

Explaining Graph Neural Networks for Vulnerability Discovery

Tom Ganz
tom.ganz@sap.com
SAP SE – Security Research
Germany

Martin Härterich
martin.haerterich@sap.com
SAP SE – Security Research
Germany

Alexander Warnecke
alexander.warnecke@tu-bs.de
TU Braunschweig
Germany

Konrad Rieck
konrad.rieck@tu-bs.de
TU Braunschweig
Germany

Abstract

Graph neural networks (GNNs) have proven to be an effective tool for vulnerability discovery that outperforms learning-based methods working directly on source code. Unfortunately, these neural networks are uninterpretable models, whose decision process is completely opaque to security experts, which obstructs their practical adoption. Recently, several methods have been proposed for explaining models of machine learning. However, it is unclear whether these methods are suitable for GNNs and support the task of vulnerability discovery. In this paper we present a framework for evaluating explanation methods on GNNs. We develop a set of criteria for comparing graph explanations and linking them to properties of source code. Based on these criteria, we conduct an experimental study of nine regular and three graph-specific explanation methods. Our study demonstrates that explaining GNNs is a non-trivial task and all evaluation criteria play a role in assessing their efficacy. We further show that graph-specific explanations relate better to code semantics and provide more information to a security expert than regular methods.

CCS Concepts

• **Computing methodologies** → **Neural networks**; • **Security and privacy** → **Software and application security**.

Keywords

Machine Learning, Software Security

ACM Reference Format:
Tom Ganz, Martin Härterich, Alexander Warnecke, and Konrad Rieck. 2021. Explaining Graph Neural Networks for Vulnerability Discovery. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security (AISec '21), November 15, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3474369.3486866

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AISEC'21, November, 2021, Seoul, South Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8657-9/21/11…$15.00
https://doi.org/10.1145/3474369.3486866

1 Introduction

Graph neural networks (GNN) belong to an emerging technology for representation learning on geometric data. GNNs have been applied successfully to a variety of challenging tasks, such as the classification of molecules [11] and protein-protein interactions [20]. Compared to other neural network architectures, GNNs can effectively make use of graph topological structures and thus constitute a versatile tool for analysis of complex data.

Because of these capabilities, GNNs have also been applied to source code to identify security vulnerabilities [39] and locate potential software defects [2]. Source code naturally exhibits graph structures, such as abstract syntax trees, control-flow structures, and program dependence graphs [10, 32], and thus is a perfect match for analysis with GNNs. Previous work could demonstrate that GNNs perform better on identifying security vulnerabilities than classical static analyzers and learning-based methods that operate directly on the source code [39]. Consequently, these neural networks are considered the basis for new and intelligent approaches in software security and engineering.

The efficacy of GNNs, however, comes at a price: neural networks are black-box models due to their deep structure and complex connectivity. While these models produce remarkable results in lab-only experiments, their decisions are opaque to security experts, which hinders their adoption in practice. Identifying security vulnerabilities is a subtle and non-trivial task. Moreover, there are even theoretical bounds as there cannot be a general approach to vulnerability detection by Rice's theorem [15], and therefore interaction with human experts is indispensable when searching for vulnerabilities. For them it is pivotal to understand the decision process behind a method to analyze its findings and decide whether a piece of code is vulnerable or not. Hence, any method for their discovery must be *interpretable*.

One promising direction to address this problem is offered by the field of *explainable machine learning*. A large body of recent work has focused on explaining the decisions of neural networks, including feed-forward, recurrent, and convolutional architectures. Similarly, some specific methods have been proposed that aim at making GNNs interpretable. Still, it is unclear whether and which of the methods from this broad field can support and track down decisions in vulnerability discovery. In this paper we address this problem and establish a link between GNNs and vulnerability discovery by posing the following research questions:

(1) *How can we evaluate and compare explanation methods for GNNs in the context of vulnerability discovery?*
(2) *Do we need graph-specific explanation methods, or are generic techniques for interpretation sufficient?*
(3) *What can we learn from explanations of GNNs generated for vulnerable and non-vulnerable code?*

To answer these questions, we present a framework for evaluating explanation methods on GNNs. In particular, we develop a set of evaluation criteria for comparing graph explanations and linking them to properties of source code. These criteria include general measures for assessing explanations adapted to graphs as well as new graph-specific criteria, such as the contrastivity and stability of edges and nodes. Based on these criteria, we are able to draw conclusions about the quality of explanations and gain insights into the decisions made by GNNs.

To investigate the utility of our framework, we conduct an experimental study with regular and graph-specific explanation methods in vulnerability discovery. For regular approaches we focus on white-box methods, such as CAM [34] and Integrated Gradients [29], which have proven to be superior to black-box techniques in the security domain [30]. For graph-specific approaches we consider GNNExplainer [35], PGExplainer [19], and Graph-LRP [24], which all have been specifically designed to provide insights on GNNs. Our study shows that explaining GNNs is a non-trivial task and all evaluation criteria are necessary to gain insights into their efficacy. Moreover, we show that graph-specific explanations relate better to code semantics and provide more information to a security expert than regular methods.

2 Neural Networks on Code Graphs

We start by introducing the basic concepts of *code graphs*, *graph neural networks*, and their application in *vulnerability discovery*.

Code graphs. We consider directed graphs $G = (V, E)$ with vertices V and edges $E \subseteq V \times V$. Nodes and edges can have attributes, formally defined as (keyed) maps from V or E to a feature space. It is well known that source code can be modeled inherently as a directed graph [1, 2, 5], and we refer to the resulting program representation as a *code graph*. In particular, the following code graphs have been widely used for finding vulnerabilities:

AST An abstract syntax tree (AST) describes the syntactic structure of a program. The nodes of the tree correspond to symbols of the language grammar and the edges to grammar rules producing these symbols.

CFG A control flow graph (CFG) models the order in which the statements of a program are executed. Therefore, each node is a set of statements and edges are directed and labeled with flow information and conditionals.

DFG A data flow graph (DFG) models the flow of information in a program. A node denotes the use or declaration of a variable, while an edge describes the flow of data between the declaration and use of variables.

PDG The program dependence graph (PDG) proposed by Ferrante et al. [13] describes control and data dependencies in a joint graph structure. It was originally developed to slice a program into independent sub-programs.

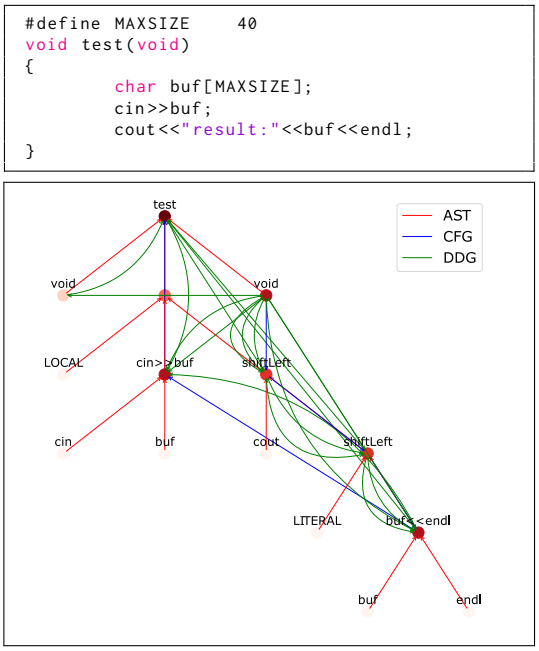


Figure 1: Source code and CPG of a simplified vulnerability. More saturated red on nodes in the CPG corresponds to more attributed relevance.

Based on these classic representations, combined graphs have been developed for vulnerability discovery. The code property graphs (CPG) by Yamaguchi et al. [32], for example, is a combination of the AST, CFG and PDG. Likewise, the code composite graph (CCG) encodes information from the AST, DFG and CFG [7]. In the remainder, we use these two combined code graphs for our experiments, as they have proven to be effective and capture semantics from multiple representations. As an example, Figure 1 shows a CPG of a simple vulnerability.

Graph neural networks. GNNs are a model of deep learning and realize a prediction function $f : G(V, E) \to \mathbb{R}^d$ [23] that can be used for classification and regression. The most popular GNN types belong to so-called message passing networks (MPNs) [31] where the prediction function is computed by iteratively aggregating information from neighboring nodes.

In the simplest GNN layer the aggregation is the sum of the neighboring feature vectors and the update forwards the aggregated feature vector per node to a multilayer perceptron (MLP). The prediction $\hat{y}$ is then given by $\hat{y} = \hat{A}XW$, where $\hat{A}$ is the normalized adjacency matrix, X the initial feature matrix, and W a weight matrix which we seek to optimize [14]. Kipf et al. coined the term *graph convolutional network* (GCN) for this specific layer. GNNs can be further extended by incorporating other graph features as well as adding pooling and readout layers. Finally, other architectures like GRUs or LSTMs can be used to update the node embeddings, too, resulting in *gated graph neural networks* (GGNNs) [17].

GNNs for vulnerability discovery. Due to the rich semantics captured by code graphs, GNNs have been applied in a series of work for vulnerability discovery. In particular, we focus on the approaches

Devign [39], ReVeal [8] and BGNN4VD [7] that can be considered state of the art and are reported to provide promising results in the respective publications.

Devign uses CPG graphs with additional edges connecting leaf nodes with their successors. These edges are called natural code sequence (NCS) and represent the natural order of the statements. The model constitutes a six-time-step GGNN. The final embedding of the sixth iteration and the initial node features are both forwarded through a novel pooling layer:

$$\sigma(\cdot) = \textsc{MaxPool}(\text{ReLu}(\textsc{Conv}(\cdot)))$$

The Conv layer is a regular 1D-convolution and is followed by a ReLu activation and max-pooling. Afterwards, the output is forwarded through an MLP. The output of both passes is multiplied pairwise and the prediction is the averaged result [39].

ReVeal is a model using the regular CPG. The pre-processing step includes a re-sampling strategy and the model consists of an eight-time-step GGNN followed by a sum aggregation and a final MLP as a prediction layer. The training involves a triplet loss incorporating binary cross-entropy, L2-regularization, and a projection loss minimizing resp. maximizing the vector distances between similar or different classes [8].

BGNN4VD differs from the other two GNN models as it uses bidirectional CCGs. The model uses an eight-time-step GGNN with the same pooling operator as Devign but followed by an MLP as a prediction layer [7].

3 Explaining Machine Learning

Vulnerability discovery using machine learning has made remarkable progress over the last years. The proposed systems, however, are opaque to practitioners and it is unclear how they arrive at their decisions. This lack of transparency obstructs their deployment in the field of security and creates a gap between research and practical demands. *Explanation methods* (EM) for machine learning have the potential to alleviate this problem and help to gain insights into the capabilities of learning-based security systems. Explainability methods turn the decision of a machine learning model into a transparent and more likely to be human interpretable result. Since all decisions depend only on the input signal of a model, conventional explainability methods try to correlate the input features to the final output. However, in the domain of GNNs, we are working with graph signals that do not only depend on features in a vector space but on discrete topological structures.

In the following, we introduce common explanation methods as well as approaches specifically designed to explain GNNs.

Graph-agnostic explanation methods. There exist a variety of general techniques for explaining learning models. For this paper, we focus on nine common approaches and adapt them for explaining GNNs. Since a node is the primitive element of code graphs, we seek explanations that indicate the relevance of nodes for the discovery of vulnerabilities. As general EMs explain only based on features, we propagate the corresponding relevance scores of edges to adjacent nodes, such that all methods yield node-level explanations.

Class Activation Maps (CAM) were originally designed for explaining Convolutional Neural Networks (CNNs). Deep layers tend to learn semantically meaningful features and CAM scales these features from the last hidden layer with the weight connecting them to the desired output node in order to generate a feature-wise explanation [38].

Linear Approximation also known as Gradient $\odot$ Input calculates for each input feature its linearized contribution to the classification output. Technically, this is the element-wise multiplication of the desired output's gradient with respect to the input node feature with the corresponding input activation [26].

GradCAM applies the idea of Linear Approximation to the intermediate activations of GNN layers instead of the input activations. This yields an advantage similar to CAM since more relevant information is aggregated. In this work, we take the GradCAM variant where activations of the last convolutional layer before the readout layer are used [25].

SmoothGrad averages the node feature gradients on multiple noisy inputs and compared to simple gradients yields noise-robust explanations [27]. We use noise sampled from a normal distribution ($\sigma = 0.15$) with 50 samples. These parameters were optimized for the descriptive accuracy of the ReVeal model.

Integrated Gradients (IG) improves the Linear Approximation by referring to a counterfactual baseline input G' and then using (1) gradients that are averaged along a straight path to the actual input G, and (2) the difference of the input activations between G and G' instead of the absolute activation [29]. We set all node features to zero for G' to achieve a very low base prediction score.

Gradient or Saliency Method simply measures the change in the prediction with respect to the change of the input by calculating the corresponding gradients. Although simple and effective it is known that the generated relevance maps are oftentimes noisy [29].

Guided Backpropagation (GB) clips the negative gradients to have a positive influence during backpropagation. This technique yields explanations that concentrate on features having an *excitatory* effect on the output prediction [28].

Layerwise Relevance Propagation (LRP) creates relevance maps by propagating the prediction back to the input such that a conservation property holds with respect to the total relevance scores of each layer. We tested the (α, β) and ϵ-rule [16] but use only the ϵ-rule since it yields better results in our experiments.

Excitation Backpropagation (EB) calculates the relative influence of the activations of the neurons in layer $l-1$ to one from layer l using backpropagation while only taking positive weights into account [37]. The gradients are normalized to sum up to 1 so that the output can be interpreted as the probability whether a neuron will fire or not given some input.

Graph-specific explanation methods. In addition to the nine general methods for explainable machine learning, we also consider three EMs that have been specifically designed for explaining GNNs. To realize a unified analysis of all methods in this paper, we adapt these graph-specific approaches, such that they also provide explanations on the level of nodes. In particular, we propagate relevance scores assigned to edges and walks to adjacent nodes in the code graphs, resulting in explanations similar to those of the graph-agnostic methods.

GNNExplainer is a black-box forward explanation technique for GNNs. Ying et al. [35] argue that general EMs fall short of incorporating graph topological properties and therefore develop this approach. For a given graph, GNNExplainer tries to maximize the *mutual information* (MI) of a prediction with respect to the prediction based on a variable discriminative subgraph S and a subset of node features. The subgraph that retains important edges is obtained by learning a mask that is applied to the adjacency matrix.
PGExplainer tackles the problem that the explanations for GNNExplainer have to be calculated for every individual graph instance. PGExplainer provides a global understanding of the inductive nature of the model by extracting relevant subgraphs S similar to GNNExplainer. Whereas GNNExplainer is not suitable for an inductive graph learning setting (cf. [19]) PGExplainer uses a so-called explanation network on a universal embedding of the graph edges to obtain a transferable version of the EM.
Graph-LRP is a method using higher order Taylor expansions to identify relevant walks over multiple layers of a GNN where the message propagation between nodes during training is considered as walks of information flow [24]. The relevance per walk is computed using a backpropagation similar to LRP for each node in the walk. Schnake et al. [24] argue that the information contained in these walks is richer compared to explanations generated by GNNExplainer or PGExplainer.

4 Evaluating Explanations of GNNs

It is evident from the previous section that a large arsenal of methods is readily available for explaining and understanding GNNs. However, the presented explanation methods considerably differ in how they characterize the decision process and derive explanations for a given input graph. As a result, it is unclear which methods are suitable for explaining the predictions of GNNs in vulnerability discovery and how the generated relevance maps relate to the semantics of code and security flaws.

To tackle this problem, we introduce a framework for evaluating explanation methods on GNNs. In particular, we build on previous work by Yuan et al. [36] and Warnecke et al. [30] who propose criteria for comparing explanation methods in security. As their work does not account for relational information and topological structure, we extend their criteria as well as introduce new ones, specifically designed for understanding how an EM characterizes nodes in code graphs.

Descriptive accuracy. To determine whether an EM captures relevant structure in a GNN, we remove a relative amount k% of the most relevant nodes from the input graph and calculate another forward pass of the model for each graph in the test set. The *descriptive accuracy (DA)* for k% is then given by the model's drop in accuracy (with respect to its original accuracy) averaged over the test data. The larger this drop is, the more relevant nodes are identified by the EM. The area under the DA curve is a single numerical quantity that summarizes its behavior, with a large area indicating a steep rise of the curve and thus an accurate explanation of the GNN's decision process.

The DA shares similarities with the fidelity measurement by Yuan et al. [36]. This measure uses thresholds on the relevance maps to evaluate the accuracy of the explanations. Instead, we first rank the nodes by relevance and then remove a fixed percentage to obtain the DA. Since there is often a high variance in the size of code graphs, we find this measure beneficial as it allows us to obtain relevant subgraphs for vulnerability localization.
Structural robustness. To measure the robustness of an input graph for a given EM we use the *remaining agreement* (RA). We compute the 10% most relevant nodes before and after perturbing the input graph by dropping its edges with a certain probability p and define the RA as the size of the intersection of these highly relevant nodes.

Our motivation comes from the desire to understand how susceptible an EM is against manipulations where an adversary tampers with the input such that the explanation method creates an arbitrary, meaningless explanation, for example, when GNN and EMs are used to assess code of unknown origin. In this context, we interpret structural robustness as the sensitivity of the model and the EM to still label the original relevant code parts despite an attacker trying to hide certain program semantics by altering control or data flow. Although structural robustness measures the stability of the EM against noise only, it provides an upper bound of the effort a potential attacker needs to successfully attack the EM.
Contrastivity. The descriptive accuracy and structural robustness provide a general view on the quality of an explanation for a GNN, yet they do not take into account the specifics of code analysis and vulnerability discovery.

To address this issue, we propose to measure the contrastivity of an explanation. This measure is calculated by comparing relevant statements for the vulnerable and non-vulnerable class. In taint-style-analysis [32], for example, identifying vulnerabilities can be approached by traversing all CPG edges that flow from user-controlled nodes to security-critical sinks (e.g. `fopen` or `memcpy`). Hence, we expect the explanations to differ for vulnerable and non-vulnerable samples in these paths. If, for instance, critical calls are completely absent in the AST, the model does not need to further look at the DFG and CFG edges.

We calculate the contrastivity of an EM using a histogram of the AST terminal and non-terminal (type) identifier of the 10% most relevant nodes. Then, we compute the chi-square distance of the normalized node histograms of negative and positive samples:

$$\chi^2(x, y) = \frac{1}{2} \sum_{i=1}^{d} \frac{(x_i - y_i)^2}{x_i + y_i}.$$

A higher difference indicates that the model takes more different node types into account when distinguishing between vulnerable or non-vulnerable samples, providing a more diverse view on the characteristics of the code.
Graph sparsity. An explanation method must stay concise during operation, since code graphs can become too large to be manually assessed by practitioners. An explanation method that marks hundreds of nodes in a code graph as relevant yields no practical benefit. To measure the conciseness of an explanation, we adapt the mass around zero (MAZ) measure [30] to GNNs: To this end, the relevance values of the nodes are normalized to be contained in the interval $[-1, 1]$ and then a cumulative distribution function $r \mapsto \int_{-r}^{r} h(x)dx$ of their *absolute values* is calculated. The larger the area under this curve is, the more relevance values of nodes are

close to 0 and hence of little influence. If all the relevance values are positive (resp. negative) then normalization is just division by the value with highest absolute value (i.e. $x_{\max}$ resp. $x_{\min}$), otherwise we use the projective transformation

$$x \mapsto \frac{(x_{\max} - x_{\min}) \cdot x}{(x_{\max} + x_{\min}) \cdot x - 2 \cdot x_{\max} \cdot x_{\min}}$$

with fixed point 0 mapping $x_{\min}$ to -1 and $x_{\max}$ to 1.

The area under the adapted MAZ provides us with a single numerical quantity that describes how concise an EM operates, where a high area indicates explanations with a sparse assignment of relevance values to nodes.

Stability. Some EMs are non-deterministic and do not provide identical results during different runs. This slight randomness can pose a problem for vulnerability discovery, where the differences between vulnerable and secure code is often nuanced and subtle. To account for this problem, we measure the stability in terms of standard deviation of the descriptive accuracy and sparsity over five runs. Note that only Smoothgrad, PGExplainer, GNNExplainer and Graph-LRP are non-deterministic, as they use randomly initialized weights or random sampling. The remaining graph-agnostic methods are deterministic by design and hence stable per definition.

Efficiency. Finally, the runtime performance of an explanation method should not drastically increase the time a security specialist needs for her traditional workflow. Especially for large and complex code graphs, it is crucial that explanations are generated in reasonable time, for instance, a few seconds. To reflect this requirement, we measure the average runtime of an EM per single graph. Note that the runtime in a practical setup also depends on details of the implementation and GNN model, and thus this criterion should only be used to provide an intuition of the performance rather than precise runtime numbers.

5 Experimental Study

After introducing our evaluation criteria, we are finally ready to empirically evaluate the performance of explanation methods on GNNs for vulnerability discovery. In particular, we consider the generic and graph-specific approaches (9+3) for explanations described in Section 3 on the three GNNs presented in Section 2.

5.1 Setup

We train Devign and ReVeal on graphs with vulnerabilities from C and C++ open-source software and BGNN4VD on a dataset containing vulnerabilities in open-source Java software. Some explainability algorithms have hyperparameters that need to be calibrated. We use Bayesian optimization to find suitable parameters for Integrated Gradients, SmoothGrad, GNNExplainer, PGExplainer and Graph-LRP. Finally, we calculate the area under curve for DA with $k \in \{1, 5, 10, 15, 30, 50, 75\}$, for the sparsity metric with interval sizes $r \in \{0.05, 0.1, 0.25, 0.5, 0.75, 1.0\}$ and for structural robustness with edge dropping probabilities $p \in \{0.005, 0.1, 0.2, 0.5, 0.75\}$. As a baseline in all experiments, we randomly generate explanations, where the relevance values for nodes are drawn independently from a uniform distribution.

Case studies. For our study, we consider three datasets and the corresponding GNNs as case studies. Each dataset consists of source code with and without security vulnerabilities. Table 1 shows an overview of the case studies and the reproduced performance of the three models. Mean and standard deviation of ten experiments, each with different stratified dataset 80/20 splits, are reported. We see a broad spectrum in the case studies' performances which is desirable, since we obtain insights in to what extent EMs depend on the underlying GNN model.

Case-Study	Accuracy	Precision	Recall	F1-Score
Devign	55.68±0.36	55.28±0.38	90.32±2.68	68.58±1.09
ReVeal	84.66±0.18	58.53±0.34	58.14±0.45	58.33±0.40
Vulas (BGNN4VD)	88.05±0.18	84.10±0.12	90.10±0.03	87.00±0.07

Table 1: Performance of all case studies for vulnerability discovery (our re-implementations).

Case study A: Devign. According to the original publication, the source code is transformed into a CPG using Joern[1] and enhanced with the NCS. Type information is label encoded. We replace the original gated graph neural network (GGNN) with six GCN message-passing networks, as this provides a slightly better model performance. We use L2-regularization during training with $\lambda_{L2} = 0.0001$ and a learning rate of 0.0001 for the Adam optimizer, since the original hyperparameters were not published. The model is originally trained to identify security vulnerabilities found in the projects FFmpeg and Qemu with excellent accuracy [39]. However, we are not able to reproduce the corresponding results and only attain a moderate F1-score.[2]

Case study B: ReVeal. In this case study, the dataset is composed of security vulnerabilities extracted from patches for the Chromium and Debian projects [8]. The code graphs are again extracted using Joern. We use L2-regularization with $\lambda_{L2} = 0.001$ and a learning rate of 0.001 for the Adam optimizer. Our accuracy (cf. Table 1) is on par with the original publication; however, we report a higher F1-score which could be due to different dataset splits.

Case study C: Vulas (BGNN4VD). As the third case study, we use a Java dataset refered to as *Vulas*[3] from Ponta et al. [21]. The dataset consists of manually curated CVEs mined from Java software repositories like Tomcat, Struts, and Spring. In contrast to memory-based vulnerabilities often found in C/C++ code, this dataset contains security issues like SQL injections, XXE vulnerabilities, directory traversals, or XSS injections. These vulnerabilities are linked to commits before and after the respective patches. We apply the BGNN4VD model and extract the CCG from each changed file in the commit both before and after the actual fix using the Fraunhofer-CPG[4]. This tools extracts graphs similar to the CCG used by Cao et al. [7]. Regarding this dataset, each Java file in a commit is merged into a single potentially disconnected graph. Furthermore, we add random Java files from the same repositories. We end up with a dataset composed of 1,000 vulnerable samples, 500 fixed samples and 500 randomly chosen benign samples. The model is trained

[1] https://github.com/joernio/joern
[2] Chakraborty et al. [8] report the same and, like them, we were not successful in contacting the authors.
[3] https://sabetta.com/post/vulas-dataset-released/
[4] https://github.com/Fraunhofer-AISEC/cpg

Criteria	Descriptive Accuracy			Structural Robustness			Contrastivity			Graph Sparsity		
Model	**Devign**	**ReVeal**	**Vulas**	**Devign**	**ReVeal**	**Vulas**	**Devign**	**ReVeal**	**Vulas**	**Devign**	**ReVeal**	**Vulas**
GNNExplainer	0.08 ±0.003	0.15 ±0.008	0.29 ±0.005	0.55 ±0.000	0.58 ±0.010	0.49 ±0.000	0.09 ±0.01	0.19 ±0.01	0.00 ±0.010	0.73 ±0.000	**0.73** ±0.001	**0.83** ±0.001
PGExplainer	0.09 ±0.003	0.16 ±0.002	0.22 ±0.013	0.37 ±0.000	0.57 ±0.010	0.49 ±0.010	0.10 ±0.010	0.21 ±0.030	0.12 ±0.040	**0.81** ±0.010	**0.73** ±0.001	0.77 ±0.140
Graph-LRP	0.09 ±0.002	0.10 ±0.000	0.23 ±0.014	0.13 ±0.000	0.71 ±0.000	0.22 ±0.010	**0.11** ±0.000	**0.35** ±0.010	0.19 ±0.000	0.79 ±0.000	0.14 ±0.000	0.79 ±0.000
Random	0.08 ±0.003	0.18 ±0.014	0.19 ±0.012	0.07 ±0.000	0.07 ±0.010	0.08 ±0.010	0.12 ±0.000	0.40 ±0.000	0.18 ±0.000	0.51 ±0.000	0.52 ±0.000	0.51 ±0.000
EB	0.09	0.10	0.12	0.48	0.71	0.39	0.02	0.00	0.22	0.80	0.14	0.32
GB	0.10	0.10	0.25	0.40	0.71	0.50	0.05	0.00	0.00	0.80	0.14	0.14
Gradient	0.10	0.10	0.25	0.40	0.71	0.50	0.05	0.00	0.00	0.80	0.14	0.14
LRP	0.09	0.10	0.25	0.16	0.71	0.34	0.08	0.00	0.27	0.77	0.14	0.75
CAM	0.26	**0.29**	0.12	0.45	0.49	0.49	0.01	0.07	0.21	0.48	0.14	0.69
SmoothGrad	0.08	0.10	**0.34**	0.30	0.71	0.55	0.03	0.00	**0.30**	0.77	0.15	0.78
GradCAM	0.11	0.10	0.33	0.42	0.71	0.49	0.01	0.00	0.28	0.56	0.14	0.77
Linear-Approx	0.09	0.10	0.13	0.42	0.71	0.49	0.02	0.00	0.17	0.80	0.14	0.67
IG	**0.31**	0.14	0.20	**0.71**	**0.72**	**0.72**	0.00	0.06	0.08	0.15	0.19	0.14

Table 2: AUC for descriptive accuracy (DA), sparsity (MAZ) and structural robustness (RA) and χ^2 distance for contrastivity. The standard deviation is omitted for deterministic methods as well as SmoothGrad as it is neglectable.

using the Adam optimizer with $lr = 0.001$ and L2-regularization with $\lambda_{L2} = 0.0001$.

5.2 Results

Equipped with three case studies on vulnerability discovery, we proceed to compare the different explanations based on our evaluation criteria. These experiments are repeated five times and mean and standard deviation are reported in Table 2.

Descriptive accuracy. We find that all graph-specific methods are inferior to the graph-agnostic ones under this criterion. Overall, the best method depends on the tested model. Graph-LRP is on par with its structure-unaware counterpart LRP. Furthermore, PGExplainer performs equal or better than GNNExplainer on two out of three tasks. Some graph-agnostic methods are even worse than the random baseline for certain models. Furthermore, as seen in Figure 2, for Vulas it is sufficient to remove less than 10% of the nodes to nearly render the prediction insignificant, since a DA of $84\% - 50\% = 34\%$ corresponds to the model predicting similar to random guess for Vulas. The DA curves show different levels of the results, which are due to the different model baselines. Compared to ReVeal, if more than 40% of the relevant nodes are removed, the accuracy drops close to random for most methods, even though Vulas has a lower node median count than ReVeal. We measure the drop in the F1-score for Devign, since this model has a low accuracy score in the first place.

As expected from the values in Table 1, the explanation methods can not reveal much for Devign as the model does not predict much better than random guessing. IG works best in the Devign case study. Our observation fits with the insights from Sanchez-Lengelin et al. [23]. Just as they suggest, we see that CAM and IG are among the best candidates. Moreover, according to our experiments SmoothGrad is a winning candidate as well.

We link the bad performance of PGExplainer to a phenomenon called *Laplacian oversmoothing* [6]. For deep GNNs, the node embeddings tend to converge to a graph-wide average. Depending on the depth of the network, the node embeddings get harder to separate and the performance of the network gets worse. Chen et al. [9] measure the mean average distance (MAD) of the node embeddings and demonstrate how networks with a higher MAD perform better. In the best runs, ReVeal, Devign and Vulas have a MAD of 1.0, 0.21 and 0.88 respectively. Because PGExplainer uses node embeddings to predict an edge's existence, we argue that this phenomenon influences such explanation methods. We can link the low MAD to the low DA from Table 2.

From descriptive accuracy to visualization. Based on the DA, we can easily extract *minimal descriptive subgraphs* that contain relevant nodes and yield insights on what paths characterize a vulnerability. As an example, BGNN4VD correctly identifies the SSRF vulnerability (CVE-2019-18394[5]) from Vulas that occurred in the OpenFire software. Figure 3 shows the vulnerability. After retrieving the 10% most relevant nodes from SmoothGrad, we can construct a minimal descriptive subgraph of this vulnerability as shown in Figure 3. We can traverse the CFG and DFG edges to reproduce the vulnerability, starting from `doGet` over `getParameter(host)` and the method call `getImage(host, defaultBytes)` and ending with the `IfStatement` where we would expect an input sanitization.

Extending descriptive accuracy to edges. Besides determining relevant nodes, it is also possible to calculate the most important edges and their descriptive accuracy. Except for GNNExplainer and PGExplainer, which both compute edge relevance scores, we calculate an edge relevance score by calculating the harmonic mean of the adjacent node relevance scores for each edge for the remaining

[5]https://nvd.nist.gov/vuln/detail/CVE-2019-18394

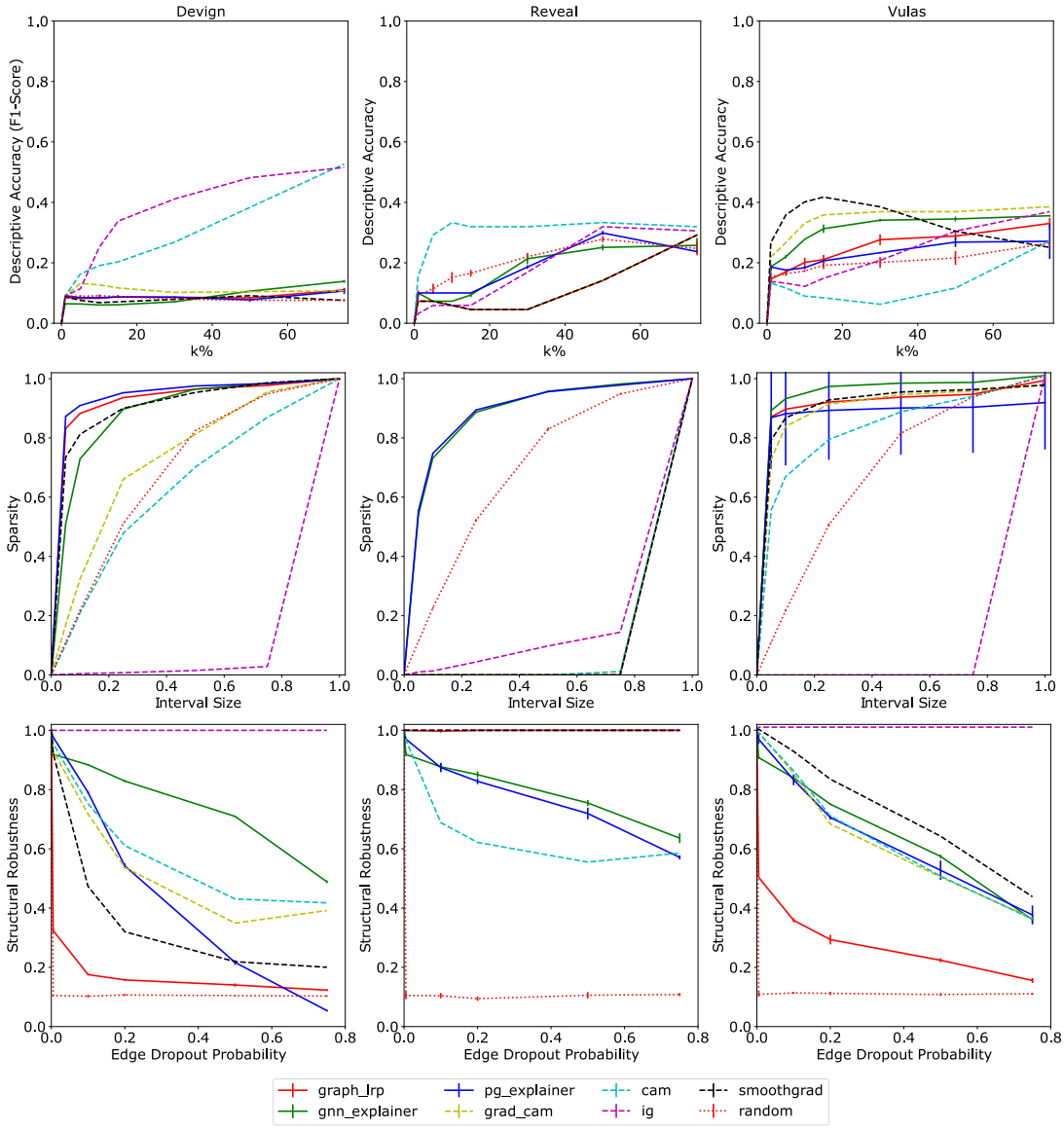


Figure 2: Descriptive accuracy (first row from top), sparsity (second row) and robustness curves (last row) for the Devign, ReVeal and Vulas case study for selected explanation methods.

EMs. An edge is only important if both adjacent nodes are similarly important. Eventually, the relevance of the edge types can be calculated by computing the histogram of the top 10% relevant edges. For space reasons, we compare the edge type attributions of the graph-specific methods only with those for the generic EMs *with the best DA*.

In this setting, SmoothGrad shows the best DA for Vulas, although, it only attributes high relevance to AST edges (Figure 4). On the other hand, PGExplainer attributes a lot more relevance to semantically important edge types, although its DA is lower. It would make sense, that assuming the model correctly learns to identify security vulnerabilities, EMs should assign more relevance on semantically meaningful edges. The AST edges should not encode much information when identifying vulnerable code. For the Vulas case study, DFG seems to be important for identifying vulnerabilities, comparing the histogram with the negative and positive samples. Unfortunately, SmoothGrad also shows the same histogram, both for negative and positive samples, while PG-Explainer attributes more scores to semantically interesting edge types.

Given the results for the ReVeal case study from Figure 4, the issue becomes more obvious: Most graph-agnostic methods fail to

```
...
public void doGet(HttpServletRequest request,
HttpServletResponse response) {
  ...
  byte[] bytes = getImage(host, defaultBytes);
  if (bytes != null) {
      writeBytesToStream(bytes, response);
  }
}
private byte[] getImage(String host,
byte[] defaultImage) {
   if (hitsCache.containsKey(host)) {
      return hitsCache.get(host);
    }
   byte[] bytes = getImage("http://" + host
                  + "/favicon.ico");
  ...
}
...
```

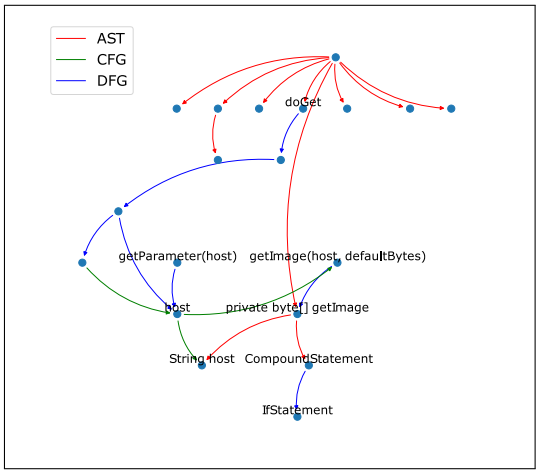


Figure 3: Minimal descriptive subgraph for the vulnerability CVE-2019-18394. The vulnerability has been detected by BGNN4VD and the graph extracted with SmoothGrad.

attribute relevance to semantically meaningful edges. Only GNNExplainer and PGExplainer attribute more relevance to meaningful edges when seeing positive samples. In general, CFG seems to be unimportant for positive samples. Graph-agnostic explanation methods attribute most relevance to semantically irrelevant AST and NCS edges for Devign (not shown in the plot).

Structural robustness. Overall, Integrated Gradients is by far the best EM according to its robustness (cf. Table 2). By contrast, Graph-LRP is the worst method on average, which makes sense since it calculates relevant walks and therefore strongly depends on edges. In Figure 2, we can see how the remaining methods compare against each other, with random being the worst method. Devign and Vulas as opposed to ReVeal show a steeper decrease, which could mean, that the model is trained to focus on the edges instead of the nodes. The random baseline is very low, as we attribute random nodes high relevance and an intersection of relevant nodes is very unlikely. Finally, ReVeal is less affected by edge perturbations.

Contrastivity. The contrastivity is rather low for most EMs, indicating that the selection of nodes is not very diverse and there is room for improvement. Still, Graph-LRP provides the largest distance in the case studies Devign and ReVeal between vulnerable and non-vulnerable code. SmoothGrad achieves the best contrastivity score for Vulas. For Devign and Vulas, all graph-agnostic EMs are

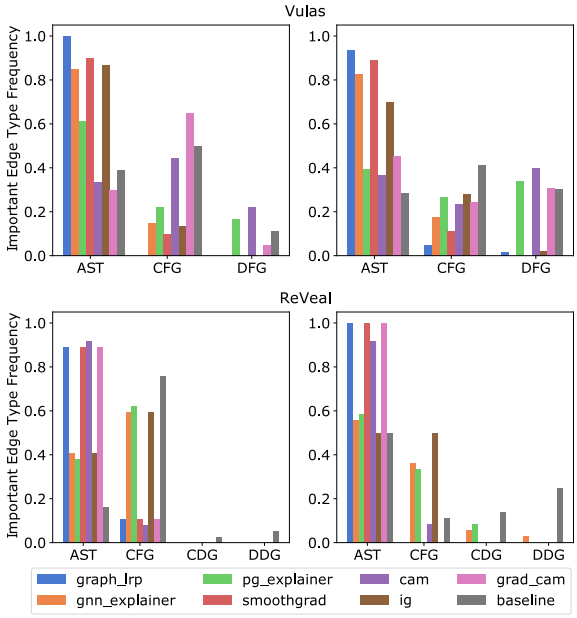


Figure 4: Important edge types for Vulas and ReVeal. Left column shows negative results and right positive.

below the baseline. In general, graph-specific methods seem to be better in identifying differences between relevant node types of vulnerable vs. non-vulnerable samples.

We observe that those EMs with a very low contrastivity attribute most relevance to the root nodes, both in the CCG and CPG.

By looking at the histogram over the most important node types (AST block identifier) labelled by graph-specific and graph-agnostic explainability methods respectively, we can clearly see a more

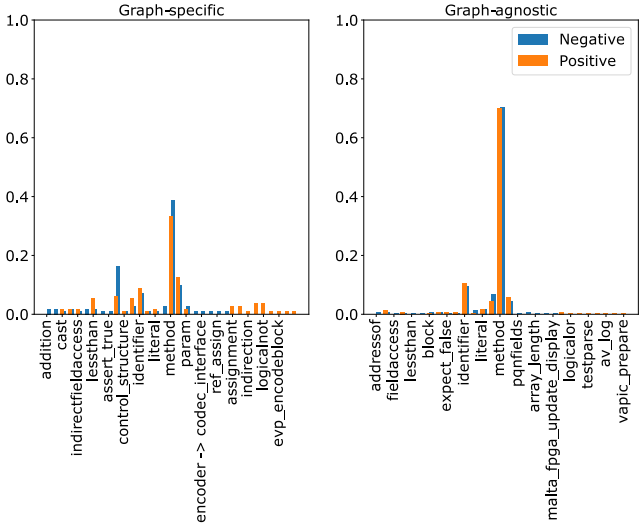


Figure 5: Important AST identifier histogram for ReVeal for negative and positive samples.

diverse distribution for the graph-specific methods in Figure 5, although the root nodes still determine the largest attribution mass for both EM classes. Some labels are skipped to be more readable.

However, we find that the contrastivity of the graph-specific methods is influenced by the root nodes of the AST. When removing the root nodes and measuring the accuracy we observe only a drop of 8%, 5% and 0% for Vulas, ReVeal and Devign, respectively. This is a hint that it is not the model that focuses on top nodes but rather the explanations do. Intuitively, it is not a desired behavior that an EM distributes relevance to nodes that do not provide any useful information to an expert. However, since the root node aggregates the relevance from nodes lower in the hierarchy, it makes sense. We can see this phenomenon in Figure 1, too, where the root node has similar relevance as the node `cin >> buf`.

Graph sparsity We see in Table 2 that for all models the graph-specific EMs yield the sparsest scores. This makes perfect sense since they are optimization algorithms that seek to maximize mutual information by maximizing the prediction score and minimizing the probability of an edge between two nodes. The random baseline has an AUC of around 50% because all nodes' relevance scores are uniformly distributed. Integrated Gradients have the worst results given the Devign and Vulas results. IG, for instance, attributes around 90% of the overall importance to approximately 60% of the nodes in the ReVeal case study.

In Figure 2 the MAZ curves (sparsity) are presented for the case studies. All graph-agnostic methods give extremely dense explanations for the ReVeal case study. Overall the graph-agnostic methods seem to be inferior compared to graph-specific methods. In Figure 1, we present an explanation of a CPG showing a vulnerability[6] that is correctly classified by ReVeal. The attribution is applied using PGExplainer which correctly attributed relevance to the `cin>>buf` node. However, unimportant nodes like the root node are highlighted as well.

Stability. Table 2 shows that all graph-specific methods yield an uncertainty that differs extremely from model to model. The graph-agnostic explanation methods do not vary at all. Furthermore, in the Sparsity and the DA column we see that PGExplainer has very different score levels across multiple runs. Each run differs in the descriptiveness from the identified important nodes and the amount of relevance distributed over all nodes.

The variance in the runs of Graph-LRP correlates to the sampled walks: Depending on the dataset and the sampling strategy, there is a difference in DA and MAZ. Graph-LRP and GNNExplainer generally have a much lower MAZ AUC standard deviation than PGExplainer, i.e. there is little variation in their conciseness. On the other hand PGExplainer's MAZ AUC varies extremely and, therefore, may yield different explanations. In addition its stability depends proportionally on the median node count of the dataset which is lowest for Devign.

We connect the large standard deviation of the graph-specific EMs in Vulas concerning the DA with the low node and edge count. A low node count means removing a single different node could have a stronger effect on the model's decision. Furthermore the CCG has less edges than the CPG, since the CCG does not contain PDG edges. Hence, a single misspredicted edge in PGExplainer or GNNExplainer could lead to a vastly different classification output.

Efficiency. Graph-specific methods are almost always slower than their conventional competitors. PGExplainer is trained one time per dataset, which in turn, renders its training time for the inference negligible. Due to the fact, that the PGExplainer uses node embeddings to predict the edge probability in a graph, we see that its runtime is extremely slow for ReVeal which can be directly linked to the large node and edge count median of 333 and 1132 respectively, for this particular case study. CAM, GB, linear approximation and EB scored the best scores in terms of runtime in our experiments. Among the graph-specific methods, PGExplainer was the fastest in 2 out of 3 tasks[7]. Graph-LRP is slow as well since it calculates one LRP run for each walk. Runtime figures can be seen in the appendix.

6 Discussion

Our evaluation of the various EMs provides a comprehensive yet also complex picture of their efficacy in explaining GNNs. Depending on the evaluation criteria, the approaches differ considerably in their performance and a clear winner is not immediately apparent, as shown in Table 3. In the following, we thus analyze and structure the findings of our evaluation by returning to the three research questions posed in the introduction.

(1) *How can we evaluate and compare explanation methods for GNNs in the context of vulnerability discovery?*

We find that existing criteria for evaluating EMs are incomplete when assessing GNNs in vulnerability discovery. Our experiments show that graph-specific criteria are crucial for understanding how an approach performs in a practical application. For example, a security expert would not only focus on high accuracy of explanations but also stability, sparsity, efficiency, robustness, and contrastivity. Theoretically, a study with human experts would provide more insights. However, this would be intractable. As a trade-off, we suggest using combinations of our proposed evaluation criteria to measure the potential to be human interpretable. The interplay of these measurements is crucial and all have to be considered.

(2) *Do we need graph-specific explanation methods, or are generic techniques for interpretation sufficient?*

Our evaluation demonstrates that generic EMs often lack sparse explanations and tend to mark more nodes as relevant than needed. For a security expert, it is necessary to spot the location of vulnerabilities. Not only have graph-specific methods larger differences between negative and positive samples more often, but also do they focus on semantically more meaningful edge types. As Yamaguchi et al. [32] show only few security vulnerability types can be found when only taking AST edges into account and hence a more contrastive view is necessary. It turns out that generic techniques often fail to provide this perspective when analyzing GNNs.

The stability and descriptive accuracy of graph-specific explanation methods, however, is inferior to generic approaches. Consequently, the sparse and more focused explanations comes with a limited accuracy in the relevant features. This opens new directions for research and developing graph-specific methods that

[6] Taken from https://samate.nist.gov/SARD/

[7] Measured on AWS EC2 p3.2xlarge instance.

Category	DA	Sparsity	Robustness	Contrastivity	Stability	Efficiency
Graph-agnostic	● ● ●	○ ○ ○	● ● ●	● ○ ○	● ● ●	● ● ●
Graph-specific	○ ○ ○	● ● ●	○ ○ ○	● ● ○	○ ○ ○	○ ○ ○

Table 3: Final evaluation comparing graph-agnostic and graph-specific EMs. One point for a *winner* EM per model.

attain the same accuracy as generic approaches. Some possible improvements could be adding regularization to focus on semantically important nodes, using node embeddings from lower layers to overcome *Laplacian oversmoothing*, or to use the contrastivity criterion already within the generation of explanations.

(3) *What can we learn from explanations of GNNs generated for vulnerable and non-vulnerable code?*

We observe that many explanation methods focus on semantically unimportant nodes and edges, while having a large descriptive accuracy. This could be a hint that the GNNs do not actually learn to identify vulnerabilities but artifacts in the data sets, so-called spurious correlations. As this phenomena occurs over several explanation methods, it seems rooted in the learning process of GNNs and thus cannot be eliminated easily. This finding is in line with recent work on problems of deep learning in vulnerability discovery [8] that also points to the risk of learning artifacts from the data sets. Hence, there is a need for new approaches that either eliminate spurious correlations early or improve the learning process, such that more focus is put on semantically relevant structures, for example, by additionally pooling AST, CFG and DFG structures.

Moreover, we show on a real-world vulnerability that the extraction of minimal relevant subgraphs from explanations is possible and provides valuable insights. These subgraphs can be used to construct detection patterns for static-analyzers [33], to guide fuzzers [40], or to find possible attack vectors for penetration testing [12]. Hence, despite the discussed shortcomings of explanation methods and GNNs in vulnerability discovery, we finally argue that they provide a powerful tool in the interplay with a security expert. Especially, the generation of subgraphs from explanations helps to understand the decision process for a discovery and to decide whether a learning-based system spotted a promising candidate for a vulnerability in source code.

7 Related Work

The variety of methods for explaining machine learning has brought forward different approaches for evaluating and comparing their performance [e.g., 18, 29, 34, 37]. In the following, we briefly discuss this body of related work, indicating similarities and differences to our framework.

Closest to our work is the study by Warnecke et al. [30] who develop evaluation criteria for EMs in security-critical contexts. For instance, they propose variants of the descriptive accuracy, sparsity, robustness, stability, and completeness for regular explanation methods. We build on this work and adapt the criteria to graph structures, such that they do not only measure the relevance of individual features but topological structures. Furthermore, we introduce new criteria that complement the evaluation and emphasize important aspects in the context of GNNs. Baldassarre and Azizpour [4] compare different explanation methods by attributing relevance to features but do not consider the underlying graph structure. Since nodes and edges are natural building blocks of a graph, it is beneficial to focus on identifying those important topological structures. This is especially important since we represent code as graphs and relevant nodes can be directly mapped to relevant code parts.

In a different research branch, explanation methods on GNNs have been evaluated by Sanchez-Lengeling et al. [23], Baldassarre and Azizpour [4] and Pope et al. [22]. Their main contributions include the reinterpretation of classical EMs to be applicable to graph neural networks and their evaluation on GNNs such as CAM, LRP and GradCAM. However, their works fall short of introducing new graph-specific criteria that are designed to explain structures not captured in common feature vectors. Besides their lack of a thorough comprehensive assessment as we introduce in our work, they do not consider any graph-specific EM.

Furthermore, Yuan et al. [36] introduce a framework for evaluating explanation methods for GNNs. They introduce the criteria fidelity, stability, and sparsity which compute the relevance for the model's prediction, the robustness against noise, and the conciseness of the methods respectively. Their work does not consider robustness against adversaries, efficiency or contrastivity and, most importantly, lacks experimental evaluations.

Pope et al. also determine the contrastivity of an explanation method by measuring the contrast between explanations for different classes [22]. However, they do not deliver insights about robustness or efficiency in their experiments which is especially important for the security domain. We adapt their contrastivity into the context of vulnerability discovery and use it to asses how well an explanation aligns with the actual code semantics. Besides that, we want to assess how the model differentiates between vulnerable and non-vulnerable samples. We try to answer whether GNN models actually learn to identify vulnerabilities. This question aligns with different works, that critically analyze the capability of models learning to represent vulnerabilities [3, 8].

In summary, current research does not offer any comprehensive framework applicable to GNNs in security related contexts. The majority of related work measures the quality of graph explanation methods with a specific ground truth [35] or domain knowledge [24] when checking whether EMs correctly detect cycles in a synthetic dataset, for example [35]. We try to evaluate models and explanations without using ground truth for the attributions, since this information rarely exists in realistic scenarios.

8 Conclusion

We compare multiple graph-agnostic and graph-specific explanation methods on three state-of-the-art GNN models which identify security vulnerabilities. For the assessment, we introduce a framework combining the evaluation criteria stability, descriptiveness,

structural robustness, efficiency, sparsity and contrastivity. Taking only the descriptive accuracy and runtime (efficiency) into account for the three GNN models under test, CAM, IG and SmoothGrad outperform all other explainability techniques. However, explanation methods for security-critical tasks, need to be thoroughly assessed using all of the above criteria. We find that all explanation methods have shortcomings in at least two criteria and therefore hope to foster research for new explanation methods. When it comes to meaningful, contrastive and sparse explanations that emphasize the underlying graph topology we find graph-specific methods to be superior.

To actually locate security vulnerabilities given human interpretable explanations we thus suggest using GNNExplainer or PG-Explainer. Our experimental results could guide development for novel graph-specific explanation methods or to overcome current shortcomings for GNNs in identifying security vulnerabilities.

Acknowledgments

This work has been funded by the Federal Ministry of Education and Research (BMBF, Germany) in the project IVAN (FKZ: 16KIS1165K).

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *CoRR* abs/1711.00740 (2017). arXiv:1711.00740 http://arxiv.org/abs/1711.00740

[3] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2020. Dos and Don'ts of Machine Learning in Computer Security. *CoRR* abs/2010.09470 (2020). arXiv:2010.09470 https://arxiv.org/abs/2010.09470

[4] Federico Baldassarre and Hossein Azizpour. 2019. Explainability Techniques for Graph Convolutional Networks. *CoRR* abs/1905.13686 (2019). arXiv:1905.13686 http://arxiv.org/abs/1905.13686

[5] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. *CoRR* abs/1806.07336 (2018). arXiv:1806.07336 http://arxiv.org/abs/1806.07336

[6] Chen Cai and Yusu Wang. 2020. A Note on Over-Smoothing for Graph Neural Networks. *CoRR* abs/2006.13318 (2020). arXiv:2006.13318 https://arxiv.org/abs/2006.13318

[7] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Information and Software Technology* 136 (2021), 106576. https://doi.org/10.1016/j.infsof.2021.106576

[8] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[9] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. 2019. Measuring and Relieving the Over-smoothing Problem for Graph Neural Networks from the Topological View. *CoRR* abs/1909.03211 (2019). arXiv:1909.03211 http://arxiv.org/abs/1909.03211

[10] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. 2020. ProGraML: Graph-based Deep Learning for Program Optimization and Analysis. arXiv:2003.10536 [cs.LG]

[11] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. 2015. Convolutional Networks on Graphs for Learning Molecular Fingerprints. *CoRR* abs/1509.09292 (2015). arXiv:1509.09292 http://arxiv.org/abs/1509.09292

[12] Mohd Ehmer and Farmeena Khan. 2012. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications* 3 (06 2012). https://doi.org/10.14569/IJACSA.2012.030603

[13] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[14] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02907

[15] Dexter C. Kozen. 1977. *Rice's Theorem.* Springer Berlin Heidelberg, Berlin, Heidelberg, 245–248. https://doi.org/10.1007/978-3-642-85706-5_42

[16] Sebastian Lapuschkin, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. 2015. On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation. *PLoS ONE* 10 (07 2015), e0130140. https://doi.org/10.1371/journal.pone.0130140

[17] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2017. Gated Graph Sequence Neural Networks. arXiv:1511.05493 [cs.LG]

[18] Pantelis Linardatos, Vasilis Papastefanopoulos, and Sotiris Kotsiantis. 2021. Explainable AI: A Review of Machine Learning Interpretability Methods. *Entropy* 23, 1 (2021). https://doi.org/10.3390/e23010018

[19] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. 2020. Parameterized Explainer for Graph Neural Network. arXiv:2011.04573 [cs.LG]

[20] Niccolò Pancino, Alberto Rossi, Giorgio Ciano, Giorgia Giacomini, Simone Bonechi, Paolo Andreini, Franco Scarselli, Monica Bianchini, and Pietro Bongini. 2020. Graph Neural Networks for the Prediction of Protein-Protein Interfaces.

[21] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and C´edric Dangremont. 2019. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories.* https://arxiv.org/pdf/1902.02595.pdf

[22] P. E. Pope, S. Kolouri, M. Rostami, C. E. Martin, and H. Hoffmann. 2019. Explainability Methods for Graph Convolutional Neural Networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* 10764–10773. https://doi.org/10.1109/CVPR.2019.01103

[23] Benjamin Sanchez-Lengeling, Jennifer Wei, Brian Lee, Emily Reif, Peter Wang, Wesley Qian, Kevin McCloskey, Lucy Colwell, and Alexander Wiltschko. 2020. Evaluating Attribution for Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 5898–5910. https://proceedings.neurips.cc/paper/2020/file/417fbbf2e9d5a28a855a11894b2e795a-Paper.pdf

[24] Thomas Schnake, Oliver Eberle, Jonas Lederer, Shinichi Nakajima, Kristof T. Schütt, Klaus-Robert Müller, and Grégoire Montavon. 2020. Higher-Order Explanations of Graph Neural Networks via Relevant Walks. arXiv:2006.03589 [cs.LG]

[25] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision.* 618–626.

[26] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning Important Features Through Propagating Activation Differences. *CoRR* abs/1704.02685 (2017). arXiv:1704.02685 http://arxiv.org/abs/1704.02685

[27] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda B. Viégas, and Martin Wattenberg. 2017. SmoothGrad: removing noise by adding noise. *CoRR* abs/1706.03825 (2017). arXiv:1706.03825 http://arxiv.org/abs/1706.03825

[28] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2015. Striving for Simplicity: The All Convolutional Net. arXiv:1412.6806 [cs.LG]

[29] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic Attribution for Deep Networks. arXiv:1703.01365 [cs.LG]

[30] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. 2020. Evaluating Explanation Methods for Deep Learning in Security. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE, Genoa, Italy, 158–174. https://doi.org/10.1109/EuroSP48549.2020.00018

[31] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[32] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy.* 590–604. https://doi.org/10.1109/SP.2014.44

[33] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy.* 797–812. https://doi.org/10.1109/SP.2015.54

[34] Wenjie Yang, Houjing Huang, Zhang Zhang, Xiaotang Chen, Kaiqi Huang, and Shu Zhang. 2019. Towards Rich Feature Discovery With Class Activation Maps Augmentation for Person Re-Identification. (2019), 1389–1398. https://doi.org/10.1109/CVPR.2019.00148

[35] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNN Explainer: A Tool for Post-hoc Explanation of Graph Neural Networks. *CoRR* abs/1903.03894 (2019). arXiv:1903.03894 http://arxiv.org/abs/1903.03894

[36] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. 2020. Explainability in Graph Neural Networks: A Taxonomic Survey. *CoRR* abs/2012.15445 (2020). arXiv:2012.15445 https://arxiv.org/abs/2012.15445

[37] Jianming Zhang, Zhe Lin, Jonathan Brandt, Xiaohui Shen, and Stan Sclaroff. 2016. Top-down Neural Attention by Excitation Backprop. *CoRR* abs/1608.00507 (2016). arXiv:1608.00507 http://arxiv.org/abs/1608.00507

[38] B. Zhou, A. Khosla, Lapedriza. A., A. Oliva, and A. Torralba. 2016. Learning Deep Features for Discriminative Localization. *CVPR* (2016).

[39] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *CoRR* abs/1909.03496 (2019). arXiv:1909.03496 http://arxiv.org/abs/1909.03496

[40] Xiaogang Zhu, Shigang Liu, Xian Li, Sheng Wen, Jun Zhang, Seyit Ahmet Çamtepe, and Yang Xiang. 2020. DeFuzz: Deep Learning Guided Directed Fuzzing. *CoRR* abs/2010.12149 (2020). arXiv:2010.12149 https://arxiv.org/abs/2010.12149

A Runtime Evaluation

Method	Devign	ReVeal	Vulas
EB	0.11	0.17	**0.07**
GB	**0.10**	0.16	0.09
Gradient	0.10	0.16	0.091
LRP	0.14	0.207	0.12
CAM	0.12	0.17	0.09
SmoothGrad	1.66	1.72	1.79
GradCAM	0.11	0.16	0.08
Linear-Approx	0.11	**0.16**	0.09
IG	1.63	2.52	1.52
GNNExplainer	3.99	5.06	8.88
PGE-Training	4.36	50.14	3.20
PGE-Inference	1.22	47.04	3.06
Graph-LRP	22.24	33.01	19.10

Table 4: Average runtime (in s) per single graph instance.

B Model Comparison

Model	Devign	BGNN4VD	Reveal
Graph	CPG	Bidirectional CCG	CPG
Network Type	6-GGNN	8-GGNN	8-GGNN
Pooling	1D-Conv	1D-Conv	Maxpooling
Prediction Head	PEM[8]	MLP	MLP
Loss	BCE + L2-Reg	BCE	Triplet Loss

Table 5: Notable differences between the models.

C Datasets

In the next three subsections we give a more detailed introduction to the datasets.

C.1 Devign

The Devign dataset consists of manually labeled C functions gathered from Qemu and FFmpeg open source projects [39]. It consists of 6000 malicious and 6000 benign samples. All bugs have been found by scraping the commit history for certain keywords including *injection* and *DoS*. Often vulnerabilities consist of *out of bounds* or other memory related security issues. For example this[9] or that[10]. Since FFmpeg and Qemu are part of the OSS-Fuzz project and are continuously fuzzed, such vulnerabilities are oftentimes detected during that process.

C.2 Reveal

Reveal consists of Debian security vulnerabilities taken from its tracker[11] and of Chromium vulnerabilities taken from its issue tracking tool[12]. Only bugs that are labeled *security* with a existent patch are scraped. Assuming a file has been patched, all its functions are extracted and labeled *benign*. Functions that differ from before and after fix are labeled *malicious*. Therefore, the dataset is unbalanced and consists of more benign than malicious functions.

```
1  static void eap_request(
2  eap_state *esp, u_char *inp, int id, int len) {
3  ...
4  if (vallen < 8 || vallen > len) {
5  ...
6  break;
7  }
8  /* FLAW: 'rhostname' array is vulnerable to overflow.*/
9  - if (vallen >= len + sizeof (rhostname)){
10 + if (len - vallen >= (int)sizeof (rhostname)){
11 ppp_dbglog(...);
12 MEMCPY(rhostname, inp + vallen,
13 sizeof(rhostname) - 1);
14 rhostname[sizeof(rhostname) - 1] = '\0';
15 ...
16 }
17 ...
18 }
```

Listing 1: Reveal example vulnerability CVE-2020-8597

In Listing 1 a sample vulnerability form the Reveal dataset taken from their original publication can be seen [8]. The sample shows a buffer overflow vulnerability due to a logic flaw in the *point to point protocol daemon* with the corresponding fix (line 9 and 10).

C.3 Vulas

Vulas is a collection of CVEs associated with large open source Java projects and their respective fix-commits [21]. We extract each changed function before and after the actual patch together with multiple randomly chosen functions from the same repository. The newest vulnerability in our dataset is CVE-2020-9489[13] and the oldest one CVE-2008-1728 [14]. A sample security issue can be seen in Figure 3.

[8] Pairwise Embedding Multiplication
[9] https://github.com/ffmpeg/ffmpeg/commit/06e5c791949b63555aa4305df6ce9d2ffa45ec90
[10] https://github.com/ffmpeg/ffmpeg/commit/5a2a7604da5f7a2fc498d1d5c90bd892edac9ce8
[11] https://security-tracker.debian.org/tracker/
[12] https://bugs.chromium.org/p/chromium/issues/list
[13] https://www.suse.com/security/cve/CVE-2020-9489.html
[14] https://nvd.nist.gov/vuln/detail/CVE-2008-1728