 : G_2 \rightarrow \{0,1\}^n$, for some $n > 0$.

The message space is $\{0,1\}^n$ and the ciphertext space is $G_1^* \times \{0,1\}^n$. The public system parameters are $(G_1, G_2, \hat{e}, n, P, P_{pub}, H_1, H_2)$, of which P_{pub} may be regarded as the master public key. The master secret key is $s \in \mathbb{Z}_p^*$.

- **Der**: Given an identity $id \in \{0,1\}^*$, compute:

1. $Q_{id} := H_1(id) \in G_1^*$.
2. Output the private key $D_{id} := sQ_{id}$.

- **Enc**: On input a message $m \in \{0,1\}^n$ and an identity id, proceed as follows:

1. Compute $Q_{id} := H_1(id) \in G_1^*$.
2. Choose a random $r \leftarrow \mathbb{Z}_p^*$.
3. Output the ciphertext $c := (rP, m \oplus H_2(\hat{e}(Q_{id}, P_{pub})^r))$.

- **Dec**: On input $c = (U, x) \in G_1^* \times \{0,1\}^n$ and a private key $D_{id} \in G_1^*$, output

$$x \oplus H_2(\hat{e}(D_{id}, U)).$$

The correctness of the scheme is given by:

Proposition 10.5 *If in the Boneh–Franklin IBE scheme the ciphertext $c = (U, x)$ is obtained by encrypting the plaintext $m \in \{0,1\}^n$ under the identity id, the result of decrypting c with the private key D_{id} is precisely m.*

Proof At encryption, the message m is Xor-ed with $H_2(\hat{e}(Q_{id}, P_{pub})^r)$ to obtain x and, during decryption, x is Xor-ed with $H_2(\hat{e}(D_{id}, U))$ to recover m. Thus it suffices to show that the two elements of G_2 to which the hash function H_2 is applied are the same, and this is indeed the case because, by the bilinearity of $\hat{e}$:

$$\hat{e}(D_{id}, U) = \hat{e}(sQ_{id}, rP) = \hat{e}(Q_{id}, P)^{sr} = \hat{e}(Q_{id}, P_{pub})^r.$$

□

The security of the basic Boneh–Franklin scheme is given by the following result for whose proof we refer to [35, Theorem 4.1]:

Theorem 10.2 *Suppose that the hash functions H_1 and H_2 are random oracles and the BDH problem is hard in the groups generated by $\mathcal{G}$. Then* `BasicIdent` *is IND-ID-CPA secure.*

The theorem shows that `BasicIdent` is secure against chosen plaintext attacks but the scheme is not secure against chosen ciphertext attacks. However, the already mentioned technique due to Fujisaki–Okamoto transforms an IND-ID-CPA secure scheme like this one into an IND-ID-CCA secure scheme and we will apply it to `BasicIdent`. The general Fujisaki–Okamoto transformation may be described as follows. Let $\mathcal{E}$ be a probabilistic public-key encryption scheme with encryption algorithm **Enc**, and denote by $\mathbf{Enc}(pk, r, m)$ the result of encrypting the message m using the public key pk and the random value r. Then Fujisaki and Okamoto define a new encryption scheme by using a random value σ and two cryptographic hash functions H_3, H_4 and setting the encryption of m to be:

$$\mathbf{Enc}^{fo}(pk, m) := (\mathbf{Enc}(pk, H_3(\sigma, m), \sigma), H_4(\sigma) \oplus m).$$

Of course, decryption is modified accordingly; we leave its description as an exercise and we will give the details in the concrete case when the transformation is applied to `BasicIdent`.

Fujisaki and Okamoto showed that if $\mathcal{E}$ is an OW-CPA secure encryption scheme, then the result of applying this transformation to $\mathcal{E}$ is IND-CCA secure in the random oracle model (the hash functions H_3 and H_4 are modeled as random oracles in the proof). Since IND-CPA implies OW-CPA, we see that from an IND-CPA secure scheme an IND-CCA secure one is obtained. In fact, Boneh and Franklin adapt this reduction to the identity-based case and show that an IND-ID-CCA secure scheme `FullIdent` can be obtained from `BasicIdent` as follows:

Definition 10.10 Boneh–Franklin `FullIdent` scheme.
The Boneh–Franklin `FullIdent` scheme is the scheme (**Setup, Der, Enc, Dec**) defined as follows.

- **Setup**: The same as in `BasicIdent` except that, in addition, two hash functions $H_3 : \{0,1\}^n \times \{0,1\}^n \to \mathbb{Z}_p^*$ and $H_4 : \{0,1\}^n \to \{0,1\}^n$ are chosen.
- **Der**: The same as in `BasicIdent`.
- **Enc**: On input a message $m \in \{0,1\}^n$ and an identity id, proceed as follows:

1. Compute $Q_{id} := H_1(id) \in G_1^*$.
2. Choose a random $\sigma \leftarrow \{0,1\}^n$.
3. Set $r := H_3(\sigma, m)$.
4. Output the ciphertext:

$$c := (rP, \sigma \oplus H_2(\hat{e}(Q_{id}, P_{pub})^r), m \oplus H_4(\sigma)).$$

- **Dec**: On input $c = (U, x, y) \in G_1^* \times \{0,1\}^n \times \{0,1\}^n$ and a private key $D_{id} \in G_1^*$, proceed as follows:

1. If $U \notin G_1^*$ then reject the ciphertext.
2. Compute $\sigma := x \oplus H_2(\hat{e}(D_{id}, U))$.
3. Compute $m := y \oplus H_4(\sigma)$.
4. Compute $r := H_3(\sigma, m)$ and check whether $U = rP$. If not, reject the ciphertext.
5. Output m.

Remark 10.4 Observe that, in accordance with the description of the Fujisaki–Okamoto transformation given above, the encryption algorithm of `FullIdent` basically uses the encryption algorithm of `BasicIdent` to encrypt the random value σ which is then used to mask the message m. Correctness of the scheme follows easily from the correctness of `BasicIdent` and from the fact that the same value of r is used both at encryption and at decryption.

The security result is then the following (cf. [35, Theorem 4.4] and also [83], where a bug in the original reduction was fixed). The proof combines the Fujisaki–Okamoto result with a translation of the reduction to the identity-based setting.

Theorem 10.3 *Suppose that the hash functions H_1, H_2, H_3 and H_4 are random oracles and the BDH problem is hard in the groups generated by $\mathcal{G}$. Then* `FullIdent` *is IND-ID-CCA secure.*

Our description of the Boneh–Franklin IBE scheme has been rather abstract because we have simply assumed the existence of a pairing relative to which the BDH assumption must hold for the scheme to be secure. We have not given any details about the pairings that can possibly be used to build the scheme because the only pairings known so far that appear to satisfy the cryptographic requirements are based on elliptic curves. We will treat elliptic curves in the next chapter and provide sufficient background to allow the interested reader to initiate study of the pairings which are suitable for IBE, namely the Weil pairing and the Tate pairing.

10.5 Final Remarks on Identity-Based Cryptography

In this chapter we have attempted to give a brief introduction to the most salient features of identity-based cryptography and, in particular, of identity-based encryption. This field is relatively new and is undergoing much development. An interesting research direction is that of *hierarchical identity-based encryption* in which ordered identity tuples are considered, giving rise to a hierarchy of PKGs that allows key delegation by an entity to the lower level entities. Another interesting recent development is the introduction of IBE schemes that go beyond the ones we have seen regarding security, since they have security reductions without random oracles. There are many more interesting subjects in this expanding field and we refer the interested reader to [153, 107] for advanced discussions of these aspects and also to [136] for an elementary introduction to IBE schemes.

Chapter 11
An Introduction to Elliptic Curve Cryptography

Elliptic curves play a very important role in central areas of mathematics and, in particular, in *arithmetic algebraic geometry* which, as suggested by its name, is a meeting point for number theory, algebra and geometry. The study of elliptic curves dates back to the nineteenth century and, to mention only a particularly relevant milestone, they played a fundamental role in Andrew Wiles' proof of Fermat's Last Theorem, completed in 1994.

The first connection between elliptic curves and cryptography was an indirect one that arose in 1984, when Hendrik Lenstra discovered his integer factoring algorithm that relies on elliptic curves (Elliptic Curve Method or ECM, for short). The running time of this algorithm is dominated by the size of the smallest factor rather than by the size of the integer being factored and ECM is not effective to factor, say, RSA moduli[1] (ECM is, however, a useful ingredient for other factoring algorithms of cryptographic interest, such as NFS). The discovery of ECM stimulated researchers to investigate other possible applications of elliptic curves to cryptography and, as a consequence, a strong connection between the two fields was established just one year later. This happened in 1985, when Koblitz [117] and Miller [142] independently suggested the use of elliptic curve groups to implement cryptographic schemes that rely on the difficulty of the DLP on these groups.

As we have seen in previous chapters, many cryptographic schemes, including those that are DLP-based, live naturally in finite abelian groups. As we shall soon see, elliptic curves provide an abundant source for these groups and hence a specific setting where these schemes may be implemented giving rise to elliptic curve cryptography (ECC). The cryptographic theory developed in the preceding chapters applies generically to elliptic curve-based schemes, for which the results obtained are valid as far as the relevant assumptions—such as, for example, hardness of the DLP or the DDH problem—hold. From this point of view it would not be strictly necessary to go into the details of elliptic curve cryptography but, on the other hand, there are also reasons to include some specific explanations as we are going to do in this chapter.

[1] The largest factor it has found as of this writing is a 73-digit one discovered in March, 2010.

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_11, © Springer-Verlag Berlin Heidelberg 2013

Perhaps the most important of these reasons is to try to provide an appreciation for the particular aspects of elliptic curve cryptography that give it an advantage over other kinds of cryptography in many specific situations. Another important reason is that some cryptographic schemes such as the pairing-based Boneh–Franklin IBE scheme studied in Chap. 10—and, similarly, other pairing-based schemes—can only, as far as we know, be implemented in the elliptic curve setting due to the fact that it is only in this context where suitable pairings have been found. Finally, the specific study of these groups is also necessary, if one wants to implement elliptic curve-based schemes, in order to be able to select appropriate parameters—for example, the elliptic curves to be used—and also to develop the specific algorithms involved.

The main reason that led Koblitz and Miller to suggest elliptic curve groups for cryptographic use was that it seemed unlikely that the index calculus method could be adapted to solve the DLP on an elliptic curve group. As we have seen, index calculus is the most efficient algorithm known to solve hard instances of the DLP and it works in the multiplicative groups of finite fields by exploiting some of their specific properties, in particular, the existence of sufficient 'smooth' elements that factor as products of 'small' elements and which, depending on the context, can be integers that are a product of small primes (the 'factor base primes') or polynomials whose irreducible factors all have low degree. In contrast with this situation, in elliptic curve groups it seems that there is no adequate notion of 'smallness' that provides sufficient elements to be used as a 'factor base', leaving generic algorithms as the only option to solve the DLP on some of these groups. Generic DLP algorithms are, as we have seen, of fully exponential complexity—in contrast with the subexponential complexity of index calculus—and this gives elliptic curve groups the important practical advantage of requiring substantially smaller parameters, in comparison with other settings, for the same cryptographic strength. For example, it has been estimated that a 4096-bit RSA modulus offers the same level of security as 313 bits in an elliptic curve scheme.

Before going into the specifics of elliptic curves, a few remarks about the scope of this chapter and the reasons why this subject was not integrated within our earlier treatment of other groups are perhaps in order. The main reason for this separate treatment is that this material is more advanced, from a mathematical point of view, than the rest of the subjects treated in the book. For this reason, this chapter aims only to give a basic understanding of elliptic curves and of the way they are used in cryptography. In doing so we will skip the more advanced aspects and we will try to keep the presentation as simple as possible. Even taking this into account, it is possible that there are some minor details in our presentation that are not fully covered by the mathematical prerequisites stated in the Introduction or the contents of the previous chapters but, in that case, the reader should not worry much about it and go on anyway, as these details will hopefully not hinder understanding of the cryptographic applications. We will give more specific references below but, for now, we just mention that two excellent advanced books on the subject—which also have some coverage of the cryptographic applications—are those by Silverman [181] and Washington [197]. For more detailed treatments of the cryptographic aspects of elliptic curves, we refer to [97], a practice-oriented reference intended as a guide for

security professionals and developers, and also to the more mathematically advanced [26, 53, 81].

11.1 Elliptic Curves and Their Group Structure

In this section we analyze the basic properties of elliptic curves and their group structure.

11.1.1 Definition of Elliptic Curve

We start with a simple definition of elliptic curves that will be sufficient for our purposes; for more general definitions see, for example [181]. For simplicity we will also assume that our curves are defined over a field of characteristic $\neq 2, 3$, although the definition can easily be extended to cover these cases.

Definition 11.1 An *elliptic curve* E over a field $\mathbb{F}$ of characteristic $\neq 2, 3$ is the curve associated with an equation of the form:

$$y^2 = x^3 + ax + b,$$

where $a, b \in \mathbb{F}$ and $4a^3 + 27b^2 \neq 0$. This equation is called the *Weierstrass equation* of the curve. If $\mathbb{K}$ is an extension field of $\mathbb{F}$ (i.e., a field containing $\mathbb{F}$ which may be equal to $\mathbb{F}$ itself) then the set of $\mathbb{K}$-rational points of E is:

$$E(\mathbb{K}) = \{\mathcal{O}\} \cup \{(x, y) \in \mathbb{K}^2 | y^2 = x^3 + ax + b\},$$

where $\mathcal{O}$ is a special point called the *point at infinity* and the remaining points are those in the affine plane $\mathbb{K}^2$ that satisfy the curve equation.

Remark 11.1 Note that the term $\mathbb{K}$-rational used in the previous definition does not refer to the rational field $\mathbb{Q}$, as $\mathbb{K}$ can be any field of characteristic $\neq 2, 3$. Elliptic curves over $\mathbb{Q}$ are often referred to as rational elliptic curves.

Our next objective is to define the operation on $E(\mathbb{K})$ that gives a group structure to this set. But before proceeding to do so, a few comments on several details of the previous definition are in order. The first is that, while in practice we will work with the group structure defined on the set $E(\mathbb{K})$ and the same curve may be defined by different equations, we will always use the Weierstrass equation to represent it.

The most natural way to introduce the point at infinity, $\mathcal{O}$, is by viewing elliptic curves as projective curves in the projective plane, as can be seen in the references. For simplicity, we shall keep the 'affine version' as in the definition above and,

for cryptographic purposes, we may simply regard $\mathcal{O}$ as a formal symbol subject to certain computational rules. However, to obtain an idea about where these rules come from, we may look at this point geometrically as sitting infinitely far up the y-axis in such a way that all vertical lines pass through it and it is the only common point of all these lines (as suggested by the saying 'parallels meet at infinity'). Since this point is unique we may also visualize it sitting infinitely far down the y-axis but one should bear in mind that the term 'infinitely far' is meaningless for finite fields which, in addition, may not have an ordering of elements that allows us to distinguish up from down. However, $\mathcal{O}$ can be simply thought of as the unique common point to all vertical lines also in this case.

Another aspect that deserves comment is the condition $4a^3 + 27b^2 \neq 0$ in the definition above, which amounts to $\Delta \neq 0$, where $\Delta = -16(4a^3 + 27b^2) \in \mathbb{F}$ is the *discriminant* of the curve. This is equivalent to the curve being *nonsingular* or *smooth*, i.e., to the fact that there is a unique well-defined tangent at each point on the curve. To understand why this is so let us elaborate this point a little further. For example, it is intuitively clear that the real curve $y^2 = x^3$—for which $\Delta = 0$—has a "corner" at the point $(0, 0)$ and there are many different lines that intersect the curve only at this point, so that the tangent is not well defined.

The concept of *singular point* on the curve of equation $F(x, y) = 0$, where $F(x, y) \in \mathbb{F}[x, y]$ is a polynomial in the variables x, y, with coefficients in $\mathbb{F}$, can then be formalized as follows. The point $P = (x_0, y_0)$ on the curve is said to be singular if, denoting by F_x and F_y, respectively, the partial derivatives of F with respect to x and y it holds that

$$F_x(x_0, y_0) = F_y(x_0, y_0) = 0.$$

Otherwise, P is said to be *smooth*[2] and the curve is smooth if all the $\mathbb{K}$-rational points on the curve are smooth for all finite extensions $\mathbb{K}$ of $\mathbb{F}$. This is usually expressed in terms of algebraic closures. A field $\mathbb{K}$ is called *algebraically closed* if every polynomial of degree ≥ 1 with coefficients in $\mathbb{K}$ has a zero in $\mathbb{K}$ or, equivalently by Theorem 2.21, every polynomial of degree ≥ 1 in $\mathbb{K}[X]$ splits into linear factors. An algebraic closure $\mathbb{K}$ of $\mathbb{F}$ is an extension of $\mathbb{F}$ which is algebraically closed and such that each element of $\mathbb{K}$ is a zero of a nonzero polynomial with coefficients in $\mathbb{F}$ (the latter property is that $\mathbb{K}$ is an algebraic extension of $\mathbb{F}$). Each field has an algebraic closure, with the best known example being $\mathbb{C}$, which is an algebraic closure of $\mathbb{R}$. Returning to the curve, we have that $F(x, y) = 0$ is smooth precisely when all the $\mathbb{K}$-rational points on the curve are smooth, with $\mathbb{K}$ being an algebraic closure of $\mathbb{F}$. We also remark here that the (partial) derivatives of polynomials may be algebraically defined by the usual rules over an arbitrary field.

Exercise 11.1 Show that if (x_0, y_0) is a singular point on the curve $y^2 = x^3 + ax + b$, then this point lies in the x-axis.

[2] The point at infinity in an elliptic curve is regarded as smooth and the tangent to the curve at this point is the line at infinity, which intersects the curve only at this point.

Exercise 11.2 Show that the real curve of equation $y^2 = x^3 - 3x + 2$ is singular but only one of the two points where it meets the x-axis is singular.

If $P = (x_0, y_0)$ is a smooth point in the curve of equation $F(x, y) = 0$, then the tangent line to the curve at P is the line of equation:

$$F_x(x_0, y_0) \cdot (x - x_0) + F_y(x_0, y_0) \cdot (y - y_0) = 0. \tag{11.1}$$

In this case the coefficients of x and y are not both zero so the equation does indeed define a line. The tangent equation comes from the Taylor expansion of $F(x, y)$ about (x_0, y_0), which has the form:

$$F(x, y) = F(x_0, y_0) + F_x(x_0, y_0)(x - x_0) + F_y(x_0, y_0)(y - y_0) + \textit{terms of higher degree}.$$

The tangent line at P is a sort of "linear approximation" to the curve at P and is obtained by taking the linear part of the Taylor expansion which, as P is a point of the curve and hence $F(x_0, y_0) = 0$, is given by Eq. 11.1.

Exercise 11.3 Use Maple's command `plots:-implicitplot` to plot the real curves $y^2 = x^3$ and $y^2 = x^3 - 3x + 2$, which allows an easy visual detection of the singular points of both curves.

We next establish the connection between the condition $27b^2 + 4a^3 \neq 0$ appearing in the definition of elliptic curve and the smoothness of the curve. We leave the following as an exercise:

Exercise 11.4 Consider the Weierstrass equation $y^2 = x^3 + ax + b$ and let $D = -(27b^2 + 4a^3)$. Show that if $x^3 + ax + b = \prod_{i=1}^{3}(x - r_i)$ then $D = \prod_{i<j}(r_i - r_j)^2$ (D is then the *discriminant* of the polynomial $x^3 + ax + b$). Deduce that $D \neq 0$ if and only if $x^3 + ax + b$ has no multiple zeros in an algebraic closure of the base field. (Hint: Expand $\prod_{i=1}^{3}(x - r_i)$ and compare the coefficients of the resulting polynomial with those of $x^3 + ax + b$ in order to express a and b in terms of two of the roots, say r_1, r_2. Finally compute D and $\prod_{i<j}(r_i - r_j)^2$ to check that they are equal. This straightforward but tedious computation may be quickly done with Maple, taking care to use the `expand` command before doing the comparison.)

We can now show the following:

Proposition 11.1 *The curve E of equation $F(x, y) = 0$, where $F(x, y) = x^3 + ax + b - y^2$, is smooth if and only if $4a^3 + 27b^2 \neq 0$ (or, equivalently, if and only if $\Delta \neq 0$)).*

Proof Suppose first, by contradiction, that $\Delta = 0$ and let us show that E is not smooth in this case, i.e., that E has a singular point. By Exercise 11.4, there exists in

an algebraic closure of $\mathbb{F}$ a multiple zero x_0 of the cubic polynomial x^3+ax+b. Thus, in this algebraic closure this polynomial is a product of linear factors as follows:

$$x^3 + ax + b = (x - x_0)^2(x - x_1).$$

Expanding the polynomial on the right and equating the coefficients of both expressions we obtain that the coefficient of the term of degree 2 is $-2x_0 - x_1 = 0$, i.e., $x_1 = -2x_0$. The coefficient of the term of degree 1 is then $a = x_0^2 + 2x_0x_1 = x_0^2 - 4x_0^2 = -3x_0^2$, which implies that, with the preceding notations, $F_x(x_0, 0) = 0$. The point $(x_0, 0)$ also satisfies that $F(x_0, 0) = 0$ and $F_y(x_0, 0) = 0$, so that $(x_0, 0)$ is a singular point of E, completing the proof of the implication.

For the converse implication suppose now, also by contradiction, that $P = (x_0, y_0)$ is a singular point of the curve. The singularity of P is then equivalent to the following identities:

$$\begin{aligned} F(x_0, y_0) &= x_0^3 + ax_0 + b - y_0^2 = 0 \ (P \text{ is on the curve}),\\ F_x(x_0, y_0) &= 0 \Leftrightarrow 3x_0^2 + a = 0 \Leftrightarrow x_0^2 = -a/3,\\ F_y(x_0, y_0) &= 0 \Leftrightarrow -2y_0 = 0 \Leftrightarrow y_0 = 0. \end{aligned}$$

If $a = 0$ we see that $b = 0$ and hence $\Delta = 0$ too, otherwise, combining these identities we obtain:

$$x_0(x_0^2 + a) + b = x_0(2a/3) + b = 0 \Rightarrow x_0^2 = 9b^2/4a^2 = -a/3,$$

from which it follows that $27b^2 + 4a^3 = 0$, completing the proof. □

11.1.2 The Group Structure on an Elliptic Curve

We are going to define the group structure on the set of points of an elliptic curve $E(\mathbb{K})$. The group operation will be denoted additively and hence, given points $P, Q \in E(\mathbb{K})$, we want to define $P + Q \in E(\mathbb{K})$. The rough idea is to consider the unique line determined by P and Q and take the sum of these points to be equal to the reflection in the x-axis of the third point at which this line intersects the curve.

In order to see why this makes sense, consider first the case in which $P \neq Q$. Then there is a unique line L_{PQ} that contains both; in case one of them is the point at infinity $\mathcal{O}$ the line $L_{\mathcal{O}P}$ is just the vertical line through P. It is easy to see that the line L_{PQ} intersects the curve at one more point, say R. Indeed, if the line is not vertical, we eliminate the variable y in the system formed by the equations of the curve and the line, and we obtain a cubic equation that has two roots in $\mathbb{K}$. Hence the remaining root must also belong to $\mathbb{K}$ since it corresponds to the linear factor

left when we divide the cubic polynomial by the product of the two linear factors corresponding to the first two roots; then the third root is the x-coordinate of the third intersection point (if the line is vertical, the third intersection point is either $\mathcal{O}$ or one of the two remaining points obtained by eliminating the variable x and solving the resulting quadratic equation). Note that this point might be equal to either P or Q because when we say that the line intersects the curve at three points, we are counting multiplicities and the third root of the cubic might be a multiple root; we illustrate this phenomenon with some examples below. We also remark that if the field is algebraically closed then every line intersects the curve at exactly three points but this is not necessarily so if the field is not algebraically closed, as is the case with $\mathbb{Q}$, $\mathbb{R}$, or any finite field. However, the preceding argument shows that, for any field $\mathbb{K}$, if the line intersects the curve at two points of $E(\mathbb{K})$, then it also intersects it at a third point.

Example 11.1 Let us consider the curve E defined by the equation:

$$y^2 = x^3 - 4x + 1.$$

The coefficients of the polynomial defining the curve belong to the rational field $\mathbb{Q}$ and we may consider the set of points $E(\mathbb{Q})$ corresponding to the elliptic curve over this field (we may also consider $E(\mathbb{K})$ for any extension $\mathbb{K}$ of $\mathbb{Q}$ such as, for example, $\mathbb{K} = \mathbb{R}$ or $\mathbb{K} = \mathbb{C}$). Since $\Delta \neq 0$ we see that E is indeed an elliptic curve. Consider the points $P = (-1, 2)$, $Q = (1/4, 1/8)$. Then the line joining these two points is the line of equation $2y + 3x - 1 = 0$. We use Maple to compute the third intersection point as follows:

```
> solve([y^2-x^3+4*x-1 = 0, 2*y+3*x-1 = 0], [x, y]);
            [[x = 3, y = -4], [x = 1/4, y = 1/8], [x = -1, y = 2]]
```

We see that the third intersection point is $R = (3, -4)$. Note that, in contrast with the line L_{PQ}, the x-axis does not intersect the rational curve E because the polynomial $x^3 - 4x + 1 \in \mathbb{Q}[x]$ is irreducible as can be checked with Maple:

```
> irreduc(x^3-4*x+1);
                                    true
```

As seen in Examples 2.4, this means that the polynomial has no zeros in $\mathbb{Q}$ and hence the curve has no points on the x-axis despite its appearance when plotted as we do in Example 11.2 below. In the graph it looks like the curve intersects the x-axis at three points and this is indeed the case if we consider the *real curve* defined by the same equation; these intersection points do not have rational coordinates and lie in $E(\mathbb{R})$ but not in $E(\mathbb{Q})$.

In the previous discussion we assumed that $P \neq Q$ but we have also to consider the case in which $P = Q$ in order to be able to define $P + P$. Then it is still true that the elliptic curve determines a unique line passing through P, namely the tangent

line at P, L_{PP}, which may be seen as the limit of the lines L_{PQ} as Q approaches P. The intersection of the tangent with the curve has multiplicity at least 2 and, arguing as above, we deduce that it intersects the curve at another point R, which could be equal to P in case the intersection has multiplicity 3 (when this happens we say that P is an inflection point of E). If the tangent is a vertical line then we have $R = \mathcal{O}$ and if $P = \mathcal{O}$ then the tangent at $\mathcal{O}$ is the *line at infinity* which intersects the curve at the point $\mathcal{O}$ with multiplicity three. Thus we regard $\mathcal{O}$ as an inflection point of the curve and we have that $R = \mathcal{O}$ in this case.

As mentioned above, the idea to define the addition of points in the curve is to set the sum of two points P, Q, to be the reflection of R in the x-axis, where R is the third point of intersection of L_{PQ} with the curve. Note that this makes sense because an elliptic curve is symmetric about the x axis, so that if the point (x_0, y_0) lies on the curve, then the point $(x_0, -y_0)$ is also on the curve. We denote the reflection of R by $-R$ and so we define $P + Q = -R$, where R is defined as above (this construction is often called the *chord-tangent method*). The point at infinity is a special case and we regard it as its own reflection about the x-axis. Then, if $P \neq \mathcal{O}$, what is the point $P + \mathcal{O}$? The line joining P and $\mathcal{O}$ is the vertical line through P and the third point of intersection with the curve is precisely $-P$. The reflection of this point in the x-axis is P so we obtain that $P + \mathcal{O} = \mathcal{O} + P = P$. Similarly, $\mathcal{O} + \mathcal{O} = \mathcal{O}$ because the tangent at $\mathcal{O}$ is the line at infinity. Thus we see that $\mathcal{O}$ acts as an additive identity (or zero element) for this operation. We also see that $P + (-P) = -P + P = \mathcal{O}$, so that the point $-P$ is actually the opposite (or additive inverse, or negative) of P with respect to the addition we have defined. The opposite element in a group is also sometimes called the symmetric so here we have that P and $-P$ are symmetric to each other both in the geometric and the group-theoretic sense.

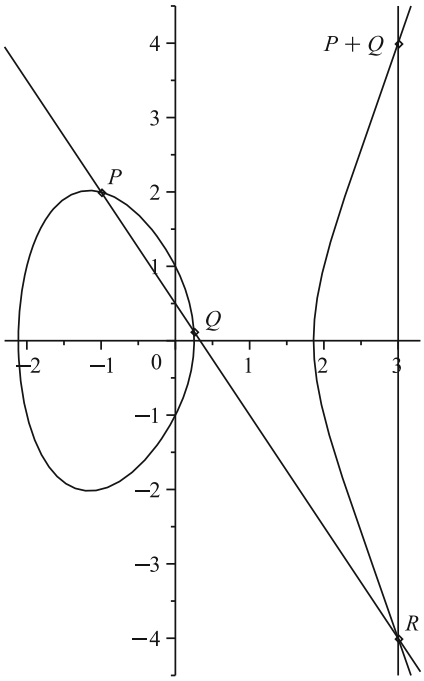
Example 11.2 Let us consider the previous curve of equation $y^2 = x^3 - 4x + 1$. Suppose we want to compute the sum of the points $P = (-1, 2)$ and $Q = (1/4, 1/8)$. As we have seen in Example 11.1, the third point of intersection of L_{PQ} with the curve is $R = (3, -4)$, so that $P + Q = -R = (3, 4)$. We may use Maple to obtain a graphic representation of this construction.

```
> with~([plots, plottools]):

> F := y^2-x^3+4*x-1:
  L := 2*y+3*x-1:
  el := implicitplot([F, L, x-3], x = -2.3 .. 3.3, y = -4.5 .. 4.5,
        resolution = 1000, crossingrefine = 20, gridrefine = 6, color = black):
  pts := pointplot([[-1, 2], [1/4, 1/8], [3, -4], [3, 4]]):
  txt := textplot([[-.8, 2.1, 'P'], [.4, .2, 'Q'], [3.2, -4, 'R'], [2.7, 4, 'P+Q']]):
```

The construction is then illustrated in the following drawing:

```
> display(el, pts, txt, labels = [" ", " "], scaling = constrained);
```



Next we are going to add P to itself. For this we have to compute the tangent to the curve at P. Using the formula for the tangent we compute the partial derivatives of the polynomial F, evaluate them at P and we obtain[3]:

```
> T := (eval(diff(F, x), [x=-1, y=2]))*(x+1)+(eval(diff(F, y), [x=-1, y=2]))*(y-2);
                                  x - 7 + 4 y
```

An alternative way of computing the tangent is to write it in the form $y = m(x + 1) + 2$ where m is the slope of the curve at the point $P = (-1, 2)$. This slope may be computed by using implicit differentiation on F with respect to x and evaluating the result at P and we use Maple to do so and to obtain the equation of the tangent (if the tangent line were vertical, which is not the case here, then the slope would not be defined and Maple would give a "division by zero error" that we could interpret as meaning that $m = \infty$):

```
> y = (eval(implicitdiff(F, y, x), [x = -1, y = 2]))*(x+1)+2;
                                y = - x/4 + 7/4
```

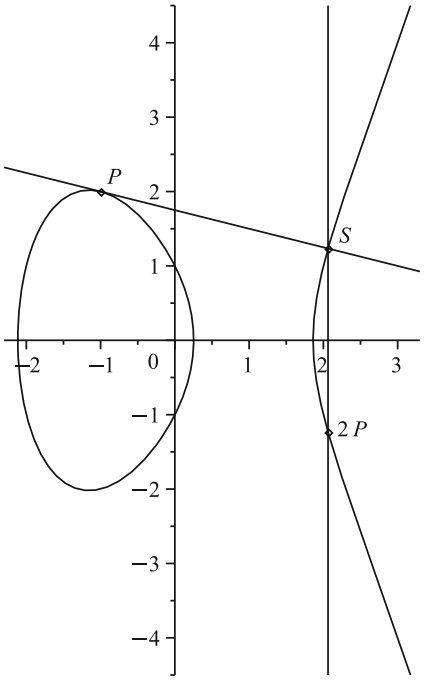
Of course, this equation defines the same line. Next we compute the third point of intersection of this line with the curve:

[3] It is also possible to write the result as an equation with the polynomial equated to 0 but this is not necessary to plot the curve because Maple knows what to do when asked to plot a polynomial.

```
> solve([F, T], [x, y]);
            [[x = 33/16, y = 79/64], [x = -1, y = 2], [x = -1, y = 2]]
```

Thus we see that the third point we were looking for is $S = (33/16, 79/64)$. Notice that since P has coordinates in $\mathbb{Q}$, we already knew that the coordinates of S would also belong to $\mathbb{Q}$ and observe also that the tangent meets the curve at P with multiplicity 2 (the coordinates of P appear twice in the result above) so the tangent intersects the curve at three points, two of which happen to be P. Next we plot the process of adding P to itself which gives as a result $2P = (33/16, -79/64)$:

```
> el2 := implicitplot([F, T, x-33/16], x = -2.3 .. 3.3, y = -4.5 .. 4.5,
         resolution = 1000, crossingrefine = 20, gridrefine = 6, color = black);
  pts2 := pointplot([[-1, 2], [33/16, 79/64], [33/16, -79/64]]);
  txt2 := textplot([[-.8, 2.1, 'P'], [2.2, 1.4, 'S'], [2.3, -1.2, '2P']]);
  display(el2, pts2, txt2, labels = [" ", " "], scaling = constrained);
```



Exercise 11.5 Prove that if an elliptic curve intersects the x-axis at three different points, then the sum of any two of these points is equal to the third one.

The previous discussion on the definition of the addition operation on $E(\mathbb{K})$ may be summarized as follows:

$$P + Q + R = \mathcal{O} \text{ if and only if } P, Q, R \text{ are collinear.}$$

where the term collinear here means that there is a line that intersects E at the three points P, Q, R counting multiplicity and including the possibility that some of these points are equal to the point at infinity.

We are now going to give the explicit formulas for the addition of points that will be used in cryptographic algorithms. These formulas are based on the *chord-tangent method* and give an expression for the coordinates of $P_1 + P_2$ in terms of those of P_1 and P_2.

Theorem 11.1 *Let E be an elliptic curve given by a Weierstrass equation*

$$y^2 = x^3 + ax + b$$

over a field of characteristic $\neq 2, 3$ and let P_1, P_2 be points in E. Then the following assertions hold:

1. *If $P_1 = \mathcal{O}$ then $P_1 + P_2 = P_2$, otherwise if $P_2 = \mathcal{O}$ then $P_1 + P_2 = P_1$.*
2. *If $P_1 \neq \mathcal{O}$ and $P_2 \neq \mathcal{O}$, write $P_i = (x_i, y_i)$ for $i = 1, 2$. The following formulas hold:*

   a. *If $x_1 = x_2$ and $y_1 = -y_2$ then $P_1 + P_2 = \mathcal{O}$.*
   b. *If $x_1 \neq x_2$ or $y_1 \neq -y_2$ then set*

$$m = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq P_2 \\ \frac{3x_1^2 + a}{2y_1} & \text{if } P_1 = P_2, \end{cases}$$

   and let

$$x_3 = m^2 - x_1 - x_2 \text{ and } y_3 = m(x_1 - x_3) - y_1.$$

   Then $P_1 + P_2 = (x_3, y_3)$.

Proof The cases 1 and 2 (a) are clear after our previous discussion. For case 2 (b) note that if $P_1 \neq P_2$, then m is the slope of the line through P_1 and P_2 and if $P_1 = P_2$ then m is the slope of the tangent line at P_1; in the latter case we may use implicit differentiation of the equation of the curve, just as we did in the computation with Maple in Example 11.2, and we find that the slope is indeed $m = \frac{3x_1^2 + a}{2y_1}$. In either case, the line $L_{P_1 P_2}$ has equation $y = mx + t$, where $t = y_1 - mx_1$. Substituting the equation for $L_{P_1 P_2}$ in the equation for E gives the identity:

$$x^3 + ax + b = (mx + t)^2,$$

which can be written in the form:

$$x^3 - m^2 x^2 + (a - 2mt)x + (b - t^2) = 0.$$

We already know that x_1 and x_2 are roots of this equation. Then the third root is x_3, namely, the x-coordinate of the sum $P_1 + P_2$, and the polynomial factors as:

$$x^3 - m^2x^2 + (a - 2mt)x + (b - t^2) = (x - x_1)(x - x_2)(x - x_3).$$

In particular, the coefficient of the term of degree 2 is the same in both polynomials and hence $m^2 = x_1 + x_2 + x_3$, from which it follows that $x_3 = m^2 - x_1 - x_2$ as stated in the theorem. To complete the proof we have to show that the y-coordinate of $P_1 + P_2$, namely y_3, also takes the value given in the statement of the theorem. This coordinate is the opposite of the y-coordinate corresponding to x_3 in the line $L_{P_1P_2}$, obtained by substituting x_3 for x in the equation of this line, so we have that $y_3 = -(m(x_3 - x_1) + y_1) = m(x_1 - x_3) - y_1$, which completes the proof. □

The important properties of the above-defined addition on $E(\mathbb{K})$ are summarized in the following result:

Theorem 11.2 *If E is an elliptic curve, then the addition of points gives $E(\mathbb{K})$ the structure of an abelian group.*

It is clear that $\mathcal{O}$ is an identity element for this operation and also that the additive inverse of $P \in E(\mathbb{K})$ is its x-axis reflection $-P$. Moreover, it is also clear that the operation is commutative, i.e., $P_1 + P_2 = P_2 + P_1$ for all $P_1, P_2 \in E(\mathbb{K})$. To show that we have a group it only remains to prove associativity, i.e., that $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$ for all $P_1, P_2, P_3 \in E(\mathbb{K})$. The elementary way to prove associativity is by direct calculation with the formulas given in Theorem 11.1. This calculation is straightforward but extremely tedious because there are many cases to consider, depending on whether or not $P_1 = P_2$ and whether or not $P_3 = P_1 + P_2$ and so on. On the other hand, this proof offers no additional insights so we do not include it here and leave it as an exercise for the sufficiently patient reader. There are other, more enlightening, proofs such as a geometrical one based on Bezout's theorem on intersections of curves [182] or an algebraic one based on the theory of divisors on curves [181].

Recall that the order of an element $a \in G$, where G is an additive group, is the smallest positive integer n such that $na = 0$ (if no such n exists then a has infinite order). It can be shown by using the *Lutz–Nagell theorem*[4] [181 (Corollary 7.2), 197 (Theorem 8.7)] that the rational curve $y^2 = x^3 - 4x + 1$ of Example 11.2 has no points of finite order other than $\mathcal{O}$.[5] However, there are rational elliptic curves with points $\neq \mathcal{O}$ of finite order and, of course, all the points of an elliptic curve over a finite field $\mathbb{F}$ have this property because in this case the group $E(\mathbb{F})$, being a subset of $\mathbb{F}^2 \cup \{\mathcal{O}\}$, is finite and hence so is the order of any point.

The smallest possible order of a point $P \neq \mathcal{O}$ is 2 and such a point has order 2 precisely when $P = -P$. Then $P = (x, y) = (x, -y)$ and hence $y = 0$ (recall

[4] This theorem says that if $P \neq \mathcal{O}$ is a point of finite order of $E(\mathbb{Q})$ then it has integral coordinates x, y and, moreover, if $y \neq 0$ then y^2 divides $4a^3 + 27b^2$.

[5] This produces a little terminological paradox: the only point of finite order in this curve is the point at infinity!

that we are assuming throughout that the characteristic of the field is $\neq 2$). Thus the points of order 2 are precisely the points whose y-coordinate is equal to 0. If $\mathbb{F}$ is algebraically closed there are always three such points (counting multiplicities) and they are $(x_1, 0), (x_2, 0), (x_3, 0)$, where x_1, x_2, x_3 are the roots of the cubic polynomial $x^3 + ax + b$ (where $y^2 = x^3 + ax + b$ is the Weierstrass equation of the curve). But for other fields the situation may be different. As we have mentioned, the curve of Example 11.2 has no points of finite order other than $\mathcal{O}$ and, in particular, that it has no points of order 2 follows easily from the already observed fact that the x-axis does not intersect the curve. The situation is different if we consider the real curve defined by the same equation, which has three points of order 2 corresponding to the three real roots of the cubic polynomial. The other possibility for a real elliptic curve is to have only one point of order 2 in case the cubic has a single real root.

The points of order 3 are the points $P \neq \mathcal{O}$ such that $3P = \mathcal{O}$, which is equivalent to $P + P = -P$ and hence happens precisely when the tangent line at P has a contact of order 3 with the curve. As we have remarked, this is the algebraic definition of inflection point and so the inflection points are just those of order 3 plus $\mathcal{O}$ (i.e., the points whose order divides 3). Again, the curve of Example 11.2 has no points of order 3, so that its only inflection point is $\mathcal{O}$. However, it can be seen by the usual calculus methods that the real curve defined by the same equation has three inflection points, two of which have order 3.

Example 11.3 Let us consider the curve $E(\mathbb{Q})$ defined by the Weierstrass equation $y^2 = x^3 + 1$. Then the point $P = (2, 3)$ is on the curve and we may compute its order by using the formulas provided by Theorem 11.1. Using the duplication formula given there we get:

$$2P = (0, 1),$$

and we may use the formula again to compute:

$$4P = 2P + 2P = 2(0, 1) = (0, -1) = -2P.$$

From $4P = -2P$ it follows that $6P = 0$. This means that $2P$ has order 3 and hence it is an inflection point. Moreover, $3P$ has order 2 (for, otherwise, $3P = \mathcal{O}$ would imply that $4P = 3P + P = P$ which is not the case) and hence the y-coordinate of $3P$ is 0 and $3P = (-1, 0)$ since -1 is the only rational root of $x^3 + 1$. Finally, $5P = -P = (2, -3)$ and so $E(\mathbb{Q})$ has the cyclic subgroup:

$$\{\mathcal{O}, (2, 3), (0, 1), (-1, 0), (0, -1), (2, -3)\},$$

in which P and $5P$ are generators. Already in the 18th century, Euler proved using the method known as infinite descent that these are the only rational points on this elliptic curve, so we have that $E(\mathbb{Q})$ is cyclic of order 6 (and hence isomorphic to $\mathbb{Z}_6$).

Exercise 11.6 We have seen an example of a rational elliptic curve such that $E(\mathbb{Q})$ has no points of order 2 and an example in which $E(\mathbb{Q})$ has just one point of order 2. Give an example in which $E(\mathbb{Q})$ has three points of order 2 and show that these three examples cover all possible cases.

11.2 Elliptic Curves Over Finite Fields

The elliptic curves of cryptographic interest are mainly elliptic curves over finite fields and we will devote special attention to the case of a prime field $\mathbb{F}_p$.

11.2.1 Some Small Examples

When the field is finite the elliptic curve defines a finite group and in order to obtain a first impression about this group we may consider a few examples over a small field. In general, to compute all the points of an elliptic curve $E(\mathbb{F}_p)$ given by the Weierstrass equation $y^2 = x^3 + ax + b$, we have to check first, for each $x \in \mathbb{F}_p$, whether $x^3 + ax + b$ is or not a quadratic residue modulo p (which can be efficiently determined by computing the Legendre symbol $\left(\frac{x^3+ax+b}{p}\right)$). As we have seen, half of the elements of $\mathbb{F}_p^* = \mathbb{F}_p - \{0\}$ are quadratic residues and each of them has just two square roots, while the quadratic non-residues do not have a square root. Thus, each time that $\left(\frac{x^3+ax+b}{p}\right) = 1$ for some $x \in \mathbb{F}_p^*$, we obtain two square roots $\pm y$ and two points $(x, \pm y) \in E(\mathbb{F}_p)$ and when $x^3 + ax + b = 0$ we obtain the point $(x, 0) \in E(\mathbb{F}_p)$ (when $\left(\frac{x^3+ax+b}{p}\right) = -1$, no point is obtained). These points, together with the point at infinity $\mathcal{O}$, form the group $E(\mathbb{F}_p)$. Note that the order $|E(\mathbb{F}_p)|$ will be even precisely when there is a point of order 2 which, as we have seen, is equivalent to the polynomial $x^3 + ax + b$ having a zero in $\mathbb{F}_p$. Thus the group has odd order if and only if this polynomial is irreducible in $\mathbb{F}_p[x]$ (which means that it has no zeros in $\mathbb{F}_p$). In the following we will refer to $|E(\mathbb{F}_p)|$ as the order of the curve E over $\mathbb{F}_p$. $|E(\mathbb{F}_p)|$ is frequently called the cardinality of the curve or, simply, the number of points of the curve, but we will use the group-theoretic term order for brevity.

The smallest prime field that satisfies our requirement of having characteristic greater than 3 is $\mathbb{F}_5 = \mathbb{Z}_5$ and this field is small enough to allow us to do the preceding computation by hand. Let us look at a couple of examples.

Examples 11.1 Consider first the elliptic curve E defined by the equation $y^2 = x^3 + 2x$ over $\mathbb{F}_5$ (note that $4a^3 + 27b^2 \bmod 5 = 2 \neq 0$ and hence the equation defines an elliptic curve). Across all values of x on $\mathbb{F}_5^*$, $x^3 + 2x$ is either 2 or 3 which are both

quadratic non-residues in $\mathbb{F}_5^*$. Thus, apart from $\mathcal{O}$, the only other point on the curve is $(0, 0)$, so that $E(\mathbb{F}_5) = \{\mathcal{O}, (0, 0)\}$, the (cyclic) group of order 2.

Consider now the curve F over $\mathbb{F}_5$ given by the equation $y^2 = x^3 + x$ (in all these examples the discriminant is nonzero, so we will not mention this fact again). For $x = 1, 4$, we obtain values of $x^3 + x$ which are quadratic non-residues but for $x = 0, 2, 3$ we obtain $x^3 + x = 0$; this is because the polynomial $x^3 + x$ splits into linear factors in $\mathbb{F}_5$ with the values of x just mentioned being its zeros. Thus $F(\mathbb{F}_5) = \{\mathcal{O}, (0, 0), (2, 0), (3, 0)\}$ is a 4-element group. Since all the nonzero elements of the group have second coordinate 0 and hence have order 2, we can see that the group is isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_2$ (the so-called "Four-Group").

Next we look at the curve G defined by the equation $y^2 = x^3 + x + 2$ over $\mathbb{F}_5$. In this case we obtain the quadratic residue $y^2 = 4$ for $x = 1$. The square roots of 4 in $\mathbb{F}_5$ are 2 and $-2 \bmod 5 = 3$ so this gives us the points $(1, 2)$, $(1, 3)$. $x = 4$ is a root of $x^3 + x + 2$ and hence gives the point $(4, 0)$. The remaining values of x give values of y^2 which are quadratic non-residues and hence they give no points on the curve. Thus $G(\mathbb{F}_5) = \{\mathcal{O}, (1, 2), (1, 3), (4, 0)\}$. In this case there is just one point of order 2, namely $(4, 0)$. The remaining points except $\mathcal{O}$ must have order $\neq 1, 2$ and, by Lagrange's theorem this order must be a divisor of 4. Thus both points have order 4, which means that $G(\mathbb{F}_5)$ is a group of order 4 (and hence isomorphic to $\mathbb{Z}_4$) of which both $(1, 2)$ and $(1, 3)$ are generators.

Let us now consider the curve H defined by $y^2 = x^3+x+4$ over $\mathbb{F}_5$. Now, the only value of $x \in \mathbb{F}_5$ such that x^3+x+4 is a quadratic non-residue (with value 2) is $x = 4$ and the remaining values give $H(\mathbb{F}_5)$=$\{\mathcal{O}, (0, 2), (0, 3), (1, 1), (1, 4), (2, 2), (2, 3), (3, 2), (3, 3)\}$, which is a group of order 9. Since the order of a point divides the order of the group, all points except $\mathcal{O}$ must have order either 3 (in which case they are inflection points) or 9 (in which case they are generators of the group). If we find a point which is not an inflection point then we know that it has order 9 and, in particular, that the group is cyclic, and generated by this point. Let us check, for example, the point $(0, 2)$. Bearing in mind that $-4 \equiv 1 \bmod 5$, the equation of the curve may be written as $y^2+4x^3+4x+1 \equiv 0 \pmod 5$ and writing $h(x, y) = y^2+4x^3+4x+1$, the tangent line to the curve at this point has the equation:

$$h_x(0, 2)(y - 2) + h_y(0, 2)x \equiv 0 \pmod 5.$$

After computing the partial derivatives and evaluating them at the point $(0, 2)$, this gives the equation $4(y-2)+4x \equiv 0 \pmod 5$ which, after some simplification, may be written as:

$$y \equiv 4x + 2 \pmod 5.$$

Next we compute the third intersection point of this line with the elliptic curve. Substituting this value of y into h we obtain:

$$(4x + 2)^2 \equiv x^3 + x + 4 \pmod 5 \Leftrightarrow x^3 + 4x^2 \equiv 0 \pmod 5.$$

The polynomial $x^3 + 4x^2$ has 0 as a double root over $\mathbb{F}_5$ (we already knew that this should happen because it was obtained from the tangent line that intersects the curve at $(0, 2)$ with multiplicity at least 2). To find the remaining root of this polynomial we factor out the term x^2 so that the remaining polynomial is $x + 4$. The only root of this polynomial in $\mathbb{F}_5$ is $x = 1$ which is then the x-coordinate of the third intersection point. To find the y-coordinate we substitute $x = 1$ in the equation of the tangent and we obtain $y = 1$, so that the intersection point is $(1, 1)$. The sum of $(0, 2)$ with itself is then the opposite of $(1, 1)$, namely:

$$2(0, 2) = (1, 4).$$

Since $(1, 4) \neq (0, -2) = (0, 3)$ we see that $(0, 2)$ does not have order 3. Hence it has order 9 and so $H(\mathbb{F}_5)$ is cyclic and $(0, 2)$ is a generator.

We have given all these details to illustrate the chord-tangent method which works over a finite field exactly the same as over $\mathbb{R}$. However, this calculation can be done more quickly if one uses the duplication formula given in Theorem 11.1.

Exercise 11.7 Show that the group of points of the elliptic curve defined by $y^2 = x^3 + 3x$ over $\mathbb{F}_5$ has 10 elements and prove, using Theorem 11.1, that $(2, 3)$ is a generator of this group.

Exercise 11.8 Show that in the elliptic curve defined by $y^2 = x^3 + x + 1$ over $\mathbb{F}_5$, the point $(2, 1)$ is an inflection point.

Exercise 11.9 Prove that the elliptic curves over $\mathbb{F}_5$ defined by the equations $y^2 = x^3 + 4x + 2$, $y^2 = x^3 + 3x + 2$, $y^2 = x^3 + 1$, $y^2 = x^3 + 2x + 1$, $y^2 = x^3 + 4x$ have groups of order 3, 5, 6, 7, 8, respectively (together with the previous examples, this shows that all integers in the interval $[2, 10]$ are orders of elliptic curves over $\mathbb{F}_5$).

11.2.2 Elliptic Curve Elementary Computations

In order to be able to work with larger examples, we are going to give a simple Maple implementation of elliptic curves over prime fields of characteristic greater than 3. An elliptic curve over $\mathbb{F}_p$ will be given by a Weierstrass equation $y^2 = x^3 + ax + b$ whose discriminant is nonzero and we will represent it by a Maple list of the form $[a, b, p]$. To initialize an elliptic curve we will use the following procedure which, on input the parameters a, b, p, given either in decimal form or as hexadecimal strings, checks that p is a (probable) prime > 3 and that the discriminant is nonzero and, if these conditions hold, outputs the list $[a, b, p]$ in decimal form.

```
> EllipticCurve := proc(a::{integer,string}, b::{integer,string}, p::{posint,string})
  local q, A, B;
  q := stringposint(p);
  if not isprime(q) or q < 5 then
    error "%1 is not a prime >3", q
  end if;
  A := stringposint(a) mod q;
  B := stringposint(b) mod q;
  if (4*A^3+27*B^2) mod q = 0 then
    error "singular curve"
  else
    [A, B, q]
  end if;
  end proc:
```

The point at infinity of an elliptic curve will be represented by 0 and the remaining points by a Maple list $[x, y]$, where x and y are the coordinates of the point. Observe that the 0 representing $\mathcal{O}$ should not be confused with the element $0 \in \mathbb{F}_p$ nor with the point $(0, 0)$ (in case this point belongs to the elliptic curve) nor with any other affine point for that matter. The next procedure takes as input either 0 (for the point at infinity) or the coordinates x, y (in decimal or hexadecimal format), and outputs the point given as either 0 or $[x, y]$, with decimal coordinates.

```
> EllipticPoint := proc(x::{integer, string}, y::{integer, string})
  if _params['y'] = NULL then
    0
  else
    stringposint~([x, y])
  end if
  end proc:
```

The next function takes as input a point and an elliptic curve and checks whether the point is on the curve, in which case it returns *true*, otherwise it returns *false*:

```
> IsEllipticPoint := proc(P, E)
  evalb(P = 0 or (P[2]^2-P[1]^3-E[1]*P[1]-E[2]) mod E[3] = 0)
  end proc:
```

The discriminant of the cubic polynomial defining an elliptic curve is computed by:

```
> discr := proc(E)
  (-4*E[1]^3-27*E[2]^2) mod E[3]
  end proc:
```

Next we give a Maple function that computes the set of points of an elliptic curve over $\mathbb{F}_p$. It runs over the field elements x and computes $x^3 + ax + b$ and its Legendre symbol modulo p, extracting the square roots when it is a quadratic residue. The input is an elliptic curve in the format output by `EllipticCurve` and the output a list of the points on the curve.

```
> EllipticPoints := proc(E::list(integer))
  local a, b, p, EP, x, z, y;
  a := E[1];
  b := E[2];
  p := E[3];
  EP := [];
  for x from 0 to p-1 do
    z := (x^3+a*x+b) mod p;
    if numtheory:-legendre(z, p) <> -1 then
```

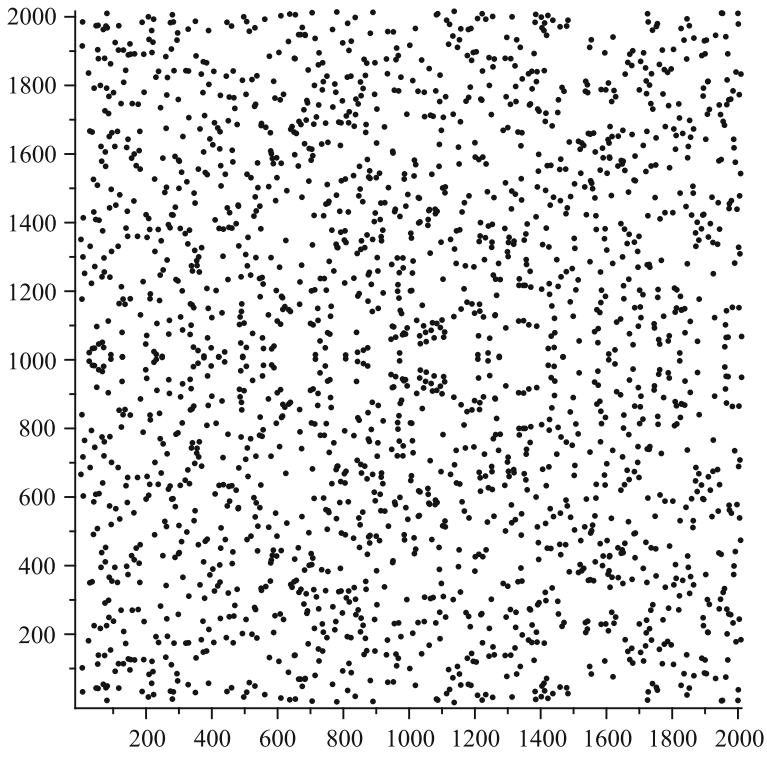
```
      y := numtheory:-msqrt(z, p);
      if y <> 0 then
        EP := [op(EP), [x, y], [x, -y mod p]]
      else
        EP := [op(EP), [x, y]]
      end if;
    end if;
  end do;
  [0, op(EP)]
end proc:
```

Example 11.4 Let us compute the list of all points on the elliptic curve over $\mathbb{F}_{2017}$ given by the equation $y^2 = x^3 + 578x + 669$. We do not print the list as it is too long but we will plot the curve to appreciate its appearance:

```
> ep :=EllipticPoints(EllipticCurve(578, 669, 2017)):
  nops(%);
                                2087
```

We see that the curve has exactly 2087 points. Next we plot it:

```
> epp := plots:-pointplot(ep[2 .. -1], symbol = solidcircle, symbolsize = 9):
> plots:-display(epp);
```



We see that the curve has symmetry about a horizontal axis but there are no points below the x-axis. The reason is that we are using the integers in the interval $[0, p-1]$ to represent the elements of $\mathbb{F}_p$ but if we use instead the "least absolute residue" then the x-axis would become the axis of symmetry of the curve. To obtain this representation in Maple we just have to replace the list `ep` by `mods~(ep,2017)`.

Exercise 11.10 Modify the procedure `EllipticPoints` so that it accepts as an additional parameter an interval $[x_0, x_1]$, where $0 \leq x_0 \leq x_1 \leq p-1$, and computes all the points in the elliptic curve whose x-coordinate belongs to this interval. Make this parameter optional and such that if no argument is passed to it then the procedure just computes the set of all points on the elliptic curve.

The method used to generate the points also gives a naive way to count their number without having to explicitly compute them. It is based on the following easy result:

Proposition 11.2 *If E is the elliptic curve over $\mathbb{F}_p$ defined by the equation $y^2 = x^3 + ax + b$, then $|E(\mathbb{F}_p)| = p + 1 + \sum_{x=0}^{p-1} \left(\frac{x^3+ax+b}{p}\right)$.*

Proof As we have seen, for a given element $x_0 \in \mathbb{F}_p$, the number of points on the curve with x-coordinate equal to x_0 is equal to 2, 0 or 1 according to whether the Legendre symbol $\left(\frac{x_0^3+ax_0+b}{p}\right)$ takes the value $+1$, -1, or 0, respectively. Thus we see that this number is precisely $1 + \left(\frac{x_0^3+ax_0+b}{p}\right)$. Adding these terms over the p elements of $\mathbb{F}_p$ and adding 1 for the point at infinity, we obtain the stated formula. □

Proposition 11.2 provides an algorithm for computing the number of points on an elliptic curve but the number of Legendre symbols to be computed is $O(p)$ and hence the running time is exponential in the size of p. As a consequence, this naive algorithm is only useful for curves over small fields but we will use it to get an idea of the distribution of orders of elliptic curves over prime fields. The Maple function below is a straightforward implementation of the algorithm:

```
> EllipticGroupOrder := proc(E::list(integer))
  local a, b, p, x;
  a := E[1];
  b := E[2];
  p := E[3];
  1 + p + add(numtheory:-legendre(x^3+a*x+b, p), x = 0 .. p-1)
  end proc:
```

For example:

```
> EllipticGroupOrder(EllipticCurve(4, 0, 5));
                                    8
```

shows that the curve $y^2 = x^3 + 4x$ over $\mathbb{F}_5$ has exactly 8 points. While this method is not suitable for curves of cryptographic size, much larger orders can be computed in a matter of seconds, for example:

```
> E := EllipticCurve(98316, 201554, 1234577):
  EllipticGroupOrder(E);
                                  1234599
```

The following Maple function generates, on input a prime $p > 3$, the list of all elliptic curves defined over $\mathbb{F}_p$.

```
> EllipticCurvesList := proc(p::posint)
  local prod, curves, E;
  if not isprime(p) or p < 5 then
    error "%1 is not a prime greater than 3", p
  end if
  prod := combinat:-cartprod([[$0 .. p-1], [$0 .. p-1]]);
  curves := [];
  while not prod[finished] do
    E := [op(prod[nextvalue]()), p];
    if discr(E) <> 0 then
      curves := [op(curves), E]
    end if;
  end do;
  curves
  end proc:
```

Example 11.5 The elliptic curves over $\mathbb{F}_5$ are the following:

```
> EllipticCurvesList(5);
[[0, 1, 5], [0, 2, 5], [0, 3, 5], [0, 4, 5], [1, 0, 5], [1, 1, 5], [1, 2, 5],
  [1, 3, 5], [1, 4, 5], [2, 0, 5], [2, 1, 5], [2, 4, 5], [3, 0, 5], [3, 2, 5],
  [3, 3, 5], [4, 0, 5], [4, 1, 5], [4, 2, 5], [4, 3, 5], [4, 4, 5]]
```

Exercise 11.11 Show that the number of elliptic curves over $\mathbb{F}_p$, for $p > 3$, is exactly $p^2 - p$. (Hint: Use the Legendre symbol to show that if $4a^3 + 27b^2 = 0 \in \mathbb{F}_p$, then there exists $c \in \mathbb{F}_p$ such that $a = -3c^2$ and $b = 2c^3$, and c is uniquely determined by a and b because $c = -3b(2a)^{-1}$ whenever $a \neq 0$. This gives p pairs $(a, b) \in \mathbb{F}_p^2$ such that the curve $y^2 = x^3 + ax + b$ is singular.)

Some of the 20 elliptic curves over $\mathbb{F}_5$ listed in Example 11.5 are isomorphic to some others, which means that the curves are essentially the same. We refer to [97, 181] for the definition of isomorphic elliptic curves and here we merely point out that the curves over $\mathbb{F}_p$ (with $p \neq 2, 3$) given by the equations $y^2 = x^3 + ax + b$ and $y^2 = x^3 + a'x + b'$ are isomorphic if and only if there exists $u \in \mathbb{F}_p^*$ such that $u^4 a' = a$ and $u^6 b' = b$. We remark that isomorphic curves have isomorphic groups of points but the converse does not hold (see Exercise 11.13 below.)

Example 11.6 We have seen in Example 11.1 that the elliptic curves over $\mathbb{F}_5$ given by the equations $y^2 = x^3 + x$ and $y^2 = x^3 + x + 2$ have groups isomorphic to

$\mathbb{Z}_2 \times \mathbb{Z}_2$ and $\mathbb{Z}_4$, respectively. Since these groups are not isomorphic, we see that these curves are not isomorphic either.

Exercise 11.12 Show that the elliptic curves $y^2 = x^3+ax+b$ and $y^2 = x^3+a'x+b'$ defined over $\mathbb{F}_5$ are isomorphic if and only if either they are equal or $a = a'$, $b \neq 0$ and $b = -b'$. Using this, compute the isomorphism classes of elliptic curves over $\mathbb{F}_5$.

Exercise 11.13 Show that the elliptic curves over $\mathbb{F}_5$ given by the equations $y^2 = x^3 + 1$ and $y^2 = x^3 + 2$ are not isomorphic but their groups of points are isomorphic (both are isomorphic to $\mathbb{Z}_6$).

Example 11.7 Let us now do a larger example in order to appreciate how the orders of the groups of rational points are distributed. Consider the prime $p = 223$. We compute the list of all elliptic curves over $\mathbb{F}_{223}$ but we do not print the list as it is too long:

```
> ecl223 := EllipticCurvesList(223):
```

The number of elliptic curves over $\mathbb{F}_{223}$ is $223^2 - 223 = 49506$:

```
> nops(ecl223);
                                  49506
```

Let us compute the orders of the groups of these curves, without printing them:

```
> egos223 := EllipticGroupOrder~(ecl223):
```

The maximum and minimum values of the orders are:

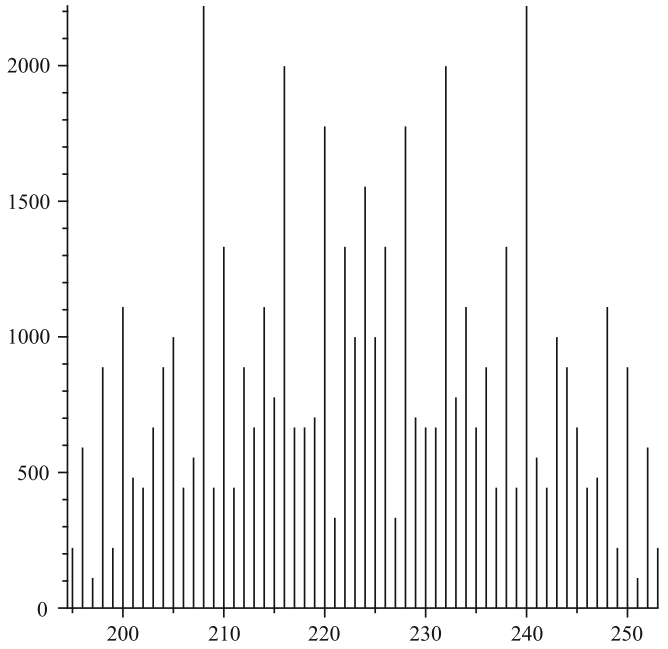
```
> min(egos223); max(egos223);
                                   195
                                   253
```

The number of curves of each order, starting with 195 and ending with 253 is:

```
> map(x -> ListTools:-Occurrences(x, egos223), [$195 .. 253]);
[222, 592, 111, 888, 222, 1110, 481, 444, 666, 888, 999, 444, 555, 2220, 444, 1332,
  444, 888, 666, 1110, 777, 1998, 666, 666, 703, 1776, 333, 1332, 999, 1554, 999,
  1332, 333, 1776, 703, 666, 666, 1998, 777, 1110, 666, 888, 444, 1332, 444, 2220,
  555, 444, 999, 888, 666, 444, 481, 1110, 222, 888, 111, 592, 222]
```

We see that all the integers in the interval [195, 253] are the order of some of these elliptic curves as none of the values in the preceding list is 0. To better appreciate how the orders are distributed, we may plot a histogram of these data, along with a summary of statistical data:

```
> Statistics:-Histogram(egos223, discrete = true, frequencyscale = absolute);
  Statistics:-DataSummary(egos223);
```




```
[mean = 224.0000000, standarddeviation = 14.93318520,
  skewness = 9.771282074 10^{-16}, kurtosis = 1.999979619,
  minimum = 195., maximum = 253., cumulativeweight = 49506.]
```

Let us look at the information provided by the example. As we have seen, the number of points of the elliptic curves over $\mathbb{F}_{223}$ are integers in the interval [195, 253] and, moreover, all these integers are the order of some elliptic curve. We also see that the plot is symmetric with respect to a vertical axis centered at the value $p+1=224$ (which is the center of the interval), i.e., the number of curves of order $p+1-t$ is equal to the number of curves of order $p+1+t$ for $t=1\ldots29$ which explains why the average value of the orders is precisely 224. This symmetry is a consequence of the fact that if $y^2=x^3+ax+b$ is an elliptic curve over $\mathbb{F}_p$ of order $p+1-t$ for some integer t, then taking its *quadratic twist*, namely the curve $y^2=x^3+u^2ax+u^3b$, where $u\in\mathbb{F}_p$ is a quadratic non-residue, induces a bijection on the isomorphism classes of elliptic curves and the group of points of the twisted curve has order $p+1+t$.

11.2.3 The Orders of Elliptic Curve Groups

An important thing we appreciate in the preceding example is that the orders of the elliptic curve groups are all contained in a relatively small interval centered at

$p+1$. We can give a simple heuristic argument that provides some justification for this. Indeed, since half of the elements of $\mathbb{F}_p^*$ are quadratic residues, given the curve $y^2 = f(x)$ one expects that approximately half of the nonzero values of $f(x)$, for x in $\mathbb{F}_p$, are quadratic residues. Since each of these values of $f(x)$ has two square roots in $\mathbb{F}_p$, we see that half the time the value of x gives two points on the curve and the remaining half it gives no points at all. The values of x for which $f(x) = 0$ give just one point each. Thus the number of points obtained this way is approximately p and hence, adding the point at infinity, we obtain $p+1$ as the expected order value.

The following important theorem, conjectured by E. Artin and proved by Hasse in the 1930s, shows that this heuristic reasoning is correct:

Theorem 11.3 (Hasse) *Let E be an elliptic curve defined over a finite field $\mathbb{F}_q$ (where $q = p^m$ for some prime p). Then the order of $E(\mathbb{F}_q)$ satisfies*

$$q + 1 - 2\sqrt{q} \leq |E(\mathbb{F}_q)| \leq q + 1 + 2\sqrt{q}.$$

We refer to [181, 197] for proofs of this theorem. The interval in the statement of the theorem is often called the *Hasse interval* and the integer $t = q + 1 - |E(\mathbb{F}_q)|$ is called the *trace* (or also the Frobenius trace) of the elliptic curve E over $\mathbb{F}_q$. Using the trace, Hasse's theorem for a curve over a prime field $\mathbb{F}_p$ may be reformulated as saying that $|E(\mathbb{F}_p)| = p + 1 - t$ with $|t| \leq 2\sqrt{p}$. If we look at our previous examples, we notice that in them there is an elliptic curve of order $p+1-t$ for each t such that $|t| \leq 2\sqrt{p}$, i.e., all the possible values in the Hasse interval are orders of some curve. It can be shown that this is always the case when the underlying field is a prime field of characteristic $p \neq 2, 3$.

Another fact which is of interest for some cryptographic applications—and also for Lenstra's elliptic curve factoring algorithm ECM—is that, as Lenstra proved, the orders of elliptic curves over a prime field $\mathbb{F}_p$ are "nearly uniformly distributed" on the interval $[p+1-\sqrt{p}, p+1+\sqrt{p}]$. The precise statement of this result as well as the way it is applied in ECM are presented in [53, Chap. 25] but suffice it to say that the orders are fairly uniformly distributed in the interval, except that their density drops off near the endpoints.

In order to better appreciate how the number of points of elliptic curves over $\mathbb{F}_p$ are distributed, we may use the following procedure that automatizes the method used in Example 11.7 and plots a histogram of these numbers:

```
> EllipticOrdersHistogram := proc(p::posint)
  local el, eo;
  el := EllipticCurvesList(p);
  eo := EllipticGroupOrder~(el);
  Statistics:-Histogram(eo, discrete=true, frequencyscale=absolute, color=black)
  end proc:
```

Exercise 11.14 Use the function `EllipticOrdersHistogram` to plot the histograms of the distribution of orders of elliptic curves over $\mathbb{F}_p$ when $p = 37$ and $p = 47$, and observe that for the former prime the most abundant orders are 36 and 40, which correspond to the curves of trace 2 and -2, respectively, while for the

second prime the most abundant order is 48, corresponding to the curves of trace 0 (we will see later that the DLP is easier than usual in curves of trace 0 or trace 2). Investigate also the prime 43 and observe that in this case the most abundant orders are 40 and 48 (corresponding to traces 4 and -4).

The previous method to compute the orders of elliptic curves runs, as we have mentioned, in exponential time. A more efficient—but still exponential—method based on the same argument consists of storing the Legendre symbols of all elements of the field in a list or array. This can be more efficiently accomplished without actually computing Legendre symbols. Instead, a one-dimensional array ls, indexed by the range $0..p-1$, is initialized by setting all entries equal to -1. Then we set $ls[0] := 0$, compute the squares $x^2 \mod p \in \mathbb{F}_p$ for $x = 1, \ldots, p-1$, and set $ls[s] := 1$ for each square $s = x^2 \mod p$. At the end of this process we have that $ls[x] = \left(\frac{x}{p}\right)$ for all $x \in [0, p-1]$. Then, given an elliptic curve, we compute the elements $x^3 + ax + b$ and we obtain $\sum_{x \in \mathbb{F}_p} \left(\frac{x^3+ax+b}{p}\right)$ as the sum of the terms $ls[x^3+ax+b]$. This way we do not have to use quadratic reciprocity and, moreover, much is gained in speed if we want to compute the orders of many elliptic curves over the same field because ls is only computed once for a given $\mathbb{F}_p$. We illustrate this method by coding it in Maple.

We start by giving a procedure to build the array just described. This is done by the following function that, on input a prime p outputs the array containing the Legendre symbols modulo p:

```
> LS := proc(p::posint)
  local ls, i;
  if not isprime(p) or p < 5 then
    error "%1 is not a prime >3", p
  end if;
  ls := Array(0 .. p-1, datatype = integer[1], fill = -1);
  ls[0] := 0;
  for i to p-1 do
    ls[i^2 mod p] := 1
  end do;
  ls
  end proc:
```

The next procedure computes the order of the group of points of an elliptic curve E over $\mathbb{F}_p$ on input the curve E and an array containing the Legendre symbols of the elements of $\mathbb{F}_p$ in the format output by the function `LS`:

```
> EllipticGroupOrderL := proc(E::list(integer), ls::Array)
  local a, b, p, i;
  a := E[1];
  b := E[2];
  p := E[3];
  1 + p + add(ls[(i^3+a*i+b) mod p], i = 0 .. p-1)
  end proc:
```

This is more efficient than the previous function `EllipticGroupOrder`. For example, the following order is quickly computed:

```
> EllipticGroupOrderL(EllipticCurve(9550684, 11443958, 12345701), LS(12345701));
                                12344442
```

Finally, the next function `EllipticGroupOrders` (not to be confused with the previous function `EllipticGroupOrder`, which only computes the order of a single curve) computes the number of curves of each possible order over a prime field $\mathbb{F}_p$. The only required input parameter is `p` for the prime characteristic and there is an additional optional parameter `ls`. This parameter is used for the array containing the precomputed values of the Legendre symbols modulo p of the elements of $\mathbb{F}_p$ and, if no argument is passed to it, this array is computed by default within the function. The output is a one-dimensional array indexed by the range $p+1-\lfloor 2\sqrt{p}\rfloor..p+1+\lfloor 2\sqrt{p}\rfloor$ (corresponding to the Hasse interval) and the value corresponding to a given index x is the number of elliptic curves of order x:

```
> EllipticGroupOrders := proc(p::posint, ls::Array := LS(p))
  local prod, t, orders, E, ord;
  if not isprime(p) or p < 5 then
    error "%1 is not a prime greater than 3", p
  end if;
  prod := combinat:-cartprod([[$0 .. p-1], [$0 .. p-1]]);
  t := floor(2*sqrt(p));
  orders := Array(p+1-t .. p+1+t, datatype = integer[8]);
  while not prod[finished] do
    E := [op(prod[nextvalue]()), p];
    if discr(E) <> 0 then
      ord := EllipticGroupOrderL(E, ls);
      orders[ord] := orders[ord] + 1
    end if;
  end do;
  orders
  end proc:
```

Example 11.8 We are going to use the previous procedure to compute the number of curves of each possible order over $\mathbb{F}_{1009}$. There are over one million elliptic curves over this field ($1009^2 - 1009 = 1017072$ curves, to be precise) and the function computes the order of each of these curves. Since the average number of points per curve is 1010, this computation involves going through over one billion points (although, of course, the number of points in $\mathbb{F}_{1009}^2$ is just 1009^2 and each of these points belongs to many curves). We store the array containing the number of curves of each order in a global Maple variable called `egos1009`:

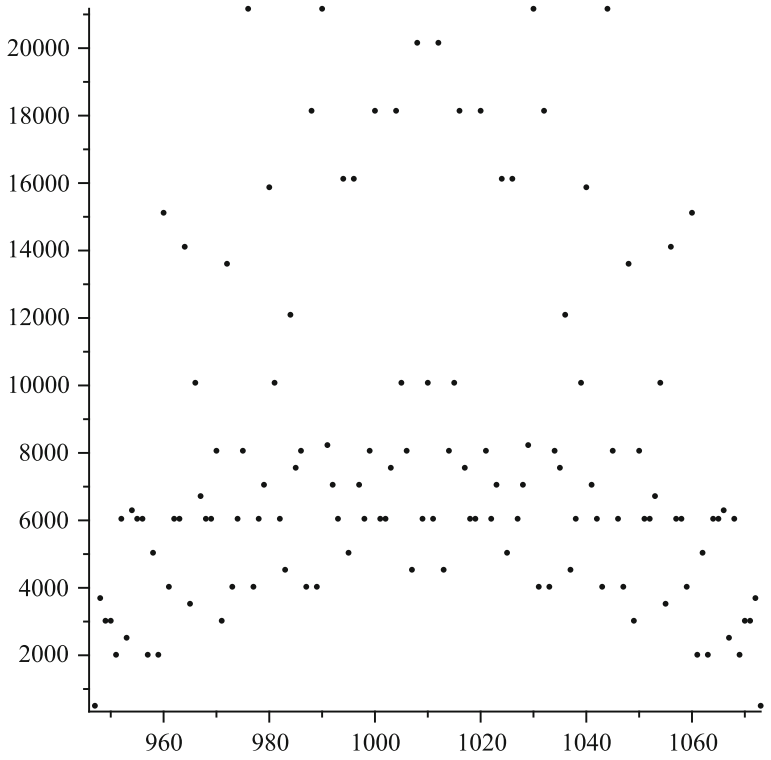
```
> egos1009 := EllipticGroupOrders(1009):
```

We obtain a 947..1073 array and, for example, we can look at the minimum and maximum values as well as at the total number of curves which are, respectively:

```
> min(egos1009);
  max(egos1009);
  add(egos1009[i], i = 947 .. 1073);
                                   504
                                 21168
                               1017072
```

We can obtain a visual representation of the distribution of the orders by plotting:

```
> Statistics:-PointPlot(egos1009, xcoords = [$947 .. 1073],
  color = black, symbol = solidcircle, symbolsize = 9);
```



The naive algorithm we have used to compute the order of an elliptic curve runs in exponential time and is not useful to compute the orders of elliptic curves suitable for cryptographic use. The computation of these orders is important, however, because one of the preferred methods to generate the curves is to choose a large finite field and to randomly select curves over that field until one is found with a suitable order which, as we will see, should be either prime or a product of a large prime by a small integer.

Fortunately, there are efficient methods to compute the order and they are based on an algorithm discovered by R. Schoof in 1985. If E is an elliptic curve over $\mathbb{F}_q$, where $q = p^n$ and p is prime, then we know by Hasse's theorem that the trace t satisfies $|t| \leq 2\sqrt{q}$ and hence falls within an interval of length $4\sqrt{q}$. Therefore, in order to determine t it is enough to compute $t \mod l$ for a set of small primes l whose product is $> 4\sqrt{q}$ which, using the Chinese remainder theorem, determines t uniquely. This is the idea of Schoof's algorithm whose running time is $O(\ln^8 q)$, which can be improved if asymptotically fast integer multiplication algorithms are used.

There were subsequent improvements of Schoof's algorithm, mainly due to Elkies and Atkin, which resulted in the so-called *SEA algorithm* which runs in time $O(\ln^6 q)$ with standard multiplication techniques. We refer to [181, 197] for descriptions of Schoof's algorithm. The SEA algorithm is also described in [197] and in more detail in Chap. 17 of [53]. There are several available implementations of these algorithms, such as those in the open source computer algebra package Sage [171], in the portable C library MIRACL (which is free for academic and non-commercial use and contains good support for elliptic curve cryptography) [143], and also in the computer algebra package Magma [132]. For example, even the basic implementation of Schoof's algorithm in MIRACL is able to compute reasonably quickly the order of an elliptic curve group over $\mathbb{F}_p$, with p a 256-bit prime—which is an adequate size for cryptographic use—and the SEA algorithm can compute the order for much larger primes.

Next we define a class of elliptic curves that play a special role in cryptographic applications:

Definition 11.2 Let E be an elliptic curve defined over a finite field $\mathbb{F}_q$ of characteristic p and let $t = q + 1 - |E(\mathbb{F}_q)|$ be the trace of E. Then E is said to be *supersingular*[6] if p divides t. Otherwise E is *ordinary* (or *non-supersingular*).

Remark 11.2 It can be shown that an elliptic curve over a field $\mathbb{F}_q$ of characteristic p is supersingular precisely when the curve has no points of order p over an algebraic closure of $\mathbb{F}_q$ [see, for example [197] (Proposition 4.31)].

In the particular case that $\mathbb{F}_q = \mathbb{F}_p$ is a prime field of characteristic $\neq 2, 3$ then E is supersingular if and only if $|E(\mathbb{F}_p)| = p+1$. This is because, by Hasse's theorem, we have that $|t| \leq 2\sqrt{p}$ and hence $p|t$ is equivalent to $t = 0$.

Example 11.9 Looking at the list containing the number of elliptic curves of each possible order over $\mathbb{F}_{223}$, computed in Example 11.7, we see that there are exactly 1554 supersingular curves, namely, the element in position 30 in the list, which corresponds to the curves of trace 0, since for $p = 223$ the maximum value of the trace is $\lfloor 2\sqrt{223} \rfloor = 29$. Similarly, in the case of $\mathbb{F}_{1009}$ studied in Example 11.8, we may use the array `egos1009` and we see that the number of supersingular curves—i.e., the number of curves of order 1010—is `egos1009[1010] = 10080`.

Exercise 11.15 Show that there are 4 supersingular elliptic curves over $\mathbb{F}_5$ distributed in two isomorphism classes containing two curves each.

Exercise 11.16 Write a Maple program that, given a prime $p > 3$ of moderate size, computes the number of supersingular elliptic curves over $\mathbb{F}_p$, as well as the number of elliptic curves with trace equal to 1 (these are called anomalous curves, see 11.3.3.2 below) and the number of curves with trace equal to 2. Use it to find the

[6] Note that this use of the term supersingular gives rise to another little terminological paradox: a supersingular elliptic curve is, by definition, nonsingular.

first prime such that the number of anomalous curves over $\mathbb{F}_p$ is greater than both the number of supersingular curves and the number of curves with trace 2 (these primes are not very abundant). Find also the prime $p < 1000$ for which the number of anomalous curves is largest.

A consequence of Hasse's theorem which is important for some cryptographic applications is that there is an efficient PPT algorithm to pick a point in $E(\mathbb{F}_p)$ uniformly at random. The algorithm simply picks a random element $x_0 \in \mathbb{F}$ and checks whether x_0 is the x-coordinate of a point in $E(\mathbb{F}_p)$. This can be accomplished by computing the Legendre symbol $\left(\frac{x_0^3+ax+b}{p}\right)$ which, as we have seen in Chap. 2 can be done in time $O(\text{len}(p)^2)$ by means of quadratic reciprocity. If the element thus obtained is a quadratic residue then a square root can be computed by using, e.g., Tonelli's Algorithm 2.8, in time $O(\text{len}(p)^4)$. We choose one of the two square roots y_0 and this gives a point on the curve, with coordinates (x_0, y_0). Since for each nonzero x-coordinate of a point there are two points on the curve and the minimum number of points is, by Hasse's theorem, $> p - 2\sqrt{p}$, we see that at least $\frac{1}{2}(p - 2\sqrt{p})$ elements of $\mathbb{F}_p$ are coordinates of points on the curve. Then, using Proposition 2.2, we obtain that the number of expected trials in order to find an x-coordinate of the curve is $\frac{2p}{p-2\sqrt{p}} = 2 + O\left(p^{-1/2}\right)$. Observe that this reasoning can also be applied to an elliptic curve over $\mathbb{F}_q$ where $q = p^m$ for some $m > 1$ and, also in this case, there exist PPT algorithms to extract square roots in $\mathbb{F}_q$. Note also that the probability of picking a given point of order 2 (in case it exists) with this algorithm is twice the probability of picking any other given point of order $\neq 2$ but this is not a serious deviation from uniformity because there are at most three points of order 2 and hence the probability of picking such a point is negligible in the size of p.

The next Maple function chooses a pseudo-random point on an elliptic curve given in the format output by `EllipticCurve`. The generator should be initialized first with a call to `RandomTools:-MersenneTwister:-SetState()` but we do not include this call inside the function because it could make the procedure repeatedly choose the same point if many points are chosen in quick succession:

```
> PseudoRandomEllipticPoint := proc(E::list(integer))
  uses RandomTools:-MersenneTwister;
  local a, b, p, ls, x, z, y, sign;
  a := E[1];
  b := E[2];
  p := E[3];
  ls := -1;
  while ls = -1 do
    x := GenerateInteger(':-range' = 0 .. p-1);
    z := (x^3+a*x+b) mod p;
    ls := numtheory:-legendre(z, p)
  end do;
  y := numtheory:-msqrt(z, p);
  sign := 2*GenerateInteger('range' = 0 .. 1)-1;
  y := sign*y mod p;
  [x, y]
  end proc:
```

Example 11.10 We initialize in Maple the "Curve P-256", which is an elliptic curve defined in [75] over a prime field corresponding to a 256-bit prime $p256$. This curve is the following:

```
> p256 := 2^256-2^224+2^192+2^96-1:
  P256 := EllipticCurve(-3,"5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e\
                        27d2604b", p256):
```

We find a pseudo-random point on the curve as follows:

```
> RandomTools:-MersenneTwister:-SetState():
  PseudoRandomEllipticPoint(P256);
[52363283436175358074013488409767837878347197462689483511863707423269106759043,
  27472337617088078407452476606541569370768112101332662796790021121765772447346]
```

We may check that the point is indeed on the curve:

```
> IsEllipticPoint(%, P256);
                                    true
```

11.2.4 Elliptic Curve Groups Over Prime Fields in Maple

In order to work with the group of rational points of an elliptic curve, we have to implement the group operation and we do this for the case of a prime field (as before, of characteristic > 3).

11.2.4.1 Elliptic Curve Point Addition

The following Maple function implements point addition closely following the steps indicated in Theorem 11.1. The input parameters are two points `P, Q` in the format output by `EllipticPoint` and an elliptic curve `E` in the format output by `EllipticCurve`, and the output is the result of adding the two points. If either of these points does not belong to the curve, an error message is returned.

```
> EllipticAdd := proc(P::{list,identical(0)}, Q::{list,identical(0)},
                      E::list(integer))
  local p, x1, y1, x2, y2, a, b, m, x3, y3;
  if not andmap(x -> IsEllipticPoint(x, E), [P, Q]) then
    error "both points must be on the curve"
  end if;
  if P = 0 then
    return Q
  elif Q = 0 then
    return P
  end if;
  p := E[3];
  x1 := P[1] mod p;
  y1 := P[2] mod p;
  x2 := Q[1] mod p;
  y2 := Q[2] mod p;
  if x1 = x2 and y1 = -y2 mod p then
    return 0
```

```
end if;
a := E[1];
b := E[2];
if P = Q then
  m := ((3*x1^2+a)/2*y1) mod p
else
  m := ((y2-y1)/(x2-x1)) mod p
end if;
x3 := (m^2-x1-x2) mod p;
y3 := (m*(x1-x3)-y1) mod p;
[x3, y3]
end proc:
```

Example 11.11 We can now build the addition table corresponding to a (small) elliptic curve group. Let us consider the group $E(\mathbb{F}_7)$ corresponding to the elliptic curve with equation $y^2 = x^3 + 3x + 2$. We compute the operation given as a map and the set of points of the elliptic curve:

```
> ec327 := EllipticCurve(3, 2, 7);
  op327 := proc (P, Q) -> EllipticAdd(P, Q, ec327);
  ep327 := EllipticPoints(ec327);
     [0, [0, 3], [0, 4], [2, 3], [2, 4], [4, 1], [4, 6], [5, 3], [5, 4]]
```

We see that the curve has order 9. Next we compute the group table using the Maple function `tabl` defined in 2.8.1.1:

```
> tabl(ep327, op327);
```

$$\begin{bmatrix}
0 & [0,3] & [0,4] & [2,3] & [2,4] & [4,1] & [4,6] & [5,3] & [5,4] \\
[0,3] & [2,3] & 0 & [5,4] & [0,4] & [5,3] & [4,1] & [2,4] & [4,6] \\
[0,4] & 0 & [2,4] & [0,3] & [5,3] & [4,6] & [5,4] & [4,1] & [2,3] \\
[2,3] & [5,4] & [0,3] & [4,6] & 0 & [2,4] & [5,3] & [0,4] & [4,1] \\
[2,4] & [0,4] & [5,3] & 0 & [4,1] & [5,4] & [2,3] & [4,6] & [0,3] \\
[4,1] & [5,3] & [4,6] & [2,4] & [5,4] & [0,3] & 0 & [2,3] & [0,4] \\
[4,6] & [4,1] & [5,4] & [5,3] & [2,3] & 0 & [0,4] & [0,3] & [2,4] \\
[5,3] & [2,4] & [4,1] & [0,4] & [4,6] & [2,3] & [0,3] & [5,4] & 0 \\
[5,4] & [4,6] & [2,3] & [4,1] & [0,3] & [0,4] & [2,4] & 0 & [5,3]
\end{bmatrix}$$

In this table, the sum of the point in position $(i, 1)$ with the point in position $(1, j)$ is the point in position (i, j). We have, for example, that if we keep adding the point $P = (0, 3)$ to itself, we successively obtain:

$$2P = (2,3),\ 3P = (5,4),\ 4P = (4,6),\ 5P = (4,1),$$

$$6P = (5,3),\ 7P = (2,4),\ 8P = (0,4),\ 9P = \mathcal{O}.$$

Thus we see that P has order 9 and hence P is a generator of the group which is, therefore, cyclic.

Exercise 11.17 Use a loop that keeps adding the point $(0, 3)$ to itself by means of `EllipticAdd`, in the curve `ec327`, to compute the order of this point.

There is a theorem of Cassels on the structure of the group $E(\mathbb{F}_q)$ that says that this group is isomorphic to $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2}$, where n_1 and n_2 are uniquely determined

positive integers such that n_2 divides both n_1 and $q-1$. Thus the group is either cyclic or the product of two cyclic groups and this theorem is often sufficient to completely determine the structure of the group (up to isomorphism). Moreover, by the structure theorem of finite abelian groups (see [45]), each finite abelian group is isomorphic (in an essentially unique way) to a product (also called a direct sum) of cyclic groups whose orders are prime powers (for the primes that divide the order of the group). Quite often the group $E(\mathbb{F}_q)$ will turn out to be cyclic. For example, this is the case when the order of the group is prime (as a consequence of Lagrange's theorem, see Corollary 2.1) and, more generally, this also happens if the order of the group is squarefree. In this case the group is a product of cyclic groups whose orders are pairwise relatively prime and hence it is cyclic by the Chinese remainder theorem (Theorem 2.14). But none of these conditions are necessary for the group to be cyclic and, indeed, there are cyclic groups of all possible orders, namely the groups (isomorphic to) $\mathbb{Z}_n$. In the case of elliptic curve groups we see, for example, that the group of order 9 in Example 11.11 is cyclic and hence isomorphic to $\mathbb{Z}_9$.

For cryptographic applications one often works with groups of prime order with the idea of making the DLP problem in the group as hard as possible (bearing in mind the Pohlig–Hellman algorithm). These groups have the additional advantage—not shared by other cyclic groups—that any element $\neq 0$ is a generator. However, cyclic groups of composite order have plenty of generators too—especially in case they have a large prime factor—because, as we have seen in Theorem 2.16, if the cyclic group has order n, then the number of generators is $\phi(n)$.

On some occasions we will also be interested in computing the order of a point $P \in E(\mathbb{F}_q)$. We know from Lagrange's theorem that the order of P is a divisor of the order of the group, $|E(\mathbb{F}_q)|$. As we will see, in cryptographic applications $|E(\mathbb{F}_q)|$ will usually be prime or have a large prime factor and, when its factorization is known, this leaves only a small number of divisors—just one in the prime case—as candidates to be the order of P. From this we obtain a polynomial-time algorithm to compute the order of P provided that multiplication of P by an integer—the additive analogue of exponentiation in a multiplicative group—can also be done in polynomial time. This is indeed the case because the binary exponentiation method works in any group and, in particular, in any elliptic curve group where it is often called the *double-and-add method*—the additive version of the "square-and-multiply" method we have seen for multiplicative groups. Since, as we have mentioned, there are polynomial-time algorithms to compute the order of the group, we see that computing the order of any point, on input the point and the Weierstrass equation of the curve, will probably be easy provided that factoring the group order is also easy, as is often the case.

We will also briefly mention that sometimes one goes in the opposite direction, i.e., one tries to compute the group order starting with the known order of some point(s). We know that the order of any point divides the group order and also that, by Hasse's theorem, the group order lies on an interval of length $4\sqrt{q}$. Therefore, if we find a point whose order is greater than $4\sqrt{q}$, there can be only one multiple of this order in the Hasse interval and it must be $|E(\mathbb{F}_q)|$. Even if the order of the point is smaller than $4\sqrt{q}$—but not too small—we obtain only a limited number of possibilities for $|E(\mathbb{F}_q)|$ in the Hasse interval. If we know the orders of several

points, this may be enough to uniquely determine $|E(\mathbb{F}_q)|$. We refer to the discussion in [197, Chap. 4] for more information on this subject.

The double-and-add algorithm mentioned above is also a crucial tool in the implementation of elliptic curve-based cryptographic algorithms such as Elgamal, just as is the case with its multiplicative counterpart when working in the multiplicative groups of finite fields. We are going to implement it in Maple and we also want to be able to multiply negative integers by points. For this we have to use the opposite point which is given by the following Maple function:

```
> EllipticOppositePoint := proc(P::{list(integer), identical(0)}, E::list(integer))
  local p;
  p := E[3];
  if not IsEllipticPoint(P, E) then
    error "the point is not on the curve"
  elif P = 0 then
    0
  else [P[1], -P[2] mod p]
  end if;
  end proc:
```

Example 11.12 We use the curve `P256` defined in Example 11.10. We generate a pseudo-random point on the curve and add it to its opposite:

```
> RandomTools:-MersenneTwister:-SetState():
  R := PseudoRandomEllipticPoint(P256):
  EllipticAdd(R, EllipticOppositePoint(R, P256), P256);
                                    0
```

11.2.4.2 Elliptic Curve Scalar Multiplication and the Order of Points

The next function implements the multiplication of a point by an integer—often called *scalar multiplication*—using the "double-and-add" algorithm. The required inputs are an integer k, a point P and an elliptic curve E, and the output is the result of adding P to itself k times in case k is positive and the result of adding $-P$ to itself $-k$ times when k is negative.

```
> EllipticMult := proc(k::integer, P::{list(integer), identical(0)},
                       E::(list(integer)))
  local t, R, S, bits, l, i;
  if not IsEllipticPoint(P, E) then
    error "the point is not on the curve"
  end if;
  if k = 0 or P = 0 then
    return 0
  end if;
  if k > 0 then
    t := k;
    R := P;
  else
    t := -k;
    R := EllipticOppositePoint(P, E)
  end if;
  S := 0;
  bits := convert(t, base, 2);
  l := nops(bits);
```

```
for i to l do
  if bits[i] = 1 then
    S := EllipticAdd(S, R, E)
  end if;
  R := EllipticAdd(R, R, E)
end do;
S
end proc:
```

Remark 11.3 The scalar multiplication operation is the one that dominates the execution time of elliptic curve cryptographic schemes. The previous function performs multiplication in a straightforward way, without exploiting any special structure of the curve or doing any precomputation. A survey of efficient algorithms that exploit some of these aspects or make use of projective coordinates is presented in [97, Chap. 3].

Example 11.13 We use the previous function to compute all the generators of the group of the elliptic curve `ec327 := EllipticCurve(3,2,7)`, which is cyclic of order 9 as we saw in Example 11.11. If we exclude the point at infinity, the remaining points have order either 3 or 9 (the divisors of 9 different from 1) and the generators are precisely the points of order 9, i.e., those points P such that $3P \neq \mathcal{O}$. They may be computed as follows:

```
> select(x -> evalb(EllipticMult(3, x, ec327) <> 0), ep327);
              [[0, 3], [0, 4], [2, 3], [2, 4], [4, 1], [4, 6]]
```

There are six generators in this case because $\phi(9) = 6$.

Exercise 11.18 Write a 'left to right' version of the Maple function implementing the multiplication operation, namely, the elliptic curve analogue of the function `ExpMod` in Sect. 2.7.

Next we give a function to pseudo-randomly choose a generator of a cyclic elliptic curve group over a prime field. This function is simply a transcription to the elliptic curve setting of our earlier function `findgenerator`, which was used in Chap. 2 to find a generator of the multiplicative group $\mathbb{Z}_p^*$. The inputs are an elliptic curve, its order and, optionally, the factors of the order in the format provided by Maple's function `ifactors`. If the factors are not supplied then they are computed inside the function. The output is a generator of the group. We remark that elliptic curve groups of prime order are often selected for cryptographic use. In that case this function is not needed, because to choose a generator pseudo-randomly it suffices to choose a point $\neq \mathcal{O}$ in the curve using the previous function `PseudoRandomEllipticPoint`.

```
> EllipticGroupGenerator := proc(E::list(integer), order::posint,
                                 {orderfactors := ifactors(order)[2]})
  local n, l, i, found, k, P, G;
  n := nops(orderfactors);
  RandomTools:-MersenneTwister:-SetState();
  l := [];
  for i to n do
    found := false;
```

```
    k := order/orderfactors[i][1];
    while not found do
      P := PseudoRandomEllipticPoint(E);
      found := evalb(EllipticMult(k, P, E) <> 0)
    end do;
    k := order/(orderfactors[i][1])^orderfactors[i][2];
    l := [op(l), EllipticMult(k, P, E)]
  end do;
  G := 0;
  for i to n do
    G := EllipticAdd(G, l[i], E)
  end do;
  G
  end proc:
```

Example 11.14 We construct an elliptic curve over $\mathbb{F}_p$, where $p = p256$ is the 256-bit prime over which the NIST curve P-256, used in Example 11.10, is defined. This prime is the following:

```
> p256 := 2^256-2^224+2^192+2^96-1:
```

The elliptic curve that we will use is:

```
> ec256 := EllipticCurve(7, 11, p256):
```

The order of this curve can be computed by means of the implementation of the SEA algorithm in [143] and it turns out to be:

```
> o256:=115792089210356248762697446949407573529756567700756856116485097055070429878012:
```

The order is easy to factor and we obtain:

```
> ifactor(o256);
 (2)^2  (72361432789) (264147774106088250771567413659195695032988900394667)
 (2260717937) (669913)
```

As we have remarked, if the order is squarefree then the group is cyclic but in this case we have a squared factor 2. By the structure theorem of finite abelian groups this group has either a factor (isomorphic to) $\mathbb{Z}_4$ or a factor $\mathbb{Z}_2 \times \mathbb{Z}_2$. The first possibility occurs precisely when there is a unique element of order 2 (the only element of order 2 in $\mathbb{Z}_4$ is 2) and the second occurs when there are (as in $\mathbb{Z}_2 \times \mathbb{Z}_2$) three elements of order 2. So, in order to see whether the group of the curve `ec256`is cyclic, it suffices to check how many points of order 2 it has. As we have seen, the points of order 2 correspond to the roots in $\mathbb{F}_p$ of the cubic polynomial $x^3 + ax + b$ defining the curve and we can compute these roots with Maple as follows:

```
> Roots(x^3+7*x+11) mod p256;
[[111237276231358922482507679868246738551900716794723608870821124038139942305782, 1]]
```

We see that there is a single root—the 1 on the right indicates the multiplicity—and hence a single element of order 2, so that the group is indeed cyclic. Thus we can use the previous function to compute a generator as follows:

```
> g256 := EllipticGroupGenerator(ec256, o256);
[69385755060832967159269382451180048774019250174739376235116761742455836744302,
  143204943736712009086413101581750065466733364427765382688727128217466183586626]
```

Exercise 11.19 Write a Maple function that checks whether a given point of an elliptic curve cyclic group is a generator and use it to check that points obtained with the `EllipticGroupGenerator` function, as in the previous example, are indeed generators.

We next give a Maple function that computes the order of a point on an elliptic curve given the group order. The input parameters are the point, the elliptic curve, the order of the group and, optionally, the sorted set of divisors of the group order, which may be passed to the procedure through the keyword parameter `orddivs`. If no value is passed to this parameter then Maple computes the divisors by default. The output is the order of the point in the elliptic curve group.

```
> EllipticPointOrder := proc(P::{list(integer), identical(0)}, E::(list(integer)),
          ord::posint, {orddivs := sort(convert(numtheory:-divisors(ord), list))})
  local d;
  option remember;
  if `not`(IsEllipticPoint(P, E)) then
    error "the point is not on the curve"
  end if;
  for d in orddivs while EllipticMult(d, P, E) <> 0 do
    NULL
  end do;
  d
  end proc:
```

Example 11.15 Using a modified version of the function `EllipticCurve Points`, one sees that the points of the curve `ec256` in Example 11.14 whose x-coordinate is ≤ 10 are the elements of the following list:

```
> ep256110 :=
[[2, 65580740334265363388806975267261614797714391452302187355066257759605956091865],
 [2, 50211348876090885373890471682145958732371751962988126840467373549261141762086],
 [10, 772399525751210806539997196000855832388411879935964660804900073269947946531851,
 [10, 385521366352351681086977273493219902912449554216938481150436239818723032007661]
```

These points appear in pairs, where one point is followed by its opposite, which has the same x-coordinate. Thus the order of the first point above is equal to the order of the second and, similarly, the order of the third is the same as the order of the fourth. We may compute the order of the first and third points as follows:

```
> map(x -> EllipticPointOrder(x, ec256, o256), subsop(2 = NULL, 4 = NULL, ep256110));
[57896044605178124381348723474703786764878283850378428058242548527535214939006,
 11579208921035624876269744694940757352975656770075685611648509705507042987801 2]
```

We see that the order of the first point in the list above is just one-half of the group order, while the order of the third point is equal to the group order, meaning that this point is a generator of the group.

Example 11.16 To illustrate the simple computations on elliptic curves that can be carried out with the previous functions, we give an example in which knowledge of the order of a single point of an elliptic curve allows us to determine the order of the group of rational points. This technique, combined with the baby-step giant-step method to compute the order of a point, leads to a subexponential algorithm—the

Mestre–Shanks method—to compute the order of the group. This algorithm is much more efficient than the naive point counting we have seen but is not efficient enough for curves of cryptographic size, where one has to apply Schoof's algorithm or the SEA algorithm.

Consider the following prime, used in [75] to define an elliptic curve:

```
> p192 := 2^192-2^64-1;
         6277101735386680763835789423207666416083908700390324961279
```

We define an elliptic curve over this prime:

```
> ec192 := EllipticCurve(-1512718932230020811709266491917131419968886211715362931724,
    1769520287457231963690687866380165522983746710433431160335, p192):
```

The point

```
> pt192 := [6268594639044626559588135250819421255947009647065456089152,
    4094928720749295414661683627820549126173799055397430561800]
```

can be shown to have order equal to:

```
> opt192 := 5626621716526886573601989555760 59:
```

To check that this number is indeed the order of the point we first show that `opt192` multiplied by the point gives the point at infinity:

```
> EllipticMult(opt192, pt192, ec192);
                                    0
```

This shows that the order of `pt192` divides `opt192` and to see that the two are equal we have to show that if `pt192` is multiplied by the result of dividing `opt192` by any one of its prime factors, then the result is different from the point at infinity. This can be done independently for each of the three prime factors of `opt192` but we do it in one pass:

```
> member(0, map(x -> EllipticMult(x, pt192, ec192),
                   iquo~(opt192, op~(1, ifactors(opt192)[2]))));
                                false
```

Next we check whether the order `opt192` is greater than $4\sqrt{p}$:

```
> evalb(opt192 > evalf(4*sqrt(p192)));
                                true
```

Since this property holds, we know from our previous discussion regarding Hasse's theorem that `opt192` has a unique multiple in the Hasse interval, which must be equal to the order of the curve. We compute this order as follows:

```
> o192:= opt192*iquo(p192+1+floor(evalf(2*sqrt(p192))), opt192);
         6277101735386680763835789423128040313757010230206649133264
```

Exercise 11.20 Consider the curve `ec192` defined in Example 11.16 and determine the structure of its group of rational points as follows (some knowledge of the structure theorem for finite abelian groups is required for this purpose).

(i) Use Maple to find the prime factors of the order o of the group, (the value `o192` which was computed in Example 11.16) and check that it has a factor 2^4 plus some odd prime factors.
(ii) Show that there are exactly three points of order 2 on the curve. Deduce that the group of points cannot be cyclic because, in that case, there could only be one element of order 2.
(iii) Find a point of order $o/2$ (pick random points until one of this order is found).
(iv) Using the point computed in the previous item, find a point of order 8 on the curve.
(v) Conclude that the group of rational points is isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_{o/2}$ (this is the form provided by Cassels' theorem) and also to $\mathbb{Z}_2 \times \mathbb{Z}_8 \times \mathbb{Z}_q$ where $q = o/16$ is the odd cofactor of o.

We briefly mention that we may use the function `EllipticPointOrder` to give an alternative method to pseudo-randomly find a generator of a cyclic elliptic curve group by just choosing pseudo-random points until one is found whose order is the same as the group order. Note that, if n is the order of the group, the expected number of trials is $O(n/\phi(n))$—because there are $\phi(n)$ generators—and this number is $O(\text{len}(n))$. This algorithm is implemented by means of the following Maple function which takes as input the curve, the group order and, optionally, the sorted list of divisors of the order (if this list is not supplied then Maple computes it by default).

```
> EllipticGroupGenerator2 := proc(E::(list(integer)), order::posint,
              {orderdivisors := sort(convert(numtheory:-divisors(order), list))})
  local found, P;
  RandomTools:-MersenneTwister:-SetState();
  found := false;
  while not found do
    P := PseudoRandomEllipticPoint(E);
    found :=
          evalb(EllipticPointOrder(P,E,order,':-orderdivisors'=orderdivisors)=order)
  end do;
  P
  end proc:
```

Example 11.17 Let us use the previous function to pseudo-randomly choose a generator of the curve `ec256` in Example 11.14. We assume that the global variables `ec256` and `o256` still hold the values assigned in that example.

```
> EllipticGroupGenerator2(ec256, o256);
[52664204410327494153641788625224005901320065157080443069487579774976347773357,
11262823617313383273490112435001523683099830283120797162056731506038016638958 5]
```

Let us check that the order of the point thus generated is the same as the order of the elliptic curve group:

```
> evalb(EllipticPointOrder(%, ec256, o256) = o256);
                                    true
```

11.3 The Elliptic Curve Discrete Logarithm Problem

We have discussed the discrete logarithm problem (DLP) in Sect. 6.5 and we have seen that the hardness of this problem is a necessary condition for the security of a number of cryptographic schemes. The DLP was formulated in an arbitrary group G and some of the algorithms to solve it which we have seen are *generic* in the sense that they do not exploit any special properties of the group nor of the encoding of its elements as strings, and hence work for arbitrary groups. It is then clear that the problem can be specialized to elliptic curve groups (over finite fields) and that the generic algorithms will work in these groups. However, the most efficient algorithms to solve the DLP, namely, those collectively known as the *index calculus method*, use specific information about the groups in which they are applied—only the multiplicative groups of finite fields for now—to achieve subexponential complexity in contrast with the fully exponential complexity of generic algorithms.

The purpose of this section is to briefly review the DLP in the concrete setting of elliptic curve groups and, in particular, the aspects related to the hardness of the problem and its cryptographic applications. First we recall the definition of the problem and establish the basic notation:

Definition 11.3 Let E be an elliptic curve over a finite field $\mathbb{F}_q$, $P \in E(\mathbb{F}_q)$ a point of order n, and $Q \in \langle P \rangle$ a point in the subgroup of $E(\mathbb{F}_q)$ generated by P. An instance of the *elliptic curve discrete logarithm problem* (ECDLP) is to find the element $l \in \mathbb{Z}_n$ such that $Q = lP$. As usual we denote:

$$l = \log_P Q,$$

and l is called the *discrete logarithm of Q to the base P.*

Remark 11.4 Note that, as we have seen in our general discussion in Sect. 6.5, the discrete logarithm is only defined modulo the order n of P and hence we may regard it as an element of $\mathbb{Z}_n$, which can also be thought of as an integer in the interval $[0, n-1]$. Thus the discrete log function defines a group isomorphism $\log_P : \langle P \rangle \to \mathbb{Z}_n$ that, as we have seen, is the inverse of the "discrete exponential isomorphism" which now, since the group $\langle P \rangle$ is additive, is really the "multiplication isomorphism" assigning to each $k \in \mathbb{Z}_n$ the point $kP \in \langle P \rangle$.

The most basic algorithm to solve the ECDLP is the brute-force method that does an exhaustive search and successively computes $P, 2P, 3P \ldots$ until Q is found. We may compare the ECDLP to the problem of computing the order of a point P, since the latter is just an instance of the former, namely, to compute $\log_P \mathcal{O}$. There is a crucial difference between the two problems though, which is the fact that if we want to compute the order of a point and we assume that the factorization of the group order is known, then Lagrange's theorem tells us that the point order is a divisor of the group order and so, instead of computing the multiples of the point by all positive integers we only have to compute the multiples by these divisors.

If the number of divisors is small, as usually happens in the case of cryptographic applications in which either $|E(\mathbb{F}_p)|$ is prime (which automatically gives the order of any nonzero point) or has a large prime factor, then the computation of the point order becomes easy. In the ECDLP problem, even if the order of the point and its factorization are known, we cannot apply this shortcut and have to run through all the possible values $1, 2, \ldots$ in succession, as it is obvious that any of them might be the discrete log we are looking for. Thus the brute-force algorithm takes $O(n)$ steps and it will require approximately n steps in the worst case and $n/2$ steps on average. Hence the running time is exponential on the size of n and it suffices to take n to be about 100 bits long to make this computation infeasible in practice. Although there are more efficient algorithms for the ECDLP (the collision-based algorithms mentioned below) we stress that, in contrast with the computation of the order, the ECDLP problem is thought to be (very) hard in general, and even more so if the group order is prime.

The generic algorithms to solve the ECDLP are, as a direct consequence of the very definition of *generic*, the same as those we studied in Sect. 6.5. These are collision-based algorithms such as the baby step-giant step method and Pollard's rho algorithm, which require $O(\sqrt{q})$ steps to solve the ECDLP in $E(\mathbb{F}_q)$—or $O(\sqrt{n})$ to solve it in a subgroup of order n—and hence, while faster than brute force, still run in exponential time.

11.3.1 The Rho Method and Pohlig–Hellman for the ECDLP in Maple

In order to appreciate how the collision-based algorithms work in the elliptic curve case, we next give a Maple implementation of the one which is probably the fastest of them, namely, Pollard's rho. We take the function `RhoDiscreteLog` of Sect. 6.5 and translate it to the elliptic curve setting. The translation is straightforward since it essentially consists of replacing the modular exponentiations used to compute powers of elements in subgroups of $\mathbb{Z}_m^*$ by multiplications of points by integers in subgroups of $E(\mathbb{F}_p)$ (in consonance with our previous implementation of elliptic curve algorithms, we will restrict ourselves to elliptic curves defined over prime fields).

In the Maple function implementing the rho method we are going to use a partition of the group $\langle P \rangle$ into three subsets similar to the one originally suggested by Pollard, using here the x-coordinate of the point to obtain a partition according to its size. We remark, however, that many other random-like partitions of this group into subsets of roughly the same size are possible and that it is common to select a larger number of subsets or *branches*, typical numbers being 16 and 32. For example, to obtain 32 branches, the points of $\langle P \rangle$ can be partitioned according to the five least significant bits of the x-coordinates as suggested in [97].

The function `EllipticRhoDiscreteLog` takes as input P, Q, E, n, where P is a point of order n on the elliptic curve E and $Q \in \langle P \rangle$, and computes $\log_P Q$:

```
> EllipticRhoDiscreteLog := proc(P::{list(integer),identical(0)},
                  Q::{list(integer),identical(0)}, E::(list(integer)), n::posint)
  local p, f, b, c, found, sols, i, finished, ans, u;
  p := E[3];
  f := proc(X, r, s)
  if X[1] < p/3 then
    EllipticAdd(P, X, E), (r+1) mod n, s
  elif X[1] < 2*p/3 then
    EllipticMult(2, X, E), 2*r mod n, 2*s mod n
  else
    EllipticAdd(Q, X, E), r, (s+1) mod n
  end if
  end proc;
  b := 0, 0, 0;
  c := 0, 0, 0;
  finished := false;
  while not finished do
    found := false;
    while not found do
      b := f(b);
      c := f(f(c));
      found := evalb(b[1] = c[1])
    end do;
    if b[3] <> c[3] or n < 1000 then
      sols := Roots((b[3]-c[3])*x-c[2]+b[2]) mod n;
      sols := [seq(sols[i][1], i = 1 .. nops(sols))];
      for i in sols do
        if Q = EllipticMult(i, P, E) then
          return i
        end if
      end do
    end if;
    ans := readstat("Not found, continue?, y/n");
    if ans = y then
      print("Computation proceeds with new initial values...");
      randomize();
      u := RandomTools:-Generate(integer(range = 1 .. n-1));
      b := EllipticMult(u, P, E), u, 0;
      c := b
    else
      finished := true;
      print("Computation interrupted")
    end if
  end do
  end proc:
```

Example 11.18 Let us consider the curve `ec192` defined in Example 11.16, which has order `o192` (we assume these variables still hold the values assigned in Example 11.16), and the point $P1$ defined below, which we see is on the curve:

```
> P1 := [4628838975151188443075672732350192991625776830582167450636,
 2329625211922066201614287684266153112582626223520455785901:

> IsEllipticPoint(P1, ec192);
                                  true
```

The order of this point is:

```
> oP1 := EllipticPointOrder(P1, ec192, o192);
                               109152774691
```

Consider now the point:

```
> Q1 := [606389957509323064350469002782367616770721864822846437400 4,
 586688481368907525155362883925707561511637469127393422165 8]
```

We are given the information that $Q1$ belongs to the subgroup generated by $P1$ and the problem is to compute $\log_{P1} Q1$. We do it as follows:

```
> EllipticRhoDiscreteLog(P1, Q1, ec192, oP1);
                                98765432101
```

11.3.1.1 The Pohlig–Hellman Algorithm in Maple

As we have seen in Sect. 6.5, when the group order n is not prime there is a generic algorithm (the Pohlig–Hellman algorithm) that essentially reduces the computation of a discrete logarithm to the computation of discrete logarithms in subgroups whose orders are the prime power factors (or even the prime factors) of n. We have also seen that the running time of the combination of the rho method with Pohlig–Hellman is $O\left(\max_i(\sqrt{n_i}) \cdot \text{polylog}(n)\right)$, where the n_i are the prime power factors of n, so that the running time is dominated by the largest of these factors (and, in fact, the running time may be reduced to $O\left(\max_i\left(\sqrt{p_i}\right) \cdot \text{polylog}(n)\right)$, where the p_i are the distinct prime factors of n).

These aspects make clear the convenience of using groups of prime order (or with order equal to the product of a prime by a small integer) to make the DL problem as hard as possible. All this applies entirely to the ECDLP and we may easily adapt the Maple function `PohligHellman` given in Sect. 6.5 for this problem. In this version we will not include the baby-step giant-step algorithm which is similarly easy to translate to the elliptic setting, something that we will leave as an exercise. The function `EllipticPohligHellmanRho` takes as input P, Q, E, n, where P is a point of order n on the elliptic curve E and $Q \in \langle P \rangle$, and computes $\log_P Q$:

```
> EllipticPohligHellmanRho := proc(P::{list(integer),identical(0)},
                Q::{list(integer),identical(0)}, E::(list(integer)), n::integer)
  local facts, mults, i, exps, Plist, Qlist, ordlist, xlist;
  if E[3] < 100000 then
    return EllipticRhoDiscreteLog(P, Q, E, n)
  end if;
  facts := ifactors(n)[2];
  facts := map(l -> l[1]^l[2], facts);
  mults := n/~(facts);
  Plist := map(x -> EllipticMult(x, P, E), mults);
  Qlist := map(x -> EllipticMult(x, Q, E), mults);
  ordlist := map(x -> EllipticPointOrder(x, E, n), Plist);
  xlist := [];
  for i to nops(mults) do
    xlist := [op(xlist), EllipticRhoDiscreteLog(Plist[i], Qlist[i], E, ordlist[i])]
  end do;
  chrem(xlist, facts)
  end proc:
```

Example 11.19 We go back to the curve `ec192` of Example 11.16 and show how the preceding function allows us to compute a discrete logarithm in a large group whose order is 191 bits long. We assume that the global variables `ec192` and `o192` still

hold the same values as in that example namely, the curve and its order, respectively. Consider the point:

```
> P2 := [346354167837191285165912031665687640342001478515158683920 8,
 568842212284238416323372900024177583272538035222864414728]:
```

`IsEllipticPoint` can be used to check that `P2` is indeed a point on the curve. We compute its order as follows:

```
> oP2 := EllipticPointOrder(P2, ec192, o192);
        3138550867693340381917894711564020156878505115103324566632
```

If we compare `oP2` to the order of the group we see that the latter is twice the former. In fact, if one pseudo-randomly chooses points on the curve, most of the time they will have order equal to one-half the curve order (Exercise: why?). It can be easily checked that the order of $P2$ is a 191-bit integer and hence it might seem that computing discrete logarithms in $\langle P2\rangle$ should be very hard (see Example 11.20 below) but it is really not so because:

```
> ifactor(oP2);
 (2)^3 (3) (97) (104109613) (109152774691) (6624920655989) (23014823341457) (778094341)
```

The order of `P2` is a product of small primes and this makes computing discrete logs in $\langle P2\rangle$ easy because of Pohlig–Hellman. Let us consider, for example, the point `Q2` obtained by multiplying `P2` by a large integer:

```
> Q2 := [227962047680369835597511679531739443271793382224806252225 9,
        3165999503627392754528245842683201607139518735263854029139]
```

Using the function `EllipticPohligHellmanRho` to compute $\log_{P2} Q2$ we obtain:

```
> EllipticPohligHellmanRho(P2, Q2, ec192, oP2);
         552875008973441621939839281572975238731135741386081064 93
```

The following checks that this value is indeed the discrete logarithm for which we searched:

```
> evalb(EllipticMult(%, P2, ec192) = Q2);
                                  true
```

Exercise 11.21 Write an elliptic curve version of the Maple function `BSGS` given in Sect. 6.5 and use it to compute the discrete logarithm of Example 11.18. Write also a function which combines this algorithm with the Pohlig–Hellman algorithm and use it to solve the discrete logarithm of Example 11.19.

In view of the Pohlig–Hellman algorithm we will henceforth assume, without loss of generality, that the group $\langle P\rangle$ where the ECDLP is considered, has prime order. In practice, this is achieved by selecting an elliptic curve whose order is either prime or has a large prime factor and then taking P to be a generator of the subgroup of this prime order. The simplest way to obtain an elliptic curve whose group of rational points has prime order is to choose curves at random and compute their orders until

one is found that is prime. Because of the near uniform distribution of orders in the Hasse interval, the probability that $|E(\mathbb{F}_p)|$ is prime or has a prime factor of minimum specified length should be close to the probability of a random integer in the interval having this property. Although not much is proved about the occurrence of primes in short intervals, it has been conjectured by Galbraith and McKee [82] that the probability that a randomly chosen elliptic curve over $\mathbb{F}_p$ has prime order can be approximated by $c_p/\ln p$ as $p \to \infty$, where c_p depends on p and lies between 0.44 and 0.62. Thus the probability of choosing a curve of prime order this way is expected to be about one half the probability that a randomly chosen integer x of the same size as p turns out to be prime, which is asymptotic to $1/\ln x$ by the PNT (Theorem 6.2).

11.3.2 The Current State of the ECDLP

The rho method has a parallelized version which is more efficient because it yields a factor m speedup when m processors are used. This is because all processors use the same iterating function to compute the sequence of points and there is a strategy that enables efficient finding of a collision in the sequences generated by different processors (see, for example, [97] for details).

The parallelized version of the rho method holds the current record for an ECDLP. We give an account of this record in the next example.

Example 11.20 Consider the following parameters, where `p112` is prime:

```
> a := 4451685225093714772084598273548424:
  b := 2061118396808653202902996166388514:
  p112 := (2^128-3)/(11*6949):
  isprime(p112);
                                  true
```

and the elliptic curve defined by the equation $y^2 = x^3 + ax + b$ over $\mathbb{F}_{p112}$:

```
> E112 := EllipticCurve(a, b, p112):
```

This curve was standardized as curve `secp112r1` in a standard issued in 2000 which is now obsolete (currently, the minimum recommended size of the prime p for elliptic curves defined over $\mathbb{F}_p$ is 192 bits). The order of the group of rational points of this curve can be easily computed using, for example, Schoof's algorithm and it turns out to be the following 112-bit prime:

```
> o112 := 4451685225093714776491891542548933:
  isprime(o112);
                                  true
```

Thus each point on the curve, with the sole exception of the point at infinity, is a generator of the group. In particular the point:

```
> P := [1882814650579725348922237787137 52, 3419875491033170827167861896082688];
  IsEllipticPoint(P, E112);
                                  true
```

is a generator and hence has order equal to `o112`. The authors of [39] considered also the following point:

```
> Q := [1415926535897932384626433832795028, 3846759606494706724286139623885544];
  IsEllipticPoint(Q, E112);
                                    true
```

and used a cluster of more than 200 PlayStation 3 (PS3) game consoles at the Laboratory of Cryptologic Research of the EPFL (Lausanne) to compute $\log_P Q$ which, as we now check, is the integer appearing in the expression below:

```
> evalb(Q = EllipticMult(3125216360147724771617673518566 99, P, E112));
                                    true
```

It is estimated that the computation would have taken 3.5 months if run continuously and that it required about $8.5 \cdot 10^{16}$ elliptic curve point additions. Although prime fields with 112-bit primes are most likely not being used much anymore in elliptic curve cryptography, this also has some significance for 160-bit prime fields. Since the rho method requires about $O(\sqrt{p})$ steps and a 160-bit prime is about 2^{48} times a 112-bit prime, we see that solving the ECDLP over a 160-bit field will require an effort around $\sqrt{2^{48}} = 2^{24} \approx 16$ million times as large as for 112-bit fields. This seems to leave some security margin for the near future but one should be cautious and NIST has recommended also phasing out this size and using a minimum of 192 bits, which adds another 2^{16} factor to the time required by the rho method and similar collision algorithms.

The current ECDLP record illustrates the fact that the subexponential collision methods—and the rho method in particular—are thought to be the fastest algorithms to solve the ECDLP. But, as happens with other problems which are presumed to be hard, there is no proof that this is really the case. Thus we have to rely on the practical evidence gathered over the years since elliptic curve cryptography was discovered, which can be summarized in the fact that, in contrast with what happens for the DLP over the multiplicative groups of finite fields, no subexponential-time algorithms for the ECDLP have been found. Moreover, Miller has given a widely accepted heuristic argument that supports the idea that the obvious candidate for a subexponential algorithm, namely, the index calculus method, cannot be successfully implemented on elliptic curve groups; we refer to [142] and also to [97, Sect. 4.1] for the details.

In addition to this, there is an interesting result of Shoup [178] which gives a lower bound of $O(\sqrt{n})$ operations for the DLP in *generic groups* of prime order n, where the term generic means that the group elements are random bit strings and the adversary only has access to the group operation through an oracle (i.e., as a black box). This proof does not imply the hardness of the ECDLP but points to the possibility that the DLP problem is really hard in some groups which have a generic-like behavior. It is fair to say, nonetheless, that elliptic curve groups come equipped with some additional structure and, even accepting the unlikelihood of a reasonable index calculus method, there is no guarantee that no other faster-than-exponential algorithm exists. Because of this and the fact that elliptic curve cryptography was introduced not so long ago, some cryptographers have expressed concern about its use. However, the

situation is not so different from other cryptographic schemes as it might appear at first sight. For example, we do not have any formal guarantee about the hardness of the integer factorization problem either, which is a necessary condition for the security of RSA. Although the IFP was already mentioned by Gauss, it was not really until the introduction of RSA in the 1970s that the problem became widely studied with the help of modern computers. Elliptic curve cryptography is a few years more recent but this difference will become less and less significant with time.

11.3.3 Reduction Attacks Against the ECDLP

We have mentioned that generic algorithms, such as the rho method, are the fastest to solve the ECDLP on subgroups of prime order of elliptic curve groups, but this is true only if the curve is carefully chosen and meets several requirements which we are going to briefly discuss. In the early years of elliptic curve cryptography, when Schoof's algorithm and its variants were not yet fully developed, it was common to choose, for cryptographic use or at least for demonstration purposes, special types of elliptic curves whose structure allowed a quick computation of the group order. This was the case with some curves which were known to be supersingular and hence to have exactly $p+1$ points in case they are defined over a prime field $\mathbb{F}_p$. For example, we have the following family of supersingular curves:

Proposition 11.3 *Let p be a prime such that $p \equiv 3 \pmod 4$ and $a \in \mathbb{F}_p^*$. Then the elliptic curve E over $\mathbb{F}_p$ given by the equation $y^2 = x^3 + ax$ is supersingular.*

Proof Since $p \equiv 3 \pmod 4$ we have that -1 is not a square in $\mathbb{F}_p$ by Euler's criterion 2.12. This implies that, for each $x \in \mathbb{F}_p^*$, either x or $-x$, but not both, is a square. If we set $f(x) = x^3 + ax$ we see that, for each $x \in \mathbb{F}_p$, $f(-x) = -f(x)$ and hence for each $x \in \mathbb{F}_p^*$ such that $f(x) \neq 0$ we have that either $f(x)$ or $f(-x)$ is a square in $\mathbb{F}_p$. Thus for each such x we have that exactly one of x, $-x$ is the x-coordinate of two points in $E(\mathbb{F}_p)$, the corresponding y-coordinates being the two square roots of whichever of $f(x)$ or $-f(x)$ is a square. If $x \in \mathbb{F}_p^*$ is such that $f(x) = 0$, then we also have the points $(x, 0)$ and $(-x, 0)$, so we see that the $p-1$ elements of $\mathbb{F}_p^*$ are the x-coordinates of exactly $p-1$ points on the curve. In addition we have the point $(0, 0)$ and the point at infinity, so $|E(\mathbb{F}_p)| = p + 1$ and the curve is supersingular. □

Example 11.21 We show how to find a subgroup of (large) prime order of $E(\mathbb{F}_p)$, where E is the supersingular elliptic curve with equation $y^2 = x^3 + x$ for a suitable prime p such that $p \equiv 3 \pmod 4$. If we choose p of the form $p = 4n - 1$, with n prime, then the condition $p \equiv 3 \pmod 4$ is satisfied and, moreover, $|E(\mathbb{F}_p)| = p+1 = 2^2 n$, so that the order of the group has a prime factor that is just 2 bits shorter than the order itself. Since $x^3+x = (x^2+1)x$ and x^2+1 is an irreducible polynomial

in $\mathbb{F}_p[x]$ (because of Euler's criterion which tells us that -1 is not a square in $\mathbb{F}_p$) we see that the cubic polynomial defining E has only one zero in $\mathbb{F}_p$ which means that there is only one point of order 2 (namely, the point $(0, 0)$) and hence that the group $E(\mathbb{F}_p)$ is isomorphic to $\mathbb{Z}_4 \times \mathbb{Z}_n$. Since 4 and n are relatively prime, this group is cyclic and the orders of its points are divisors of $4n$. Thus these orders are either divisors of 4 or multiples of the prime n by a divisor of 4. The points whose order is a divisor of 4 form a subgroup isomorphic to $\mathbb{Z}_4$ and hence there are only four of them (more specifically, there is a point of order 1, namely $\mathcal{O}$, a point of order 2, namely $(0, 0)$, and two points of order 4). Thus if p is large and we randomly choose an element of $E(\mathbb{F}_p)$, there is an overwhelming probability (equal to $\frac{n-1}{n}$) that the order of the resulting point is a multiple of n. If the order of this point is cn (where $c \in \{1, 2, 4\}$ is the *cofactor* of n) then, using Proposition 2.5, we obtain the point of order n we are looking for by multiplying the point of order cn by c. Note that the same reasoning applies to the curve of equation $y^2 = x^3 + ax$ where a is any quadratic residue in $\mathbb{F}_p$.

Example 11.22 Let us use Maple to construct a subgroup of large prime order of an elliptic curve group. Suppose that we want to find a subgroup of order a 256-bit prime with the idea of making the ECDLP in this group hard. Following the preceding discussion, we first choose a prime p of the form $p = 4n - 1$, where n is a 256-bit prime. To do this we can use the following Maple procedure that, on input a positive integer m, finds the smallest (probable) prime p of the form $p = 4n - 1$, where n is also (probable) prime and $m < n$.

```
> specialprime := proc(m)
  local n, p;
  n := m;
  p := 1;
  while not isprime(p) do
    n := nextprime(n);
    p := 4*n-1
  end do;
  p
  end proc:
```

Now, suppose we want n to be a 256-bit prime. Then we choose m a 256-bit integer (randomly or otherwise) and apply the preceding procedure to find p. For example, let us take $m = 2^{255}$ and proceed as follows:

```
> p := specialprime(2^255);
231584178474632390847141970017375815706539969331281128078915168015826259517507

> isprime(p);
                                  true

> n := (p+1)/4;
57896044618658097711785492504343953926634992332820282019728792003956564879377

> isprime(n);
                                  true

> ilog2(n)+1;
                                   256
```

We see that p and n are primes and that n is 256 bits long. The supersingular elliptic curve we want is the following:

```
> E258 := EllipticCurve(1, 0, p):
```

Let us now generate a pseudo-random point on this curve:

```
> RandomTools:-MersenneTwister:-SetState();
  R258 := PseudoRandomEllipticPoint(E258);
[690002811149448820703264628382299758401438862786190261016999980212659223794l9,
  6831553969084445545817246100236354529289534806106553748977214484854813193530 9]
```

We now compute the order of this point or, rather, we compute the quotient of its order divided by n; from the previous discussion we know that, with overwhelming probability, this quotient will be either 1, 2 or 4:

```
> EllipticPointOrder(R258, E258, p+1)/n;
                                    4
```

The result is 4, which means that the order is exactly $4n = p + 1$ and hence the point $R258$ is a generator of $E(\mathbb{F}_p)$. To obtain a point of order n we simply multiply $R258$ by 4:

```
> P258 := EllipticMult(4, R258, E258);
[173111335492990338254801713908264336074248773279792702929290254354011194404717,
168998047512458939398710531199066686924425997870916399851718955296173447826793]
```

Now we know that the order of the subgroup $\langle P258 \rangle$ is n, a 256-bit prime. Pohlig–Hellman is of no help since n is prime and, on the other hand, the size of n makes it infeasible to solve the ECDLP in this group by means of a collision attack such as Pollard's rho. We could be tempted to conclude that the ECDLP in this group of large prime order is hard but, as we explain below, *this is not the case*.

Exercise 11.22 Consider the supersingular curve of equation $y^2 = x^3 - x$ over $\mathbb{F}_p$, where $p = 4n - 1$ is the prime in Example 11.22. Show that its group of rational points is isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_n$ (and also to $\mathbb{Z}_2 \times \mathbb{Z}_{2n}$) and find a point of order n on the curve. Show that the same is true for the curve over $\mathbb{F}_p$ of equation $y^2 = x^3 + 2x$ and find a point of order n on this curve.

11.3.3.1 The MOV and FR Reductions

As we have mentioned, the fastest known algorithms to solve the ECDLP that work for all elliptic curves are the exponential collision algorithms such as the rho method. However, there are special classes of curves for which the ECDLP is easier to solve and one of them includes the supersingular curves such as the one considered in Example 11.22. The idea is to exploit some additional structure of the elliptic curve that allows a reduction of the problem to an easier DLP in a different group, such as the multiplicative group of a finite field where index calculus methods are applicable. Thus if $P \in E(\mathbb{F}_q)$ is a point of prime order n on an elliptic curve E defined over

$\mathbb{F}_q$ and if one has an efficiently computable isomorphism $\psi : \langle P \rangle \to G$ where G is another group, then the ECDLP instances in $\langle P \rangle$ can be efficiently reduced to instances of the DLP in G because, since the isomorphism preserves scalar multiplication, given another point $Q \in \langle P \rangle$ one simply has to compute:

$$\log_P Q = \log_{\psi(P)} \psi(Q).$$

The first attack of this kind arose in 1993, when Menezes, Okamoto and Vanstone presented what is now known as the MOV attack or MOV reduction. This attack applied originally to the class of supersingular curves but was later extended to apply to all elliptic curves of small embedding degree in the sense of the definition below.

Definition 11.4 Let $\mathbb{F}_q$ be a finite field, E an elliptic curve over $\mathbb{F}_q$, and n a divisor of $|E(\mathbb{F}_q)|$ such that $\gcd(n, q) = 1$. Then the *embedding degree* of E with respect to n is the smallest integer k such that $n | q^k - 1$.

Note that, since $\gcd(n, q) = 1$, $q \bmod n$ belongs to $\mathbb{Z}_n^*$ and so k is simply the order of this element in $\mathbb{Z}_n^*$, namely, the smallest positive integer such that $q^k \equiv 1 \pmod{n}$. Suppose now that n is the prime order of a point $P \in E(\mathbb{F}_q)$ and $\gcd(n, q) = 1$. Then the multiplicative group of the field $\mathbb{F}_{q^k}^*$ is cyclic of order $q^k - 1$ (Corollary 2.5) and, since n divides it, there is a unique subgroup $G \subseteq \mathbb{F}_{q^k}^*$ such that $|G| = n$ (Exercise 2.12). Then, under the additional assumption that $gcd(n, q - 1) = 1$, the MOV reduction constructs an isomorphism from $\langle P \rangle$ to G by using the *Weil pairing* (see Chap. 10 for the general definition of pairings). There is another similar attack, called the Frey–Rück attack or FR reduction, that also constructs such an isomorphism without requiring the additional constraint that $\gcd(n, q - 1) = 1$. The FR reduction is based on another pairing, namely the *Tate pairing* (also called the *Tate–Lichtenbaum pairing*).

It is interesting to point out that both pairings play a relevant role in elliptic curve cryptography, not only because of these attacks, which are often called 'destructive applications' of pairings but, more importantly, because of their constructive applications in other aspect of elliptic curve cryptography, namely, the construction of IBE schemes. The study of these pairings is beyond the scope of this book and we refer to the excellent presentations in [181, 197], and in particular to [181, XI.6] and [197, 5.3], for elliptic curve pairings and pairing-based attacks on the ECDLP. A more elementary introduction to pairings and their cryptographic applications is contained in [102] and more advanced presentations are contained in [26, 53], with the latter including also detailed information on pairing implementation.

A necessary condition for these attacks to be viable is that the embedding degree not be too high. The most optimistic estimates for the subexponential running time that can be expected for an index calculus method over $\mathbb{F}_{q^k}^*$ mean that this time becomes fully exponential for $k \geq \ln^2 q$, and Balasubramanian and Koblitz [8] proved that for most randomly generated elliptic curves we have an embedding degree $k > \ln^2 q$ (and that, in fact, for most of these curves the embedding degree is

proportional to the large prime cofactor n). Thus, for a randomly chosen elliptic curve over $\mathbb{F}_q$ the DLP over $\mathbb{F}^*_{q^k}$ is in fact much harder than the original ECDLP problem. However, there are some special classes of curves for which the embedding degree is low and the MOV reduction gives a subexponential-time algorithm. Some of these curves are given in the following example:

Example 11.23 Let E be a supersingular elliptic curve over $\mathbb{F}_p$ where $p > 3$ is prime. Then $|E(\mathbb{F}_p)| = p + 1$ and, since we are assuming that the prime n divides $|E(\mathbb{F}_p)|$, we have that $n|p + 1|p^2 - 1$, so that the embedding degree is $k = 2$. Thus the ECDLP in $\langle P \rangle$, where P is a point of prime order n, can be reduced to a DLP on $\mathbb{F}^*_{p^2}$, for which there are subexponential algorithms. This is the reason why the ECDLP considered in Example 11.22 above is not hard.

Some of the most important classes of elliptic curves with small embedding degree are:

- Supersingular curves.
- Elliptic curves of trace 2, i.e., such that $|E(\mathbb{F}_q)| = q - 1$.

All these curves have embedding degree ≤ 6 and hence they should not be used in cryptographic algorithms that rely on the hardness of the ECDLP unless the underlying finite field is large enough for the DLP in $\mathbb{F}^*_{q^k}$ to be hard. This fact discouraged the cryptographic use of supersingular elliptic curves for a while, at least until constructive applications of these curves were discovered with the arrival of IB cryptography.

To ensure that an elliptic curve defined over $\mathbb{F}_q$ is resistant to the MOV and FR reductions it suffices to check that the order n of the base point P relative to which the ECDLP is computed does not divide $q^k - 1$ for the small values of k for which the DLP in $\mathbb{F}^*_{q^k}$ is considered feasible. In [97] it is estimated that if $n > 2^{160}$ then it suffices to check this condition for all $k \in [1, 20]$ (so that the embedding degree is > 20) and in the more recent recommendation in [173], the embedding degree is required to be ≥ 100.

Example 11.24 Let us consider again the NIST elliptic curve P-256 introduced in Example 11.10. The prime modulus and the order of the curve are the following:

```
> p256 := 2^256-2^224+2^192+2^96-1:
  oP256 := 115792089210356248762697446949407573529996955224135760342422259061068512\
  2044369:
```

The order is prime, and hence any point on the curve distinct from the point at infinity is a generator and has order `oP256`. Let us check that the embedding degree of the curve is not ≤ 100:

```
> member(0, Power~(p256, [$1 .. 100]) mod oP256 -~1);
                                    false
```

Thus this curve is resistant to the MOV and FR reductions and seems, from this point of view, adequate for cryptographic use.

Exercise 11.23 Write a Maple program that searches, up to a specified bound, for the embedding degree of an elliptic curve over $\mathbb{F}_p$ with respect to a factor of $|E(\mathbb{F}_p)|$ and use it to show that the embedding degree of the curve P-256 in Example 11.24 with respect to its order is greater than one million.

11.3.3.2 Anomalous Elliptic Curves

The elliptic curves of small embedding degree are not the only ones that are vulnerable to a reduction attack. After the MOV reduction was known, it was suggested that curves of trace one, namely those curves—called *anomalous curves*—such that $|E(\mathbb{F}_q)| = q$, could be adequate for cryptographic use. In the case of curves defined over a prime field $\mathbb{F}_p$, the number of points is p and hence prime and, moreover, p does not divide $p^k - 1$ for any value of k, so that the group of rational points of the curve cannot be embedded in the multiplicative group of $\mathbb{F}_{p^k}$. Thus these curves are immune to the MOV and FR reductions. However, there exist even more efficient reductions for this class of curves, which provide a polynomial-time algorithm to solve the ECDLP in this case.

These reductions were independently found by Semaev, Satoh–Araki, and Smart, and they show that there is an efficiently computable isomorphism $\psi : E(\mathbb{F}_p) \to \mathbb{Z}_p$ where $\mathbb{Z}_p$ is, as usual, the additive group of integers modulo p. This reduces the ECDLP on an anomalous curve over a prime field to the DLP in $\mathbb{Z}_p$ which, as we have seen, is easy because it essentially reduces to the application of the Euclidean algorithm. As a consequence of this fact, anomalous elliptic curves should not be used for cryptographic purposes. This poses no problem because very few elliptic curves have this property and, on the other hand, it is easy to determine whether an elliptic curve over $\mathbb{F}_p$ is anomalous by computing $|E(\mathbb{F}_p)|$ and checking whether it is equal to p. We refer to [181, 197] for details on anomalous curves and their ECDLP reductions.

11.3.4 Final Remarks on the ECDLP

The preceding discussion suggests that in order to ensure that the ECDLP is hard, one should select an elliptic curve of prime order or whose order has a large prime factor and work in the corresponding (sub)group. Moreover, additional precautions should be taken such as, in particular, checking that the curve is not anomalous and that the embedding degree is sufficiently large. Besides these considerations, experience tells us that it is dangerous to use curves which have a rich structure in order to ease point counting—such as was the case with supersingular curves—or to make the arithmetic computations faster. Even if no special purpose attacks against the DLP are currently known, the existence of some additional structure moves us further away from the generic group model where we know that the ECDLP is hard and opens a door for

potential attacks that are as yet unknown. This remark has to be qualified, however, bearing in mind the fact that for some purposes such as IB cryptography, the additional structure in the form of efficient pairings is a prerequisite for which elliptic curves of low embedding degree are required and, in this context, even supersingular curves are useful in some cases. Of course, even when subexponential methods are available for the ECDLP, reasonable security can be attained by using appropriate key sizes.

The idea that additional structure on elliptic curves is dangerous leads us not only to discard those classes of curves for which an efficient reduction for the ECDLP is known but, in the view of many cryptographers, to altogether discard all curves built in some special way. For example, some curves are built by the *complex multiplication method* [181, 197] which makes it easier to count the number of points. Although no special weaknesses for these curves have been found so far, the conservative approach is not to use them. This leaves us with randomly selected curves which can also be checked for vulnerability to known reductions and eventually discarded. In practice such checks might not even be necessary because the probability that an attack such as the MOV or FR-reduction is applicable to a randomly chosen curve is negligible. An alternative to this approach is to use precomputed curves which have been approved by some trusted third party. For example, one might consider some of the 15 NIST recommended curves appearing in [75], one of which is the curve P-256 that we have used in several of our examples.

In view of the greater difficulty of the ECDLP in comparison with the DLP in the multiplicative groups of finite fields and also in comparison with the integer factorization problem, elliptic curve cryptography has the important advantage of requiring smaller key sizes than traditional cryptographic schemes, such as Elgamal or RSA, for a given security level. This is especially important when cryptographic schemes are implemented in computationally constrained environments such as, for example, smart cards. Even if this is not the case, the difference in parameter sizes makes elliptic curve cryptography much more efficient than, say, DL-based cryptography over the multiplicative groups of finite fields.

There have been various studies that compare the relative difficulty of the main problems used in cryptography to try to obtain estimates of the key sizes that should be used for the schemes based on these problems in order to attain a given security level. Several of these estimates, produced by different bodies and researchers, are collected in the web page [30] by Giry, which allows many comparisons between them. In the following table we summarize the estimated equivalent sizes corresponding to an ECDLP in an elliptic curve group with 256-bit order. The sizes are in bits and, for example, the sizes on the 'Asymmetric' column correspond to asymmetric keys and refer either to the size of an RSA modulus or to the size of a multiplicative group of a finite field where a DL-based scheme is implemented. The item 'Symmetric' corresponds to the size of a symmetric encryption scheme such as AES and is 128 bits in most cases, just one-half of the output size of a hash function with the same security level. In the first column, the name of the body or the person that produced the estimate is given, and the last column gives an estimation of the year until which the key and group sizes on the corresponding row are thought to offer sufficient security.

The most interesting comparison is perhaps between a given ECDLP size, such as the 256-bits considered in the table, and the size of the corresponding asymmetric keys which, as we can see, is estimated to be much larger as was to be expected from our previous discussion. We also see that, although there is a wide variation in the year estimates, 256-bit elliptic curve cryptography is thought to offer adequate security for the near future:

Table 11.1

Method	Elliptic curve	Asymmetric	Symmetric	Hash	Year
A. Lenstra	256	4440	128	256	2090
ECRYPT II	256	3248	128	256	2031–2040
NIST	256	3072	128	256	> 2030
SECG	256	3072	128	–	2040
FNISA	256	4096	128	256	> 2020
RFC3766	257	3707	136	–	–

In order to monitor the algorithmic advances against the ECDLP there is an ongoing challenge sponsored by Certicom, a company that produces ECC software. The Certicom ECC Challenge offers prizes ranging from \$20,000 to \$100,000 for the solution of several ECDL problems over fields of size varying from 131 bits[7] to 359 bits; see [47] for details.

11.4 Elliptic Curve Schemes

Standard elliptic curve cryptography—as opposed to IB cryptography—is defined over elliptic curve groups in a way similar to the standard DL-based public-key cryptography. In some specific cases there may be the added technical difficulty of having to represent a message as a point on an elliptic curve but this is generally not the case, as the points are often used only to 'mask' other values. For example, it is quite frequent that the elliptic curve group is used in a hybrid scheme where a symmetric encryption scheme is the one that actually encrypts the messages. For a simple method, due to Koblitz, of representing messages as points on an elliptic curve by embedding them into the x-coordinate, we refer to [197, p. 174].

11.4.1 ECC Schemes and Their Domain Parameters

One of the EC encryption schemes that requires the representation of messages as elliptic curve points is the plain version of Elgamal which was described on a generic cyclic group $G = \langle g \rangle$ in Sect. 8.5. The EC version of Elgamal is the particular case

[7] As of this writing, there is an ongoing effort, expected to be completed soon, to solve one of these 131-bit challenges using the rho method and massive parallelism, see [23] for the details.

in which G is an elliptic curve cyclic group generated by a point P and we leave as an exercise to formulate it explicitly.

Exercise 11.24 Describe the Elgamal elliptic encryption scheme.

Exercise 11.25 In a paper circulating on the Internet, the authors implement an EC-based encryption scheme similar to Elgamal but, in their implementation, the 'double-and-add' method is not used for scalar multiplication and, instead, they compute kP by adding P to itself $k-1$ times. Explain the reasons why such a scheme does not satisfy Definition 8.2 and, moreover, is completely insecure.

In the elliptic curve version of Elgamal, each ciphertext consists of a pair of points of an elliptic curve. Since the message string has to be previously embedded in the x-coordinate of a point, we see that this scheme has a 4-to-1 message expansion, which is an undesirable feature. There is a simple procedure that allows the representation of a point on an elliptic curve over $\mathbb{F}_q$ by using only $\text{len}(q)+1$ bits instead of the usual $2\text{len}(q)$ bits required by using $\text{len}(q)$ bits for each coordinate. If the Weierstrass equation of the curve is $y^2 = x^3 + ax + b$, then the y-coordinate corresponding to an x-coordinate $x \in \mathbb{F}_q$ is one of the two square roots of $x^3 + ax + b$ in $\mathbb{F}_q$ and hence the x-coordinate determines the y coordinate up to the sign. Thus instead of transmitting a point (x, y), it is sufficient to transmit x together with one additional bit that determines which of the two square roots corresponds to the point. With this technique, known as *point compression*, each point is represented by $\text{len}(q)+1$ bits and hence the message expansion of plain Elgamal is reduced from 4-to-1 to 2-to-1.

An alternative to using point compression is to transmit only the x-coordinate of a point P, without any additional information. The receiver would not know whether the point P or the point $-P$ is intended but in the majority of schemes the only elliptic group operation involved at decryption is scalar multiplication. Since $k(-P) = -kP$, the x-coordinates of kP and $k(-P)$ are the same, and if one uses the x-coordinate alone the scheme will work exactly the same.

Another cryptographic protocol that we defined in a generic cyclic group $G = \langle g \rangle$ is the Diffie–Hellman Key Agreement that is discussed in Sect. 7.3. The definition applies in particular to the elliptic curve setting, where the roles of G and g are represented by an elliptic curve $E(\mathbb{F}_q)$ and a point $P \in E(\mathbb{F}_q)$ of large prime order. The DH problem and its variants have corresponding elliptic curve versions to which the same remarks apply so that they are generally believed to be difficult.

Exercise 11.26 Give an explicit formulation of the elliptic curve version of the Diffie–Hellman protocol.

Hardness of the ECDLP is a necessary condition for the security of ECC schemes such as Elgamal or Diffie–Hellman but, as we have seen when we described these schemes over arbitrary cyclic groups, their security is also related to the DH problem and its variants. These problems may be specialized to the EC setting and we will leave the specific details to the reader.

An EC scheme makes use of a set of *domain parameters* which specify, at least, an elliptic curve E defined over a finite field $\mathbb{F}_q$, a base point $G \in E(\mathbb{F}_q)$ and its order. We are going to consider the case in which $q = p$ is prime and to describe the domain parameters over the prime field $\mathbb{F}_p$ according to the specification in the SEC 1 standard by the Standards for Efficient Cryptography Group (SECG) [173].

11.4.1.1 Domain Parameters

EC domain parameters over $\mathbb{F}_p$ are a sextuple:

$$T = (p, a, b, G, n, h)$$

consisting of the following items:

- A prime p that specifies the prime field $\mathbb{F}_p$.
- Two elements $a, b \in \mathbb{F}_p$ that define the elliptic curve $E(\mathbb{F}_p)$ with equation $y^2 = x^3 + ax + b$.
- A base point $G = (x_G, y_G) \in E(\mathbb{F}_p)$, given by its affine coordinates x_G, y_G.
- A prime n which is the order of G in the group $E(\mathbb{F}_p)$.
- The cofactor $h = |E(\mathbb{F}_p)|/n$.

These parameters are chosen so that they satisfy the constraints we have mentioned in order to make the ECDLP hard in the subgroup $\langle G \rangle$ of $E(\mathbb{F}_p)$. Thus the curve should not be anomalous and it is recommended that the embedding degree should be ≥ 100 to resist the MOV and FR attacks. In [173] the prime p is selected such that, for a given security level in bits $t \in \{80, 112, 128, 192, 256\}$ (which approximately corresponds to the key length of a symmetric scheme of comparable strength as in Table 11.1):

$$\lceil \log_2 p \rceil = \begin{cases} 192 \text{ if } t = 80, \\ 2t \quad \text{if } 80 < t < 256, \\ 521 \text{ if } t = 256. \end{cases}$$

In the same reference the cofactor h is required to be $\leq 2^{t/8}$ and, in practice, it is often $h = 1$ which means that $E(\mathbb{F}_p)$ has prime order n. In [173] there is the additional requirement that $n - 1$ and $n + 1$ should have a large prime factor in order to prevent some recent exponential attacks that, anyway, are very unlikely to be feasible in practice.

11.4.1.2 Domain Parameter Generation

There are several methods to generate elliptic curve domain parameters of a given strength. The first step is to generate the finite field over which the curve is going to be defined and then the curve itself must be generated. There are methods (based on 'complex multiplication') that allow one to specify the order first and then to generate

a curve with this order and there is also the possibility of choosing the curve from some special family for which the order is easily computable. The preferred method, however, is to generate the curve at random and then to count the number of points using the SEA algorithm and to repeat this process until a curve is found for which this number is 'nearly prime', i.e., the product of a prime by a small cofactor. Since the probability that a random curve is vulnerable to one of the known reduction attacks is negligible, these attacks are also prevented, although the curve can additionally be checked to ensure that this is indeed the case. Moreover, generating the curve at random also offers a probabilistic protection against future special purpose attacks and hence is the most conservative choice from a security point of view.

A variation on the random method of generating elliptic curves is to select them in such a way that the parameters are *verifiable*, which is accomplished by choosing the coefficients of the curve from the outputs of a one-way function—SHA-1 is often used in practice for this purpose—by means of some specified algorithm. This guarantees that these parameters cannot be predetermined, which is verifiable with the help of the input seed of the function used and provides evidence that the elliptic curve was not intentionally constructed with some hidden weaknesses that could subsequently be used to break the cryptographic schemes. Algorithms to generate and to validate these parameters are given in [97, 173].

11.4.2 The Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is the elliptic curve variant of DSA (see 9.4.1) and is perhaps the most-used EC scheme, having been included in several standards. We next give a description of this scheme.

11.4.2.1 Definition of ECDSA

We will assume that EC domain parameters (p, a, b, G, n, h), corresponding to an elliptic curve E over a prime field $\mathbb{F}_p$, are given. In ECC the domain parameters are often selected from the choices available in standards like [75] or [174]. This has the advantage that, besides making interoperability easier, it provides parameters that meet the security requirements established by the designers of the standard. In addition, this frees the user from having to implement the domain parameter generation algorithm which may include point-counting algorithms and hence be relatively costly. For simplicity, we will denote the x-coordinate of a point $P \in E(\mathbb{F}_p)$ by $x(P)$.

In addition to the EC domain parameters, ECDSA also uses, like DSA, a hash function H such as one of the NIST-approved functions specified in [74], namely, either SHA-1 or one of the SHA-2 variants, SHA-224, SHA-256, SHA-384 or SHA-

512. The hash function should have a security strength at least equal to that provided by the ECDLP on the EC subgroup of order n so that, for example, if n is a 256-bit prime (see Table 11.1) then we would use a hash function with a $\geq$ 256-bit output, such as SHA-256 (in our implementation below we will actually use the same length, as we did in the case of DSA). Given these parameters, the ECDSA algorithm based on an elliptic curve defined over a prime field may be described as follows:

Definition 11.5 *The Elliptic Curve Digital Signature Algorithm* is defined as the signature scheme:

$$\text{ECDSA} = (\textbf{Gen}_{\text{ECDSA}}, \textbf{Sign}_{\text{ECDSA}}, \textbf{Ver}_{\text{ECDSA}})$$

with the following algorithms:

- $\textbf{Gen}_{\text{ECDSA}}$.
  Input: The EC domain parameters (p, a, b, G, n, h).
  Output: (d, Q), with $d \in [1, n-1]$ the private key and $Q \in E(\mathbb{F}_p)$ the public key.

  1. `Pick` a (pseudo-) random $d \leftarrow [1, n-1]$.
  2. `Compute` $Q := dG \in E(\mathbb{F}_p)$.
  3. `Output` (d, Q).

- $\textbf{Sign}_{\text{ECDSA}}$.
  Input: The EC domain parameters, the hash function H, a private key $d \in [1, n-1]$ and a message $m \in \{0, 1\}^*$.
  Output: A signature (r, s) of the message m.

  1. `Choose` $k \leftarrow [1, n-1]$.
  2. `Compute` $R := kG \in E(\mathbb{F}_p)$.
  3. $r := x(R) \bmod n$; `if` $r = 0$ `then go to` step 1.
  4. `Compute` $e := H(m)$ (as in 9.4.1 we assume that H maps $\{0, 1\}^*$ to $\mathbb{Z}_n$).
  5. `Compute` $s := k^{-1}(e + dr) \bmod n$; `if` $s = 0$ `then go to` step 1.
  6. `Output` (r, s).

- $\textbf{Ver}_{\text{ECDSA}}$.
  Input: The EC domain parameters, the hash function H, a public key $Q \in E(\mathbb{F}_p)$, a message m, and a signature (r, s).
  Output: A bit $b \in \{0, 1\}$. If $b = 1$ the signature is *valid*, otherwise it is *invalid*.

  1. `Check` that $0 < r < n$ and $0 < s < n$. `If` either of these conditions does not hold `then output` 0, otherwise `compute`:
  2. $e := H(m)$.
  3. $w := s^{-1} \bmod n$.
  4. $u_1 := ew \bmod n$.
  5. $u_2 := rw \bmod n$.
  6. $R := u_1 G + u_2 Q \in E(\mathbb{F}_p)$.
  7. `If` $R \neq \mathcal{O}$ and $x(R) \bmod n = r$ `then output` 1, otherwise `output` 0.

Remarks 11.1

1. Observe first that a correctly generated signature is declared valid by the verification algorithm (with valid corresponding to output 1). Indeed, in this case we have that $u_1G + u_2Q = (u_1G + u_2dG) = (u_1 + u_2d)G = kG$ because $u_1 + u_2d \equiv s^{-1}(e + dr) \equiv k \pmod{n}$ and hence $x(u_1G + u_2Q) \bmod n = x(kG) \bmod n = r$, which is the condition that must be verified for the signature to be declared valid.
2. If the signature (r, s) is valid for the message m then $(r, -s \bmod n)$ is also a valid signature for m. However, this is not regarded as a forgery because the same message is signed in both cases.
3. The cofactor h in the domain parameters was not explicitly used in the ECDSA algorithms. However, this parameter is required in several standards such as, for example, [75]. The reason is that maximum sizes for h (relative to the size of n) are specified in the standard and these sizes should be taken into account either when generating the domain parameters or when validating them.
4. We have defined ECDSA for prime fields but we mention that there is also a version of the scheme that works on curves defined over binary fields $\mathbb{F}_{2^m}$.
5. Observe that the signing algorithm does not make use of the y-coordinate of the point R computed therein. Thus in ECDSA it is not necessary to use point compression as no elliptic curve points—only the x-coordinate of a point and a value deduced from it—are included in the signature.

11.4.2.2 The Security of ECDSA

The ECDSA private key can be recovered from the corresponding public key by solving a DL problem, so hardness of this problem in the subgroup of order n generated by G is an obvious necessary condition for the scheme to be secure. Another necessary condition is that the "ephemeral key" k generated by the signing algorithm be truly random. As we have seen when discussing Elgamal signatures, an extreme case in which this condition is violated (except with negligible probability) is when the same value of k is used to sign two different messages m, m_1. Then the corresponding signatures (r, s) and (r_1, s_1) satisfy:

1. $r = r_1 = x(kG) \bmod n$,
2. $s := k^{-1}(H(m) + dr) \bmod n$,
3. $s_1 := k^{-1}(H(m_1) + dr) \bmod n$,

from which, working in the field $\mathbb{F}_n = \mathbb{Z}_n$, we can eliminate k and obtain $s_1(H(m) + dr) \equiv s(H(m_1) + dr) \pmod{n}$ which, in turn, gives:

$$d = (s_1H(m) - sH(m_1))r^{-1}(s - s_1)^{-1} \bmod n,$$

and recovers the private key. We also see that if an adversary is able to guess k then it can obtain the private key, thus the random number generator used to produce k should be unpredictable. A failure to use a random value for the ephemeral key in the

ECDSA implementation in Sony's PS3 console allowed a key-recovery attack that was presented by a group named "Fail Overflow" at the 27th Chaos Communication Congress held in Berlin in 2010.

Other conditions that are necessary for the security of ECDSA refer to the hash function H that should be one-way and collision resistant. If H is not one-way, then an adversary $\mathcal{A}$ may forge a signature as follows:

- $\mathcal{A}$ picks an integer $u \leftarrow [1, n-1]$.
- $\mathcal{A}$ sets $r := x(uG + Q) \bmod n$ and $s := r$.
- $\mathcal{A}$ sets $e := ur \bmod n$ and finds a message m such that $H(m) = e$.

Then (r, r) is a valid signature for m because in the verification algorithm we have that $u_1 = er^{-1} \bmod n = u$ and $u_2 = rr^{-1} \bmod n = 1$. Thus $R = uG + Q$ and $x(R) \bmod n = x(uG + Q) \bmod n = r$, so that verification succeeds.

On the other hand, if the hash function H is not collision resistant, then it is trivial to construct, in a chosen message attack, a signature forgery from a collision between two messages m, m_1 exactly in the same way we have seen for hashed RSA signatures in 9.3.1, since a signature valid for m is also valid for all messages with the same hash value as m (see also the collision attack of Example 5.12).

When all these necessary conditions are satisfied and, moreover, the elliptic curve group is modeled as a generic group, then it was proved by Brown (see [42]) that ECDSA is unforgeable under a chosen message attack. We see that this security reduction assumes the ideal condition that the group is generic and, although this is not satisfied in practice, the proof gives a good indication that reinforces the reputation of ECDSA as a secure scheme, which was already high because no substantial weaknesses have been found. Note that the generic group hypothesis is not applicable in the case of DSA because in $\mathbb{Z}_p^*$ there are subexponential index calculus algorithms for the DLP.

11.4.3 ECDSA in Maple

Here we give a Maple implementation of ECDSA over a prime field $\mathbb{F}_p$. We will use precomputed domain parameters which can be taken, for example, from [75] or [174] and for implementing the algorithms we will mainly follow [173] with some simplifications. The hash function used will be SHA-256 as implemented in 5.6.3.

We will start with a function to test the domain parameters. The parameters contained in [75, 174] often include a seed that can be used to verify that the parameters have been generated from it, but we shall not include this seed in our parameter set and we will just check that they define an elliptic curve over a prime field and a base point of prime order on this curve and, moreover, that the embedding degree is ≥ 100 as required in [173]. The cofactor h will always be equal to 1 in the examples we will use.

The next function tests the domain parameters and, as usual, makes use of several previously defined functions. The input is a list $[p, a, b, [x_G, y_G], n]$ whose elements

may be in either decimal or hexadecimal format and that may contain, in addition, the cofactor h at the end (but h will not be used by the function). The function checks that the following conditions are satisfied: (i) p is a (probable) prime greater than 3, (ii) a, b and p define an elliptic curve $E(\mathbb{F}_p)$, (iii) $[x_G, y_G] \in E(\mathbb{F}_p)$, (iv) n is (probably) prime, (v) n is the order of $G = [x_G, y_G]$ in the group $E(\mathbb{F}_p)$, and (vi) the embedding degree is ≥ 100. The output is either an error message in case any of these conditions does not hold or a message informing that the parameters are valid.

```
> ECDomainTest := proc(domain::list)
  local dom, p, a, b, G, n, E;
  dom := stringposint~(domain);
  p := dom[1];
  a := dom[2];
  b := dom[3];
  G := dom[4];
  n := dom[5];
  if p < 5 or not isprime(p) then
    error "%1 must be a prime >3",p
  end if;
  E := EllipticCurve(a, b, p);
  if not IsEllipticPoint(G, E) then
    error "%1 is not a point on the curve",G
  end if;
  if not isprime(n) then
    error "%1 is not prime",n
  end if;
  if EllipticMult(n, G, E) <> 0 then
    error "%1 does not have order %2",G,n
  end if;
  if member(0, Power~(p, [$1 .. 99]) mod n -~ 1) then
    error "embedding degree less than 100"
  end if;
  printf("Valid parameters")
  end proc:
```

Example 11.25 Let us consider the EC domain parameters contained in [75, D.1.2.3], where an elliptic curve, called P-256, is defined over a prime field, as well as a base point whose order is a 256-bit prime. This curve was already used in several of our Examples, including 11.10 and 11.24. The parameters are the following (note that some of them are in decimal and others in hexadecimal format but our functions will accept both):

```
> p256 := 2^256-2^224+2^192+2^96-1:
  a := -3:
  b := "5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b":
  G := ["6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296",
  "4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5"]:
  oP256 := 115792089210356248762697446949407573529996955224135760342422259061068512044369:
```

The notation used differs slightly from the generic notation for EC domain parameters but we use it because it was already used in previous examples. Specifically, $p256$ is the parameter generically denoted p, and $oP256$ is the parameter generically denoted n. The full set of EC domain parameters corresponding to these values is then:

```
> dp256 := [p256, a, b, G, oP256, 1]:
```

We may test these parameters with the `ECDomainTest` function:

```
> ECDomainTest(dp256);
Valid parameters
```

We next give the key generation function which generates an "EC key pair" formed by an integer $d \in [1, n-1]$ and a point $Q \in E(\mathbb{F}_p)$ such that $Q = dG$. As on other occasions, we will use the Blum–Blum–Shub PRG to generate d and, to this purpose, the PRG should be supplied with a random seed. Thus the input parameters of the function are `domain`, for the list of EC domain parameters, and `seed`, for the seed. The optional parameters are `bbslength` to set the length of the primes used by Blum–Blum–Shub and `format` to specify whether the output will be decimal or hexadecimal (with the latter as default). The output is a list $[d, Q]$ containing the integer d and the point Q.

```
> ECKeyGen := proc(domain::list, seed::{posint, string},
  {bbslength::{512, 768, 1024} := 1024, format::identical(decimal, hex) := hex})
  local dom, p, a, b, G, n, E, sd, B, l, c, d, Q;
  dom := stringposint~~(domain);
  p := dom[1];
  a := dom[2];
  b := dom[3];
  G := dom[4];
  n := dom[5];
  E := EllipticCurve(a, b, p);
  l := intlog[2](n)+1;
  sd := stringposint(seed);
  B := RandomTools:-BlumBlumShub:-NewBitGenerator(sd, primes = bbslength);
  c := n-1;
  while n-2 < c do
    c := convert(cat(seq(B(), i = 1 .. l)), decimal, binary)
  end do;
  d := c+1;
  Q := EllipticMult(d, G, E);
  if format = decimal then
    [d, Q]
  else
    StringTools:-LowerCase~~(convert~~([d, Q], hex))
  end if
  end proc:
```

The next function tests EC key pairs. The input parameters are `domain` for the EC domain parameters and `key` for the key pair. The function checks whether the point Q in the key pair $[d, Q]$ belongs to the curve and whether $dG = Q$, for the value of G in the domain parameters. The output is either "Valid key" or "Invalid key".

```
> ECKeyTest := proc(domain::list, key::list)
  local k, dom, E;
  dom := stringposint~~(domain);
  k := stringposint~~(key);
  E := EllipticCurve(dom[2], dom[3], dom[1]);
  if not IsEllipticPoint(k[2], E) then
    error "%1 is not a point on the curve", k[2]
  end if
  if evalb(EllipticMult(k[1], dom[4], E) = k[2]) then
    printf("Valid key")
  else
    printf("Invalid key")
  end if
  end proc:
```

Example 11.26 We generate an EC key pair for the domain parameters in Example 11.25. We need a random seed for the BBS generator and we supply the function with an externally generated one which is the following:

```
> kgseed := "663434a0148ccea7ff7ef7cd8b53866c":
```

Next, we generate the key pair:

```
> eckey := ECKeyGen(dp256, kgseed);
   ["75902d710db8b5b08cf7e27326dc86ed779298a33081bf1a7171193f77af9266",
     ["c2ed2f06cd6e1648c2eff6e12208b4effcb791ccf51908ab226ad2e731af798d",
     "fca203d638f4afeefbf260f25cb4c4e4b8c0d3b415d3ef5cfbfcf892dea65a58"]]
```

We check that the key is valid for the domain parameters:

```
> ECKeyTest(dp256, eckey);
Valid key
```

Next we give the ECDSA signing function. The required parameters are `domain, privatekey, message and seed`, where the last serves to specify a random seed to be used by the BBS generator to generate the "ephemeral key pair" (k, R) required by the signing algorithm. This ephemeral key pair is generated by calling the previous function `ECKeyGen`. The optional keyword parameters are `messagetype`, used to specify whether the message is a hexadecimal string or an ASCII-8 string (in which case, `messagetype = text` should be set), and `H` for the name of the hash function used which, in our case, will be `SHA256` by default. The output is the signature $[r, s]$, where r and s are hexadecimal byte strings.

```
> ECDSASign := proc(domain::list, privatekey::{posint,string}, message::string,
   seed::{posint,string}, {messagetype::identical(hex,text):=hex, H::name:='SHA256'})
  local dom, d, p, a, b, G, n, E, r, s, sec, k, R, e;
  dom := stringposint~(domain);
  d := stringposint(privatekey);
  p := dom[1]; a := dom[2]; b := dom[3]; G := dom[4]; n := dom[5];
  E := EllipticCurve(a, b, p);
  r := 0;
  while r = 0 or s = 0 do
    sec := stringposint~(ECKeyGen(domain, seed));
    k := sec[1];
    R := sec[2];
    r := R[1] mod n;
    e := OS2IP(H(message, messagetype));
    s := ((e+d*r)/k) mod n
  end do;
    I2OSP~([r, s])
  end proc:
```

Remarks 11.2

1. If the default value `SHA256` is used for the hash function, the length of the prime p defining the field should not exceed 256 bits (for longer primes SHA-384 or SHA-512 should be used). If, on the other hand, a prime n fewer than 256 bits long is used together with SHA-256 then the function should be modified in order to make the value of e equal to the leftmost len(n) bits of the output of applying the hash function to the message. This procedure should be used, more generally, whenever the length of n is less than the output length of the hash function used.

2. For the security reasons already mentioned in relation with the ephemeral key, it is important that the function `DSASign` uses a freshly generated random seed each time it is used.

We next give the function that implements the ECDSA verification algorithm. The input parameters are the same as in `ECDSASign` except that `privatekey` is replaced by `publickey`, `seed` is no longer required, and there is a new parameter `signature`, used to specify the signature pair. The output of the function is either `"Valid"`, if the signature is accepted, or `"Invalid"` otherwise.

```
> ECDSAVer := proc(domain::list, publickey::list, message::string, signature::list,
                   {messagetype::identical(hex, text) := hex, H::name := 'SHA256'})
  local dom, Q, sig, r, s, p, a, b, G, n, E, w, e, u1, u2, R;
  dom := stringposint~~(domain);
  Q := stringposint~(publickey);
  sig := stringposint~(signature);
  r := sig[1]; s := sig[2];
  p := dom[1]; a := dom[2]; b := dom[3]; G := dom[4]; n := dom[5];
  E := EllipticCurve(a, b, p);
  if r = 0 or s = 0 or n <= r or n <= s then
    error "invalid signature"
  end if;
  w := s^(-1) mod n;
  e := OS2IP(H(message, messagetype));
  u1 := e*w mod n;
  u2 := r*w mod n;
  R := EllipticAdd(EllipticMult(u1, G, E), EllipticMult(u2, Q, E), E);
  if R <> 0 and evalb(R[1] mod n = r) then
    "Valid"
  else
    "Invalid"
  end if
  end proc:
```

Exercise 11.27 Modify the functions `ECDSASign` and `ECDSAVer` so that they are able to accept values of n whose length is less than the output length of the hash function used (see the remarks after `ECDSASign`).

Exercise 11.28 Write a Maple function that tests the functions `ECDSASign` and `ECDSAVer` by generating pseudo-random strings and, given valid domain parameters, compute signatures for these messages and verify them.

Example 11.27 We use ECDSA to sign and verify a couple of messages in different formats. Consider the messages:

```
> m1 := "This is a message to test ECDSA signatures":
  m2 := "0123456789abcdefedcba98765432101":
```

We are going to use the EC domain parameters `dp256` of Example 11.25 and the key pair `eckey` generated in Example 11.26. To sign each message we need a randomly generated seed and we will use the following:

```
> snseed1 := "c34070a96e17de75ad3dd5a6d7945996":
  snseed2 := "36a56ab2e6eedf6a0781863af96fff7c":
```

The signatures of these messages are the following (note that in the case of the ASCII message `m1` we have to specify `messagetype=text`):

```
> s1 := ECDSASign(dp256, eckey[1], m1, snseed1, messagetype = text);
    ["1c9852d60e45b864ab2227e66447caf678202c5ce1a13135c2423120b9edc870",
      "293f8bd87436edfff89abf5dd72e27bceebd777439b5ab677bfd2fa5a7d38538"]

> s2 := ECDSASign(dp256, eckey[1], m2, snseed2);
    ["510b4b98b7fad7b33f5ca5ea82cf2c2fdfc1ba90329966cea3a0837fe82427af",
      "aed245c193e0448c053e056498a50de461a3ae19bf7a1b93fe3151e69c980f6d"]
```

The verification of the signatures is then as follows:

```
> ECDSAVer(dp256, eckey[2], m1, s1, messagetype = text);
                                  "Valid"

> ECDSAVer(dp256, eckey[2], m2, s2);
                                  "Valid"
```

11.4.4 Elliptic Curve Encryption

The only EC encryption scheme we have mentioned so far is Elgamal but, as is shown in Sect. 8.5, this scheme is malleable and hence is not CCA secure. We now present a more secure encryption scheme which is essentially a hashed version of Elgamal that also uses a MAC for added protection against active attacks. It is called "Elliptic Curve Integrated Encryption Scheme" (ECIES) and was introduced in [1] and subsequently included in several standards.

ECIES is a hybrid encryption scheme in which a Diffie–Hellman protocol is used to derive two symmetric keys k_1, k_2. The first of them is used to encrypt the plaintext with a symmetric encryption scheme and the second is used to authenticate the resulting ciphertext with a MAC. Thus, in addition to the EC domain parameters (p, a, b, G, n, h), the following cryptographic primitives are also required:

- An encryption function E_k and a decryption function D_k corresponding to a private-key encryption scheme such as AES.
- A *key derivation function*, KDF, which is built from a hash function and, on input a pair of points on the elliptic curve (and, possibly, a desired length l), outputs a bit string. KDF is usually defined as a concatenation of hash values similar to that used in the function `MGF1` defined in 8.3.7. Actually, a variant of `MGF1` can be used for this purpose.
- A message authentication code, generically denoted MAC (for example, HMAC can be used).

Given these parameters, the ECIES encryption scheme for elliptic curves over a prime field may be defined as follows (the definition for other fields is similar):

Definition 11.6 The *Elliptic Curve Integrated Encryption Scheme* is the public-key encryption scheme:

$$\text{ECIES} = (\mathbf{Gen}_{\text{ECIES}}, \mathbf{Enc}_{\text{ECIES}}, \mathbf{Dec}_{\text{ECIES}}),$$

with the following algorithms:

- **Gen**$_{\text{ECIES}}$.
  Input: The EC domain parameters (p, a, b, G, n, h).
  Output: (d, Q), where $d \in [1, n-1]$ is the private key and $Q \in E(\mathbb{F}_p)$ is the public key.

  1. `Pick` a (pseudo-) random $d \leftarrow [1, n-1]$.
  2. `Compute` $Q := dG \in E(\mathbb{F}_p)$.
  3. `Output` (d, Q).

- **Enc**$_{\text{ECIES}}$.
  Input: The EC domain parameters, the key derivation function KDF, the message authentication code MAC, the symmetric encryption and decryption functions, a public key $Q \in E(\mathbb{F}_p)$, and a message $m \in \{0, 1\}^*$.
  Output: A ciphertext (R, c, t).

  1. `Choose` $k \leftarrow [1, n-1]$.
  2. `Compute` $R := kG \in E(\mathbb{F}_p)$ and $S := hkQ \in E(\mathbb{F}_p)$. `If` $S = \mathcal{O}$ `then go to` Step 1.
  3. `Set` $k_1||k_2 := \text{KDF}(R, S)$ (where k_1 and k_2 have some specified lengths).
  4. `Compute` $c := E_{k_1}(m)$ and $t := \text{MAC}_{k_2}(c)$.
  5. `Output` (R, c, t).

- **Dec**$_{\text{ECIES}}$.
  Input: The EC domain parameters, KDF, MAC, the symmetric encryption and decryption functions, a private key $d \in [1, n-1]$, and a ciphertext (R, c, t).
  Output: The message m or an "Invalid ciphertext" message.

  1. $S := hdR$; `if` $S = \mathcal{O}$ `then output` "Invalid ciphertext".
  2. $k_1||k_2 := \text{KDF}(R, S)$.
  3. $t' := \text{MAC}_{k_2}(c)$; `if` $t' \neq t$ `then output` "Invalid ciphertext".
  4. $m := D_{k_1}(c)$.
  5. `Output` m.

Remarks 11.3

1. If (R, c, t) is an encryption of the message m, then when decrypting we obtain $S = hdR = hd(kG) = hk(dG) = hkQ$. Thus the value of S computed by the decryption algorithm is the same as that computed at encryption and hence the computed values of k_1 and k_2 are also the same. Therefore, the output of the decryption algorithm is $D_{k_1}(c) = D_{k_1}(E_{k_1}(m)) = m$ and the message m is recovered.
2. ECIES is similar to a one-pass Diffie–Hellman key agreement in which one of the parties uses a long-term key (the private key d) instead of an ephemeral one. From dG and k, a shared secret dkG is computed as in the DH protocol, and used to obtain a "shared secret point" $S = hkdG$ which, in turn, is used to derive the keys k_1 and k_2.

3. There are several variants regarding the inputs of the KDF function. In the simplest of them, KDF is applied only to the x-coordinate of the point S (together with the output length l if this length is not fixed). This is efficient but has the disadvantage of leading to a phenomenon called *benign malleability*, which means that an adversary is able to construct, given a ciphertext (R, c, t) corresponding to the message m, another valid ciphertext for m. In this case $(-R, c, t)$ is also valid for m because, at decryption, the opposite point $-S = hd(-R)$ is obtained instead of S. Since we are assuming that the output of KDF depends only on the x-coordinate of S, and $x(S) = x(-S)$, the same values of k_1 and k_2 are obtained and m is recovered. Thus the scheme is malleable and since, as we have seen, CCA security implies non-malleability, this version of the scheme cannot be CCA secure: more directly, an adversary may ask for a decryption of $(-R, c, t)$ to obtain a decryption of (R, c, t). The term "benign" refers to the fact that the definition of CCA security can be replaced by another one which admits this kind of malleability and yet it is also useful because there are no known real-world attacks that are able to exploit benign malleability. This kind of malleability may be prevented by requiring that the KDF function takes as input the point S instead of its x-coordinate. This way, if R is replaced by $-R$, then the decryption algorithm replaces S by $-S$ and the values of k_1 and k_2 which are obtained are different. Another possibility is to use, in addition to the x-coordinate $x(S)$, the point R as input for KDF. It is clear that this suffices to prevent benign malleability. Of course, this is also the case when the two points, R and S, are used as inputs for KDF as, for simplicity, we assumed in the above description of the algorithm.
4. As indicated in Sect. 5.4, there is a generic construction that uses a MAC to produce a CCA secure encryption scheme from a CPA secure scheme. The idea is that, since the ciphertext is authenticated by a CMA secure MAC, an adversary will be unable to generate a valid ciphertext to query the decryption oracle unless the ciphertext has been obtained by querying the encryption oracle first—which is useless because in that case the adversary already knows the plaintext. In ECIES, the MAC is used as in this construction and with the same purpose.

We will leave writing a Maple implementation of ECIES as an exercise for the industrious reader:

Exercise 11.29 Write an implementation of the ECIES algorithms using the EC domain parameters of Example 11.25 (or some similar ones), the AES encryption and decryption functions of Sect. 4.3, the SHA-256-based `HMAC` function of Sect. 5.6 and the `MGF1` function of 8.3.7 used as KDF.

11.4.4.1 The Security of ECIES

The previous remarks contain some information about security aspects of ECIES but do not tell us whether the scheme is secure or not. In [1] it was proved that ECIES is CCA secure assuming that a non-standard variant of the Diffie–Hellman problem

involving the KDF (called the *hash Diffie–Hellman problem*) is hard and that the private-key encryption scheme and the MAC are also secure. There is another security reduction that relies on a more standard variant of the Diffie–Hellman problem, namely, the *gap Diffie–Hellman problem* which essentially is the problem of solving the CDH problem given access to an oracle that solves the DDH problem. This proof, however, requires another additional assumption, namely, that the KDF function is a random oracle, and hence is a proof in the random oracle model. Yet another security reduction for ECIES, due to Smart, trades off the assumptions on the KDF function for the assumption that the elliptic group is modeled as a generic group: it is, therefore, a proof in the 'generic group model'. A detailed discussion of these proofs, written by A.W. Dent, can be found in Chap. III of [26].

11.5 Final Remarks on Elliptic Curve Cryptography

In this chapter we have given a brief account of the basics of ECC but there are many other aspects that fall outside the scope of this book. We have limited our specific treatment to elliptic curves over prime fields of characteristic > 3 but ECC may also be developed over other fields and, in particular, over binary fields of the form $\mathbb{F}_{2^m}$, where m is a large positive integer. Not only is ECC being continuously developed but also new cryptographically interesting discoveries on the mathematical theory of elliptic curves are being made. A relevant example is the introduction by H.M. Edwards, in 2007, of *Edwards curves* or *Edwards coordinates*. Edwards introduced a new normal form for elliptic curves which is different from the Weierstrass equation and has interesting cryptographic applications because it allows faster point addition and scalar multiplication. A short introduction to Edwards coordinates is given in [97, 2.6.3] and an account of some recent work on this subject by Bernstein and Lange is given in the web page [24].

We have seen that one of the main advantages of elliptic curve cryptography is that it requires smaller key sizes, which can be exploited either to obtain better performance for a given security level or to obtain better security for a given key size. There is also another important, more qualitative reason that we have only mentioned in passing and gives ECC a special status because it allows things that are very difficult to achieve in other cryptographic settings: the existence of bilinear pairings on certain families of elliptic curves that allow the full development of identity-based encryption.

Although some IBE schemes exist that do not rely on elliptic curves—such as the Cocks IBE scheme described in Chap. 10—it is really in the context of pairing-based cryptography—which, for now, is an exclusive feature of ECC—where these schemes fully realize their enormous potential. Pairing-based IBE schemes, such as the Boneh–Franklin IBE of Chap. 10, rely on the Weil and Tate pairings (and their variants) on elliptic curves and on the fact that, for elliptic curves of low embedding degree, these pairings can be efficiently computed. We refer to [26, 53, 81] for in-depth treatments of pairing-based cryptography, as well as to [107], which is a reference book for identity-based cryptography.

Appendix A
Some Maple Conversion Functions

Here we give some auxiliary Maple conversion functions that are used in many of the implementations of schemes and algorithms in the text. Although the alphabet of choice when discussing the theoretical aspects of cryptography is the binary alphabet $\{0, 1\}$, when implementing a cryptosystem we usually prefer to work with the alphabet formed by the base-256 digits, which correspond to the integers in the 0..255 range and which we refer to as *bytes*. Each of the 256 bytes can be represented in several ways and we try to use whichever is most convenient. For example, they can be represented as decimal numbers in the set $\{0, 1, \ldots, 255\}$, or as hexadecimal numbers in the set $\{00, 01, \ldots, ff\}$, where each of these numbers is represented by two hexadecimal digits, with the rightmost one being the least significant. It is customary and convenient to use two hexadecimal digits—even for numbers less than 16, for which the left digit will be a 0—because this makes it possible to group several bytes in an even-length hexadecimal string (also called a byte string or an octet string) and go back to the list of bytes without ambiguity. Of course, one byte can also be represented in binary as a string (or as a list) of 8 bits. In general, we use the big-endian notation in which the most significant digit is written to the left.

The use of bytes for the basic alphabet is convenient because our example messages are usually English text strings which are easily converted to and from lists or strings of bytes. The preferred format for the messages, keys, etc., is either even-length hexadecimal strings or lists of bytes given as decimal numbers. Of course, the conversion between these two formats is easy and fast. Moreover, working at the byte level is also useful from an efficiency point of view. For example, in Chap. 2 the bitwise Xor is implemented at the byte level by constructing a table where the result of computing the Xor of any two bytes (viewed as 8-bit strings) is stored. This table is quickly built and uses only a little memory. When computing the bitwise Xor of two long bit strings we only have to make a table look-up for each pair of 8-bit substrings to be Xor-ed. Combining this with the use of Maple's function `convert/bytes` to convert between ASCII-8 text messages (where each character corresponds to one byte) and lists of bytes, we can

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5, © Springer-Verlag Berlin Heidelberg 2013

perform encryption and decryption of messages without having to go to the lower bit level. Working at the byte level is even more convenient in the case of byte-oriented algorithms such as AES in which a great number of byte operations are performed.

We start with a function to pass from bytes, given as integers, to 8-bit lists. Note that Maple uses the little-endian convention in its implementation of `convert` and, since we use the big-endian convention, we reverse the list:

```
> byte2bits := proc(byte)
  local l, k;
  l := ListTools:-Reverse(convert(byte, base, 2));
  k := nops(l);
  [0$(8-k), op(l)]
  end proc:
```

The following tables are helpful to speed up the process of converting between integers in the 0..255 range and 8-bit lists.

```
> bitstobytetable := table(map(x -> byte2bits(x) = x, [$0 .. 255])):
  bytetobitstable := table(map(x -> x = byte2bits(x), [$0 .. 255])):
```

Using these tables we define a very efficient conversion between bit lists and bytes.[1] Thus, for efficiency, we generally use `bytetobits` instead of `byte2bits`; the only purpose of the latter function was to help to build the two preceding tables that once evaluated in the Maple session can also be called directly from other functions.

```
> bitstobyte := bitlist -> bitstobytetable[bitlist]:
  bytetobits := byte -> bytetobitstable[byte]:
```

A quick check that these functions behave as intended is the following:

```
> evalb(map(bitstobyte, map(bytetobits, [$0 .. 255])) = [$0 .. 255]);
                                    true
```

As already mentioned, a more compact way to represent bytes is by means of hex digits, each two-digit hex string corresponding to one byte. More generally, we can represent any list of bytes by the even-length hex string obtained by concatenating the hex representations of the bytes in the list. We next give conversions between bytes or lists of bytes and hex strings, which are used by the encryption/decryption functions. To make these conversions faster, we use tables. Note that Maple's `convert/hex` function outputs the alphabetic hex digits as upper case letters but we use lower case letters instead, as it is more usual in cryptographic texts and code. We start by building a table (in Array format) with the correspondences between bytes and hex digits.

[1] Alternatively, from Maple v12 onwards, the functions `Join` and `Split`, from the package `Bits`, can also be used for this purpose.

```
> bytestohex := Array(0 .. 255,
  proc(i)
  if i < 16 then
    cat("0", StringTools:-LowerCase(convert(i, hex)))
  else
    StringTools:-LowerCase(convert(convert(i, hex), string))
  end if
  end proc):
```

Based on the above table, the following is a function to convert a list of bytes to a hex string.

```
> bytestohexstring := proc(bytelist::{integer[0 .. 255], list(integer[0 .. 255])})
  if bytelist = [] then
    ""
  else
    cat(op(map(x -> bytestohex[x], bytelist)))
  end if
  end proc:
```

Next, we "invert" the preceding table, this time to convert strings formed by two hex digits to the corresponding bytes in integer form, and use this table to give the associated function that converts hex strings to lists of bytes.

```
> hextobytes :=
              table(map(x -> x = convert(x,decimal,hex), convert(bytestohex,list)))

> hexstringtobytes := proc(str::string)
  if str = "" then
    []
  else
    map(x -> hextobytes[x], [StringTools:-LengthSplit(str, 2)])
  end if;
  end proc:
```

A quick check that these functions behave as intended is the following:

```
> evalb(hexstringtobytes(bytestohexstring([$0 .. 255])) = [$0 .. 255]);
  evalb(bytestohexstring(hexstringtobytes(bytestohexstring([$0 .. 255]))) =
    bytestohexstring([$0 .. 255]));
                                    true
                                    true
```

Some of our implementations of schemes are able to encrypt and decrypt binary files. We use the following procedure to read a file to a list of bytes.

```
> filetobytes := proc(filename::string)
  local f;
  f := FileTools:-Binary:-Read(filename, integer[1]) mod 256;
  FileTools:-Binary:-Close(filename);
  f
  end proc:
```

For the inverse process we want to check, before writing a list of bytes to a file, whether a file with the same name already exists and, in this case, whether it can be overwritten. The following function checks this and prompts the user about which

action to take if the file already exists. In this case the user can choose between overwriting the existing file, aborting the process or choosing a new file name. The output of the function is either an error message if the operation is aborted or a file name:

```
> checkfile := proc(filename::string)
  local r, f;
  f := filename;
  if FileTools:-Exists(f) then
    r := readstat("File exists, o/c/a to overwrite/change/abort");
    if r = o then NULL
    elif r = a then
      error "Operation aborted by user"
    else
      f := readstat("Please enter new file name:");
      f := procname(f)
    end if
  end if;
  f
  end proc:
```

Now, to write a list of bytes to a file, we have to convert bytes (integers in the 0..255 range) to integers of type integer[1]. One would like to use the function `mods(-, 256)` but this function does not map the 0..255 range to the $-128..127$ range but to the $-127..128$ range instead. This problem is easily solved since, after applying `mods(-,256)`, it suffices to take care of converting the 128 byte to -128 instead of keeping its original value. We do it by using a slight modification of this function, namely the function `x` $\rightarrow$ `mods(x+1,256)-1`. In order to speed up this process, we build a table to convert bytes to integers in the integer[1] range:

```
> bytetointeger1table := Array(0 .. 255, x -> mods(x+1, 256)-1);
```

The function that writes the byte list to a file is `bytestofile` below. The input is the list of bytes to be written to the file, the file name (given as a string), a Boolean-valued parameter called `filecheck` and a parameter called `logoutput`. If the value `true` is passed to `filecheck`, the function calls `checkfile` to check whether a file with the same name exists in the current directory and eventually prompts the user about which action to take. On the other hand, the value of `logoutput` can be `none`, `file` or `terminal`, and it specifies whether logging information (indicating the number of bytes written and the name of the file) is provided and, in this case, whether it is written to a file or to the terminal (i.e., printed on the screen).

```
> bytestofile := proc(l::(list(integer[0 .. 255])), filename::string,
                      filecheck::truefalse := true, logoutput::name := terminal)
  local f, n;
  f := filename;
  if filecheck then
    f := checkfile(f)
  end if;
  n := nops(l);
  FileTools:-Binary:-Write(f, integer[1], map(x -> bytetointeger1table[x], l));
  FileTools:-Binary:-Close(f);
  if logoutput = file then
    writeto(cat(filename, ".log"));
    printf("%d bytes saved to %s\n\n", n, f);
    writeto(terminal)
  elif logoutput = terminal then
    printf("%d bytes saved to %s\n\n", n, f)
  else
    NULL
  end if
  end proc:
```

Example A.1 We write a text string to a file called `"testfile"` in the current directory and then we read the text string from the file:

```
> bytestofile(messagetobytes("This is a test", text), "testfile");
14 bytes saved to testfile

> convert(filetobytes("testfile"), bytes);
                              "This is a test"
```

Some of the previous functions are combined in the next function, `messagetobytes`, which converts a message to a list of bytes, where the message can be either an even-length hexadecimal (byte) string, an ordinary text string, or a file. These conversions are useful to process the messages prior to encryption or authentication in some schemes. The first input parameter of the function corresponds to the message string or a string that is the name of a file in the current directory and the second parameter is a name that tells the function the type of message; the possible values for this parameter are `file`, `text` or `hex` (of course, hex strings are also text strings except that we use a different method to convert them to lists of bytes). The output of the function is a list of bytes given as integers in the 0..255 range and we often use this format to perform on the message the operations required by several cryptographic primitives.

```
> messagetobytes := proc(message::string, messagetype::name)
  if messagetype = file then
    filetobytes(message)
  elif messagetype = text then
    convert(message, bytes)
  elif messagetype = hex then
    hexstringtobytes(StringTools:-LowerCase(message))
  else
    error "incorrect message type specification"
  end if
  end proc:
```

On some occasions it is interesting to convert messages to and from hexadecimal form. Using the previously defined functions we can do it easily. The next function, `messagetohex`, converts a message (in either text, hex, or file format) to an even-length hexadecimal string. There are two input parameters, `message` (either an even-length hex or a text string) and `messagetype`, a name which tells the function whether the message is a hex string (with value `hex`), a message given as a text string (`text`), or a text string which is the name of a file (`file`). The output is a byte string given as an even-length hexadecimal string.

```
> messagetohex := proc(message::string, messagetype::name)
  if messagetype = hex then
    message
  elif messagetype = text then
    bytestohexstring(convert(message, bytes))
  elif messagetype = file then
    bytestohexstring(filetobytes(message))
  else
    error "incorrect message type"
  end if
  end proc:
```

The inverse function of the preceding one converts an even-length hexadecimal string to a message in any one of the three formats mentioned above. It is the function `hextomessage` with three input parameters, `hexstring` (the string to be converted), `messagetype` (the type of message to be converted to) and `filename`, which is only used when `messagetype = file`, in which case it serves to specify the name of the file to which the message is to be written.

```
> hextomessage := proc(hexstring::string, messagetype::name, filename::string)
  if messagetype = hex then
    hexstring
  elif messagetype = text then
    convert(hexstringtobytes(hexstring), bytes)
  elif messagetype = file then
    bytestofile(hexstringtobytes(hexstring), filename)
  else
    error "incorrect message type"
  end if
  end proc:
```

Example A.2 Let us convert a text message to a hex string and back:

```
> messagetohex("The magic words are squeamish ossifrage", text);
"546865206d616769632077726473206172652073717565616d697368206f7373696672616765"

> hextomessage(%, text);
                  "The magic words are squeamish ossifrage"
```

The next function plays an auxiliary role in several functions where a key or a seed that can be given either as a hexadecimal string or an integer is used. It checks whether the input is a string and, if so, whether it is a hexadecimal string and, in that case, it converts it to an integer which can be used as a seed. If the input is an integer, then the output is this same integer.

```
> stringposint := proc(x)
  if type(x, string) then
    if not StringTools:-IsHexDigit(x) then
      error "This value must be either a hex string or an integer"
    end if;
    convert(x, decimal, hex)
  else
    x
  end if;
  end proc:
```

References

1. Abdalla, M., Bellare, M., & Rogaway, P. (2001). The oracle Diffie-Hellman assumptions and an analysis of DHIES. In *Topics in Cryptology-CT-RSA 2001, Lecture Notes in Computer Science 2020* (pp. 143–158). Springer.
2. Adams, C., & Lloyd, S. (2002). *Understanding PKI: Concepts, standards, and deployment considerations* (2nd ed.). Reading: Addison-Wesley.
3. Agrawal, M., Kayal, N., & Saxena, N. (2004). PRIMES is in P. *Annals of Mathematics,160*, 781–793.
4. Ajtai, M., & Dwork, C. (1997). A public-key cryptosystem with worst-case/average-case equivalence, In *Proceedings of the 29th ACM STOC*, pp. 284–293.
5. Alford, W. R., Granville, A., & Pomerance, C. (1994). There are infinitely many Carmichael numbers. Annals of Mathematics (2) *139*(3), 703–722.
6. Arora, S., & Barak, B. (2009). *Computational complexity: A modern approach*. Cambridge: Cambridge University Press.
7. Bach, E., & Shallit, J. (1996). *Algorithmic number theory: Volume 1 efficient algorithms.* Cambridge: The MIT Press.
8. Balasubramanian, R., & Koblitz, N. (1998). The improbability that an elliptic curve has subexponential discrete log problem under the Menezes-Okamoto-Vanstone algorithm. *Journal of Cryptology,11*, 141–145.
9. Bard, G.V. (2009). *Algebraic cryptanalysis*. Heidelberg: Springer.
10. Barker, E., Johnson, D., Smid, M. (2007). *Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (revised)*. NIST Special Publication 800–56A. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf
11. Barker, E., et al. (2007). *Recommendation for key management, Part 1: General (revised)*, NIST Special Publication 800–57. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf
12. Bassham III, L. E. (2002). *The Advanced Encryption Standard algorithm validation suite*, NIST publication. Retrieved from http://csrc.nist.gov/groups/STM/cavp/documents/aes/AESAVS.pdf
13. Bellare, M., Canetti, R., & Krawczyk, H. (1996). Keying hash functions for message authentication. In *Proceedings of CRYPTO '96*, *Lecture Notes in Computer Science* (Vol. 1109, pp. 1–15). Springer.
14. Bellare, M., Desai, A., Pointcheval, D., & Rogaway, P. (1998). Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology, Proceedings of CRYPTO '98*, *Lecture Notes in Computer Science* (Vol. 1462, pp. 26–46). Springer.

DOI: 10.1007/978-3-642-32166-5, © Springer-Verlag Berlin Heidelberg 2013

15. Bellare, M., Kilian, J., & Rogaway, P. (1994). The security of cipher block chaining, In *Advances in Cryptology, Proceedings of CRYPTO '94, Lecture Notes in Computer Science* (Vol. 839, pp. 341–358). Springer.
16. Bellare, M., & Namprempre, C. (2000). Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology—Proceedings of ASIACRYPT 2000, Lecture Notes in Computer Science* (Vol. 1976, pp. 531–545). Springer.
17. Bellare, M., Namprempre, C., & Neven, G. (2004). Security proofs for identity-based identification and signature schemes. In *Advances in Cryptology—EUROCRYPT 2004, Lecture Notes in Computer Science* (Vol. 3027, pp. 268–286). Springer.
18. Bellare, M., & Rogaway, P. (1993). Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (pp. 62–73). ACM.
19. Bellare, M., & Rogaway, P. (1995). Optimal asymmetric encryption. In *Advances in Cryptology, Procedings of EUROCRYPT '94, Lecture Notes in Computer Science* (Vol. 950, pp. 92–111). Springer.
20. Bellare, M. & Rogaway, P. (1996). The exact security of digital signatures—How to sign with RSA and Rabin. In *Advances in Cryptology, Proceedings of EUROCRYPT '96, Lecture Notes in Computer Science* (Vol. 1070, pp. 399–416). Springer.
21. Bellare, M., & Rogaway, P. *Introduction to modern cryptography*. Course notes. Retrieved from http://cseweb.ucsd.edu/∼mihir/cse207/classnotes.html
22. Bernstein, D. J., Buchmann, J., & Dahmen, E. (Eds.) (2009), *Post-quantum cryptography*. Heidelberg: Springer.
23. Bernstein, D. J., et al. *Breaking ECC2K-130*. Retrieved from http://www.ecc-challenge.info/
24. Bernstein, D. J. & Lange, T. *Edwards coordinates for elliptic curves*. http://cr.yp.to/newelliptic/newelliptic.html
25. Biryukov, A., & Khovratovich, D. (2009). Related-key cryptanalysis of the full AES-192 and AES-256. In *Advances in Cryptology, Proceedings of ASIACRYPT 2009, Lecture Notes in Computer Science* (Vol. 5912, pp. 1–18). Springer.
26. Blake, I. F., Seroussi, G., & Smart, N. P. (2005). *Advances in elliptic curve cryptography. London Mathematical Society Lecture Note Series* (Vol. 317). Cambridge: Cambridge University Press.
27. Bleichenbacher, D. (1996). Efficiency and security of cryptosystems based on number theory. *Dissertation*. ETH. Retrieved from http://www.bell-labs.com/user/bleichen/diss/thesis.html
28. Bleichenbacher, D. (1996) Generating El Gamal signatures without knowing the secret key. In *Advances in Cryptology, Proceedings of EUROCRYPT '96, Lecture Notes in Computer Science* (Vol. 1070, pp. 10–18). Springer
29. Bleichenbacher, D. (1998) Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology, Proceedings of CRYPTO '98, Lecture Notes in Computer Science* (Vol. 1462, pp. 1–12). Springer.
30. BlueKrypt, *Cryptographic length recommendation*. http://www.keylength.com/
31. Blum, L., Blum, M., & Shub, M. (1986). A simple unpredictable random number generator. *SIAM Journal on Computing*, *15*, 364–383.
32. Bogdanov, A., Khovratovich, D., Rechberger, C., *Biclique cryptanalysis of the full AES*. Retrieved from http://eprint.iacr.org/2011/449
33. Boneh, D. (1999). Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society*, *46*, 203–213.
34. Boneh, D. (2001). Simplified OAEP for the RSA and Rabin functions. In *Proceedings of CRYPTO 2001, Lecture Notes in Computer Science* (Vol. 2139, pp. 275–291). Springer.
35. Boneh, D., & Franklin, M. (2003). Identity-based encryption from the Weil pairing. *SIAM Journal on Computing*, *32*(3), 586–615.
36. Boneh, D., Gentry, C., & Hamburg, M. (2007). Space-efficient identity based encryption without pairings. In *Proceedings of 48th Annual Symposium on Foundations of Computer Science (FOCS 2007)*, (pp. 647–657). IEEE Computer Society Press.

37. Boneh, D., Joux, A., & Nguyen, P. Q. (2000). Why textbook ElGamal and RSA encryption are insecure. In *Proceedings of ASIACRYPT 2000, Lecture Notes in Computer Science* (Vol. 1976, pp. 30–43). Springer.
38. Boneh, D., & Venkatesan, R. (1998). Breaking RSA may not be equivalent to factoring. In *Proceedings of EUROCRYPT '98, Lecture Notes in Computer Science* (Vol. 1403, pp. 59–71). Springer.
39. Bos, J. W., et al. *112-bit prime ECDLP solved*. http://lacal.epfl.ch/page81774.html
40. Bradford, A., Monagan, M., Percival, C. *Integer factorization and computing discrete logarithms in Maple*. Retrieved from http://www.cecm.sfu.ca/pborwein/MITACS/papers/percival.pdf
41. Brent, R. P., & Pollard, J. M. (1981). Factorization of the eighth Fermat number. *Mathematics of Computation*, *36*, 627–630.
42. Brown, D. On the provable security of ECDSA. Chapter 2 in [26].
43. Buchmann, J. A. (2004). *Introduction to cryptography* (2nd ed.). New York: Springer.
44. Burthe, R. (1996). Further investigations with the strong probable prime test. *Mathematics of Computation*, *65*, 373–381.
45. Cameron, P. J. (2008). *Introduction to algebra* (2nd ed.). New York: Oxford University Press.
46. Canetti, R., Goldreich, O., & Halevi, S. (1998). The random oracles methodology, revisited. In *Proceedings of the 30th STOC,* (pp. 209–218). ACM Press.
47. Certicom, *The Certicom ECC challenge*. http://www.certicom.com/index.php/the-certicom-ecc-challenge
48. Childs, L. N. (2009). *A concrete introduction to higher algebra* (3rd ed.). New York: Springer.
49. Cid, C., Murphy, S., & Robshaw, M. (2006). *Algebraic aspects of the Advanced Encryption Standard*. Berlin: Springer.
50. Clay Mathematics Institute, *Millennium prize problems*. http://www.claymath.org/millennium/
51. Cocks, C. (2001). An identity based encryption scheme based on quadratic residues. In *Cryptography and Coding, Proceedings of the 8th IMA Intlernational Conference, Lecture Notes in Computer Science* (Vol. 2260, pp. 360–363). Springer.
52. Cohen, H. (1993). *A course in computational algebraic number theory*. Graduate texts in mathematics (Vol. 138). Berlin: Springer.
53. Cohen, H., Frey, G., et al. (2006). *Handbook of elliptic and hyperelliptic curve cryptography*. Boca Raton: Chapman & Hall/CRC.
54. Coppersmith, D. (1997). Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology,10*, 233–260.
55. Coron, J. S., & May, A. (2007). Deterministic polynomial-time equivalence of computing the RSA secret key and factoring. *Journal of Cryptology,20*, 39–50.
56. Courtois, N., & Pieprzyk, J. (2002). Cryptanalysis of block ciphers with overdefined systems of equations. In *Proceedings of ASIACRYPT 2002, Lecture Notes in Computer Science* (Vol. 2501, pp. 267–287). Springer.
57. Cramer, R., & Shoup, V. (1998). A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *Advances in Cryptology, Proceedings of CRYPTO '98, Lecture Notes in Computer Science* (Vol. 1462, pp. 13–25). Springer.
58. Cramer, R., & Shoup, V. (2000). Signature schemes based on the strong RSA assumption. *ACM Transactions on Information and System Security,3*, 161–185.
59. Cramer, R., & Shoup, V. (2003). Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing ,22*, 167–226.
60. Crandall, R., & Pomerance, C. (2005). *Prime numbers: A computational perspective* (2nd ed.). New York: Springer.

61. Curtin, M. (2005). *Brute force: Cracking the Data Encryption Standard*. New York: Copernicus Books.
62. Daemen, J., & Rijmen, V. (2002). *The design of Rijndael: AES—the Advanced Encryption Standard*. Heidelberg: Springer.
63. Damgård, I., Jurik, M., & Nielsen, J. B. (2010). A generalization of Paillier's public-key system with applications to electronic voting. *International Journal of Information Security*, *9*(6), 371–385.
64. Dawson, E., & Nielsen, L. (1996). Automated cryptanalysis of XOR plaintext strings. *Cryptologia,2*, 165–181.
65. Delfs, H., & Knebl, H. (2007). *Introduction to cryptography*. *Principles and applications* (2nd ed.). New York: Springer.
66. Diaconis, P., Holmes, S., & Montgomery, R. (2007). Dynamical bias in the coin toss. *SIAM Review,49*(2), 211–235.
67. Diffie, W., Hellman, M. E. (1976). New directions in cryptography. *IEEE Transactions on Information Theory, IT-22*(6), 644–654.
68. Dworkin, M. (2001). *Recommendation for block cipher modes of operation: Methods and techniques*. NIST Special Publication 800–38A. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf
69. Dworkin, M. (2005). *Recommendation for block cipher modes of operation: The CMAC mode for authentication*. NIST Special Publication 800–38B. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf
70. Dworkin, M. (2007). *Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication 800–38D. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf
71. Elgamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory,31*(4), 469–472.
72. Erra, R., & Grenier, C., *The Fermat factorization method revisited,* preprint, Retrieved from http://eprint.iacr.org/2009/318.ps
73. Federal Information Processing Standards Publication 197 (2001). *Specification for the Advanced Encryption Standard (AES)*. Retrieved from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
74. Federal Information Processing Standards Publication 180-3 (2008). *Secure Hash Standard*. Retrieved from http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf
75. Federal Information Processing Standards Publication 186-3 (2009). *Digital Signature Standard (DSS)*. Retrieved from http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf
76. Federal Information Processing Standards Publication 198–1 (2008). *The Keyed-Hash Message Authentication Code (HMAC)*. Retrieved from http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf
77. Fine, B., & Rosenberger, G. (2007). *Number theory: An introduction via the distribution of primes*. Boston: Birkhäuser.
78. Friedlander, J. B., Pomerance, C., & Shparlinski, I. (2001). Period of the power generator and small values of Carmichael's function. *Mathematics of Computation,70*, 1591–1605.
79. Fujisaki, E., & Okamoto, T. (1999). Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology, Proceedings of CRYPTO '99, Lecture Notes in Computer Science* (Vol. 1666, pp. 537–554). Springer.
80. Fujisaki, E., Okamoto, T., Pointcheval, D., & Stern, J. (2004). RSA-OAEP is secure under the RSA assumption. *Journal of Cryptology,17*, 81–104.
81. Galbraith, S. D. (2012). *Mathematics of public key cryptography*. Cambridge: Cambridge University Press.
82. Galbraith, S. D., & McKee, J. F. (2000). The probability that the number of points on an elliptic curve over a finite field is prime. *Journal of the London Mathematical Society,62*, 671–684.

83. Galindo, D. (2005). Boneh-Franklin identity based encryption revisited. In *Proceeding of Automata, Languages and Programming (ICALP 2005), Lecture Notes in Computer Science* (Vol. 3580, pp. 791–802). Springer.
84. Gentry, C., *A fully homomorphic encryption scheme*. Retrieved from http://crypto.stanford.edu/craig/craig-thesis.pdf
85. GGNFS. Retrieved from http://sourceforge.net/projects/ggnfs/
86. Goldreich, O. (2001). *Foundations of cryptography, vol. 1: Basic tools*. Cambridge: Cambridge University Press.
87. Goldreich, O. (2004). *Foundations of cryptography, vol. 2: Basic applications*. Cambridge: Cambridge University Press.
88. Goldwasser, S. (1990). The search for provably secure cryptosystems. In *Cryptology and Computational Number Theory, Proceedings of Symposia in Applied Mathematics* (Vol. 42, pp. 89–113). American Mathematical Society.
89. Goldwasser, S. (2002). Mathematical foundations of modern cryptography: Computational complexity perspective. In *Proceedings of the ICM2002* (Vol. 1, pp. 245–272). Higher Education Press.
90. Goldwasser, S., & Bellare, M., *Lecture notes on cryptography*. Retrieved from http://cseweb.ucsd.edu/∼mihir/papers/gb.pdf
91. Goldwasser, S., & Micali, S. (1984). Probabilistic encryption. *Journal of Computer and System Sciences,28*, 270–299.
92. Goldwasser, S., Micali, S., & Rivest, R. (1988). A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing,17*, 281–308.
93. Gómez Pardo J. L., & Gómez-Rodríguez, C., *The Advanced Encryption Standard and its modes of operation*. Retrieved from http://www.maplesoft.com/applications/view.aspx?SID=6618
94. Gordon, H. (1997). *Discrete probability*. New York: Springer.
95. Grantham, J., *Pseudoprimes/probable primes*. http://www.pseudoprime.com/pseudo.html
96. Gurevich, Yu. (1989). The challenger-solver game: Variations on the theme of P=?NP. *Bulletin of the European Association for Theoretical Computer Science,39*, 112–121.
97. Hankerson, D., Menezes, A., & Vanstone, S. (2004). *Guide to elliptic curve cryptography*. Heidelberg: Springer.
98. Håstad, J., & Näslund, M. (2004). The security of all RSA and discrete log bits. *Journal of the ACM,51*, 187–230.
99. Hazay, C., & Lindell, Y. (2010). *Efficient secure two-party protocols*. Heidelberg: Springer.
100. Heck, A. (2003). *Introduction to Maple* (3rd ed.). New York: Springer.
101. Hinek, M. J. (2010). *Cryptanalysis of RSA and its variants*. Boca Raton: CRC Press.
102. Hoffstein, J., Pipher, J., & Silverman, J. H. (2008). *An introduction to mathematical cryptography*. New York: Springer.
103. IBM, *IBM's test-tube quantum computer makes history*. Retrieved from http://www-03.ibm.com/press/us/en/pressrelease/965.wss
104. Impagliazzo, R. (1995). A personal view of average-case complexity. In *Proceedings of the 10th Annual Structure in Complexity Theory Conference (SCT'95),* (pp. 134–147).
105. Iwata, T., & Kurosawa, K., *Stronger security bounds for OMAC, TMAC and XCBC*. Retrieved from http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/General_Comments/papers/Iwata.pdf
106. Joux, A. (2009). *Algorithmic cryptanalysis*. Boca Raton: Chapman & Hall/CRC.
107. Joye, M., & Neven, G. (Eds.) (2009). *Identity-based cryptography*. Amsterdam: IOS Press.
108. Kahn, D. (1996). *The codebreakers: The story of secret writing*. New York: Scribner.
109. Katz, J., & Lindell, Y. (2008). *Introduction to modern cryptography*. Boca Raton: Chapman & Hall/CRC.
110. Kelly, S., & Frankel, S. (2007). *Using RFC 4868, HMAC-SHA256, SHA384, and SHA512 with IPsec*. Retrieved from http://www.rfc-editor.org/rfc/rfc4868.txt

111. Kiltz, E., & Neven, G., Identity-based signatures, in [107] (pp. 31–44).
112. Kleinjung, T. *Discrete logarithms inGF*(p)—160 *digits*. Retrieved from http://old.nabble.com/Discrete-logarithms-in-GF(p)—160-digits-td8810595.html
113. Kleinjung, T., et al., *Factorization of a 768-bit RSA modulus*. Retrieved from http://eprint.iacr.org/2010/006.pdf
114. Knudsen, L. R., & Robshaw, M. J. B. (2011). *The block cipher companion*. New York: Springer.
115. Knuth, D. E. (1998). *The art of computer programming, vol. 2: Seminumerical algorithms* (2nd ed.). Reading: Addison-Wesley.
116. Knuth, D. E. (1998). *The art of computer programming, vol. 3: Sorting and searching* (2nd ed.). Reading: Addison-Wesley.
117. Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of Computation,48*, 203–209.
118. Koblitz, N. (1994). *A course in number theory and cryptography* (2nd ed.). New York: Springer.
119. Koblitz, N. (1998). *Algebraic aspects of cryptography*. New York: Springer.
120. Koblitz, N., & Menezes, A. (2007). Another look at "provable security". *Journal of Cryptology,20*, 3–37.
121. Kocher, P. (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology, Proceedings of CRYPTO '96, Lecture Notes in Computer Science* (Vol. 1109, pp. 104–113). Springer.
122. Krawczyk, H. (1994). LFSR-based hashing and authentication. In *Advances in Cruyptology, Proceedings of CRYPTO '94, Lecture Notes In Computer Science* (Vol. 839, pp. 129–139). Springer.
123. Krawczyk, H. (2001). The order of encryption and authentication for protecting communications (Or: how secure is SSL?). In *Advances in Cryptology, Procedings of CRYPTO 2001, Lecture Notes in Computer Science* (Vol. 2139, pp. 310–331). Springer
124. Krawczyk, H., Bellare, M., & Canetti, R. (1997). *RFC 2104*, *HMAC: Keyed-hashing for message authentication*. Retrieved from http://www.ietf.org/rfc/rfc2104.txt
125. Lemmermeyer, F., *Proofs of the quadratic reciprocity law*. Retrieved from http://www.rzuser.uni-heidelberg.de/∼hb3/fchrono.html
126. Lenstra, H. W., Jr. (2002). *Rijndael for algebraists*. Retrieved from http://math.berkeley.edu/∼hwl/papers/rijndael0.pdf
127. Lenstra, A. K., & Lenstra, H. W. Jr. (1990). Algorithms in number theory. In J. van Leeuwen (Ed.), *Handbook of theoretical computer science* (Vol. A, pp. 673–716). New York :Elsevier
128. Lenstra, A. K., & Verheul, E. R. (2001). Selecting cryptographic key sizes. *Journal of Cryptology,14*, 255–293.
129. Levin, L. A. (1986). Average case complete problems. *SIAM Journal on Computing,15*(1), 285–286.
130. Lewand, R.E. (2000). *Cryptological mathematics*. Washington: The Mathematical Association of America.
131. Lidl, R., & Niederreiter, H. (1994). *Introduction to finite fields and their applications*, (Revised edition). Cambridge: Cambridge University Press.
132. Magma Computational Algebra System. http://magma.maths.usyd.edu.au/magma/
133. Manger, J. (2001). A chosen ciphertext attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as standardized in PKCS #1 v2.0. In *Advances in Cryptology, Proceedings of CRYPTO 2001, Lecture Notes in Computer Science* (Vol. 2139, pp. 230–238). Springer.
134. Mao, W. (2003). *Modern cryptography: Theory and practice*. Upper Saddle River: Prentice Hall PTR.
135. Maplesoft. http://www.maplesoft.com
136. Martin, L. (2008). *Introduction to identity-based encryption*. Boston: Artech House.
137. Martin, M., *Primo—primality proving*. http://www.ellipsa.eu

138. McGrew, D. A., & Viega, J. (2004). The security and performance of the Galois/Counter Mode (GCM) of operation. In *Progress in Cryptology, Proceedings of INDOCRYPT 2004, Lecture Notes in Computer Science* (Vol. 3348, pp. 343–355). Springer.
139. Menezes, A., Okamoto, T., & Vanstone, S. A. (1993). Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory,39*, 1639–1646.
140. Menezes, A., van Oorschot, P. C., & Vanstone, S. A. (1996). *Handbook of applied cryptography*. Boca Raton: CRC Press.
141. Micciancio, D., & Regev, O. Lattice-based cryptography. in [22] (pp. 147–191).
142. Miller, V. (1986). Uses of elliptic curves in cryptography. In *Advances in Cryptology, Proceedings of CRYPTO '85, Lecture Notes in Computer Science* (Vol. 218, pp. 417–426), Springer.
143. MIRACL, *Multiprecision Integer and Rational Arithmetic C/C++ Library*. Retrieved from http://certivox.com
144. Monagan, M., & Fee, G., *A cryptographically secure random bit generator for Maple*. Retrieved from http://www.cecm.sfu.ca/CAG/papers/monfeeMSWS04.pdf
145. Moore, G. E. (1965). *Cramming more components onto integrated circuits*, Electronics (Vol. 38). Retrieved from ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf
146. Morain, F., *The ECPP home page*. http://www.lix.polytechnique.fr/∼morain/Prgms/ecpp.english.html
147. Msieve. Retrieved from http://sourceforge.net/projects/msieve/
148. NIST SHA-256 Examples. Retrieved from http://csrc.nist.gov/groups/ST/toolkit/documents/Examples/SHA256.pdf
149. NSA Suite B Cryptography. Retrieved from http://www.nsa.gov/ia/programs/suiteb_cryptography/
150. Odlyzko, A. M. (1990). The rise and fall of knapsack cryptosystems. In *Cryptology and number theory, Proceedings of Symposia in Applied Mathematics* (Vol. 42, pp. 75–88). American Mathematical Society.
151. Odlyzko, A. M., *The future of integer factorization*. Retrieved from http://www.dtc.umn.edu/odlyzko/doc/future.of.factoring.pdf
152. Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology, Proceedings of EUROCRYPT '99, Lecture Notes in Computer Science* (Vol. 1592, pp. 223–238). Springer.
153. Paterson, K. G., Cryptography from pairings. in [26] (pp. 215–251).
154. *PKCS #1 v2.1: RSA Cryptography Standard* (2002). RSA laboratories. Retrieved from http://www.rsa.com/rsalabs/node.asp?id=2125
155. Pointcheval, D. (2005). Provable security for public key schemes. In D. Catalano et al. (Eds.), *Contemporary cryptology* (pp. 133–190). Basel: Birkhäuser.
156. Pointcheval, D., & Stern, J. (2000). Security arguments for digital signatures and blind signatures. *Journal of Cryptology,13*, 361–396.
157. Polymath Wiki, *Finding primes*. http://michaelnielsen.org/polymath1/index.php?title=Finding_primes
158. Pomerance, C. (1985). The quadratic sieve factoring algorithm. In *Advances in Cryptology, Proceedings of EUROCRYPT 84, Lecture Notes in Computer Science* (Vol. 209, pp. 169–182). Springer.
159. Pomerance, C., Selfridge, J. L., & Wagstaff, S. S, Jr. (1980). The pseudoprimes up to $25 \cdot 10^9$. *Mathematics of Computation,35*, 1003–1026.
160. Pomerance, C., Smith, J. W., & Tuler, R. (1988). A pipeline architecture for factoring large integers with the quadratic sieve algorithm. *SIAM Journal on Computing,17*, 387–403.
161. Project Gutenberg web page. http://www.gutenberg.org/wiki/Main_Page
162. Ribenboim, P. (2004). *The little book of bigger primes* (2nd ed.). New York: Springer.
163. Rivest, R., Adleman, L., & Dertouzos, M. (1978). On data banks and privacy homomorphisms. In *Foundations of Secure Computation* (pp. 169–180).

164. Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM,21*(2), 120–126.
165. Rivest, R., & Silverman, R., *Are 'strong' primes needed for RSA?*. Retrieved from http://eprint.iacr.org/2001/007
166. Rompel, J. (1990). One-way functions are necessary and sufficient for secure signatures. In *Proceedings of the 22nd STOC* (pp. 387–394).
167. Ross, S. M. (2009). *A first course in probability* (8th ed.). Upper Saddle River: Prentice-Hall.
168. Ross, S. M. (2010). *Introductory statistics* (3rd ed.). New York: Elsevier.
169. Rothe, J. (2005). *Complexity theory and cryptology: An introduction to cryptocomplexity*. Berlin: Springer.
170. RSA Factoring Challenge. Retrieved from http://www.rsa.com/rsalabs/node.asp?id=2092
171. Sage, free open-source mathematics software system. Retrieved from http://www.sagemath.org/
172. Schneier, B. (1996). *Applied cryptography: Protocols, algorithms, and source code in C* (2nd ed.). New York: Wiley.
173. Standards for Efficient Cryptography 1 (SEC 1): Elliptic curve cryptography, version 2.0. Retrieved from http://www.secg.org/download/aid-780/sec1-v2.pdf
174. Standards for Efficient Cryptography 2 (SEC 2): Recommended elliptic curve domain parameters, version 2.0. Retrieved from http://www.secg.org/download/aid-784/sec2-v2.pdf
175. Shamir, A. (1984). Identity-based cryptosystems and signature schemes. In *Advances in Cryptology, Proceedings of CRYPTO 84, Lecture Notes in Computer Science* (Vol. 196, pp. 47–53). Springer
176. Shannon, C. E. (1949). Communication theory of secrecy systems. *Bell System Technical Journal,28*, 656–715.
177. Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing,26*(5), 1484–1509.
178. Shoup, V. (1997). Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology, Proceedings of EUROCRYPT '97, Lecture Notes in Computer Science* (Vol. 1233, pp. 256–266). Springer.
179. Shoup, V. (2002). OAEP reconsidered. *Journal of Cryptology,15*, 223–249.
180. Shoup, V. (2009). *A computational introduction to number theory and algebra* (2nd ed.). Cambridge: Cambridge University Press. Retrieved from http://www.shoup.net/ntb/
181. Silverman, J. H. (2009). *The arithmetic of elliptic curves* (2nd ed.). Graduate texts in mathematics (Vol. 106). Dordrecht: Springer.
182. Silverman, J. H., & Tate, J. (1992). *Rational points on elliptic curves*. New York: Springer.
183. Singh, S. (1999). *The code book: The science of secrecy from ancient Egypt to quantum cryptography*. New York: Doubleday.
184. Smart, N., *Cryptography, an introduction*. Retrieved from http://www.cs.bris.ac.uk/~nigel/Crypto_Book/
185. Sotirov, A., Stevens, M., Appelbaum, J., Lenstra, A., Molnar, D., Osvik, D. A., & de Weger, B., *MD5 considered harmful today*. Retrieved from http://www.win.tue.nl/hashclash/rogue-ca/
186. Stamp, M., & Low, R. M. (2007). *Applied cryptanalysis: Breaking ciphers in the real world*. New York: Wiley.
187. Stangl, W. D. (1996). Counting squares in $\mathbb{Z}_n$. *Mathematics Magazine*, *69*, 285–289.
188. Stevens, M., Lenstra, A., & de Weger, B., *Chosen-prefix collisions for MD5 and applications*. Retrieved from https://documents.epfl.ch/users/l/le/lenstra/public/papers/lat.pdf
189. Stinson, D. R. (2006). *Cryptography: Theory and practice* (3rd ed.). Boca Raton: Chapman & Hall/CRC.
190. Swenson, C. (2008). *Modern cryptanalysis: Techniques for advanced code breaking*. New York: Wiley.

191. Talbot, J., & Wells, D. (2006). *Complexity and cryptography: An introduction*. Cambridge: Cambridge University Press.
192. Tao, T., Croot III, E. & Helfgott, H. (2011). Deterministic methods to find primes. *Mathematics of Computation*. Retrieved from http://www.ams.org/journals/mcom/0000-000-00/S0025-5718-2011-02542-1/S0025-5718-2011-02542-1.pdf
193. Vaudenay, S. (2006). *A classical introduction to cryptography: Applications for communications security*. New York: Springer.
194. Wagstaff, S. S. Jr. (2003). *Cryptanalysis of number theoretic ciphers*. Boca Raton: Chapman & Hall/CRC.
195. Wang, X., Yin, Y. L., & Yu, H. (2005). Finding collisions in the full SHA-1. In *Advances in Cryptology, CRYPTO 2005, Lecture Notes in Computer Science* (Vol. 3621, pp. 17–36). Springer.
196. Wang, X., & Yu, H. (2005). How to break MD5 and other hash functions. In *Advances in Cryptology, Proceedings of EUROCRYPT 2005, Lecture Notes in Computer Science* (Vol. 3494, pp. 19–35). Heidelberg: Springer.
197. Washington, L. C. (2008). *Elliptic curves: Number theory and cryptography* (2nd ed.). Boca Raton: Chapman& Hall/CRC.
198. de Weger, B. (2002). Cryptanalysis of RSA with small prime difference. *Applicable Algebra in Engineering, Communication and Computing,13*, 17–28.
199. Wright, E. V., *Gadsby*. Retrieved from http://www.holybooks.com/wp-content/uploads/Gadsby-by-Ernest-Vincent-Wright.pdf
200. Wu, H., *The misuse of RC4 in Microsoft Word and Excel*. Retrieved from http://eprint.iacr.org/2005/007
201. Yan, S. Y. (2008). *Cryptanalytic attacks on RSA*. New York: Springer.
202. Yan, S. Y. (2009). *Primality testing and integer factorization in public-key cryptography* (2nd ed.). New York: Springer.
203. Yao, A. C. (1982). Theory and applications of trapdoor functions. In *Proceedings 23rd IEEE Symposium on Foundations of Computer Science* (pp. 80–91).

Index

$620 problem, 314

A
A posteriori CCA attack, *see* CCA2 attack
A priori CCA attack, *see* CCA1 attack
Adaptive CCA attack, *see* CCA2 attack
Adaptive chosen message attack, 541
Advanced Encryption Standard, *see* AES
Advantage, 171, 185
Adversary, xix
  computationally bounded, 8, 33, 138
  computationally unbounded, 131, 139
  PPT, 168, 170–177
AES, 186, 189–203, 675
  AddRoundKey, 196
  decryption, 199
  encryption, 191
  expanded key, 190
  forward cipher function, 191
  InvMixColumns, 201
  InvShiftRows, 201
  InvSubBytes, 199
  key expansion, 189, 196
  MixColumns, 195
  operations, 190
  round, 189
  round keys, 190
  S-box, 191
  ShiftRows, 195
  state, 189
  SubBytes, 191
Ajtai–Dwork public-key encryption scheme, 537
AKS test, 315
Algebraic homomorphic encryption scheme, 536
Algorithm
  AES encryption, 191
  AES key expansion, 197
  Atlantic City, 58
  baby-step giant-step, 374
  binary exponentiation, 85
  CBC decryption, 205
  CBC encryption, 204
  CFB decryption, 206
  CFB encryption, 206
  Chinese remainder theorem, 76
  CMAC, 244
  compression function of SHA-256, 266
  computation of a generator of a finite cyclic group, 91
  computation of square roots modulo an odd prime, 125
  computation of the 4 square roots of a quadratic residue modulo a Blum integer, 478
  computation of the inverse of the Blum–Williams function, 478
  computation of the Jacobi symbol, 122
  CTR decryption, 211
  CTR encryption, 210
  decryption, 169, 424, 595
  deterministic, 58
  division, 39
  encryption, 169, 424, 595
  Euclidean, 61
  expected polynomial-time, 58, 315, 317, 318
  exponential-time, 46
  extended Euclidean, 67
  Fermat factoring, 331
  Floyd's cycle-finding algorithm, 326
  generating a random prime, 317

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5, © Springer-Verlag Berlin Heidelberg 2013

A (*cont.*)
HMAC, 270
index calculus method, 390
key derivation, 591, 603
key generation, 168, 423
Las Vegas, 58
Merkle–Damgård construction, 263
Miller–Rabin test, 311
modular exponentiation, 85
Monte Carlo, 58
no-biased, 58
yes-biased, 58
OFB decryption, 208
OFB encryption, 208
Pohlig–Hellman, 410, 550
Pollard's rho for discrete logarithms, 381
polynomial division, 102
polynomial-time, 46, 168
primality test, 286
probabilistic, 58
quadratic sieve, 356
quantum, 372
randomized, *see* probabilistic
recover, 569
RSA key generation, 432
Shor's quantum factoring, 372
sieve of Eratosthenes, 51
signing, 592, 594
strong probable prime test, 303
subexponential-time, 47, 408, 504, 656
superpolynomial-time, 315
tag generation, 235
trial division, 53
verification, 235, 592, 594
Alice, xxi
Arithmetic algebraic geometry, 613
Asymptotic functions, 43
Asymptotic notation, 43
big-O, 43
big-Omega, 43
big-Theta, 43
little-o, 43
Attack
birthday, 276–284
Bleichenbacher's attack against PKCS #1 v1.5, 462
brute-force, 5, 9, 163
chosen ciphertext, xxv
chosen message, 237, 541
chosen plaintext, xxv
ciphertext-only, xxv
common modulus, 457–458
cycling, 438
dictionary, 24
FR, 660
Håstad's broadcast, 455–457
key-only, 541
known plaintext, xxv
man-in-the-middle, 409, 414–418
meet-in-the-middle, 451
message, 541
MOV, 660
no-message, 541
related-key, 202
replay, 236
Semaev, Satoh–Araki, Smart, 662
short pad, 460
shortcut, 188, 201
side-channel, 187, 203, 459
spoofing, 235
timing, 459
Wiener's low exponent, 450
Attribute certificate, 583
Authenticated encryption, 248
Average complexity, 60

B
Bayes' formula, 37
BDH problem, 609
is hard, 609
Benign malleability, 677
Bernoulli experiment, 38
Big-endian, 223, 255, 256, 681
Bilinear Diffie–Hellman problem, *see* BDH problem
Bilinear pairing, *see* pairing
Binary search, 49
Birthday attack, 276–284
Birthday paradox, 273–276
Bit complexity, 45
Bitwise Xor, 73, 96, 223, 464, 681
Bleichenbacher's attack against PKCS #1 v1.5, 462
Blind signatures, 580
Blinder, 580
Blinding, 459
Blinding function, 581
Block cipher, 26, 181
key-iterated, 189
Block length, 26, 181
Block size, *see* block length
Blum integer, 150, 477
Blum prime, 124, 149
Blum–Blum–Shub PRG, 154–163, 557
Blum–Williams function, 150, 477
Blum–Williams permutations, 153
Bob, xxi

Boneh–Franklin IBE scheme, 609–612
  BasicIdent, 610
  FullIdent, 611
Boneh–Rabin encryption scheme, *see* Rabin-SAEP$^+$
Bounded-error probabilistic polynomial time, *see* $\mathcal{BPP}$
$\mathcal{BPP}$, 59
Byte, 681

C
CA, 584
  root, 586
Caesar cipher, 1
Carmichael lambda function, 433, 437, 469
Carmichael number, 298–301
Cassels' theorem, 642
CBC mode, *see* modes of operation
CBC-MAC, 240
CCA secure, 177, 216, 428, 502, 509, 529, 677
CCA1 attack, 428
CCA1 secure, 467
CCA2 attack, 428
CCRA, 532
CDH problem, 406, 609
Certicom ECC Challenge, 664
Certificate, 583
  X.509, 584
Certificate revocation, 587
Certificate revocation list, 587
Certification Authority, *see* CA
CFB mode, *see* modes of operation
Chain of trust, 586
Challenge ciphertext, 170, 173, 174, 176, 425–427, 596
Chaum's protocol, 581
Chi-squared goodness-of-fit test, 22
Chinese remainder isomorphism, 78
Chinese remainder theorem, 76, 87, 477
  complexity, 77
Chord-tangent method, 620
Chosen ciphertext attack, xxvi
Chosen message attack, 237
Chosen plaintext attack, xxv
Ciphertext, xxii
Ciphertext integrity, 251
Ciphertext-only attack, xxv
CMAC, 242–246
Cocks IBE scheme, 597–606
Collision, 260
Common modulus attack, 457–459
Composite number, 40
Compression function, 264
Computational composite residuosity assumption, 532
Computational DH problem, *see* CDH problem
Computational Diffie–Hellman assumption, 406
Conditional probability, 37
Confusion, 30
Congruence, 73
Coppersmith's short pad attack, 460
Co$\mathcal{RP}$, 59
Counter, 205
CPA advantage, 175
CPA secure, 175–178, 203, 205, 212–216, 426, 454, 499, 677
CPP, 315
Cramer–Shoup encryption scheme, 507–525
Cramer–Shoup signature scheme, 578–580
CRT, *see* Chinese remainder theorem
Cryptanalysis, xxiv
Cryptography, xxi
Cryptology, xxiv
CTR mode, *see* modes of operation
Cyclic, 69
Cycling attack, 438

D
Data encapsulation mechanism, 430
DCRA, 533
DDH problem, 406
  easy, 407
  hard, 407
Decisional composite residuosity assumption, 533
Decisional composite residuosity problem, 533
Decisional DH problem, *see* DDH problem
Decisional Diffie–Hellman assumption, 406
Decryption oracle, 176
Delegation of trust, 586
DEM, *see* data encapsulation mechanism
DES, 187
  challenge, 188
  cryptanalysis, 188
DES Cracker, 188
DH problem, 406
  bilinear, 609
  computational, 406, 609
  decisional, 406, 608
  gap, 678
  hash, 678
DH protocol, 405, 675
Dictionary attack, 24

D (*cont.*)
Diffie–Hellman key agreement, *see* DH protocol
Diffie–Hellman problem, *see* DH problem
Diffie–Hellman protocol, *see* DH protocol
Diffusion, 29, 202
Digital signature, xxiii, 262, 277, 402, 404, 539
Digital Signature Algorithm, *see* DSA
Digital signature scheme, *see* signature scheme
Digital Signature Standard, 551
Direct product, 78
Disavowal protocol, 582
Discrete logarithm function, 147
Discrete logarithm problem, 373, 405
  baby-step giant-step algorithm, 374–378
  elliptic curve, 650
  generic algorithms, 373, 614
  index calculus method, 387–399
  lambda method, 380
  Pohlig–Hellman algorithm, 384–387
  Pollard's rho method, 378–384
Distinguisher, 140
DLP, *see* discrete logarithm problem
Double-and-add method, 643
DSA, 550–562

E
Eavesdropper, 166–175, 425 , 427, 430
ECB mode, *see* modes of operation
ECC, 613
  Certicom Challenge, 664
  SEC 1 standard, 665
ECDLP, 650–664
  FR reduction, 660
  generic algorithms, 651–657
    Pohlig–Hellman algorithm, 653
    Pollard's rho method, 651
  hardness, 656–665
  MOV reduction, 660
  record, 655
  reduction attacks, 657–664
  rho method, 651–653
  Semaev, Satoh–Araki, Smart reductions, 662
ECDSA, 667–675
ECIES, 675
ECM, 371, 613
ECPP, 315
Edwards curves, 678
Elgamal encryption scheme, 497–507
Elgamal signature scheme, 545
Elliptic curve, 615
  abelian group, 624
  anomalous, 662
  complex multiplication method, 663
  counting points
    Schoof's algorithm, 638
    SEA algorithm, 638
  cryptography, *see* ECC
  digital signature algorithm, *see* ECDSA
  discrete logarithm problem, *see* ECDLP
  discriminant, 616
  domain parameters, 665
  Edwards coordinates, 678
  embedding degree, 660
  encryption, 675–678
  integrated encryption scheme. *see* ECIES
  ordinary, 639
  P-256, 671
  point addition, 618–624
  point addition formulas, 623
  point at infinity, 615
  point compression, 665
  quadratic twist, 634
  rational points, 615
  scalar multiplication, 644
  supersingular, 639
  tangent at P, 617
  Weierstrass equation, 615
Elliptic Curve Cryptography. *see* ECC
Elliptic Curve Method. *see* ECM
EME-OAEP
  decoding, 470
  encoding, 469
EMSA-PSS, 469, 573
Encrypt-and-Mac, 248
Encrypt-then-Mac, 250
Encryption oracle, 174
Encryption scheme, xxii
  asymmetric, *see* public-key
  homomorphic, 451, 525
  hybrid, 429
  non-malleable, 177, 454
  private-key, 167–179
  public-key, 421–437
  stateful, 173
  symmetric, *see* private-key
Enigma machine, 33
Ephemeral key, 549
Euler ϕ function, 80
Euler's criterion, 118, 480
Euler's theorem, 80
Eve , xxi
Event, 36
Existentially unforgeable, 237, 542, 593
Expected value, 38
Experiment

authentication CMA unforgeability, 237
CCA indistinguishability
identity-based, 595
private-key, 177
public-key, 427
ciphertext integrity, 250
CPA indistinguishability
private-key, 174
public-key, 426
CPA one-wayness, 425
DDH, 407
eavesdropping indistinguishability, 170
eavesdropping multiple-encryption, 172
factoring, 151
FDH signature unforgeability, 564
IB signature CMA unforgeability, 592
inversion, 145
key agreement, 404
pseudo-random function, 185
QR, 526
quadratic residuosity, 526
RSA, 433
signature CMA unforgeability, 542
square root computation, 151

F
Factor base, 338
Factor base factoring, 336–355
Factoring assumption, 151–155, 434
Fail-stop signatures, 582
FDH signatures, 563–566
Federal Information Processing Standard, *see* FIPS
Fermat prime, 439
Fermat test, 294–297
Fermat's factoring method, 328–336
complexity, 329
Fermat's little theorem, 81
Fibonacci numbers, 64
Field, 73
algebraically closed, 616
characteristic, 96
finite, 95, 108
prime, 95
FIPS, 187
First witness, 310
Flexible RSA problem, 578
Forger, 541, 564
Forgery, 541, 548, 565, 568
existential, 542, 544, 545, 548, 582
selective, 542, 544, 547
universal, 542
FR reduction, 660
Frequency analysis, 11
Frobenius trace, 635
Fujisaki–Okamoto transformation, 601, 611
Full Domain Hash signatures, *see* FDH signatures
Fully homomorphic encryption scheme, 536
Fundamental theorem of arithmetic, 40

G
Galois counter mode, *see* GCM
Gap Diffie–Hellman problem, 678
GCM, 253–258
GCTR, 255
Generic group, 656
Generic group model, 678
Geometric distribution, 38
GGNFS, 370
GHASH, 254
GMAC, 254, 258–259
GMP, 314
GNU multiple-precision bignum library, *see* GMP
Goldwasser–Micali encryption scheme, 526–528
Greatest common divisor, 61
Group, 68
abelian, 68
homomorphism, 69
order, 69
quotient, 71
Group generating algorithm, $\mathbf{Gen}_G$, 406

H
Hard-core predicate, 148, 153
Hash Diffie–Hellman problem, 678
Hash function, 259
collision resistant, 262, 508, 548
ε-almost universal, 253
ε-almost Xor universal, 253
keyed, 253
MD5, 271, 276, 277
preimage resistant, 261, 547
second preimage resistant, 261
SHA-256, 266–270, 284, 469, 471, 493, 515, 546, 603, 667, 670
SHA-1, 261, 265, 271, 276, 284, 551, 554
SHA-2 family, 264, 265, 551
universal, 252, 508
weak collision resistant, 261
Hash oracle, 564–566

H (*cont.*)
Hash-based MAC, *see* HMAC
Hash-then-Decrypt, 546
Hash-then-Sign, 546
Hashed Elgamal signatures, 548
Hashed RSA signatures, 547
Hasse interval, 635
Hasse's theorem, 635
Håstad's broadcast attack, 455–457
Hill cipher, 25
HMAC, 270–272, 675
HMAC-SHA-256, 271
Homomorphic encryption, 525, 529

I
I2OSP, 471
IBC, 589
IBE, 589
  hierarchical, 612
IBE scheme, 591–592
IBM, 187
IBS scheme, 592–595
Ideal lattice, 537
Identity-based cryptography, *see* IBC
Identity-based encryption, *see* IBE
Identity-based encryption scheme, *see* IBE scheme
Identity-based signature scheme, *see* IBS scheme
IFP, *see* integer factorization problem
IND-CCA secure, *see* CCA secure
IND-CCA2 secure, 428, 467
IND-CPA secure, *see* CPA secure
IND-EAV security, 172
IND-ID-CCA secure, 596, 611
IND-ID-CPA secure, 601, 611
Index calculus, 554
Index of a subgroup, 71
Index of coincidence, 17
Indifferent CCA attack, *see* CCA1 attack
Indistinguishable encryptions, 171–177, 426–428
Initialization vector, 204
INT-CTXT, *see* ciphertext integrity
INT-PTXT, *see* plaintext integrity
Integer
  b-adic expansion, 41
  composite, 40
  digits, 41
  length, 41
  positional representation, 40
  prime, 40
Integer factorization problem, 48, 322
  decision problem, 49
  factor base method, 336–355
  factoring assumption, 151–155
  Fermat's method, 328–335
  number field sieve, 368–371
  Pollard's rho method, 325–328
  quadratic sieve, 365–367
  search problem, 48
  trial division, 323
Invalid signature, 540
ISO standard, 270
ISO/IEC 8859 standard, 279
ISO/IEC 9797-2 standard, 270
IV, *see* initialization vector

J
Jacobi symbol, 119, 480, 598

K
Kasiski test, 16, 25
KDF, *see* key derivation function
KEM, *see* key encapsulation mechanism
KEM/DEM scheme, 430
Kerckhoffs' principle, xxiv
Key agreement protocol, 404
  DH protocol, 405
  secure in the presence of an eavesdropper, 404
Key derivation function, 413, 675
Key encapsulation mechanism, 430
Key generation center, 590
Key-only attack, 541
Keyed function, 184
Keyed permutation, 184
Known plaintext attack, xxv
Kocher's timing attack, 459
Korselt's criterion, 299

L
L-notation, 47
Lagrange's theorem, 71
Lamé's Theorem, 65
Lattice, 537
Lattice-based cryptography, 537
Leading coefficient, 99
Least non-negative residue, 62, 74
Legendre symbol, 117, 407
Line at infinity, 620
Linear dependency, 341

Little-endian, 86, 255, 682
Lunchtime attack, *see* CCA1 attack
Lutz–Nagell theorem, 624

M
MAC, 235–259
  tag, 235
  with unique tags, 249
Mac-then-Encrypt, 249
Magma, 639
Malleability, 454
  benign, 677
Mallory, xxi
Man-in-the-middle attack, 409, 414, 418
Maple implementations/examples
  768-bit factorization account, 370–371
  AES encryption and decryption for lists of bytes, 220–221
  AES forward and inverse cipher functions, 219–220
  AES inverse S-box, 199–200
  AES `KeyExpansion`, 197–199
  AES operations, 217–219
  AES S-box, 192–194
  AES-based modes OFB and CTR, 222–231
  baby-step giant-step method, 376–378
  binary exponentiation variants, 85–89
  birthday attack, 277–280
  bitwise Xor, 96–97
  Blum–Blum–Shub example, 156–158
  brute-force attack against the pseudoran-dom one-time pad, 163–164
  Caesar cipher, 3–5
  Carmichael numbers, 289–301
  chosen ciphertext attack against plain RSA, 452–455
  CMAC, 242–246
  Cocks IBE scheme, 597–607
  computing a modular square root example, 126–129
  computing a primitive root modulo a prime, 92–94
  conversion functions, 681–687
  Cramer–Shoup encryption scheme, 515–522
  cryptanalysis of the Caesar cipher, 5
  cryptanalysis of the Hill cipher, 29–33
  cryptanalysis of the Vigenère cipher, 16–25
  Diffie–Hellman protocol example, 398–399
  DSA, 555–562
  ECDLP record account, 655–657
  ECDSA, 670–675
  Elgamal example, 504–507
  elliptic curve chord-tangent method examples, 619–622
  elliptic curve elementary computations, 628–640
  elliptic curve group, 641–649
  elliptic curve rho method and Pohlig–Hellman algorithm, 652–654
  Eratosthenes sieve, 51–53
  Euclidean algorithm variants, 62–65
  factor base factoring method, 342–355
  factoring an RSA modulus n from $\phi(n)$, 434–435
  Fermat test and pseudoprimes, 294–298
  Fermat's factoring method, 331–335
  field of 256 elements, 111–113
  GCM, 254–258
  generation of pseudo-random primes, 321–322
  GMAC, 258–259
  Håstad's broadcast attack against plain RSA, 455–457
  Hill cipher, 27–29
  HMAC-SHA-256, 266, 271–272
  index calculus method, 391–398
  irreducible polynomials over finite fields, 150–111
  man-in-the-middle attack against the Diffie–Hellman protocol, 414–418
  Miller–Rabin test, 311–313
  NIST's AES known answer tests, 221–222
  one-time pad does not provide authentica-tion, 233–235
  one-time pad example, 166
  Paillier encryption scheme, 534–536
  plain Rabin encryption, 482–485
  plain RSA, 442–449
  Pohlig–Hellman algorithm, 385–387
  Pollard's rho factoring method, 327
  Pollard's rho method for discrete logarithms, 382–384
  PSS signatures, 572–578
  PSW test, 315
  QS/NFS complexity comparison, 368–370
  quadratic sieve factoring algorithm, 360–366
  Rabin-SAEP$^+$, 491–497
  RSAES-OAEP, 471–476
  searching for primes, 491–492
  searching for squares, 286
  SHA-256, 266–270
  speed comparison of Cramer–Shoup/Rabin-SAEP$^+$/RSAES-OAEP implementations, 522–525
  strong probable prime test, 301–306

M (*cont.*)
substitution cipher, 7–9
the (pseudo-random) one-time pad, 158–162
trial division, 53
trial division as a factoring algorithm, 323–325
trial division timing, 54–57
Vigenère cipher, 14–16
Maple package, 3
Mask generating function, 469
Master key, 591
Master private key, 591
Master public key, 591
MD5, 271, 276–277
Merkle–Damgård construction, 263
Merkle–Hellman knapsack cryptosystem, 408
Mersenne prime, 324
Message attack, 541
Message Authentication Code *see* MAC
Message expansion, 499
Miller–Rabin test, 306–322, 556
MIRACL, 639
Modes of operation, 203–216
CBC mode, 204
CFB mode, 206
CTR mode, 210
ECB mode, 203
GCM, 253
GMAC, 258
OFB mode, 207
Modular exponentiation, 83
Monier–Rabin theorem, 307
Moore's law, 9
MOV reduction, 608, 659
MPQS, *see* quadratic sieve
Msieve, 370
Multiple polynomial quadratic sieve, *see* MPQS
Multiple root, *see* multiple zero
Multiple zero, 103
Multiplicative function, 81
Multiplier, 347

N
nth residue modulo n^2, 531
nth residuosity class, 531
Negligible function, 140, 171, 424, 426
Negligible success probability, 139
NFS, *see* number field sieve
NIST, 187, 189, 203, 242, 254, 265, 270, 277, 551, 553, 646, 656
No-message attack, 541
Non-adaptive CCA attack, *see* CCA1 attack
Non-deterministic polynomial time, *see* $\mathcal{NP}$
Non-repudiation, 402, 540
Nonce, 207
Nonsingular curve, 616
$\mathcal{NP}$, 48
NSA, 187
NTRU encryption scheme, 538
Number field sieve, 368–373
general, 368

O
OAEP, 463
OAEP encoding, 469
OFB mode, *see* modes of operation
OMAC, 242
One-time pad, 136, 159–169
One-way against chosen plaintext attacks, *see* OW-CPA secure
One-way function, 144, 405, 543
family, 146
One-way permutation, 148
Optimal Asymmetric Encryption Padding, *see* OAEP
Oracle, 593, 596
Oracle access, 174–178, 427, 428
Order
of a group, 69
of a group element, 69
Order notation, *see* asymptotic notation
OS2IP, 471
Oscar, xxi
OW-CCA secure, 454
OW-CPA secure, 426, 431, 454, 480, 499, 611

P
$\mathcal{P}$, 48
$\mathcal{P}$ versus $\mathcal{NP}$, 46
$p-1$ method, 439
Padding, 203
10^i, 203
Paillier encryption scheme, 529–536
Pairing, 607, 614
Tate, 660
Weil, 660
Perfect secrecy, 132
Petabyte, 183
PGP, 587
PKCS #1 v1.5, 462
PKCS #1 v2.1, 468, 569
PKG, 590–599
PKI, 584

Plain Rabin encryption scheme, 479
Plain RSA, 436–459
Plain RSA signature scheme, 544
Plaintext , xxi
Plaintext awareness, 466
Plaintext integrity, 251
PlayStation 3 console, 656
PNT, *see* prime number theorem
Pollard's lambda method, 380
Pollard's rho method, 325–328
Polynomial, 97
  degree of, 99
  greatest common divisor, 102
  irreducible, 105
  monic, 99
  zero of, 103
Polynomial ring, 100
PPT adversary, 142, 143, 169, 178
PPT algorithm, 139
Predictor, 143
PRG, *see* pseudo-random generator
Primality test, 286
Prime gap, 292
Prime number, 40
Prime number theorem, 45, 290
  for arithmetic progressions, 293
Primitive root, 89
Private key, 403
Private key delegation, 597
Private Key Generator, *see* PKG
Probabilistic Signature Scheme, *see* PSS
Probability distribution, 36
Probability space, 36
Protocol transcript, 404
Pseudo-random function, 185, 239
Pseudo-random generator, 140, 148
  stretch, 141
Pseudo-random one-time pad, 142, 154–164
Pseudo-random permutation, 185
  strong, 186
Pseudoprime, 295
  Fibonacci, 314
  strong, 301
PSS, 566–578
PSW test, 315
Public key, 403
Public key revocation, 597
Public-Key Infrastructure, *see* PKI

Q

QR problem, *see* quadratic residuosity problem
QS, *see* quadratic sieve
Quadratic non-residue, 115
Quadratic reciprocity, 121
Quadratic residue, 115, 411
Quadratic residuosity problem, 480, 526
  is hard, 526, 599, 601
Quadratic sieve, 355–368
  complexity, 357
  large prime variation, 366
  MPQS, 367
  self-initializing, 367
Quantum computer, 61, 371, 399, 537
Qubit, 371

R

Rabin encryption, 477
  plain, 477–485
Rabin functions, 153
Rabin-SAEP$^+$, 487–497
Random oracle, 429, 554, 563–565
Random oracle model, 429, 464, 554, 568, 601, 678
Random self-reducible, 60, 408, 434
Random variable, 38
Randomized polynomial time, *see* $\mathcal{RP}$
Reduction
  loose, 467
  tight, 467
Related-key attack, 202
Relation (over a factor base), 341
  partial, 366
Relatively prime integers, 61
Replay attack, 237
Residue class ring, 73
Riemann hypothesis, 291
  extended, 293
Ring, 72
  homomorphism, 72
  integral domain, 72
  polynomial, 100
$\mathcal{RP}$, 59
RSA, 431–476
  decryption exponent, 436
  encryption exponent, 436
  modulus, 436
  OAEP, 463–476
  padded, 460
  plain, 436–457
RSA assumption, 431, 434, 437
RSA Labs, 431
RSA problem, 431, 433, 437
  flexible, 578
  is hard, 433, 564
RSA-OAEP, 463–476

R (*cont.*)
RSA768, 370
RSAES-OAEP, 468, 523
RSASSA-PSS, 569, 575

S
Safe prime, 320, 411, 508, 578
Sage, 638
Sample space, 35
Schnorr's signature scheme, 550
Schoof's algorithm, 638
SEA algorithm, 638
Secure, xxiii
Secure Hash Algorithm, 265
Secure Hash Standard, 265
Secure signature scheme, 542
Security parameter, 145, 168, 175, 236, 404, 422–427, 464, 487–497, 508, 540, 542, 566, 578, 592, 595, 609, 610
Security through obscurity, xxv
Semantic security, 170, 171
Sender, 580
Session key, 429
SHA-256, 265–269, 276, 284, 469, 471, 515, 546, 603, 667, 670
  message schedule, 266
SHA-1, 261, 264, 265, 271, 276, 284, 551, 553
SHA-2, 264, 265, 276, 551
SHA-3, 276
Shannon's theorem, 134, 498
Shift cipher, *see* Caesar cipher
Shor's quantum factoring algorithm, 372
Side-channel attack, 459
Signature scheme, 540
  blind, 580
  fail-stop, 582
  undeniable, 582
  with appendix, 569
  with message recovery, 569
Signer, 539, 580, 582
Signing algorithm, 540
Signing oracle, 547, 564, 565
Singular point, 616
Smooth curve, 616
Smooth point, 616
Smoothness bound, 340
Sophie Germain prime, 316, 320, 411, 508, 515, 578
Spoofing attack, 235
SS-2-IBS, 593
Stirling's formula, 183
Stream cipher, 165
Strong liar, 306
Strong one-way function, 145
Strong prime, 439
Strong probable prime test, 301–305
Strong pseudo-random permutation, 186
Strong RSA assumption, 578
Strong unforgeability, 246
Subexponential function, 47
Subgroup, 69
Substitution cipher, 7
  monoalphabetic, 12
  polyalphabetic, 12
SUF-CMA secure, 246
Symmetric group, 69

T
Tate pairing, 660
Textbook RSA, *see* plain RSA
The Gold Bug, 9
Tonelli's algorithm, 125
Total break, 541
Totient function, *see* Euler ϕ function
Trace, *see* Frobenius trace
Trapdoor one-way, 422, 477
Trial division, 323

U
UF-CMA secure, 238, 246, 248, 542, 564, 568, 580, 593, 594
Unary notation, 143
Unblinding function, 581
Unconditional security, *see* perfect secrecy
Undeniability, 540
Undeniable signatures, 582
Unicity distance, 5, 135
Uniform distribution, 36
Universal hash function, 252
Unpredictable function, 144

V
Valid signature, 540
Verification algorithm, 540
Vernam cipher, see one-time pad
Vigenère cipher, 12
  cryptanalysis, 15–25
Vigenère table, 13
Von Neumann's coin flipping trick, 142

W
Weakly plaintext aware, 466
Web trust model, 587
Weil pairing, 660
Wilson's theorem, 294
Witness (to compositeness), 303

X
X.509, 584

Y
Yao's theorem, 144

Z
Zero divisor, 39, 72
Zero-error probabilistic polynomial time, *see* $\mathcal{ZPP}$
$\mathcal{ZPP}$, 59