Interpreters for GNN-Based Vulnerability Detection: Are We There Yet?

Yutao Hu[*†]
Huazhong University of Science and Technology
China
yutaohu@hust.edu.cn

Suyuan Wang[*†]
Huazhong University of Science and Technology
China
stevewong@hust.edu.cn

Wenke Li[*]
Huazhong University of Science and Technology
China
winkli@hust.edu.cn

Junru Peng
Wuhan University, China
China
pengjunru@whu.edu.cn

Yueming Wu[‡]
Nanyang Technological University
Singapore
wuyueming21@gmail.com

Deqing Zou[*†]
Huazhong University of Science and Technology
China
deqingzou@hust.edu.cn

Hai Jin[†§]
Huazhong University of Science and Technology
China
hjin@hust.edu.cn

ABSTRACT

Traditional vulnerability detection methods have limitations due to their need for extensive manual labor. Using automated means for vulnerability detection has attracted research interest, especially deep learning, which has achieved remarkable results. Since graphs can better convey the structural feature of code than text, *graph neural network* (GNN) based vulnerability detection is significantly better than text-based approaches. Therefore, GNN-based vulnerability detection approaches are becoming popular. However, GNN models are close to black boxes for security analysts, so the models cannot provide clear evidence to explain why a code sample is detected as vulnerable or secure. At this stage, many GNN interpreters have been proposed. However, the explanations provided by these interpretations for vulnerability detection models are highly inconsistent and unconvincing to security experts. To address the above issues, we propose principled guidelines to assess the quality of the interpretation approaches for GNN-based vulnerability detectors based on concerns in vulnerability detection, namely, stability, robustness, and effectiveness. We conduct extensive experiments to evaluate the interpretation performance of six famous interpreters (*i.e., GNN-LRP*, *DeepLIFT*, *GradCAM*, *GNNExplainer*, *PGExplainer*, and *SubGraphX*) on four vulnerability detectors (*i.e., DeepWukong*, *Devign*, *IVDetect*, and *Reveal*). The experimental results show that the target interpreters achieve poor performance in terms of effectiveness, stability, and robustness. For effectiveness, we find that the instance-independent methods outperform others due to their deep insight into the detection model. In terms of stability, the perturbation-based interpretation methods are more resilient to slight changes in model parameters as they are model-agnostic. For robustness, the instance-independent approaches provide more consistent interpretation results for similar vulnerabilities.

CCS Concepts

• **Security and privacy → Vulnerability scanners**.

Keywords

Vulnerability Detection, Interpretation, GNN Interpreters

ACM Reference Format:
Yutao Hu, Suyuan Wang, Wenke Li, Junru Peng, Yueming Wu, Deqing Zou, and Hai Jin. 2023. Interpreters for GNN-Based Vulnerability Detection: Are We There Yet?. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598145

1 INTRODUCTION

With the rapid development of Internet technology, the number of source code vulnerabilities in software grows, posing a significant threat to enterprise and individual users' software security [45]. Although it is difficult to avoid source code vulnerabilities during the software development process, it is also a reasonable solution to identify vulnerabilities as early as possible and fix them as soon

[*]Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
[†]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, HUST, Wuhan, 430074, China
[‡]Yueming Wu is the corresponding author
[§]Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598145

as possible. At the moment, static code vulnerability detection methods can be divided into two categories: code similarity-based methods and pattern-based methods. The code similarity method is primarily used to discover vulnerabilities caused by code cloning and has a significant false negative rate for vulnerabilities produced by other factors. Traditional pattern-based methods require experts to manually define vulnerability characteristics, which wastes time and effort. Furthermore, because identifying characteristics is a subjective activity. Expert judgment will influence the detection results. As a result, there is an urgent need for a mechanism that can detect vulnerabilities without relying on experts.

Deep learning is a new field of machine learning research that has received extensive attention in recent years. Since it can automatically extract features from code without human intervention, it is widely used in source code vulnerability detection. For example, *VulDeePecker* [25] treats the code as text and applies a *long short-term memory* (LSTM) network to train a vulnerability detector. *ASTGRU* [13] extracts the *abstract syntax tree* (AST) to represent the code and leverages a *gated recurrent units* (GRU) network to analyze the tree sequence obtained by preorder traversal searching. *Devign* [59] collects the *code property graph* (CPG) of the code to maintain the program details and feeds it into a GNN model for vulnerability detection. In reality, a recent study [6] that performed a detailed comparison of different deep learning-based vulnerability detectors discovered that graph-based methods outperform other representation-based techniques (*e.g.,* text-based and tree-based). It makes sense because transforming the code into a graph representation can preserve program semantics such as control flow and data flow between lines of code.

Meanwhile, because of the GNN's superior ability to process graph structures, it has been adopted by an increasing number of researchers for vulnerability detection. However, since the GNN model is essentially a black box for security analysts, it cannot provide clear evidence to explain why a code sample is classified as vulnerable or safe. In practice, some GNN work related to interpretability has been proposed at this point, but whether the explanations provided by them can be convinced by security experts remains to be studied. Additionally, we also observe that the explanations provided by different GNN interpreters on the vulnerability detection task are inconsistent (see Section 4.2). This brings us to a question: *Which interpreters can we trust more?*

To answer the question, we present the first empirical study to research the ability of GNN interpreters on vulnerability detection in this paper. We select six famous GNN interpreters from multiple categories for our survey: *GNN-LRP* [35], *DeepLIFT* [38], *GradCAM* [30], *GNNExplainer* [51], *PGExplainer* [28], and *SubGraphX* [55]. Specifically, we evaluate them from three different perspectives, which are effectiveness, stability, and robustness. As for **effectiveness**, we analyze the performance of GNN interpreters on the vulnerability detection task, including *Intersection over Union* (IoU) and accuracy evaluation. Based on our results, we see that the effectiveness of all target interpreters is not ideal. Limited by the inaccuracy of the vulnerability detection model, the instance-independent interpretation methods (gradient-based and decomposition-based) outperform the instance-dependent ones (perturbation-based). As for **stability**, we explore whether the interpretations given by GNN interpreters are consistent when interpreting vulnerability detection models under subtle parameter differences. Through the results, we find that the stability of different interpreters varies enormously. Among them, the perturbation-based methods achieve better stability than decomposition- and gradient-based ones because they are model-agnostic. As for **robustness**, we investigate whether GNN interpreters can give similar interpretations when interpreting similar vulnerabilities. According to the results, we observe that all interpreters achieve poor robustness. To be specific, they cannot even provide consistent interpretations for a pair of similar vulnerabilities with different variable names.

Contributions. In summary, our paper makes the following contributions:

- We propose principled guidelines to assess the quality of the interpretation approaches for GNN-based vulnerability detectors based on concerns in the vulnerability detection domain, namely, stability, robustness, and effectiveness.
- We conduct extensive experiments on six popular interpreters applied to four state-of-the-art GNN-based vulnerability detection models.
- We open source our dataset and experimental results so that other researchers can replicate our findings. The website is *https://github.com/CGCL-codes/vdgraph*.

Paper Outline. Section 2 presents the background and the vulnerability detection models and GNN interpreters we investigated. Section 3 describes our study design, including effectiveness, stability, and robustness. Section 4 reports the study results. Section 5 discusses the threat to validity, future work, and actionable insights. Section 6 presents the related work. Section 7 concludes the paper.

2 BACKGROUND

In this section, we introduce the general pipeline of deep learning-based detection and interpretation. Then we describe the detection and interpretation methods used in our survey.

2.1 Overview

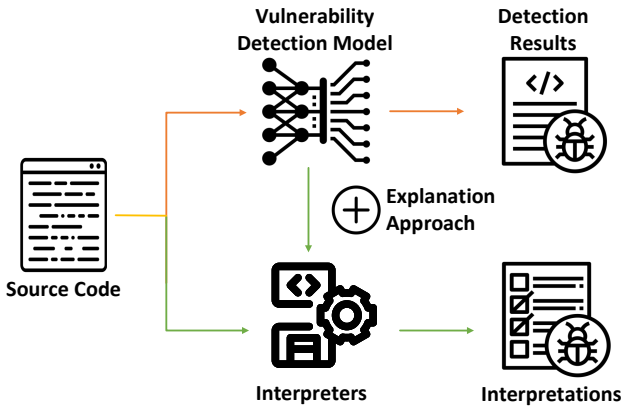

Figure 1: The general pipeline of deep learning-based vulnerability detection and interpretation

The pipeline of the interpretable detection method includes the detection and interpretation phase, as shown in Fig. 1. First, they preprocess the source code, embed it as vectors, and then use them

for training a vulnerability detection model. Then the interpreter generates vulnerability features (*i.e.,* vulnerable statements) so that security analysts can easily understand the basis of the detection and the vulnerability's principle to fix it. For interpreter training, the trained detection model and the detected vulnerability are fed into the interpretation model. Finally, the interpreter provides an interpretation for the vulnerability.

2.2 GNN-Based Vulnerability Detection Methods

Graphs represent the structural and semantic features of codes. In this part, we introduce the GNN-based vulnerability detectors used in our survey, which are the most representative GNN detectors. They have all been published in top conferences and journals and are not only the most used but are also all open source.

Devign [59] is a vulnerability detection model for multiple code representation graphs. Specifically, it embeds AST, *control flow graph* (CFG), *data flow graph* (DFG), and *natural code sequence* (NCS) into a joint graph for feature learning. *Devign*'s model consists of a gated graph recurrent layer and a convolutional layer. The former layer uses the GRU model to aggregate and pass node information to generate node features. The goal of the convolutional layer is to output prediction results, which perform two convolutions with pooling and then use *multilayer perceptron* (MLP) for classifications.

IVDETECT [21] is an interpretable vulnerability detection model using *program dependence graph* (PDG). For each node in the PDG, Glove+GRU is used to generate vector representations of *sub-token sequences*, *variables and types*, and *surrounding contexts*, respectively. Then attention-based Bi-GRU transforms the above vectors into the final vector. To obtain the vector representation of the target node, the vector of all the adjacent nodes are computed and weighted to sum.

REVEAL [6] is a vulnerability detection model using *code property graph* (CPG). It mainly uses the *gated graph neural network* (GGNN) for graph feature extraction. The model assigns a GRU to each node in the CPG, which updates the vertex's embedding by assimilating all its neighbors. Then, it sums the vectors of each node to obtain a vector representation of the CPG. The final result is obtained by using MLP with softmax for classification prediction. Same as *Devign* [59], to pre-train GGNN, it adds a classification layer on top of GGNN for feature extraction.

DeepWukong [8] relies on PDG for detection. It uses program slicing to generate XFG (*i.e.,* a program slice) for model training. Its GNN model mainly consists of graph convolutional/pooling layers, graph readout layer, and MLP, where the graph readout layer is to integrate the graphs' features with different sizes. Note that *DeepWukong* is slice-level method, while the others are function-level (with the entire function as the detection target). To fairly compare interpretation, we adjust the input to *DeepWukong* to match others, *i.e.,* we use PDG of the function as input.

2.3 GNN Interpretation Methods

As mentioned, the interpreter's input is the pre-trained detection model and the detected vulnerability. The output is the features that contribute significantly to detection, such as critical nodes, edges, or subgraphs. This part introduces six GNN interpreters employed in our survey, which cover perturbation-based, decomposition-based, and gradient-based methods.

2.3.1 Perturbation-Based Methods. It is an instance-dependent interpretation method whose key motivation is to investigate the instance's detection variation under different perturbations. Specifically, when the prediction differs from the original, the instance's perturbed information is an essential feature, *i.e.,* the vulnerability feature. The main difference between them is that they interpret different target features, such as essential nodes, edges, or subgraphs.

GNNExplainer [51] aims to explore the most relevant subgraph for explanation, using edge and feature masks to select structures and features, respectively. Specifically, its key idea lies in accomplishing a maximum mutual information optimization task. First, it performs an update training on the masks generated by perturbing the edge to generate a new graph. Then, the new graph are fed into the trained detector. A new graph with changed detection results (*i.e.,* from vulnerable to non-vulnerable) indicates that these perturbed features are essential for the instance to be detected as vulnerable. Finally, *GNNExplainer* extracts key subgraphs based on these features and outputs them as an interpretation.

PGExplainer [28] enables simultaneous interpretation of multiple instances, whereas *GNNExplainer* is developed for single-instance interpretation. It trains a parameterized mask predictor to predict edge masks. Specifically, it generates the edge's embedding by connecting the node embeddings. Then, the predictor uses edge embeddings to predict the probability that each edge will be selected, which can be used as an importance score. As a result, the mask predictor is trained by maximizing the mutual information between the original and new predictions. Its interpretation includes a subgraph and a set of node features.

SubgraphX [55] provides significant subgraphs as interpretations. However, *GNNExplainer* and *PGExplainer* interpret the graph with essential nodes and edges, which are not guaranteed to be connected. They ignore the interactions between nodes and edges, which may contain important information. *SubgraphX* uses the *Monte Carlo Tree Search* (MCTS) algorithm to efficiently explore different subgraphs by node pruning. It selects the most crucial subgraph from the tree's leaves as its output. Although *SubgraphX* does not learn masks directly, the MCTS can be considered a mask-generation algorithm.

2.3.2 Decomposition-Based Methods. This method calculates the importance of input features by decomposing the detector's predictions into several terms. They examine the detection model's parameters to reveal the relationship between the input space features and the output predictions. Therefore, they interpret from the model's perspective and are viewed as instance-independent approaches. Specifically, they distribute the prediction scores layer by layer in a back-propagation way to the input layer. The scores are then decomposed and assigned to neurons in each layer according to decomposition rules. By repeating this process until the input space, the importance scores of node features can be obtained, aggregated to represent the importance of edges, nodes, and walks. The main differences between these approaches are the score decomposition rules and the interpreted objects.

GNN-LRP [35] follows a *higher-order* Taylor decomposition of model prediction, decomposing the scores into the importance

of different walks. When performing neighborhood information aggregation, the walks correspond to the message flow, making it more consistent with GNN. The Taylor decomposition (at root zero) contains only T-order terms, where T is the number of layers of the trained GNN. Then each term corresponds to a T-order walk, considered an importance score. The walk records the paths of the message distribution process from one layer to another. These paths are considered different and scored from their corresponding nodes. The final output is the set of walks associated with the prediction.

DeepLIFT [38] is a novel algorithm for assigning importance scores to inputs with a given output. Specifically, it uses the differences in some "reference" inputs to explain the inconsistencies in some "reference" outputs. The "reference" inputs represent some default or "neutral" inputs selected based on their suitability for the problem. Using reference differences allow information to be propagated even when the gradient is zero, thus avoiding placing potentially misleading importance on bias terms.

2.3.3 Gradients-Based Methods. Using gradients to interpret deep models is the most straightforward solution and is widely used for image, text, graph, and node classification tasks. The key insight is to employ the gradient mapping values as an approximation of the input importance. They calculate the gradient by back-propagation. Note that the gradients are highly correlated with the model parameters, so the interpretations reflect the hidden information in the model. Therefore, the gradients-based approaches are instance-independent. The difference between these methods lies in the process of gradient back-propagation.

GradCam [37] is a deep network visualization method based on gradient localization. Specifically, it back-propagates the predicted values to obtain the gradient information, whose each element represents the contribution of the output of the last convolutional layer to the predicted values. The more significant the contribution, the more critical the neural network considers it to be. Note that *GradCam* is an interpretation method designed for *convolutional neural network* (CNN). To make it applicable to GNN, we map the scores of image classification to the nodes and edges of the graph as in previous work [54], *i.e.,* we use gradients as weights to combine different graph features.

3 STUDY DESIGN

This section presents the intuition of our three proposed quantitative metrics and their detailed calculation formulas.

3.1 Effectiveness

3.1.1 IoU Evaluation Approach. For any interpretation task, effectiveness is one of the most important goals. The first effectiveness evaluation method is inspired by manual vulnerability detection [46]. For human experts, a sample is considered vulnerable if it covers comprehensive vulnerability features. Specifically, vulnerability features are a set of code lines with a data dependent and a control dependent from the root cause of the vulnerability (*i.e.,* the location where the vulnerability was introduced) to the vulnerability trigger. Likewise, interpreters for deep learning-based detection methods are expected to provide vulnerability features to convince security analysts of the detection results and to help provide the cause of the vulnerability for quick patching. Therefore, an interpretation that covers the vulnerability features can be considered accurate.

Based on the above observations, we define the first method of effectiveness calculation. As mentioned before, the calculation of this metric relies on ground truth, that is, the vulnerability features. To obtain vulnerability features, we refer to previous work [48] and perform forward slicing from the modified statements of the vulnerability patch (*i.e.,* the root cause of the vulnerability). The generated slices cover complete vulnerability features and are used as the ground truth for interpretation. On that basis, we measure the effectiveness of the interpretation by comparing the overlap (*IoU*) between the interpretations and the ground truth. In the case that interpreter e outputs the top $k\%$ most significant nodes for all samples in dataset D, the specific formula for the effectiveness evaluation is as follow.

$$IoU(e, D, k) = \frac{1}{|D|} \sum_{x_i \in D} IoU(e, x_i, k) \tag{1}$$

$IoU(e, x_i, k)$ denotes the overlap of the interpretation p_i of sample x_i with its ground truth g_i. Its calculation is as follows, where e represents the interpreter, and v denotes the vulnerability detector.

$$IoU(e, x_i, k) = \frac{|p_i \cap g_i|}{|p_i \cup g_i|} \tag{2}$$

$$p_i = e(x_i, k, v) \tag{3}$$

3.1.2 Accuracy Evaluation Approach. Apart from this, previous work [21] proposed a more straightforward but less accurate method of assessing the effectiveness of the interpretation. It only depends on the modifications in the vulnerability patch instead of the ground truth. This evaluation method considers the explanation result correct if it overlaps with the modification in the vulnerability patch. The insight behind it is that the modified statement in the patch must be related to the vulnerability, which matches the intuition of our *IoU* metrics. In this way, if the interpreter points out the vulnerability-related statements, the interpretation is valid. As for an interpreter e, it outputs the top $k\%$ for each sample. The interpretation effectiveness $Acc(e, D, k)$ is calculated by summing and averaging $Acc(e, x_i, k)$ of each sample x_i in the dataset D. The formula is as follows, where r_i denotes the modified statements in the vulnerability patch and p_i is the interpretation results as Eq.3.

$$Acc(e, D, k) = \frac{1}{|D|} \sum_{x_i \in D} Acc(e, x_i, k) \tag{4}$$

$$Acc(e, x_i, k) = \begin{cases} 1, r_i \cap p_i \neq \emptyset \\ 0, r_i \cap p_i = \emptyset \end{cases} \tag{5}$$

3.2 Stability

The interpretation results generated for critical security systems should be stable. The interpreters we investigate are all instance-level, providing an interpretation for each input. Thus, the interpreters are expected to be consistent for multiple interpretations provided for a vulnerability sample. In this way, it can be shown whether the interpreter is learned the actual cause of the detected vulnerability, *i.e.,* the vulnerability signature. However, the parameters and structure of deep learning-based vulnerability detection models can be affected by several factors. In practice, influenced by different server configurations, including memory size, GPU performance, and hard drive capacity, people may run the model with targeted fine-tuning. For example, the batch size or the sample

embedding dimensionality may be adjusted due to the poor performance of the GPU, or the number of GGNN layers may be modified due to memory limitations. In this case, it is difficult to gain users' trust if the interpreters provide significantly varying explanations for similar models. In other words, the interpretations of a superior interpreter should remain constant on a similar pre-trained vulnerability detection model. Based on this intuition, stability is an essential property for evaluating interpreters.

We define the stability metric as the similarity of explanations of the same vulnerability on similar detection models. Therefore, we first define the formula for calculating the similarity of the two interpretations P_1 and P_2, following the *Dice coefficient* [2]. The interpretation P of sample x_i on model v using the interpreter e is as follows:

$$P = p(e, x_i, k, v) \tag{6}$$

$$Sim(P_1, P_2) = 2 \times \frac{|P_1 \cap P_2|}{|P_1| + |P_2|} \tag{7}$$

Then, the stability calculation formulas that an explanation method e outputs the first $k\%$ nodes on dataset D are as follows.

$$Stb(e, D, k, v) = \frac{1}{|D|} \sum_{x_i \in D} Stb(e, x_i, k, v) \tag{8}$$

$$Stb(e, x_i, k, v) = Sim(p(e, x_i, k, v), p(e, x_i, k, v')) \tag{9}$$

3.3 Robustness

Previous work [16, 17] has demonstrated that tiny changes in the input samples can confuse the interpretation results. In fact, such small changes in the samples do not change the prediction results of the detection model. Such problems can be fatal for the interpretation task of vulnerability detection. It is well known that in addition to some newly disclosed 0-day vulnerabilities, numerous cloned vulnerabilities exist in the wild. Therefore, the detection of cloned vulnerabilities is also an important research area for vulnerability detection [20, 23, 41, 48]. Security analysts expect the interpreter to provide consistent explanations for similar vulnerabilities. In particular, semantic clones of vulnerabilities should be interpreted with the same vulnerability characteristics, although they are textually very different for users to analyze. In this way, the interpreter can provide experts with practical help to quickly locate the vulnerable statements to fix the vulnerabilities. On the contrary, if the interpreter's interpretation of similar vulnerabilities varies significantly, not only will it not convince the experts, but it will lead them to doubt the vulnerability detection results and increase their workload. Therefore, we consider robustness another important evaluation metric of the interpretation approach by measuring the similarity of explanations of similar vulnerabilities.

Based on the above description, we give the following calculation formula for robustness metric, where X_i is the set of similar vulnerabilities of x_i.

$$Rob(e, D, k, v) = \frac{1}{|D|} \sum_{x_i \in D} Rob(e, x_i, k, v) \tag{10}$$

$$Rob(e, x_i, k, v) = \frac{1}{|Xi|} \sum_{x_i^{'} \in X_i} Sim(p(e, x_i, k, v), p(e, x_i^{'}, k, v)) \tag{11}$$

4 STUDY RESULTS

In this section, our experiments are centered on answering the following *Research Questions* (RQs):

- RQ1: To what extent is the inconsistency of interpretations with different interpreters for a detection model?
- RQ2: Can the interpreters provide effective interpretation results for different detection models?
- RQ3: Can the interpreters provide stable interpretation results for similar detection models?
- RQ4: Can the interpreters provide robust interpretation results for cloned vulnerabilities?

4.1 Experiment Settings

4.1.1 Dataset and Implementations. We run all experiments on a widely-used vulnerability dataset, *Big-Vul* [12], covers vulnerabilities in 348 open-source projects from 2002 to 2019, with 11,834 vulnerable functions and 253,096 non-vulnerable functions. In addition, it is necessary to have vulnerability patches and the corresponding functions before and after being patched, since the computation of the *IoU* requires to generate ground truth from the modifications of the patches. The calculation of *Acc* also requires the modified lines in the patch to obtain whether the interpretation is correct. We choose *Big-Vul* because it satisfies the above requirements. For implementations, we run all experiments on a standard server with 128GB RAM, 16 cores of CPU, and a GTX 5000 GPU.

4.1.2 Vulnerability Detection. In this part, we present the detection effectiveness of the four vulnerability detection models in our research. The interpreters directly interpret these pre-trained detectors with the pipeline, as in Fig. 1.

We use all vulnerable functions from the *Big-Vul* dataset and their corresponding non-vulnerable functions for detection model training. The specific training parameters are shown in Table 1. The evaluation metrics for the detection effectiveness of the vulnerability detection models follow existing work [8, 22, 24, 25]. The detection results are also shown in Table 1, and it can be seen that all four detectors achieve more than 60% F1-Score. Moreover, their results are similar overall, with high Recall and low Precision. Note that the goal of the interpreters is to provide vulnerability characteristics as the basis for vulnerability detection. So it is meaningless to interpret the non-vulnerable samples and unfair to interpret the samples that are incorrectly detected as vulnerable. Therefore, in all subsequent interpreter evaluation experiments, we consider only those vulnerable samples that are detected correctly.

4.1.3 Effectiveness Setup. There are two ways to evaluate the effectiveness (*i.e., IoU* and *Accuracy*), as mentioned in Section 3.1. Therefore, we need to prepare the ground truth for *IoU* evaluation and the modified statements of the patch for *Accuracy* evaluation in advance.

For the preparation of ground truth, we resort to an open-source code analysis tool, *Joern* [1, 49], to extract the PDG of the vulnerability. Then we identify the critical variables of the modification in the patch following previous work [48] and perform forward slicing based on these variables. The final slice is the ground truth of the vulnerability, *i.e.,* the set of all vulnerability-related statements. For the modified statements in the vulnerability patch, we directly parse the vulnerability patch and record the line numbers of the

Table 1: The summary of the vulnerability detection models

Vulnerability Detection Model	Parameters Setting						Detection Performance		
	Loss Function	Activation	Optimizer	Batch Size	Learning Rate	Epoch Num	F1-Score (%)	Precision (%)	Recall (%)
DeepWukong	CrossEntropyLoss	Relu	Adam	64	0.002	50	63.39	53.32	78.15
Devign	BCELoss	Relu	Adam	16	0.0001	100	62.37	53.72	74.33
IVDetect	CrossEntropyLoss	Relu	Adam	8	0.0001	100	65.26	52.51	86.18
Reveal	NLLloss	Relu	Adam	16	0.001	100	63.98	52.42	82.08

Table 2: The summary of the similar vulnerability detection models for stability evaluation

Vulnerability Detection Model	Adjusted Parameters						Detection Performance		
	Batch Size	Embedding Dimensions	Hidden Size	Dropout	GGNN Layers	Classifier	F1-Score (%)	Precision(%)	Recall(%)
DeepWukong	64 → 32	200 → 256	400 → 256	0.3 → 0.5	-	*softmax* → sigmoid	62.18	52.94	75.34
Devign	16 → 32	200 → 256	-	-	6 → 4	*softmax* → sigmoid	60.04	53.28	68.76
IVDetect	8 → 32	200 → 256	-	0.3 → 0.5	-	*softmax* → sigmoid	65.58	52.25	88.03
Reveal	16 → 32	200 → 256	-	0.5 → 0.2	6 → 4	-	59.54	53.67	66.85

additions and deletions (statements starting with "+" or "-" in the patch). In this way, the calculation of *Acc* is done by directly checking whether the modified statements' line numbers are included in the interpretations' line number set. Note that we only consider the interpretations of vulnerabilities correctly detected by the four vulnerability detectors to ensure that the interpretation is valid.

4.1.4 Stability Setup. Stability is estimated by the fact that given a sample, the explanations delivered on similar models should be consistent, then the interpretation method is considered stable. Therefore, we need to construct similar models for the four vulnerability detectors and train these similar ones. Specifically, We tune the parameters of the models to construct similar models. These parameters are selected to emulate the situations that may occur in reality, which may be adjusted by users when the model running is limited by physical reasons, as described in Section 3.2.

The specific parameter adjustments are shown in Table 2, which can be divided into two categories. The first is the batch size and embedding dimension. They are generic hyperparameters applicable to any model tuning, and the user generally adjusts them to adapt the model training to the server configuration. The other four parameters are oriented for specific model structures and are adjusted to improve the detection effectiveness. They are the parameters often tuned in deep learning model training by users, and such tuning does not cause significant changes in the model structure. Therefore, we choose to change them to construct a similar model of the vulnerability detection model.

The detection results of constructed similar models are shown in Table 2. The discrepancy between the similar model's and the original model's detection effectiveness is not remarkable, with the difference in F1-Score being within 5%. These results confirm that tuning these parameters has little effect on the model, so the models can be regarded as a pair of similar models before and after the parameters adjustment. Note that when we evaluate the stability of the interpreters, we only interpret the vulnerabilities detected correctly in both the original and similar models.

4.1.5 Robustness Setup. The robustness evaluation is oriented to the interpreter's ability to resist interference from similar samples, which is evaluated by the similarity of the interpretations for similar samples. An ideal interpreter should output consistent vulnerability features for similar vulnerabilities. Therefore, generating similar vulnerabilities is a significant challenge for this research question.

Table 3: Description of code transformation for robustness evaluation

	Code Transformations	Description
1	changeRename	Rename all variables
2	changeCompoundForAndWhile	Swap `for` and `while` loops
3	changeCompoundWhile	Change `while` basic block
4	changeCompoundDoWhile	Change `Do-while` basic block
5	changeCompoundIf (if-else)	Convert `if-else` to `if-else-if`
6	changeCompoundIf (if-else-if)	Convert `if-else-if` to `if-else`
7	changeCompoundSwitch	Convert `switch` to `if-else-if` structure
8	changeCompoundLogicalOperator	Change logical expressions
9	changeSelfOperator	Change SelfOperator (*e.g.,* `i++`)
10	changeCompoundIncrement	Change increment operation (*e.g.,* `+=`)
11	changeConstant	Change Constant
12	changeVariableDefinitions	Change Variable and Function Definitions
13	changeAddJunkCode	Insert Junk Code
14	changeExchangeCodeOrder	Exchange the order of statements without dependencies (*e.g.,* declaration statements)
15	changeDeleteCode	Delete code without semantics (*e.g.,* `printf()`)

To address this problem, we consider using code transformation techniques to perform code transformation on vulnerabilities. It is semantic-preserving, meaning that the label of an instance (vulnerable or non-vulnerable) will not be changed. Specifically, we employ *CloneGen* [3, 56] for code transformation, which generates similar code for vulnerabilities in the *Big-Vul* dataset, *i.e.,* cloned vulnerabilities. *CloneGen* is an open-source code transformation tool that provides 15 different ways to change code, with detailed descriptions in Table 3. For a vulnerability, we overlay different transformation methods to generate various cloned vulnerabilities. Specifically, we generate four cloned vulnerabilities for each vulnerability x_i, namely x_{i1} (using Transformation 1 only, *i.e.,* changeRename), x_{i5} (overlaying Transformations 1-5), x_{i10} (overlaying Transformations 1-10), and x_{i15} (overlaying Transformations 1-15). The number of transformations from x_{i1} to x_{i15} increases in order, that is, the lower the similarity to the original vulnerability.

For the generated cloned vulnerabilities, we first detect them with the four pre-trained vulnerability detectors. Finally, for each vulnerability detection model, we retain only the original vulnerability and its four cloned vulnerabilities that can all be detected correctly, which share the exact reason, as mentioned earlier.

4.2 RQ1: Interpretation Consistency Evaluation

To answer RQ1, we interpret each vulnerability with an interpreter and then observe the consistency of its interpretations by different interpreters. Specifically, we apply each of the six interpreters to the four pre-trained vulnerability detectors to obtain the interpretations, with the overall pipeline shown in Fig. 1.

Table 4: The interpretation of running example for *Devign*

	GNN-LRP	DeepLIFT	GradCAM	GNNExplainer	PGExplainer	SubgraphX
RQ2	1, 4, 5, 6, 10, 11, 13, 15, 19, 21	1, 2, 4, 5, 6, 10, 11, 13, 14, 15, 19	1, 4, 5, 6, 10, 11, 14, 15, 19	1, 2, 4, 10, 11, 15, 21	1, 4, 5, 10, 11, 14, 15, 19	1, 4, 5, 10, 11, 14, 15, 19
RQ3	1, 4, 5, 10, 11, 13, 15, 19, 21	2, 4, 5, 6, 10, 11, 13, 14, 15, 19, 21	1, 4, 5, 10, 11, 13, 14, 15, 19	4, 5, 6, 11, 13, 14, 15, 19	1, 4, 5, 11, 15, 19	1, 4, 5, 10, 11, 14, 15, 19
RQ4	2, 4, 5, 6, 10, 11, 13, 15, 19	1, 4, 5, 6, 10, 11, 13, 15, 19, 21	1, 2, 5, 6, 10, 11, 13, 15, 19	4,5,10, 11, 13, 15, 21	1, 2, 4, 10, 11, 14, 15, 19	4, 5, 10, 11, 13, 14, 19

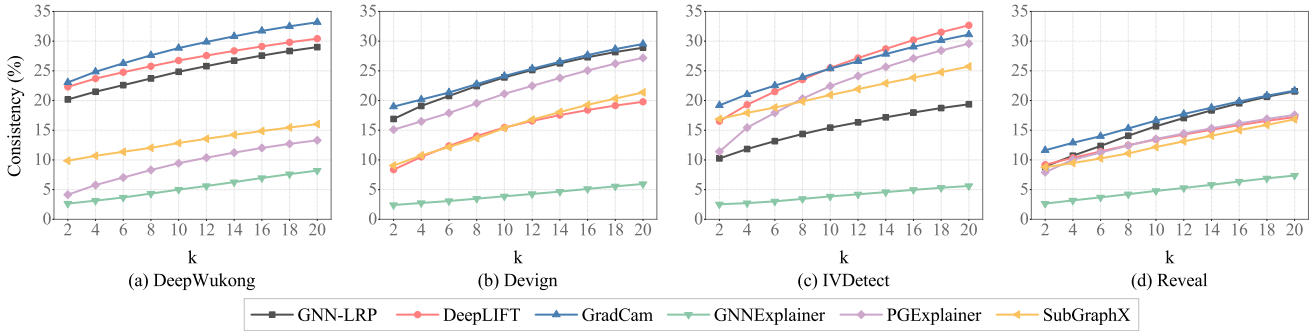

Figure 2: Consistency of the interpretation results for different vulnerability detectors with k from 1 to 20

For an interpreter's interpretation of each sample, we calculate its similarity to the interpretations of the other five interpreters separately and take the average to obtain its consistency on this interpreter. The final consistency of interpreter e on model v is calculated as follows. For example, sample x_i is interpreted in six interpreters as $p_1, p_2, \ldots, p_6$. To obtain the consistency of sample x_i on interpreter 1, calculate the similarity between p_1 and p_2 to p_6, respectively, and then average them to obtain $Con(e, x_i, k, v)$.

$$Con(e, D, k, v) = \frac{1}{|D|} \sum_{x_i \in D} Con(e, x_i, k, v) \tag{12}$$

$$Con(e, x_i, k, v) = \frac{1}{|E-1|} \sum_{e' \in \{E-e\}} Sim(p(e, x_i, k, v), p(e', x_i, k, v)) \tag{13}$$

The consistency of the interpretations under each vulnerability detection model using different interpreters is shown in Fig. 2. It can be seen that the overall consistency increases slowly with increasing k. It is reasonable since the larger k is, the more nodes the interpreter outputs, and the more likely the interpretation of different interpreters will overlap. Unfortunately, the consistency of all interpreters for all vulnerability detectors is below 35%. Among them, the highest consistency of all interpreters for *Reveal* is less than 25%, indicating that the interpretations of different interpreters vary greatly and fail to meet the requirement of trustworthiness. The above results show that the existing interpreters suffer from significant variability in the interpretation of the same detection model and the same vulnerability. This indicates that the interpreters are not credible. Therefore, there is an urgent need for principled guidelines for interpreter evaluation to help select the optimal vulnerability interpreter.

Summary: The consistency of different interpreters against vulnerability detection models is less than 35%, making it difficult for security analysts to select proper vulnerability interpretations. Therefore, the systematic evaluation methods for GNN interpretation proposed in our work have considerable practical value.

```
1 jbig2_page_add_result(Jbig2Ctx *ctx, Jbig2Page *page, Jbig2Image
                *image, int x, int y, Jbig2ComposeOp op)
2 {
3   /* ensure image exists first */
4   if (page->image == NULL) {
5       jbig2_error(ctx, JBIG2_SEVERITY_WARNING, -1, "page info possibly
                missing, no image defined");
6       return 0;
7   }
8
9   /* grow the page to accomodate a new stripe if necessary */
10  if (page->striped) {
11    -  int new_height = y + image->height + page->end_row;
      +  uint32_t new_height = y + image->height + page->end_row;
12
13    if (page->image->height < new_height) {
14     jbig2_error(ctx, JBIG2_SEVERITY_DEBUG, -1, "growing page
              buffer to %d rows " "to accomodate new stripe", new_height);
15     jbig2_image_resize(ctx, page->image, page->image->width, new_height);
16     }
17  }
18
19  jbig2_image_compose(ctx, page->image, image, x, y + page->end_row, op);
20
21  return 0;
22 }
```

Figure 3: A running example (the diff file of CVE-2017-9995)

4.3 RQ2: Effectiveness Evaluation

Effectiveness is an essential property of the usability of an interpretation method. As described in Section 4.1.3, we employ six interpretation methods for the trained vulnerability detection models to obtain interpretations. To answer RQ2, we use two metrics, where *IoU* is calculated by comparing interpretation with ground truth and *Acc* is measured based on patch modifications.

A running example is provided to illustrate the interpreter's interpretation, as shown in Fig. 3. We pick a vulnerability in CWE-119 (CVE-1016-9601), which is patched by modifying the variable type of `new_height` at line 11. According to the analysis, this function triggers a buffer overflow vulnerability at line 19 due to a possible negative value for `new_height` with type `int` at line 11, which makes the `if-condition` at line 13 not satisfied. Therefore, the characteristics of this vulnerability (ground truth) are as follows: the definition of `new_height` at line 11 is used as the root cause, and then the `new_height` is sliced forward until the vulnerability

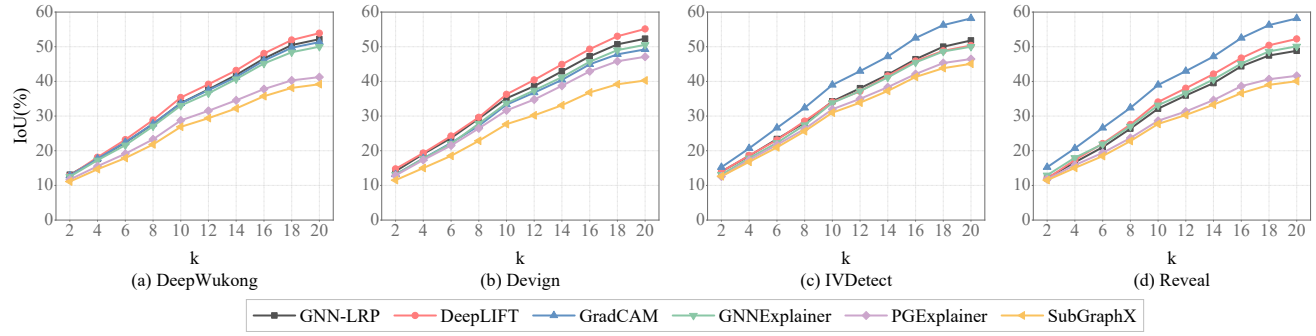

Figure 4: Effectiveness (*IoU*) of the interpretation results for different vulnerability detectors with k from 1 to 20

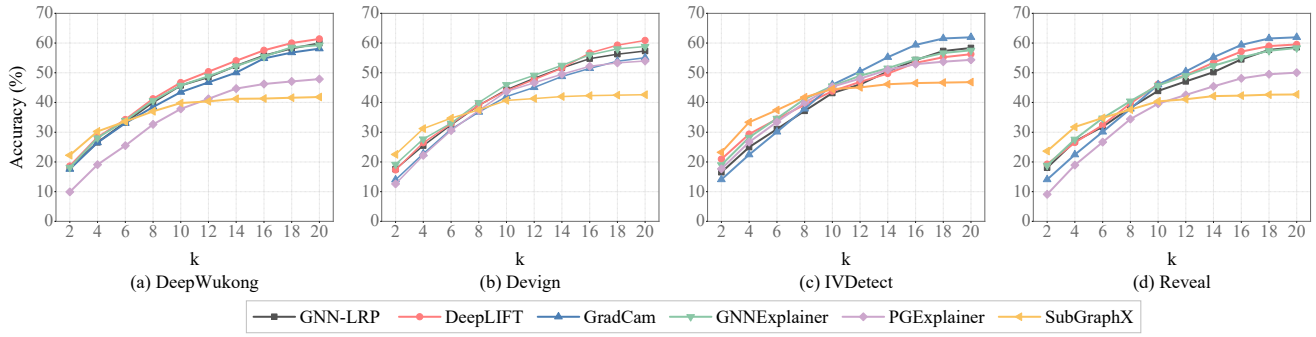

Figure 5: Effectiveness (*Acc*) of the interpretation results for different vulnerability detectors with k from 1 to 20

is triggered at line 19. In summary, the ground truth for this vulnerability contains lines 11, 13, 14, 15, and 19, and a modification of the vulnerability patch is line 11.

Due to space limitations, we provide the output of the six interpreters (*i.e.,* a set of the line numbers) for *Devign* under different research questions, as shown in Table 4. *Devign* was chosen as an example because it is the first proposed and most cited GNN-based vulnerability detector. Note that for presentation purposes, the data in Table 4 is the raw output for interpreters without specified k. Row RQ2 represents the interpretation of the six interpreters for *Devign* for the running example detection. It can be seen that the output of all interpreters contains line 11 (modification of the patch), so all interpreters achieve an *Acc* of 1.0 for this example. For the *IoU*, we use the GNNExplainer example, where ground truth is list $[11, 13, 14, 15, 19]$ and the output of GNNExplainer is list $[1, 2, 4, 10, 11, 15, 21]$. Thus, its *IoU* can be calculated as 0.2 according to Eq. 2.

The *IoU* and *Acc* results of the interpreters are shown in Fig. 4 and Fig. 5, where we can observe that: **(1)** Instance-independent methods (decomposition- and gradient-based) are more effective than instance-dependent methods (perturbation-based). Specifically, *GradCAM* (gradient-based) and *DeepLIFT* (decomposition-based) are the most effective, while the perturbation-based methods *PGExplainer* and *SubgraphX* have the worst effects. **(2)** The effectiveness of all six interpretation methods improves as the value of k increases. It can also be observed that at $k = 18$, the growth of *IoU* tends to flatten out, and the growth of *Acc* likewise slows down around $k = 10$. **(3)** The detection model's performance affects the interpretation methods' effectiveness. For example, Table 1 shows that the detection performance of *IVDetect* is better than that of *Devign*. In general, all interpreters are better for *IVDetect* than for *Devign*. Especially for *GradCAM*, it can be seen that its interpretation of *IVDetect* is significantly better than that of *Devign*.

Perturbation-based methods focus on the instances rather than the detection model's features, so they are viewed as instance-dependent. They perturb the nodes or edges of the input and then determine the critical feature by the changed detection results. For example, after a node is removed from a vulnerable PDG, the new instance is detected from vulnerable to non-vulnerable, indicating that this node is a crucial vulnerability feature (*i.e.,* interpretation). However, Table 1 shows that none of the vulnerability detectors is satisfactory, whose precisions are below 55%. Therefore, the detection result for the post-perturbation sample may be incorrectly detected by the inaccurate detectors. This is the reason for the poor effectiveness of the perturbation-based interpretation methods.

The effectiveness of the perturbation-based methods are highly variable, with *GNNExplainer* being significantly better than *PGExplainer* and *SubgraphX*. The reason is that *GNNExplainer* trains for each instance individually to obtain the interpretation. In contrast, *PGExplainer* pre-trains the interpreter for the entire training set, *i.e.,* it provides explanations for the instances with a global view. However, since the causes of vulnerabilities are complex and the vulnerability characteristics vary greatly between vulnerability types, the global features of all vulnerabilities are difficult to generalize. *SubGraphX* expects to obtain a key subgraph as the explanation, thus paying more attention to the connectivity between nodes and edges. However, connectivity is not a critical concern

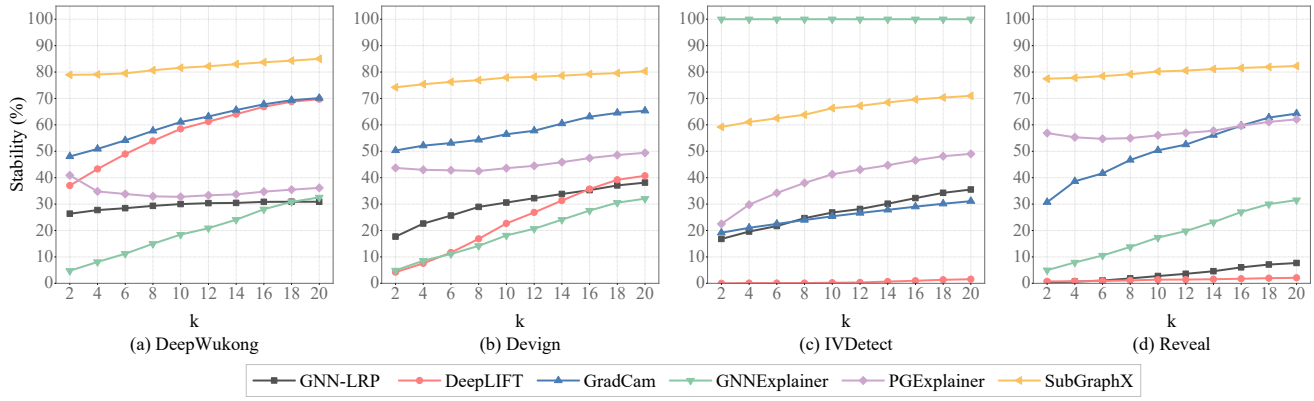

Figure 6: Stability of the interpretation results for different vulnerability detectors with k from 1 to 20

for the vulnerability detection task, so *SubGraphX* is inappropriate for the vulnerability detection interpretation.

Decomposition-based and **gradient-based** methods mainly generate the interpretation based on the parameters and structure of the detection model. Therefore, the interpretation results can reveal the vulnerability's critical nodes from the model's deeper perspective, thus achieving better interpretation performance. In addition, there is no particularly significant difference in the effectiveness of these methods.

Summary: The effectiveness of all interpreters for the vulnerability detection model is unsatisfactory, and all of their effectiveness is below 70%. Meanwhile, the experiments show that the instance-independent interpretation methods (decomposition-based and gradient-based) are more effective than the instance-dependent (perturbation-based) methods.

4.4 RQ3: Stability Evaluation

In order to answer RQ3, we apply the six GNN interpreters on similar pre-trained vulnerability detection models. The similar models are constructed as described in Section 4.1.4.

Row RQ3 in Table 4 represents the interpretation of *Devign*'s similar model. From the definition of stability, we can learn that we compare the similarity of the interpretation of the original detector and that of the similar detector to assess the stability of the interpreter. Since Row RQ2 is the interpretation of *Devign*'s original model, we can combine Row RQ3 and Row RQ2 to obtain the stability results. Using the *PGExplainer* as an example, the original interpretation is [1, 4, 5, 10, 11, 14, 15, 19], while the interpretation of the similar model is [1, 4, 5, 11, 15, 19]. Therefore, we can calculate its stability as 0.86 according to Eq. 9.

As shown in Fig. 6, the average stability is calculated according Eq. 8. From the figure, we can observe that: **(1)** *SubgraphX* shows the best stability among the six interpretation methods. It achieves around 80% stability in three detection models. **(2)** *GNN-LRP*, *DeepLIFT*, and *GNNExplainer* have poor performance in terms of stability, staying below 40% on average. However, they have better explanation results for individual vulnerability detection models. For example, *GNNExplainer* performs best in interpreting *IVDetect* with a score of 100%.

Perturbation-based methods have better stability in general, which directly infer the critical features that explain the detection results by modifying the nodes and edges in the graph. *SubGraphX* obtains the critical subgraphs for interpretation. It is because it pursues node and edge connectivity and uses a key subgraph to explain vulnerabilities, which is less disturbed by changes in the model when generating explanations. *PGExplainer* and *GNNExplainer* consider only critical nodes and edges and are more susceptible to model changes. *PGExplainer* trains the mask for all training data and interprets the test set based on this pre-trained mask. In contrast, *GNNExplainer* has poor stability as it lacks a global view of the dataset, resulting in an exaggeration of the sample's noise. Specifically, it trains a new mask for each instance individually to provide an interpretation of that instance. As a result, *PGExplainer* is more resistant to changes in the model than *GNNExplainer*.

Decomposition-based and **gradients-based methods** both have poor stability. Among them, the decomposition-based methods analyze the relationship between the input features and prediction results according to the model's structure and parameters. Similarly, the gradients-based approaches calculate the gradient of the input features by backpropagating between the model's hidden layers to obtain the input features' importance score. Note that the gradient is highly correlated with the model parameters, so its interpretation results can capture the hidden information in the model. Therefore, both methods are susceptible to model variations. In other words, tiny modifications in the model can lead to dramatic changes in the explanation results, especially for the two complex vulnerability detection models, *Reveal* and *IVDetect*.

Summary: Perturbation-based methods have better stability than decomposition-based and gradients-based methods. The former focus on the input features, while the latter is more sensitive to the GNN models, indicating that model-agnostic interpretation methods achieve good performance in term of stability.

4.5 RQ4: Robustness Evaluation

In order to answer RQ4, we evaluate the robustness by calculating the overlap of the interpreter's interpretation results in cloned vulnerabilities. We generate cloned vulnerabilities by code transformations one time, five times, ten times, and 15 times for experiments, as described in Section 4.1.5.

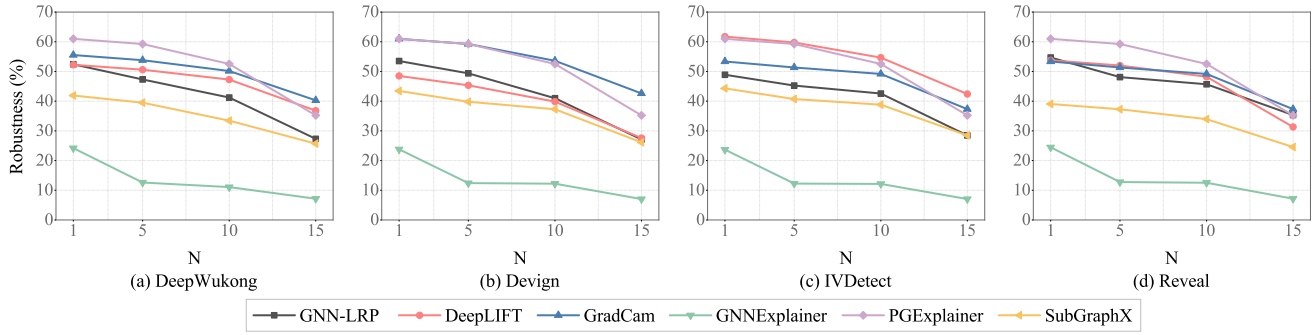

Figure 7: Robustness of the interpretation results for different vulnerability detectors with the time of code transformations N from 1 to 15

Row RQ4 in Table 4 represents the interpretation of the cloned vulnerability (with once code transformation) of CVE-2017-9995. From the definition, we can learn that we compare the similarity of the interpretations of the vulnerability and the cloned vulnerability to evaluate the robustness. Row RQ2 is the interpretation CVE-2017-9995, so we can combine Row RQ4 and RQ2 to obtain the robustness results. Using the *PGExplainer* as an example, the original interpretation is $[1, 4, 5, 10, 11, 14, 15, 19]$, while the interpretation of the cloned vulnerability is $[1, 2, 4, 10, 11, 14, 15, 19]$. Therefore, we can calculate its robustness as 0.875 according to Eq. 11.

Then the robustness of these six interpretation methods is calculated separately according to Eq. 10, and the calculation results are shown in Fig. 7. Due to space limitation, Fig. 7 only shows the experimental results with $k = 20$ because the best interpretation is achieved at $k = 20$ according to effectiveness evaluation. The rest of the data are published in our GitHub repository. From the figure, we can notice that: **(1)** The robustness of all the interpretation methods is unsatisfactory, and all fail to reach 61%. It indicates that these interpreters are easily disturbed by small changes in the sample. **(2)** The robustness shows a decreasing trend with the increase of the times of code transformations. Among them, the effect of the first 10 code transformations on the robustness is relatively weak, and the robustness decreases significantly after 15 transformations. **(3)** *PGExplainer* shows high robustness on all detection models, and the robustness can reach more than 60% for one code transformation and more than 35% even for 15 times. On the contrary, *GNNExplainer* achieves poor robustness on all detection models, less than 25% on single code transformation, and less than 10% on 15 times. **(4)** The detection model impacts the robustness of the interpretation method. For example, *DeepLIFT* is significantly more robust than the other three models in explaining the *IVDetct*. Its robustness to *IVDetect* is 60%, which is about 10% better than its robustness of the other three detection models.

Perturbation-based methods differ significantly in their robustness performance. Among them, *PGExplainer* has the best robustness while *GNNExplainer* is the worst, with similar reasons described in the stability evaluation. *PGExplainer* provides a global view for all samples, so its interpretation results depend mainly on the interpreter's training. Therefore, *PGExplainer* can effectively extract its important nodes and edges for similar codes after code transformation, so it has good robustness. On the contrary, *GNNExplainer* retrains the interpretation model for each instance, which lacks a global perspective and cannot effectively distinguish the noise introduced after transformation, thus having poor robustness. The robustness of the **decomposition-based** and **gradient-based methods** are not significantly different. As introduced before, both methods are model-sensitive. Since small changes in the input do not affect the parameters and gradients of the model, their robustness performance is acceptable, with slight variations.

The figure shows that even for one code transformation, *i.e.,* renaming variables in the code, the robustness of all interpreters is only 60%, indicating that the robustness of all interpreters is very poor. In other words, these interpreters can hardly deliver the same interpretation, even for two vulnerabilities that differ only in variable names. In addition, we analyze the PDGs after code transformation and find that the first ten transformations have a small impact on robustness due to the minor effect of the graph structure. However, after the eleventh to fifteenth transformations, there are significant changes in the graph structure, such as adding nodes and edges. Therefore, it can be observed that the robustness after 15 transformations is significantly reduced compared to the previous ones.

Summary: The robustness of all interpreters is not ideal and cannot provide consistent interpretations for a pair of similar vulnerabilities with different variable names. Additionally, the robustness of the perturbation-based methods varies widely, while the decomposition-based and gradient-based ones achieve relatively similar robustness.

5 DISCUSSION

5.1 Threats to Validity

First, the effect of parameter k on interpretation is very significant. To mitigate this threat, we conduct extensive experiments on choosing k values from 2-20 to demonstrate more clearly the interpretation results at different k. Second, the effect of different interpreters varies when applied to different detectors. To mitigate the threat, we choose four state-of-the-art detectors and employ each interpreter on them separately to ensure the integrity of the experiments. Third, for the interpreters, we select six representative interpreters from multiple categories. In this way, the threat of evaluating the applicability of the metrics to all GNN interpreters is mitigated. Fourth, the interpretations are only meaningful for correctly detected vulnerable inputs. For this reason, all explained samples are correctly detected in all our experiments to ensure the fairness of the experiments.

5.2 Lessons

Based on our study results, we provide top-5 actionable insights for GNN-based vulnerability interpreters.

Firstly, the detection effect of the vulnerability detector has a fatal impact on the interpreter. For instance-dependent interpreters, it provides interpretation by perturbing the nodes and edges of the instance. Therefore, it takes a superior vulnerability detector to confirm whether a perturbed instance is vulnerable and thus influences the interpreter's judgment. For model-related interpreters, there is a greater reliance on the detector, as it interprets the vulnerability in terms of the model's perspective. Unfortunately, current vulnerability detectors only achieve an F1-Score of around 65%, which is far from ideal. Therefore, we suggest that the main goal at this stage is to improve detection effectiveness of the detectors.

Secondly, instance-related interpreters explain from the perspective of vulnerabilities more in line with the logic of the human. Their core insight is that it becomes non-vulnerable when vulnerability's features (the important nodes and edges) are removed from a vulnerable instance. It is possible that the current instance-related interpreters are not as effective as they could be due to the poor performance of the detectors, *i.e.,* they cannot provide the correct detection results for the perturbed instances. Therefore, we recommend trying the instance-related interpreters again after the detectors has been improved, and they may perform well.

Third, the current instance-related interpreters are proposed based on graph theory and do not take into account the features of vulnerabilities that are different from other graphs. We suggest that some perturbation algorithms related to the vulnerability features could be designed for the vulnerability representation graphs, thus making the perturbation more effective and consistent with the vulnerability principles. Also, this makes the interpreter more focused on vulnerability-related features, which can improve the robustness of the instance-related interpreters.

Fourth, model-related interpreters currently achieve better effectiveness and have an inherent advantage for robustness. For their stability, decomposition-based interpreters focus more on graph-related input features, thus reducing the impact of unimportant model parameters. We suggest that the gradient-based approach be able to take into account some a prior experience by pre-setting some important nodes and edges that may be relevant to the vulnerability, thus mapping the back-propagation results of the predictions to more reliable nodes and edges and reducing the interference of irrelevant information in the graph.

Fifth, there is an urgent need for a GNN interpreter designed for vulnerability detectors. The current general GNN interpreters are not effective enough for vulnerability detection. We suggest that any interpreter combined with consideration of vulnerability features would effectively improve interpreters' performance (*i.e.,* effectiveness, robustness, and stability).

6 RELATED WORK

Many deep learning-based vulnerability detectors have been proposed. The text-based approaches treat the source code as plain text and then embed it as a vector. Then they employ traditional deep neural networks for training and detection [11, 13, 24–27, 31–33, 47, 61]. Since code has more structure and semantics than text, text-based methods do not perform well. Graphs are ideal code representations that convey the code's structural and semantic features [50]. Therefore, recent works [5, 6, 8, 9, 18, 43, 59] employ GNN models for detection and achieve good results.

Deep learning-based vulnerability detection methods lack explainability, resulting in unreliable detection results. To this end, interpretable detection methods are proposed. Some studies use the detection model to interpret the detection results. *Linevul* [14] provides the statement-level interpretation of vulnerabilities with the help of a self-attentive mechanism of *Transformer*. *LineVD* [18] uses *graph attention networks* (GAT) for detection and interpretation. *mVulPreter* [60] incorporates two granularity models to detect and explain vulnerabilities. However, these approaches using detection models are not generic. Therefore, some work design or apply a specialized interpretation model to interpret the results of vulnerability detection. Zou *et al.* [62] design an explainable framework that identifies combinations of tokens contributing significantly to prediction by perturbing samples near the decision boundary. *IVDetect* [21] interprets with *GNNExplainer* [51].

Current interpretable methods provide different perspectives for interpretation. There are two main categories of GNN interpretation methods depending :*instance-level methods* and *model-level methods*. *Instance-level approaches* interpret by identifying the predicted important input features. By different ways of obtaining importance scores,the instance-level methods can be divided into four branches. The first is *Gradient/Feature-based* [4, 30], which uses the gradient [39, 40] or hidden feature [37, 58] mapping values as an approximation of the input importance. The *Perturbation-based methods* [15, 28, 34, 44, 51] is to study the output variation under input perturbations [7, 10, 52]. The *Surrogate Methods* [19, 42, 57] use interpretable surrogate models to approximate the predictions of complex models for adjacent input regions. The *Decomposition Methods* [4, 35, 36] decompose the prediction of the original model into several terms to measure the importance of the input features. *Model-level methods* are designed to deliver general insights to interpret the GNN model [29], whose representative work is *XGNN* [53]. It trains the graph generator so that the generated graph maximizes the target graph prediction. However, the model-level interpreter explains the model rather than interpretations for the instances, so it is not chosen as our investigated tool. As can be seen, the six interpreters we selected cover most of the interpreter classes.

7 CONCLUSION

This paper proposes principled guidelines to assess the interpretation methods for GNN-based vulnerability detectors based on concerns in vulnerability detection, stability, robustness, and effectiveness. We conduct experiments using six widely used interpreters and four state-of-the-art detectors. The experimental results show that the interpretation results provided by different interpreters for vulnerability detection vary significantly, and the performance could be more satisfactory in all the above three metrics.

ACKNOWLEDGEMENTS

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the the National Science Foundation of China under grant No. 62172168 and Hubei Province Key R&D Technology Special Innovation Project under Grant No. 2021BAA032.

REFERENCES

[1] 2021. Open-source code analysis platform for C/C++ based on code property graphs. https://joern.io/.

[2] 2022. Dice similarity coefficient. https://radiopaedia.org/articles/dice-similarity-coefficient.

[3] 2023. CloneGen. https://github.com/CloneGen/CLONEGEN.

[4] Federico Baldassarre and Hossein Azizpour. 2019. Explainability techniques for graph convolutional networks. *arXiv preprint arXiv:1905.13686* abs/1905.13686 (2019). https://doi.org/10.48550/arXiv.1905.13686

[5] 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

[6] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48, 9 (2022), 3280–3296. https://doi.org/10.1109/TSE.2021.3087402

[7] Jianbo Chen, Le Song, Martin Wainwright, and Michael Jordan. 2018. Learning to explain: An information-theoretic perspective on model interpretation. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*. 883–892. https://doi.org/10.48550/arXiv.1802.07814

[8] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33. https://doi.org/10.1145/3436877

[9] Lei Cui, Zhiyu Hao, Yang Jiao, Haiqiang Fei, and Xiaochun Yun. 2020. Vuldetector: detecting vulnerabilities using weighted feature graph comparison. *IEEE Transactions on Information Forensics and Security* 16 (2020), 2004–2017. https://doi.org/10.1109/TIFS.2020.3047756

[10] Piotr Dabkowski and Yarin Gal. 2017. Real time image saliency for black box classifiers. *Advances in Neural Information Processing Systems* 30 (2017), 6967–6976. https://doi.org/10.48550/arXiv.1705.07857

[11] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. 2019. VulSniper: Focus your attention to shoot fine-grained vulnerabilities. In *Proceedings of the 2019 International Joint Conference on Artificial Intelligence (IJCAI'19)*. 4665–4671. https://doi.org/10.24963/ijcai.2019/648

[12] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'2020)*. 508–512. https://doi.org/10.1145/3379597.3387501

[13] Hantao Feng, Xiaotong Fu, Hongyu Sun, He Wang, and Yuqing Zhang. 2020. Efficient Vulnerability Detection based on abstract syntax tree and Deep Learning. In *Proceedings of the 2020 IEEE INFOCOM Conference on Computer Communications Workshops (INFOCOM'20 WKSHPS)*. 722–727. https://doi.org/10.1109/INFOCOMWKSHPS50562.2020.9163061

[14] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR'22)*. 608–620. https://doi.org/10.1145/3524842.3528452

[15] Thorben Funke, Megha Khosla, and Avishek Anand. 2021. Hard masking for explaining graph neural networks. In *Proceedings of the 2021 International Conference on Learning Representations (ICLR'21)*.

[16] Amirata Ghorbani, Abubakar Abid, and James Zou. 2019. Interpretation of neural networks is fragile. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'19)*. 3681–3688.

[17] Juyeon Heo, Sunghwan Joo, and Taesup Moon. 2019. Fooling neural network interpretations via adversarial model manipulation. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS'19)*. https://doi.org/10.1609/aaai.v33i01.33013681

[18] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. 2022. LineVD: statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR'22)*. 596–607. https://doi.org/10.1145/3524842.3527949

[19] Qiang Huang, Makoto Yamada, Yuan Tian, Dinesh Singh, and Yi Chang. 2022. Graphlime: Local interpretable model explanations for graph neural networks. *IEEE Transactions on Knowledge and Data Engineering* abs/2001.06216 (2022). https://doi.org/10.48550/arXiv.1909.10911

[20] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P'17)*. 595–614. https://doi.org/10.1109/SP.2017.62

[21] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. 292–303. https://doi.org/10.1145/3468264.3468597

[22] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2022. VulDeeLocator: A Deep Learning-Based Fine-Grained Vulnerability Detector. *IEEE Transactions on Dependable and Secure Computing* 19, 04 (2022), 2821–2837. https://doi.org/10.1109/TDSC.2021.3076142

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*. 201–213. https://doi.org/10.1145/2991079.2991102

[24] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258. https://doi.org/10.1109/TDSC.2021.3051525

[25] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23158

[26] Guanjun Lin, Wei Xiao, Jun Zhang, and Yang Xiang. 2020. Deep learning-based vulnerable function detection: A benchmark. In *Proceedings of the 21st International Conference on Information and Communications Security (ICICS'19)*. 219–232. https://doi.org/10.1007/978-3-030-41579-2_13

[27] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. 2017. POSTER: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. 2539–2541. https://doi.org/10.1145/3133956.3138840

[28] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. 2020. Parameterized explainer for graph neural network. *Advances in Neural Information Processing Systems* 33 (2020), 19620–19631. https://doi.org/10.48550/arXiv.2011.04573

[29] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. 2017. Feature visualization. *Distill* 2, 11 (2017), e7. https://doi.org/10.23915/distill.00007

[30] Phillip E. Pope, Soheil Kolouri, Mohammad Rostami, Charles E. Martin, and Heiko Hoffmann. 2019. Explainability methods for graph convolutional neural networks. In *Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'19)*. 10772–10781. https://doi.org/10.1109/CVPR.2019.01103

[31] Gao Qiang. 2022. Research on Software Vulnerability Detection Method Based on Improved CNN Model. *Scientific Programming* 2022 (2022). https://doi.org/10.1155/2022/4442374

[32] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA'18)*. 757–762. https://doi.org/10.1109/ICMLA.2018.00120

[33] Canan Batur Şahin. 2021. DCW-RNN: Improving Class Level Metrics for Software Vulnerability Detection Using Artificial Immune System with Clock-Work Recurrent Neural Network. In *Proceedings of the 15th International Conference on INnovations in Intelligent SysTems and Applications (INISTA'21)*. 1–8. https://doi.org/10.1109/INISTA52262.2021.9548609

[34] Michael Sejr Schlichtkrull, Nicola De Cao, and Ivan Titov. 2020. Interpreting graph neural networks for NLP with differentiable edge masking. *arXiv preprint arXiv:2010.00577* (2020). https://doi.org/10.48550/arXiv.2010.00577

[35] Thomas Schnake, Oliver Eberle, Jonas Lederer, Shinichi Nakajima, Kristof T. Schütt, Klaus-Robert Müller, and Grégoire Montavon. 2022. Higher-order explanations of graph neural networks via relevant walks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 11 (2022), 7581–7596. https://doi.org/10.1109/TPAMI.2021.3115452

[36] Robert Schwarzenberg, Marc Hübner, David Harbecke, Christoph Alt, and Leonhard Hennig. 2019. Layerwise relevance visualization in convolutional text graph classifiers. *arXiv preprint arXiv:1909.10911* (2019), 58–62.

[37] 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 (ICCV'17)*. 618–626. https://doi.org/10.1007/s11263-019-01228-7

[38] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning important features through propagating activation differences. In *Proceedings of the 34th International Conference on Machine Learning (PMLR'17)*. 3145–3153. https://doi.org/10.48550/arXiv.1704.02685

[39] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034* (2013). https://doi.org/10.48550/arXiv.1312.6034

[40] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. 2017. Smoothgrad: removing noise by adding noise. *arXiv preprint arXiv:1706.03825* abs/1706.03825 (2017). https://doi.org/10.48550/arXiv.1706.03825

[41] Hao Sun, Lei Cui, Lun Li, Zhenquan Ding, Zhiyu Hao, Jiancong Cui, and Peng Liu. 2021. VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches. *Computers & Security* 110 (2021), 102417. https://doi.org/10.1016/j.cose.2021.102417

[42] Minh Vu and My T. Thai. 2020. Pgm-explainer: Probabilistic graphical model explanations for graph neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 12225–12235. https://doi.org/10.48550/arXiv.2010.05788

[43] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2020. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security* 16 (2020), 1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

[44] Xiang Wang, Yingxin Wu, An Zhang, Xiangnan He, and Tat-seng Chua. 2021. Causal screening to interpret graph neural networks. In *Proceedings of the 2021 International Conference on Learning Representations (ICLR'21)*.

[45] Chensi Wu, Tao Wen, and Yuqing Zhang. 2019. A revised CVSS-based system to improve the dispersion of vulnerability risk scores. *Science China Information Sciences* 62, 3 (2019), 1–3. https://doi.org/10.1007/s11432-017-9445-4

[46] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS'20)*.

[47] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An Image-inspired Scalable Vulnerability Detection System. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. 2365–2376. https://doi.org/10.1145/3510003.3510229

[48] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, and Wei Zou. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures.. In *Proceedings of the 2020 USENIX Security Symposium (USENIX Security'20)*. 1165–1182.

[49] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceddings of the 2014 IEEE Symposium on Security and Privacy (S&P'14)*. 590–604. https://doi.org/10.1109/SP.2014.44

[50] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*. 590–604. https://doi.org/10.1109/SP.2014.44

[51] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNNExplainer: Generating Explanations for Graph Neural Networks. In *Proceedings of the 32nd Annual Conference on Neural Information Processing Systems (NeurIP'19)*. 9240–9251. https://doi.org/10.48550/arXiv.1903.03894

[52] Hao Yuan, Lei Cai, Xia Hu, Jie Wang, and Shuiwang Ji. 2020. Interpreting image classifiers by generating discrete masks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 4 (2020), 2019–2030. https://doi.org/10.1109/TPAMI.2020.3028783

[53] Hao Yuan, Jiliang Tang, Xia Hu, and Shuiwang Ji. 2020. Xgnn: Towards model-level explanations of graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'20)*. 430–438. https://doi.org/10.1145/3394486.3403085

[54] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. 2022. Explainability in graph neural networks: A taxonomic survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022). https://doi.org/10.1109/TPAMI.2022.3204236

[55] Hao Yuan, Haiyang Yu, Jie Wang, Kang Li, and Shuiwang Ji. 2021. On explainability of graph neural networks via subgraph explorations. In *Proceedings of the 38th International Conference on Machine Learning (PMLR'21)*. 12241–12252. https://doi.org/10.48550/arXiv.2102.05152

[56] Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. 2023. Challenging machine learning-based clone detectors via semantic-preserving code transformations. *IEEE Transactions on Software Engineering* abs/2111.10793 (2023). https://doi.org/10.1109/TSE.2023.3240118

[57] Yue Zhang, David Defazio, and Arti Ramesh. 2021. Relex: A model-agnostic relational model explainer. In *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society (AIES'21)*. 1042–1049. https://doi.org/10.1145/3461702.3462562

[58] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. 2016. Learning deep features for discriminative localization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 2921–2929. https://doi.org/10.1109/CVPR.2016.319

[59] 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. In *Proceedings of the 32nd Annual Conference on Neural Information Processing Systems (NeurIPS'19)*. 10197–10207. https://doi.org/10.48550/arXiv.1909.03496

[60] Deqing Zou, Yutao Hu, Wenke Li, Yueming Wu, Haojun Zhao, and Hai Jin. 2022. mVulPreter: A Multi-Granularity Vulnerability Detection System With Interpretations. *IEEE Transactions on Dependable and Secure Computing* (2022). https://doi.org/10.1109/TDSC.2022.3199769

[61] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2021. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2021), 2224–2236. https://doi.org/10.1109/TDSC.2019.2942930

[62] Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. 2021. Interpreting Deep Learning-based Vulnerability Detector Predictions Based on Heuristic Searching. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 23:1–23:31. https://doi.org/10.1145/3429444

Received 2023-02-16