mVulPreter: A Multi-granularity Vulnerability Detection System with Interpretations

Deqing Zou, Yutao Hu, Wenke Li, Yueming Wu, Haojun Zhao, and Hai Jin, *Fellow IEEE*

Abstract—Due to the powerful automatic feature extraction, deep learning-based vulnerability detection methods have evolved significantly in recent years. However, almost all current work focuses on detecting vulnerabilities at a single granularity (*i.e.,* slice-level or function-level). In practice, slice-level vulnerability detection is fine-grained but may contain incomplete vulnerability details. Function-level vulnerability detection includes full vulnerability semantics but may contain vulnerability-unrelated statements. Meanwhile, they pay more attention to predicting whether the source code is vulnerable and cannot pinpoint which statements are more likely to be vulnerable. In this paper, we design *mVulPreter*, a multi-granularity vulnerability detector that can provide interpretations of detection results. Specifically, we propose a novel technique to effectively blend the advantages of function-level and slice-level vulnerability detection models and output the detection results' interpretation only by the model itself. We evaluate *mVulPreter* on a dataset containing 5,310 vulnerable functions and 7,601 non-vulnerable functions. The experimental results indicate that *mVulPreter* outperforms existing state-of-the-art vulnerability detection approaches (*i.e., Checkmarx*, *FlawFinder*, *RATS*, *TokenCNN*, *StatementLSTM*, *SySeVR*, and *Devign*).

Index Terms—Vulnerability Detection, Deep Learning, Interpretable AI, Graph Neural Network.

1 Introduction

Most cyber-attacks [1], [2], such as hacker ransomware and botnet attacks, originate from software vulnerabilities, which have caused vast harm in our daily life [3]. Therefore, security researchers have invested much energy in developing various vulnerability prediction and detection tools. In general, source code vulnerability detection methods can be classified into similarity-based [4], [5], [6], [7], [8] and pattern-based [9], [10], [11], [12], [13], [14], [15], [16], [17]. Similarity-based methods can detect vulnerabilities caused by code reuse, but they suffer from high false negatives for newly emerging vulnerabilities. Traditional pattern-based methods require human experts to define vulnerability features to represent vulnerabilities, leading to high false positive rates. Therefore, an ideal vulnerability detection method requires less labor and achieves low false positive and false negative rates.

Automated approaches, especially deep learning, have received extensive attention due to their powerful modeling and intelligent pattern learning capabilities. Therefore, security researchers have applied them to vulnerability detection. According to the code granularity processed by the models, current deep learning-based vulnerability detectors can be classified into two main categories, that is, function-level [14], [18], [19] and slice-level [12], [13], [20]. At function-level, a complete function is labeled and used as a training sample. At slice-level, the training slices are generated by the data-dependent analysis starting from vulnerability interesting points (*i.e.,* sensitive APIs, pointer usages, array usages, and integer usages) that frequently introduce vulnerabilities. In practice, a vulnerable function can cover the complete vulnerability feature but introduce many vulnerability-unrelated statements. A slice can better capture a vulnerability with less noise than the function [21]. However, the criterion of slices is the root cause of the vulnerabilities, which does not guarantee that slicing will cover their trigger location. So some of the necessary vulnerability features may be absent in a vulnerable slice. Moreover, most of these studies cannot interpret the vulnerability detection results. They focus on predicting the program source code as either vulnerable or not, but cannot pinpoint which lines of code are more likely to be vulnerable.

In this paper, we aim to combine the fine-granularity of the slice-level with the complete semantics of the function-level to construct an accurate and interpretable vulnerability detector. Specifically, we mainly address two challenges.

- *How to combine the advantages of slice-level and function-level vulnerability detection to reduce false positives and false negatives?*
- *How to give a detailed interpretation of the vulnerability detection results?*

To address the first challenge, we leverage two deep neural networks to perform vulnerability detection at both function-level and slice-level. Given the source code of a

- *D. Zou, Y. Hu, W. Li, and H. Zhao are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China. E-mail: deqingzou, yutaohu, winkli, haojunzhao@hust.edu.cn*
- *Y. Wu (corresponding author) is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore, 639798. E-mail: wuyueming21@gmail.com*
- *H. Jin is with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China. E-mail: hjin@hust.edu.cn*

function, we first apply static analysis to extract the *Program Dependency Graph* (PDG). Then we split PDG into subgraphs (*i.e.,* slices) via program slicing based on the interesting points of vulnerabilities. These slices will be fed into a trained slice-level detection model to calculate their predicted probabilities. The lower the probability, the less likely it is to contain vulnerable code. Therefore, we discard those slices with the lowest probability. Based on this, we can purify the function semantics to achieve more accurate vulnerability detection at function level.

To address the second challenge, we incorporate an attention mechanism into the function-level vulnerability detection model to determine the weight of each slice. Meanwhile, we also consider the probability of the slice predicted by the slice-level vulnerability detector. After combining the weight with the probability, we can compute a specific score for each slice. In this way, each statement is assigned a score by accumulating the slice scores to which it belongs. The higher the score of the statement, the more likely it is a vulnerable statement.

We implement *mVulPreter* and evaluate it on a dataset consisting of 5,310 vulnerable and 7,601 non-vulnerable functions. The experimental results report that *mVulPreter* is superior to three rule-based tools (*i.e., Checkmarx* [9], *FlawFinder* [11], and *RATS* [10]) and four deep learning-based systems (*i.e., TokenCNN* [17], *StatementLSTM* [22], *SySeVR* [23], and *Devign* [14]). Additionally, the detection results of *mVulPreter* can be interpreted by the attention module and slice-level vulnerability detector, which achieve the best performance compared to current SOTA interpreters (*i.e., GNNExplainer* [24] and *PGExplainer* [25]). We also conduct a case study to examine the ability of *mVulPreter* to scan vulnerabilities in real-world open-source products. As a result, *mVulPreter* discovers 28 new vulnerabilities that are not reported in *National Vulnerability Database* (NVD). We have reported them to their vendors and hope that they can be patched as soon as possible.

In summary, our paper makes the following contributions:

- To the best of our knowledge, we are the first to combine the function-level and slice-level vulnerability detection models to achieve effective vulnerability detection.
- We design *mVulPreter*[1], a novel multi-granularity graph-based vulnerability detector that can give detailed interpretations of the detection results.
- We conduct an extensive evaluation to demonstrate the effectiveness of *mVulPreter*. Through the results, we find that *mVulPreter* outperforms seven state-of-the-art vulnerability detection methods (*i.e., Checkmarx* [9], *FlawFinder* [11], *RATS* [10], *TokenCNN* [17], *StatementLSTM* [22], *SySeVR* [23], and *Devign* [14]).
- We perform a case study on more than 25 million lines of code to check the practicability of *mVulPreter* on real-world vulnerability scanning. Through the results, we discover 28 new vulnerabilities that are not reported in NVD.

1. https://github.com/tao7777/mVulPreter

2 Motivation

To illustrate the limitations of existing method and motivate the idea of our approach, we adopt a real-world vulnerability (CVE-2012-0850 [26]) from *FFmpeg* [27] as a running example, which is a buffer errors vulnerability (CWE-119).

Fig 1 shows the vulnerable function and corresponding slices of CVE-2012-0850. Through manual analysis, we can infer that the vulnerability is introduced by an incorrect control condition (*i.e.,* the root cause). In detail, the branch condition in the vulnerable function is `*v_off==0` (line 6). So the decreasing shift operation `*v_off-=128>div` in the *else* branch (line 11) will be handled when `*v_off` is not equal to 0. It may lead to a negative value of `v` in the forward data dependency of `*v_off` (lines 21-22, 31-32), resulting in buffer underflow when accessing the array `v`, thus triggering a buffer error vulnerability.

Based on the vulnerability description above, we can learn that a code sample can only capture the vulnerability when it covers both the root cause (line 6) and the vulnerability trigger location (lines 21-22, 31-32). At slice-level, the slices containing the deletions in the patch (*i.e.,* vulnerable slices) extracted from this function are *S1*, *S2*, and *S3*, as shown in Fig 1. However, none of these three slices can contain both the causes and the trigger location of the vulnerability. Specifically, *S1* and *S3* cover the trigger positions, while *S2* is related to the root cause. Besides, there are some non-vulnerable slices extracted (*S4*-*S9*). For example, *S4* shows a slice performed from the integer variable `saved_samples` as a criterion. At function-level, the function contains the comprehensive vulnerability features, but it also includes many statements that are not related to the vulnerability. Too much noise within the training samples makes it difficult for the model to learn the accurate vulnerability feature.

The above analysis shows that when slicing a vulnerable function, the vulnerability semantics may be split into different slices so that some slices only contain part of the vulnerable code. In other words, a slice containing vulnerable code may not be able to represent complete vulnerability knowledge, resulting in inaccuracies in slice-level vulnerability detection. Moreover, some slices from a vulnerable function may be non-vulnerable slices, which means a vulnerable function may contain normal behaviors. Normal behaviors in the function may confuse the function-level detector, making the detection less accurate.

In short, slice-level vulnerability detection is fine-grained but may contain incomplete vulnerability semantics. However, function-level vulnerability detection includes full vulnerability semantics but may contain normal behaviors. In this paper, we aim to combine the fine-granularity of slice-level methods with the full semantics of function-level detectors to achieve more accurate vulnerability detection. To this end, we design *mVulPreter*, a novel multi-granularity graph-based vulnerability detection system.

3 System Architecture

This section introduces *mVulPreter*, a novel multi-granularity vulnerability detection system that combines function-level with slice-level learning models.

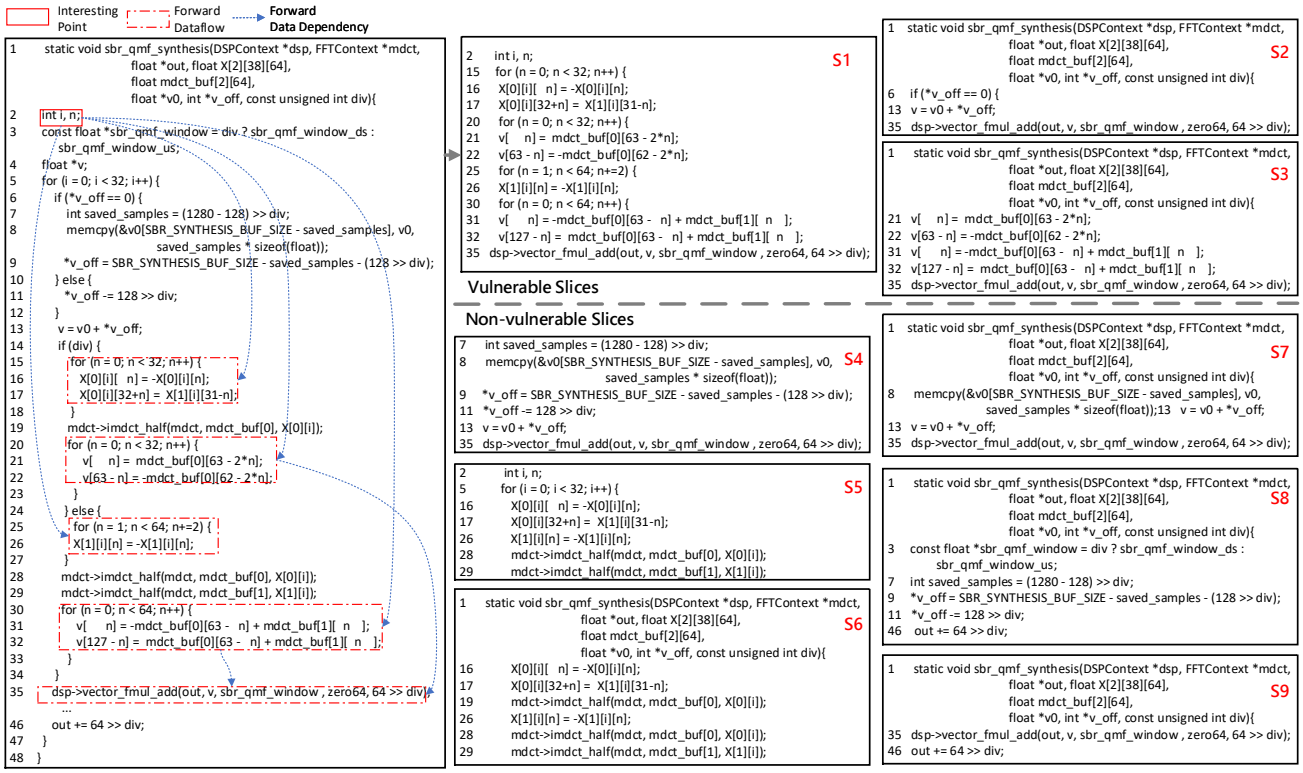


Fig. 1: The vulnerable function and slices of CVE-2012-0850

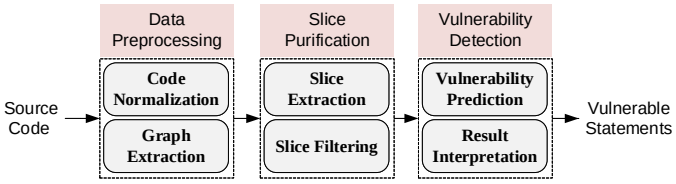


Fig. 2: System overview of *mVulPreter*

3.1 Overview

As shown in Fig 2, *mVulPreter* consists of three main phases: *Data Preprocessing*, *Slice Purification*, and *Vulnerability Detection*. The input to *mVulPreter* is the source code of the target function, and the output is whether the function is vulnerable and the critical vulnerable statements.

- *Data Preprocessing:* Given the source code of a function, we first perform code normalization on it and then extract the PDG of the normalized function.
- *Slice Purification:* Given the PDG, we first split it into multiple slices via program slicing. Next, we filter them with low relevance to the vulnerability according to the output (*i.e.*, predicted probability) of the pre-trained slice-level detector.
- *Vulnerability Detection:* Our final phase is designed to output the detection results of functions and corresponding interpretations. We first train an attention-based function-level model to detect vulnerability, and then interpret the detection results by combining attention weights with the predicted probabilities of the slice-level detector.

3.2 Data Preprocessing

mVulPreter is targeted to detect function-level vulnerabilities, which is an appropriate granularity as it contains more comprehensive vulnerability features than slice-level and less noise than file-level. Before extracting the graph representation of a function, we first abstract and normalize the source code. In particular, we utilize three levels of normalization, which make *mVulPreter* resilient to common code modifications while preserving program semantics.

Steps 1-3 in Fig 3 illustrate the detailed normalization process of a function at different levels.

- **Step 1:** Remove comments that do not affect the semantics of the program.
- **Step 2:** Map user-defined variables one by one to symbolic names (*i.e.*, `VAR1`).
- **Step 3:** Map user-defined functions one by one to symbolic names (*i.e.*, `FUN1`).

To obtain a complete representation of the dependencies within a function, we resort to the PDG. It is constructed based on the nodes of the *Abstract Syntax Tree* (AST), part of which are connected by data-dependence and control-dependence edges. As for implement, *mVulPreter* utilizes Joern [28], [29], an open-source code analysis platform for C/C++, to extract the PDG of the abstracted function.

3.3 Slice Purification

In this phase, *mVulPreter* splits the PDG of a function into slices. With the help of a slice-level GNN model, the slices within a function that have little contribution to vulnerability detection are filtered out.

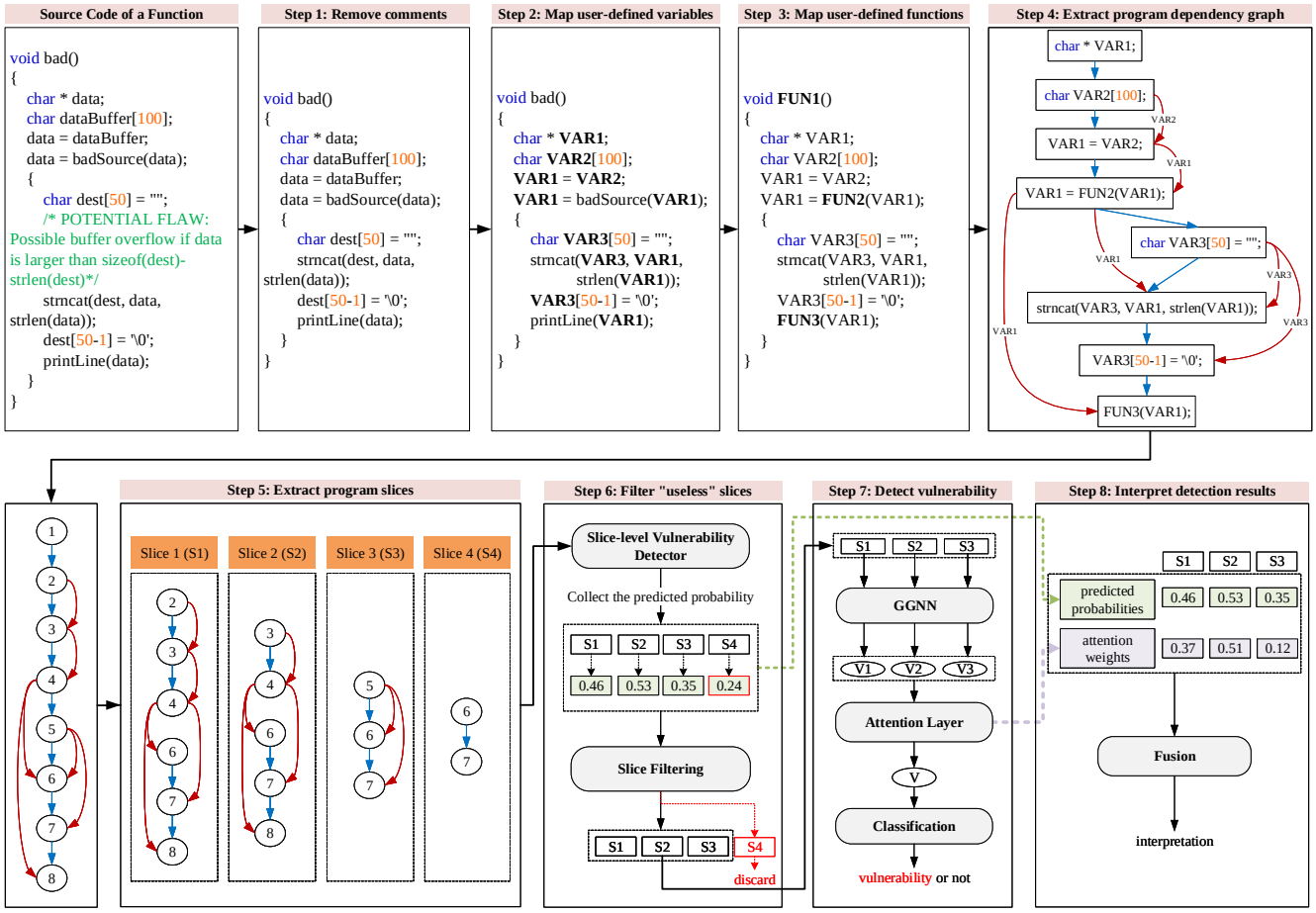


Fig. 3: An example to illustrate the detailed phases of *mVulPreter*

3.3.1 Slice Extraction

As shown in Step 5 of Fig 3, *mVulPreter* performs program slicing from the interesting points of vulnerabilities to extract slices based on the PDG of a target function.

First, we introduce the interesting points of vulnerabilities, *i.e.*, the root causes of vulnerabilities summarized by manual analysis. Previous work [23] has demonstrated four code features that often lead to vulnerabilities, including sensitive APIs, arrays, integers, and pointers. Therefore, *mVulPreter* also regards the above features as the interesting points of vulnerabilities, which are starting points for program slicing. For example, the identified interesting points are `* data` (pointer), `dataBuffer` (array), `dest` (array), and `strncat` (sensitive API) in the example of Fig 1.

After identifying above interesting points, we set each of them as a slicing criterion. In detail, we perform forward- and backward- slicing [30], [31] on the PDG of the function. Then the forward and backward slices are concatenated to generate the program slices, which exclude vulnerability-unrelated statements while preserving the source code structure information. S1-S4 of Step 5 in Fig 3 are the slices we obtained. Each slice generated in this step contains two parts of information: *nodes* (*i.e.*, node code) and *edges* (*i.e.*, the dependencies between nodes). Red lines and blue lines show the data flows and control flows between the code lines within the function, respectively. To display the PDG of the vulnerability more clearly, we replace each line of code with a numbered circular node. Eight lines of code correspond to eight circular nodes, as shown in Fig 3.

Note that program slicing from various code lines for the same variable may generate different slices. For example, the variable `n` is an integer variable considered a vulnerability interesting points in Fig 1. *S1* is a slice generated by data dependency analysis on the variable `n` starting from line 2. However, when extracting slices for the variable `n` starting from line 15, the generated slice *S1′* contains lines 15-17. Obviously, all the nodes within the *S1′* are included in the *S1*. In this case, after obtaining all the slices, we perform simple preliminary filtering on them. For the slices with inclusion relations, only retain those that contain the most lines of code. That is, for *S1* and *S1′* we filter out *S1′*. In contrast to *S3* and *S4*, since they are extracted from different variables, it is not necessary to filter out *S4* even if *S3* contains all the nodes of *S4*.

3.3.2 Slice Filtering

In this part, we introduce the slice-level model that is used to filter a part of slices within a function. Each slice extracted from a function can be regarded as a behavior. Those behaviors unrelated to vulnerabilities contribute little and even interfere with the function's vulnerability detection. So this phase of *mVulPreter* is designed to filter out these "useless" slices within a function.

Node Embedding Module. For each slice, we first process them into a proper data structure to train the *Graph*

Neural Network (GNN) [32] model. We treat a line of code as a sentence and apply sentence embedding to transform it into a fixed-length vector. Specifically, we choose a widely used method, *Sent2Vec* [33], to complete our node embedding. It adopts a simple but efficient unsupervised objective to train distributed representations of sentences. Each slice is presented by a feature matrix V_i'' after node embedding, whose length is $m \times n$, where m denotes the number of nodes within the slice and n indicates the dimension of the embedding vector, which is 100 in this paper.

Slice Embedding Module. To better capture the structure of a graph, we leverage a graph-based model to embed the slices further. The reason is that the graph-based model regards the source code as graphs with comprehensive syntactic and semantic information. Moreover, there are various kinds of graph-based networks whose insights are aggregating neighborhood information [34], [35], [36]. Among them, *Gated Graph Recurrent Network* (GGNN) is more adaptive for our task of detecting vulnerability based on structured and semantically informative graph information. So we adopt GGNN for slice embedding.

For the node feature matrix V_i'' of the slice, GGNN converts it into a slice feature matrix by embedding each node with its neighborhoods. The output is V_i', whose the length is $m \times n'$, where n' means the dimension of the slice feature matrix, here we set it to 200.

Slice Detecting and Filtering Module. This module is designed to detect vulnerabilities at slice-level and filter out the "useless" slices based on the detection results.

The slice-level detector fuses further the matrices (V_i', V_i'') extracted by the previous modules and maps them into a score, *i.e.,* the final binary detection result. In detail, we perform convolutions for V_i' and (V_i', V_i''), respectively. After each convolution layer, we employ *Relu* [37] as the activation function and then use *Maximum Pooling* to generate Y_i and Z_i. Finally, Y_i and Z_i are fed into a *Fully Connected Layer* for dot product operation with a *Sigmoid* function after *Averaging Pooling*. Moreover, the loss function for penalizing the incorrect classification is *Binary Cross Entropy* (BCELoss). We train the model with *Adam* [38] with a learning rate of 0.0001. The output is the predicted probability of the slice, as illustrated in Step 6 of Fig 3.

In practice, the slices with the lowest predicted probability are considered the least relevant to the vulnerability and are referred to as "useless" slices. They contribute little to the results of function-level vulnerability detection. Therefore, we rank each slice's predicted probability p from a function and discard a certain percentage of slices according to a set threshold (set to 25% in this paper). For example, we drop out the slice *S4* with the lowest value of probability in Step 6 of Fig 3.

Note that we train the slice-level detector before using it to complete the filtering task. To this end, we first annotate the slices in the same way as in previous work [23] [12], *i.e.,* the slice that contains the deletions in the vulnerability patch is labeled as a 1 (vulnerable). Otherwise it is 0 (not vulnerable). Then we use them for training our slice-level vulnerability detector. Additionally, the slice whose value of predicted probability p is greater than 0.5 is regarded as vulnerable when evaluating the effectiveness of the slice-level detector.

3.4 Vulnerability Detection

This phase is designed to detect vulnerabilities and output the interpretations. We leverage an attention-based GNN model as a functional-level detector. In addition, the interpretation is derived by combining the attention weight with the predicted probability of slice-level detector.

3.4.1 Vulnerability Prediction

The GNN model with the attention mechanism improves detection effectiveness and provides a coarse-grained explanation for the detection results.

GNN Model. In this part, we introduce the overall structure of the function-level detector, as shown in Step 7 of Fig 3. The input is the remaining slices of the function filtered by the slice-level detector. Similar to the process in Step 6, we first generate the node feature matrix (*i.e.,* V_i'', whose length is $m \times n$) and then generate the slice feature matrix (*i.e.,* V_i', whose length is $m \times n'$) via GGNN.

To simply the slice feature matrices, we employ pooling operations. In detail, we perform *Average Pooling* on the node feature matrix V_i'' and corresponding slice feature matrix V_i' of the remained slices within a function, respectively. Then we concatenate them into a matrix V_i, which presents the slice's feature. A function is comprised of all the slice representations $(V_i, i \in [1, k])$, where k denotes the number of slices after filtering.

Attention Module. Given the slice representations, we utilize the *Attention Mechanism* to strengthen the impact of important slices and assign the attention score for each slice within a function.

Specifically, the slice representation V_i is regarded as *key* and *query*. We multiply the *key* and *query* 1, whose result is used for *weight* calculation 2. After that, the product of *query* and *weight* is connected with *key*, and the attention weight s is the output after *linear* and *activation* layers 3. The formulas are as follows:

$$x = key \times query^T \tag{1}$$

$$weight = \frac{exp(x_i)}{\sum_{i=1}^{k} exp(x_i)} \tag{2}$$

$$s = tanh(linear(weight * query, key)) \tag{3}$$

In addition, the *Attention Mechanism* helps to aggregate all the slice representations V_i within the function into a vector V. Then, it is fed into the function-level model to generate the prediction result of the function, as shown in Step 7 of Fig 3. Since the structure of the function-level detector is the same as that of the slice-level one, it will not be described in detail due to the page limitation. Finally, the output of the function-level detector is detection results, showing whether the targeted function is vulnerable or not, and the attention weight for each slice.

3.4.2 Result Interpretation

In this part, we describe the process of interpreting the results of vulnerability detection.

To obtain the importance score for each code line, we rely on the slice's predicted probability p outputted by the slice-level detector and each slice's attention weight s within

a function outputted by the function-level detector. The predicted probability p represents the probability that this slice is vulnerable. The weight s indicates the importance of this slice in the detection of the overall function.

For each statement, we calculate its importance score W, which is the sum value of each slice's score containing this statement. The initial value of W is set to 0. First, we scan all slices for this statement. For each slice i that includes this statement, we calculate the slice's importance score W_i by $s \times p$. Note that the case of $p <= 0.5$ means the slice is not vulnerable; thus, the importance of this statement is weakened by $W = W - W_i$. Conversely, $p > 0.5$ implies that the slice is vulnerable and contains statements that contribute to the function detection result. Therefore, the importance of this statement is reinforced by $W = W + W_i$.

To better describe the interpretation method, we give a detailed example to illustrate the importance score W calculation for line 13 of the function in Fig 1. Its predicted probability p and the attention weight s of each slice within the function are shown in Table 1. We can see that line 13 appears in *S2*, *S4*, and *S7*. Among them, the predicted expectation of *S2* is greater than 0.5, whereas that of *S4* and *S7* are less than 0.5. According to the above formula, the final importance score W of line 13 can be calculated by $W2-W4-W7$ ($0.572\times0.336-0.395\times0.028-0.451\times0.052$), whose value is 0.157680, as shown in Fig 6.

After performing the above calculation, we can obtain the importance score of each statement within the function. The top-ranked ones are the explanations of the vulnerability detection results after sorting the importance scores of all statements. In other words, these statements are considered to be ones that make a significant contribution to whether the function is vulnerable or not.

4 Experiments

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

- RQ1: What is the detection performance of *mVulPreter* on detecting source code vulnerability?
- RQ2: Can *mVulPreter* give an accurate interpretation for the vulnerability detection results?
- RQ3: Can *mVulPreter* be used to scan for vulnerabilities in real-world products?

4.1 Experiment Settings

4.1.1 Dataset

We conduct all experiments on a subset of the widely used dataset *Big-Vul* [39]. We adopt it for our experiments for two reasons. First, it is a high-quality vulnerability dataset consisting of real vulnerability functions. Specifically, it covers CVE entries from 2002 to 2019 in 348 different open source projects, with a total of 11,834 vulnerable functions and 253,096 non-vulnerable functions. Second, it is necessary to have vulnerability patches and the corresponding functions before and after being patched, which are used to annotate samples at function-level and slice-level. *Big-Vul* provides those that satisfy our needs.

It should be noted that the *Big-Vul* dataset crawls all the functions in the source files where the vulnerability patches are present. Then, the functions before the patch are labeled as vulnerable, and those after the patch are labeled as non-vulnerable. Moreover, the functions without any modifications are also kept and labeled non-vulnerable. Due to the limitations of the slice annotation approach described in Section 3.3.2, we have to remove functions without any deletions. Finally, our vulnerability dataset consists of 5,310 vulnerable and 7,601 non-vulnerable functions, obtaining 20,485 vulnerable slices and 164,110 non-vulnerable slices.

4.1.2 Implementations

We run all experiments on a machine with 128G RAM, 16 cores of CPU and a GTX 5000 GPU. Phases of *mVulPreter* are implemented with Joern [40], sent2vec [33], and PyTorch [41]. For the dataset, we randomly split the dataset according to the number of function samples into a training set, a validation set, and a test set, with a ratio of 8:1:1, which is similar to the common approaches [12]. Finally, the training set has 10,329 functions with 147,590 slices; the validation set has 1,291 functions consisting of 18,395 slices; and the test set has 1,291 functions including 18,610 slices. It should be noted that the above three subsets are not only used for the slice-level detector but also for the function-level one.

4.1.3 Metrics

The metrics used to measure the effectiveness of *mVulPreter* are the same as others [8], [12], [14].

- *True Positive* (TP): the number of samples correctly predicted as vulnerable.
- *True Negative* (TN): the number of samples correctly predicted as non-vulnerable.
- *False Positive* (FP): the number of samples incorrectly classified as vulnerable.
- *False Negative* (FN): the number of samples incorrectly classified as non-vulnerable.
- Accuracy=(TP+TN)/(TP+TN+FP+FN)
- Recall=TP/(TP+FN)
- Precision=TP/(TP+FP)
- F1=2∗Precision∗Recall/(Precision+Recall)

4.2 RQ1: Detection Effectiveness

We first present the detection performance of *mVulPreter* under different thresholds to filter "useless" slices. Specifically, we select nine thresholds (0%, 5%, 10%, 15%, 20%, 25%, 30%, 35%, and 40%) to commence our evaluations. 0% denotes that we do not filter any slices, which is a general function-level vulnerability detection process. 5% means that we discard the slices with the predicted probability in the last 5% and only keep the first 95% of that for subsequent function-level vulnerability detection. The predicted probability is the output of the trained slice-level vulnerability detector. In detail, we train and validate the detector using the slices from the training and validation sets (as Section 4.1.2), respectively. And it achieves an F1 score of 78.8% in the test. The final experimental results (*i.e.,* function-level detector's output) are presented in Fig 4. We see that the threshold is positively correlated with the detection performance at first. The higher the threshold, the better the detection performance. This suggests that our slice-level model can indeed filter some slices that do not contain

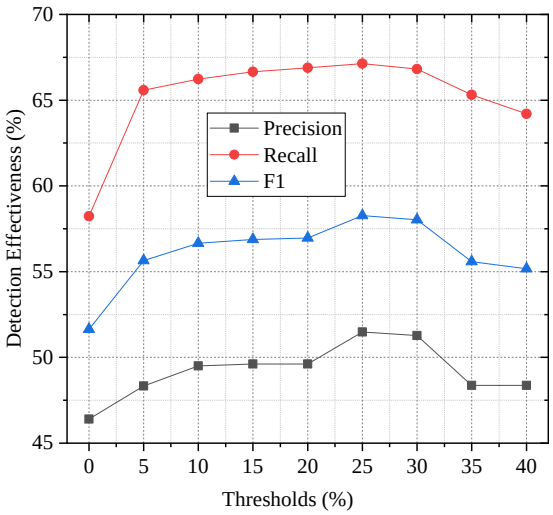


Fig. 4: The detection effectiveness of *mVulPreter* with different thresholds to filter program slices

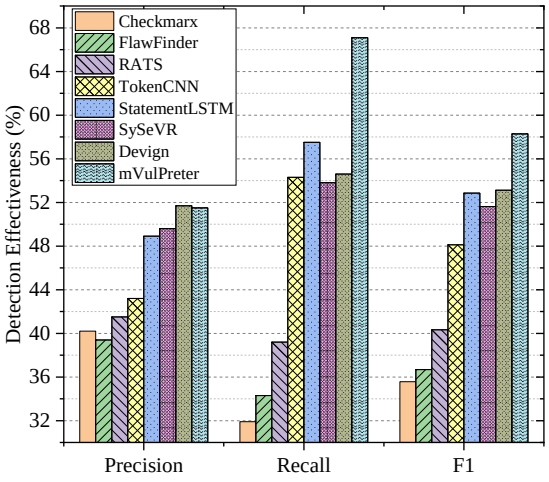


Fig. 5: The detection effectiveness of *Checkmarx*, *FlawFinder*, *RATS*, *TokenCNN*, *StatementLSTM*, *SySeVR*, *Devign*, and *mVulPreter*

vulnerable code, which helps to reduce the noise within the function. As a result, it makes the subsequent function-level vulnerability detection better. When the threshold increases to 25%, the model can obtain the best detection performance. However, if we continue to increase the filtering threshold, the detection performance of the model will decrease as the threshold increases. It is reasonable because more slices will be discarded if the threshold is set too high. But some of those may be vulnerable, reducing the detection performance. Overall, *mVulPreter* achieves the best detection performance when we choose to filter slices with predicted probability in the bottom 25% as shown in Fig 4 .

We then compare *mVulPreter* with several vulnerability detection tools, including one commercial static vulnerability detector (*i.e., Checkmarx* [9]), two static analysis systems (*i.e., FlawFinder* [11] and *RATS* [10]), and four deep learning-based vulnerability detection methods[2] (*i.e., TokenCNN* [17], *StatementLSTM* [22], *SySeVR* [23], and *Devign* [14]).

As for the commercial tool (*i.e., Checkmarx*) and two static analysis systems (*i.e., FlawFinder* and *RATS*), the detection performance in Fig 5 shows that their precision, recall, and F1 are not ideal. For example, the recall of *Checkmarx* is only 31.9%, which means that *Checkmarx* can only detect 31.9% of vulnerabilities in the experimental dataset. They depend on rules or patterns defined by human experts. However, it is impossible for the experts to define all patterns of vulnerabilities, resulting in poor detection performance.

As for deep learning-based detectors, we compare *mVulPreter* with four state-of-the-art ones (*i.e., TokenCNN* [17], *StatementLSTM* [22], *SySeVR* [23], and *Devign* [14]).

With token-based method. *TokenCNN* first transforms the source code into a token sequence by lexical analysis and then embeds it into a vector, which is input for *Convolutional Neural Network* (CNN) model. Obviously, *TokenCNN* regards the source code as plain text, which lacks consideration of the semantic and structural information of the source code. It leads to a weaker performance in vulnerability detection than *mVulPreter*.

2. For convenience, we name the unnamed models based on their code representations (*e.g.,* token) and neural networks (*e.g.,* CNN).

With statement-based method. *StatementLSTM* treats each line of code as a sentence and embeds it into a fixed-length vector representation. Then they are fed into a *Long Short-Term Memory* (LSTM) model for training a vulnerability detector. Similar to *TokenCNN*, *StatementLSTM* does not consider any program structure details, resulting in poor detection performance compared with that of *mVulPreter*.

With slice-based method. At slice-level, *SySeVR* is one of the most representative works. It performs slicing on the targeted program to generate the code gadgets and then embeds them into corresponding vector representations. Unlike methods at other levels, the extraction of slices purifies the training samples, *i.e.,* it removes many irrelevant statements with vulnerabilities. However, the entire slice is also treated as a piece of text. So the dependencies between code lines are not captured in the slice, resulting in poor detection performance.

With graph-based method. As for *Devign*, it first applies complex program analysis to extract a graph representation (*i.e., Code Property Graph* (CPG)) and then uses a general graph neural network to detect vulnerability. The representation of the CPG enables the training sample to contain comprehensive semantic and syntactic information about the code. However, such complex graph also contains certain normal behaviors, making the detector can not perform as well as *mVulPreter*.

Summary: mVulPreter can achieve the best performance when we discard the slices with the predicted probability in the last 25%. At this time, the detection performance of mVulPreter is better than that of Checkmarx, FlawFinder, RATS, TokenCNN, StatementLSTM, SySeVR, and Devign.

4.3 RQ2: Interpretability

An Interpretation Example. First, we take the Fig 1 as an example to illustrate that how *mVulPreter* interprets the detection results. Table 1 presents the output of the two models of *mVulPreter*. The column "Slice-level model prediction" shows the predicted probability value of each slice within the function, which is the output of the slice-level model. And the column "Function-level model attention weight"

TABLE 1: The output of mutiple-granularity models for the slices in Fig 1

	Criterion line number	Variable name	Vulnerability interesting points type	Slice-level model prediction	Function-level model attention weight	Lines within the slice
S1	2	*n*	integer	0.547	0.186	2,15,16,17,20,21,22,25,26,30,31,32,35
S2	1	*v_off*	pointer	0.572	0.336	1, 6, 13, 35
S3	1	*mdct_buf*	pointer	0.553	0.197	1, 21, 22, 31, 32, 35
S4	7	*saved_sample*	integer	0.395	0.028	7, 8, 9, 11, 13, 35
S5	2	*i*	integer	0.466	0.068	2, 5, 16, 17, 26, 28, 29
S6	1	*X*	integer	0.472	0.075	1, 16, 17, 19, 26, 28, 29
S7	1	*v0*	integer	0.451	0.052	1, 8, 13, 35
S8	1	*div*	integer	0.410	0.035	1, 3, 7, 9, 11, 46
S9	1	*out*	pointer	0.446	0.023	1, 35, 46

shows the weight value of the slices output by the function-level model. The score of each code line is calculated by accumulating the values of the above two columns, as described in Section 3.4.2. Finally, the scores of code lines in the function are ranked, where higher scores represent a higher probability of being vulnerable. This ranked list of statement scores interprets the detection results and provides security researchers with a guideline for vulnerability analysis.

```
35  dsp->vector_fmul_add(out, v, sbr_qmf_window , zero64, 64 >> div);   0.358105
21  v[    n] =  mdct_buf[0][63 - 2*n];                                  0.210683
22  v[63 - n] = -mdct_buf[0][62 - 2*n];                                 0.210683
31  v[    n] = -mdct_buf[0][63 -  n] + mdct_buf[1][  n  ];              0.210683
32  v[127 - n] =  mdct_buf[0][63 -  n] + mdct_buf[1][  n  ];            0.210683
1   static void sbr_qmf_synthesis(DSPContext *dsp, FFTContext *mdct,    0.206613
            float *out, float X[2][38][64],
            float mdct_buf[2][64],
            float *v0, int *v_off, const unsigned int div) {
6   if (*v_off == 0) {                                                  0.192192
13  v = v0 + *v_off;                                                    0.157680
15  for (n = 0; n < 32; n++) {                                          0.101742
25  for (n = 1; n < 64; n+=2) {                                         0.101742
```

Fig. 6: The interpretation results for Fig 1

For the scores of all code lines within the function in Fig 1, we select the top 10 for display (see Fig 6). The value shown in red on the right side denotes its corresponding final importance score. According to the analysis in Section 2, the statements related to the vulnerability are line 6 (*i.e.,* the root cause) and lines 21-22, 31-32 (*i.e.,* trigger location). As shown in Fig 6, the top-scoring statements are lines 35, 21-22, 31-32, 1, and 6, where all the interpretation results we expect are included. This proves that *mVulPreter* can provide a valid interpretation for the vulnerability detection.

Aside from the correct interpretation, some intriguing phenomena merit our attention. The statement at line 35 obtains the highest score, whose type is a function-call. Multiple parameters are passed in it, such as `out`, `v`, and `sbr_qmf_window`. Since function-call statements generally have more variables than others, they tend to have more data dependencies with others. In other words, the number of slices that contain function-call statements is greater than other types of statements, resulting in a higher importance score than others. To avoid "false positives" caused by function-call statements, we recommend ignoring them if they are not sensitive APIs.

In addition, the statement at line 1 obtaining a high score is a function header statement (*i.e.,* Joern parsed as METHOD type), which is also a type with multiple parameters. Therefore, at least one slice is derived from each parameter when conducting data dependency analysis. It means that many slices cover this statement, resulting in it achieving a high score. However, we rarely consider function header statements as vulnerability-related statements.

Comparisons. In this part, we pay attention to accuracy and time performance of *mVulPreter* and two comparative interpreters. As shown in Table 2, the evaluation metrics, *accuracy*, is adopted by [42]. Specifically, [42] has mentioned that if an interpretation result has an overlap with any statement in the code changes that fix the vulnerability, it is considered correct. And *accuracy* can be calculated as a ratio between the number of correct interpretations over the total number. They also have proven that when the number of statements is higher than 5, the accuracy of interpreters increases more slowly. Thus, we evaluate the accuracy of the top 5 most important statements in the interpreted results of each interpreter. For scalability, we randomly select 500 functions from our dataset to estimate runtime. Therefore, the time performance reported in Table 2 is the seconds required for each interpreter to interpret above 500 functions.

TABLE 2: The accuracy and runtime of interpretation

	GNNExplainer	*PGExplainer*	*mVulPreter*
Accuracy	0.53	0.55	**0.65**
Scalability	10595.3s	167.5s	**0.98s**

For accuracy, *PGExplainer* [25] performs slightly better than *GNNExplainer* [24]. *GNNExplainer* is designed to explain a single instance. Its explanation result is not learned from the model being explained, so it may be affected by the suboptimal generalization performance. In contrast, *PGExplainer* uses a parametric interpreter that considers the structure of the model being explained, making its explanation result accurate. Among them, *mVulPreter* achieves the best interpretation performance. Unlike the above two comparative tools that are common interpreters designed for GNN, *mVulPreter*'s interpretation is derived from the analysis of vulnerability characteristics. That is, *mVulPreter* believes that a statement derived from a vulnerable slice may contribute more to function-level vulnerability detection and is more likely to be a vulnerable statement. Moreover, the weights assigned to each slice by the attention-based function-level detector also indicate its contribution. Therefore, by intelligently combining the probabilities with weights of the slices, the importance score of each statement can be obtained. The higher the score is achieved, the more the statement contributes to vulnerability detection. Thus,

for the interpretation of vulnerability detection, *mVulPreter* is superior as it is an interpreter designed based on vulnerability features.

For scalability, the runtime of *mVulPreter* is significantly less than the two comparative tools. *GNNExplainer* must be retrained for each interpreted instance, so it is time-consuming when interpreting a large number of nodes. In contrast, *PGExplainer* is trained to parameterize the interpreter, making it directly interpret data. As a result, it takes less time than *GNNExplainer*. However, *mVulPreter* is not an additional interpretation model and therefore does not require any high overhead training process. It only requires a simple arithmetic operation on each statement's weights and probabilities to obtain the importance score, whose interpretation consumes a short time.

Summary: By combining the predicted probabilities of the slice-level vulnerability detector with the attention weights of the function-level vulnerability detector, mVulPreter can interpret the detection results and has the ability to pinpoint vulnerable statements. Moreover, the effectiveness and scalability of mVulPreter outperform state-of-the-art interpreters (GNNExplainer and PGExplainer).

4.4 RQ3: Case Study

In this subsection, we perform a case study to examine the practicability of *mVulPreter* on real-world vulnerability discovery. Specifically, we download four popular programs as our targets: Libav [43], Xen [44], Openssl [45], and Thunderbird [46]. The versions of these products include one old version and the latest version. Table 3 presents the summary of our collected products. The total number of functions that can be successfully analyzed by Joern [28] in these products is 635,451. In other words, *mVulPreter* analyzes a total of 635,451 functions, with a total of more than 25 million lines of code. Moreover, it costs *mVulPreter* 92.9 minutes to detect.

TABLE 3: The details of our selected open-source products

OpenSource Products	#Files	#Function	#Lines of code
Libav-11.12	1,343	9,807	552,768
Libav-12.3	1,509	10,760	625,034
Xen-4.14.0	5,151	71,230	2,872,957
Xen-4.15.1	5,453	74,524	2,942,773
Openssl-1.0.0s	912	5,360	324,060
Openssl-3.0.0-beta2	1,335	11,713	519,096
Thunderbird-80.0b5	17,297	223,838	8,603,285
Thunderbird-91.3.2	17,678	228,219	8,851,564
Total	50,678	635,451	25,291,537

In practice, our scanning results are also encouraging since we discovered 28 vulnerabilities. Specifically, we first train *mVulPreter* using the experimental dataset used in the former subsections. After completing the training phase, we feed 635,451 functions from eight products into the trained *mVulPreter* and collect the corresponding predictions. To check if the predicted vulnerabilities are real vulnerabilities or not, we then apply further manual analysis to compare them with our collected real vulnerabilities. If the two are found to belong to the same pattern, it will be judged to be a real vulnerability.

The analysis result shows that 28 predicted functions correspond to patterns of known vulnerabilities in NVD. Table 4 presents the details of our detected vulnerabilities, including the corresponding CVE ID in NVD, vulnerable products reported in NVD, and so on. From four old versions of our selected products, we detect 17 vulnerabilities. From the four latest versions of these products, *mVulPreter* discovers 11 vulnerabilities.

Summary: mVulPreter discovers 28 real-world vulnerabilities by scanning eight open-source products. Such results demonstrate the capability of mVulPreter in real-world vulnerability detection.

5 Discussion

5.1 Threats to Validity

Filtering "useless" slices to purify the function semantics for accurate vulnerability identification may cause some inaccuracies. We mitigate the threat by adopting a comprehensive evaluation based on nine different thresholds to find a suitable candidate. Also, inaccuracies in detecting vulnerabilities in open-source products (*i.e.,* Libav, Xen, Openssl, and Thunderbird) are inevitable since *mVulPreter* may cause some false positives. The threat is mitigated by deeply comparing the pattern of detected vulnerabilities with the pattern of real-world vulnerabilities in NVD.

5.2 Future Work

In practice, the most time-consuming phase of *mVulPreter* is PDG extraction of functions. In our future work, we plan to design a new static analysis tool or try other static analysis tools (*e.g., Frama-C* [47]) to achieve more efficient PDG generation. Moreover, since most vulnerability detection systems are closed source, we only compare *mVulPreter* with seven tools. In the future, we will conduct detailed comparative analyses on additional systems. Although *mVulPreter* can maintain better effectiveness than comparative tools, its TNR is not ideal. In other words, some of our detected vulnerabilities may be false positives. For interpretation, *mVulPreter* employs a simple formula to calculate the importance scores of the statements. It can also be further enhanced by further experiments to optimize the method that combines weight and predicted probability. In our future work, we plan to leverage directed fuzzing [48], [49] on our detected vulnerabilities to mitigate the situation.

6 Related Work

Deep Learning-based Vulnerability Detection. There are two main granularity models for deep learning-based vulnerability detection: function-level and slice-level. Function-level ones have the advantage of covering relatively complete vulnerability features. But they introduce more irrelevant statements and have a coarser granularity of detection (*e.g.,* [50], [14], [51], [18]). For example, Feng *et al.* design a tree-based vulnerability detector. They first apply static analysis to extract the AST of programs and then apply a preorder traversal search algorithm to convert them into sequences. Finally, these sequences are used to train a *Bidirectional Gated Recurrent Unit* (BGRU) model to detect vulnerability. *Devign* [14] applies a general GNN to detect vulnerability. It contains a novel convolutional module to extract useful features from the learned rich node representation for graph-level classification. Conversely, slice-level

TABLE 4: 28 Vulnerabilities discovered by *mVulPreter* from eight real-world products

Target product	CVE ID	Vulnerable product reported	Vulnerability release date	Vulnerable file in the target product	Newest version of target product patched or not
Libav 11.12	CVE-2011-3893	Google Chrome	11/11/2011	libavcodec/vorbis.c	Not patched
	CVE-2014-8543	FFmpeg	11/5/2014	libavcodec/smc.c	Patched
	CVE-2015-8662	FFmpeg	12/23/2015	libavcodec/jpeg2000dwt.c	Not patched
	CVE-2018-12460	FFmpeg	6/15/2018	libavcodec/mpegvideo.c	Not patched
Libav 12.3	CVE-2011-3893	Google Chrome	11/11/2011	libavcodec/vorbis.c	Not patched
	CVE-2015-8662	FFmpeg	12/23/2015	libavcodec/jpeg2000dwt.c	Not patched
	CVE-2018-12460	FFmpeg	6/15/2018	libavcodec/mpegvideo.c	Not patched
Openssl-1.0.00s	CVE-2015-0205	OpenSSL	1/8/2015	crypto/asn1/x_name.c	Not exist
	CVE-2017-3737	OpenSSL	12/7/2017	crypto/asn1/tasn_dec.c	Patched
	CVE-2018-0737	OpenSSL	4/16/2018	crypto/x509/x509_cmp.c	Patched
	CVE-2019-1563	OpenSSL	9/10/2019	crypto/cms/cms_smime.c	Patched
Openssl-3.0.0-beta2	CVE-2018-0737	OpenSSL	4/16/2018	crypto/conf/conf_ssl.c	Not patched
Thunderbird-80.0b5	CVE-2007-5947	Mozilla Firefox, SeaMonkey	11/13/2007	docshell/base/nsDocShell.cpp	Patched
	CVE-2014-1494	Mozilla Firefox	3/19/2014	./prefetch/nsOfflineCacheUpdate.cpp	Not exist
	CVE-2015-0818	Mozilla Firefox, Firefox ESR, SeaMonkey	3/23/2015	./base/nsDocShell.cpp	Patched
	CVE-2017-14062	Libidn2	8/31/2017	netwerk/dns/punycode.c	Not patched
	CVE-2019-12904	Libgcrypt	6/19/2019	./cipher/rijndael.c	Patched
Thunderbird-91.3.2	CVE-2007-5947	Mozilla Firefox, SeaMonkey	11/13/2007	.base/nsDocShell.cpp	Not patched
	CVE-2016-1952	Mozilla Firefox, Firefox	3/13/2016	dom/console/Console.cpp	Not patched
	CVE-2017-14062	Libidn2	8/31/2017	netwerk/dns/punycode.c	Not patched
	CVE-2018-5156	Thunderbird, Firefox ESR, Firefox	10/18/2018	./html/HTMLMediaElement.cpp	Not patched
Xen-4.14.0	CVE-2013-4533	QEMU	11/4/2014	./hw/pxa2xx.c	Patched
	CVE-2016-10028	QEMU	2/27/2017	./vhost-user-gpu/virgl.c	Patched
	CVE-2017-7471	QEMU	7/9/2018	./9pfs/9p-proxy.c	Not patched
	CVE-2017-8309	QEMU	5/23/2017	./audio/audio.c	Patched
Xen-4.15.1	CVE-2012-2652	QEMU	8/7/2012	.qemu-xen-traditional/block.c	Not patched
	CVE-2014-9801	Android	7/10/2016	./libfdt/fdt_rw.c	Not patched
	CVE-2017-7471	QEMU	7/9/2018	./9pfs/9p-proxy.c	Not patched

vulnerability detection has the advantage of fine granularity. They regard manually summarized vulnerability points as the slicing criterion [52], [20], [23], [13], [18]. For example, *VulDeePecker* [12] first extracts the program slices of a program and then trains a detector using *bidirectional long short-term memory* (BLSTM). *muVulDeePecker* [13] improves the implementation of *VulDeePecker*. It introduces the concept of code attention and uses it to help *VulDeePecker* complete multiclass vulnerability detection. However, not all vulnerabilities are related to the above vulnerability points, so sliced samples may contain incomplete vulnerability information.

In contrast to the previous work, *mVulPreter* intend to combine the fine-granularity of slice-level vulnerability detection with the full semantics of function-level vulnerability detection to achieve more effective vulnerability detection.

Vulnerability Detection Interpretation. Interpreting detection is an important research problem for deep learning-based vulnerability detection. Zou *et al.* [53] propose an interpretable framework that enables the identification of a small number of tokens that contribute significantly to the vulnerability prediction of the detector, which perturbs the samples near the decision boundary by applying perturbations to them to find out the significant token combinations in the samples to obtain important features. Similarly, Li *et al*. [42] leverage *GNNExplainer* [24] to give an interpretation (*i.e.,* subgraphs of PDG) of prediction results for graph neural network-based vulnerability detection models, where *GNNExplainer* is also a perturbation-based model interpretation method. Specifically, it selects the most relevant subgraph structures to the prediction results by perturbing the graph structure and features.

These approaches rely on an additional interpreter to explain the results of vulnerability detection. However, the interpretation and detection models work in serial, increasing the time overhead. Conversely, *mVulPreter*'s interpretation is based on the characteristics of the vulnerability, which relies on the vulnerability detection model itself and therefore does not require additional time and computational effort.

7 Conclusion

In this paper, we propose to use slice-level vulnerability detection to assist function-level vulnerability detection.

By this, we design a novel multiple granularity graph-based vulnerability detector namely *mVulPreter*. To demonstrate the effectiveness of *mVulPreter*, we perform evaluations on a dataset containing 5,310 vulnerable functions and 7,601 non-vulnerable functions. The experimental results suggest that *mVulPreter* is superior to *Checkmarx* [9], *FlawFinder* [11], *RATS* [10]), *TokenCNN* [17], *StatementLSTM* [22], *SySeVR* [23], and *Devign* [14]. Meanwhile, *mVulPreter* can pinpoint which statements are more likely to be vulnerable and outperform two state-of-the-art interpreters (*i.e.*, *GNNExplainer* [24] and *PGExplainer* [25]). Finally, to validate the ability of *mVulPreter* on real-world vulnerability detection, we conduct a case study on more than 25 million lines of code. Through the scanning results, we discover 28 new vulnerabilities that are not reported in NVD.

ACKNOWLEDGMENTS

This work is supported by the Key Program of National Science Foundation of China under Grant No. U1936211.

REFERENCES

[1] "What is wannacry ransomware?" https://www.kaspersky.com/resource-center/threats/ransomware-wannacry, 2021.
[2] "The exactis breach: 5 things you need to know," https://blog.infoarmor.com/individuals-and-families/the-exactis-breach-5-things-you-need-to-know, 2020.
[3] "Cyber security vulnerabilities and their business impact," https://www.verizon.com/business/resources/articles/s/cyber-security-vulnerabilities-and-their-business-impact/, 2021.
[4] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P'12)*, 2012, pp. 48–62.
[5] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P'17)*, 2017, pp. 595–614.
[6] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the 2010 International Conference on Automated Software Engineering (ASE'10)*, 2010, pp. 447–456.
[7] J. Li and M. D. Ernst, "Cbcd: Cloned buggy code detector," in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 310–320.
[8] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: An automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*, 2016, pp. 201–213.
[9] "Checkmarx," https://www.checkmarx.com/, 2022.
[10] "Rough audit tool for security," https://code.google.com/archive/p/rough-auditing-tool-for-security/, 2022.
[11] "Flawfinder," http://www.dwheeler.com/flawfinde/r, 2022.
[12] Z. Li, D. Zou, S. Xu, X. Ou, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018, pp. 1–15.
[13] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2021.
[14] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "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)*, 2019, pp. 10 197–10 207.
[15] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "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)*, 2017, pp. 2539–2541.
[16] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "Vulsniper: Focus your attention to shoot fine-grained vulnerabilities," in *Proceedings of the 2019 International Joint Conference on Artificial Intelligence (IJCAI'19)*, 2019, pp. 4665–4671.
[17] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "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)*, 2018, pp. 757–762.
[18] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "Vulsniper: Focus your attention to shoot fine-grained vulnerabilities," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*, S. Kraus, Ed., 2019, pp. 4665–4671.
[19] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "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)*, 2018, pp. 757–762.
[20] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.
[21] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "{MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, 2020, pp. 1165–1182.
[22] G. Lin, W. Xiao, J. Zhang, and Y. Xiang, "Deep learning-based vulnerable function detection: A benchmark," in *Proceedings of the 21st International Conference on Information and Communications Security (ICICS'19)*, 2019, pp. 219–232.
[23] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.
[24] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," in *Proceedings of the 32nd Annual Conference on Neural Information Processing Systems (NeurIP'19)*, 2019, pp. 9240–9251.
[25] H. Yuan, H. Yu, S. Gui, and S. Ji, "Explainability in graph neural networks: A taxonomic survey," *arXiv preprint arXiv:2012.15445*, 2020.
[26] "CVE-2012-0850," https://nvd.nist.gov/vuln/detail/CVE-2012-0850., 2022.
[27] "Ffmpeg," http://www.ffmpeg.org/, 2021.
[28] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P'14)*, 2014, pp. 590–604.
[29] "Open-source code analysis platform for c/c++ based on code property graphs," https://joern.io/, 2021.
[30] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*, pp. 439–449.
[31] J. Silva, "A vocabulary of program slicing-based techniques," *ACM computing surveys (CSUR)*, vol. 44, no. 3, pp. 1–41, 2012.
[32] "What are graph neural networks?" https://venturebeat.com/2021/10/13/what-are-graph-neural-networks-gnn/, 2021.
[33] M. Pagliardini, P. Gupta, and M. Jaggi, "Unsupervised learning of sentence embeddings using compositional n-gram features," *arXiv preprint arXiv:1703.02507*, 2017.
[34] M. S. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*, ser. Lecture Notes in Computer Science, vol. 10843. Springer, 2018, pp. 593–607.
[35] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *stat*, vol. 1050, p. 20, 2017.
[36] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *Proceedings of the 4th International Conference on Learning Representations (ICLR'16)*, 2016.
[37] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for LVCSR using rectified linear units and dropout," in *Proceedings of the 38TH IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'13)*, 2013, pp. 8609–8613.
[38] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*, Y. Bengio and Y. LeCun, Eds., 2015.

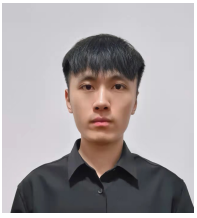
[39] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'2020)*, 2020, pp. 508–512.
[40] "Open-Source Code Querying Engine for C/C++." https://joern.io/, 2020.
[41] "Tensors and Dynamic neural networks in Python with strong GPU acceleration (PyTorch)," https://pytorch.org., 2022.
[42] Y. Li, S. Wang, and T. N. Nguyen, "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)*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds., pp. 292–303.
[43] "Libav," https://libav.org/, 2021.
[44] "Xen," https://xenproject.org/xen-project-archives/, 2021.
[45] "Openssl," https://www.openssl.org/, 2021.
[46] "Thunderbird," https://www.thunderbird.net/, 2021.
[47] "Frama-c," http://frama-c.com/, 2021.
[48] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, 2018, pp. 2095–2108.
[49] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, 2017, pp. 2329–2344.
[50] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. abs/2009.07235, 2020.
[51] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2020.
[52] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*, 2018.
[53] D. Zou, Y. Zhu, S. Xu, Z. Li, H. Jin, and H. Ye, "Interpreting deep learning-based vulnerability detector predictions based on heuristic searching," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 23:1–23:31, 2021.

Deqing Zou received the Ph.D. degree at Huazhong University of Science and Technology (HUST), in 2004. He is currently a professor of School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, China. His main research interests include system security, trusted computing, virtualization and cloud security. He has always served as a reviewer for several prestigious journals, such as IEEE TDSC, IEEE TOC, IEEE TPDS, and IEEE TCC. He is on the editorial boards of four international journals, and has served as PC chair/PC member of more than 40 international conferences.

Yutao Hu received the B.E. degree in Information Security from the China University of Geosciences (Wuhan), Wuhan, China, in 2019, and is currently pursuing the Ph.D. degree in Cyberspace Security at Huazhong University of Science and Technology, Wuhan, China. Her primary research interests lie in vulnerability anlysis based on deep learning.

Wenke Li is currently a senior undergraduate student at Huazhong University of Science and Technology, Wuhan, China. His research interests is in vulnerability detection and malicious code analysis based on deep learning.

Yueming Wu received the B.E. degree in Computer Science and Technology at Southwest Jiaotong University, Chengdu, China, in 2016 and the Ph.D. degree in School of Cyber Science and Engineering at Huazhong University of Science and Technology, Wuhan, China, in 2021. He is currently a research fellow in the School of Computer Science and Engineering at Nanyang Technological University. His primary research interests lie in malware analysis and vulnerability analysis.



Haojun Zhao received the B.S. degree in information security from the China University of Geosciences (Wuhan), Wuhan, China, in 2019, and is currently pursuing the Ph.D. degree in cyberspace security at Huazhong University of Science and Technology, Wuhan, China. His primary research interests include vulnerability detection and malware analysis.

Hai Jin received the Ph.D. degree in computer engineering from Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994. He is a Cheung Kung Scholars Chair Professor of computer science and engineering at HUST in China. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of the IEEE, a fellow of the CCF, and a member of the ACM. He has co-authored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.