VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities

Xu Duan[1,2], **Jingzheng Wu**[1,3], **Shouling Ji**[4], **Zhiqing Rui**[1,5], **Tianyue Luo**[1,6], **Mutian Yang**[1,6] and **Yanjun Wu**[1,3]
[1]Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences
[2]School of Computer & Communication Engineering, University of Science and Technology Beijing
[3]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[4]College of Computer Science and Technology, Zhejiang University
[5]Artificial Intelligence Academy, University of Chinese Academy of Sciences
[6]Beijing VuLab Technology Co.Ltd
duanxu1997@gmail.com, {tianyue, jingzheng08}@iscas.ac.cn

Abstract

With the explosive development of information technology, vulnerabilities have become one of the major threats to computer security. Most vulnerabilities with similar patterns can be detected effectively by static analysis methods. However, some vulnerable and non-vulnerable code is hardly distinguishable, resulting in low detection accuracy. In this paper, we define the accurate identification of vulnerabilities in similar code as a fine-grained vulnerability detection problem. We propose VulSniper which is designed to detect fine-grained vulnerabilities more effectively. In VulSniper, attention mechanism is used to capture the critical features of the vulnerabilities. Especially, we use bottom-up and top-down structures to learn the attention weights of different areas of the program. Moreover, in order to fully extract the semantic features of the program, we generate the code property graph, design a 144-dimensional vector to describe the relation between the nodes, and finally encode the program as a feature tensor. VulSniper achieves F1-scores of 80.6% and 73.3% on the two benchmark datasets, the SARD Buffer Error dataset and the SARD Resource Management Error dataset respectively, which are significantly higher than those of the state-of-the-art methods.

1 Introduction

Vulnerabilities have become a major threat to software security. Once the vulnerabilities are exploited by attackers, serious consequences will be caused. According to the data released by MITRE[1], as of Jan. 31. 2019, there have been 112,364 entries in CVE (Common Vulnerabilities and Exposures), and the amount of vulnerabilities is still exploding. The known vulnerabilities are often recurring throughout different software, which costs software security developers a lot of time to deal with. Detecting vulnerabilities accurately is a great concern in the field of software security.

```
int foo(char *str, size_t n)
{
    char buf[BUF_SIZE], *ar;
    size_t len = strlen(str);
    if(len >= BUF_SIZE) return ERROR;
    memcpy(buf, str, len);
    ar = malloc(n);
    if(!ar) return ERROR;
}
```

(a) Non-vulnerable Program

```
int foo(char *str, size_t n)
{
    char buf[BUF_SIZE], *ar;
    size_t len = strlen(str);
    if(len >= 2*BUF_SIZE) return ERROR;
    memcpy(buf, str, len);
    ar = malloc(n);
    if(!ar) return ERROR;
}
```

(b) Vulnerable Program

Figure 1: An example of the Buffer Error vulnerability. There are not many differences between the code of the two programs, which shows the necessity of fine-grained vulnerability detection.

To improve the effectiveness and efficiency of vulnerability detection and reduce manual auditing, many detection approaches have been proposed. The mainstream methods can be divided into two categories, including dynamic program analysis and static program analysis. Dynamic analysis identifies vulnerabilities in running programs, such as constructing abnormal inputs to trigger the vulnerable code. This kind of method can achieve high precision, but it is hard to reach all potential vulnerabilities and often trapped by the path explosion problem [Cadar *et al.*, 2008]. On the other side, static analysis methods detect vulnerabilities based on the static information, such as searching for defect patterns in the program source code. Nevertheless, such methods rely on manual efforts to define defect patterns and induce relatively high false positive and false negative.

Since a small number of code can lead to a vulnerability or fix a vulnerability, the code difference between a vulnerable and a non-vulnerable program can be very slight. For example, compared with Figure 1(a), the program shown in Figure 1(b) can cause a buffer overflow only because there is an additional "2" in the predicate statement. It can directly lead to a lower accuracy of traditional static analysis methods. The root cause is that these methods can not exactly identify and capture the critical factors that determine whether code is vulnerable, resulting in their wrong judgment.

In this paper, fine-grained vulnerability detection refers to distinguishing the vulnerabilities accurately from the vulner-

[1]https://cve.mitre.org/

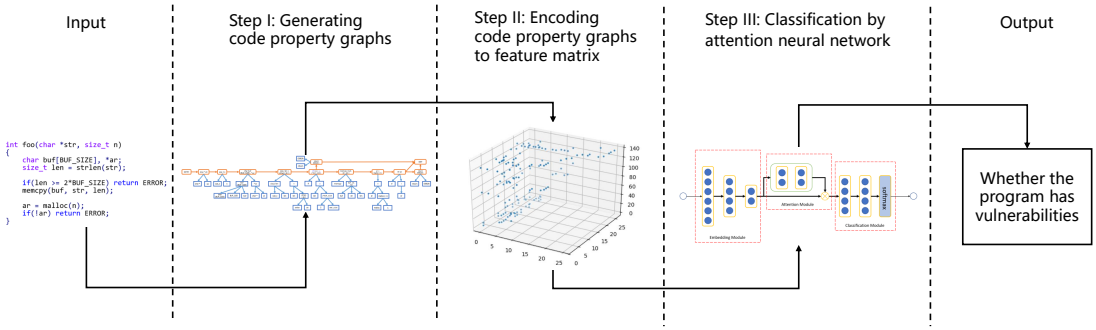

Figure 2: The general framework of VulSniper.

able and the non-vulnerable program that have slight differences, which is as shown in Figure 1. The essence of this problem is similar to the fine-grained classification problem in computer vision. In the field of computer vision, fine-grained classification refers to classifying images into more fine-grained categories, such as distinguishing between chihuahuas and bulldogs, rather than that between dogs and cats [Xiao *et al.*, 2014]. The input of this problem only has slight differences, and the key is how to capture and learn the slight differences exactly. To accurately capture those differences, the attention mechanism is proposed and has been studied in depth previously [Mnih *et al.*, 2014; Jaderberg *et al.*, 2015; Wang *et al.*, 2017; Fu *et al.*, 2017].

Although attention neural network is widely used and has achieved good performance in the field of computer vision, applying it to vulnerability detection has two challenges. First, we need a method to encode the source code of the program with minimal information loss. Second, the structure of the attention neural network should be elaborately designed to better adapt to the vulnerability detection problem. The general framework of VulSniper is shown in Figure 2. VulSniper firstly extracts the Code Property Graph (CPG) from the source code of the program to obtain more semantic features than the plain text of the source code. After that, the CPG is encoded to a feature tensor and input into the neural network. In our neural network model, we first embed the feature tensor through an embedding module, and then take advantage of the attention model to adequately learn the weights of different nodes in the CPG. Finally, fully connected layers are utilized to summarize the features and achieve the binary classification of the feature tensor. The result of the classification indicates whether the program has vulnerabilities.

We train the neural network model and evaluate VulSniper on the SARD Buffer Error dataset and the SARD Resource Management Error dataset, which are two sub-dataset of SARD[2]. There are both vulnerable and non-vulnerable code in each test case of SARD, where few code differences exist. Therefore, it is suitable for evaluating a fine-grained vulnerability detection model. Experimental results show that VulSniper significantly outperforms the state-of-the-art methods in fine-grained vulnerability detection.

[2]https://samate.nist.gov/SRD/index.php

The contributions of our work are summarized as follows:

- We propose a novel method to encode source code into a feature tensor, which has more semantic information and is beneficial for extracting program features and detecting vulnerabilities.
- We propose an attention neural network model for fine-grained vulnerability detection. This model learns attention weights on the different parts of the program code, effectively detecting vulnerabilities.
- We evaluate VulSniper on the two benchmark datasets, the SARD Buffer Error dataset and the SARD Resource Management Error dataset, and the results show that VulSniper can achieve much higher accuracy than the state-of-the-art methods.

The rest of this paper is organized as follows. In Section 2, we discuss the related work. In Section 3, we present our method of encoding the program source code. In Section 4, we present our neural network model with the attention mechanism. The experiments are discussed in Section 5, and finally, we conclude and summarize the paper in Section 6.

2 Related Work

Recently, many different approaches have been studied to improve the accuracy of vulnerability detection. The existing methods include dynamic program analysis [Wu *et al.*, 2017; Wu *et al.*, 2015; Chen *et al.*, 2018; Karamcheti *et al.*, 2018] and static program analysis [Yamaguchi *et al.*, 2014; Wu and Yang, 2017; Li *et al.*, 2018; Perl *et al.*, 2015; Yamaguchi *et al.*, 2015; Li and Ernst, 2012]. Static analysis is widely used due to its convenience on operation. Among such methods, it is common to use the graph structure to detect vulnerabilities bacause it contains the high semantic abstracted information which can better describe the features of the program. For example, Pham *et al.* proposed SecureSync which uses the graph search to detect the vulnerabilities in the software [Pham *et al.*, 2010]. It searches for the Abstract Syntax Tree (AST) and the Program Dependency Graph (PDG) of known vulnerabilities in those of software. However, the graph search method is less flexible, which can only detect the same vulnerabilities that are repeated in the software.

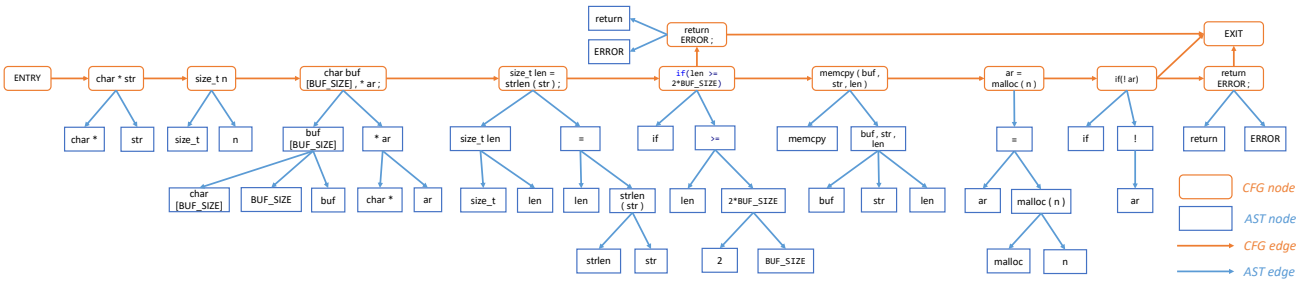

Figure 3: The simplified CPG of the program in Figure 1(b). The meaning of the graphics and colors is shown in the legends.

Nowadays, machine learning and deep learning models are very popular and have achieved enormous success in many different fields. Meanwhile, many vulnerability detection approaches employ machine learning to improve their performance [Grieco *et al.*, 2016; Feng *et al.*, 2016; Li *et al.*, 2018; Yamaguchi *et al.*, 2011; Neuhaus *et al.*, 2007; Yamaguchi *et al.*, 2012; Wang *et al.*, 2016]. Feng *et al.* proposed a method for detecting vulnerabilities with the high-level features of the graph structure, which has higher accuracy than the methods based on graph search [Feng *et al.*, 2016]. This method clusters the Control Flow Graphs (CFG) of all known vulnerabilities, and compares the CFG of program with the centroid of each class to determine if the program has vulnerabilities. Furthermore, several studies have tried deep learning models on vulnerability detection. For example, Li *et al.* proposed VulDeePecker which takes advantage of neural networks [Li *et al.*, 2018]. It first performs the program slicing on source code to exclude irrelevant code. Then, it embeds the code and trains BiLSTM to classify the sequences of word vectors to identify vulnerabilities, which achieves an accuracy of more than 90%. There are also some methods which can infer potential vulnerabilities by self-learning [Yun *et al.*, 2016; Yamaguchi *et al.*, 2013]. Despite that the above methods have successfully improved the performance in common vulnerability detection problems, their accuracy is still reduced in the face of fine-grained vulnerability detection problems.

3 Encoding Code Features

To detect fine-grained vulnerabilities, we design a neural network with the attention mechanism. Before the source code is input to the neural network, it needs to be encoded into a vector with a certain shape. In this section, we introduce how to encode the source code of a program into a feature tensor.

3.1 Generating Code Property Graph

For a program, it is not a good choice to treat the code as plain text, because it has more semantic structures, which is different from natural language. Instead, the graph structure (e.g., AST, CFG, and PDG) is a proper form for program representation, where the semantic information can be extracted and abstracted initially. Each graph structure describes a specific perspective of the program. For example, AST describes the syntactic structure, while CFG describes the execution path and the transfer of control flow of the program. The code property graph [Yamaguchi *et al.*, 2014] integrates three kinds of graphs, including CFG, AST and PDG, which contains complete semantic features.

As for implementation, VulSniper uses Joern[3] for CPG generation, which is an open-source tool for robust analysis of C/C++ code. It can efficiently generate a CPG and store it in the Neo4j[4] graph database. We generate the CPG for the code in Figure 1(b), and the result is shown in Figure 3. It should be noted that since the data dependency and control dependency information can be indirectly reflected by the encoding of AST and CFG in the subsequent steps, VulSniper simplifies the CPG by keeping the CFG and AST and removing the PDG information in it.

3.2 Encoding Feature Tensor

We continue to encode the simplified CPG as a feature tensor. The feature tensor is similar to the matrix representation of the graph, the difference is that we use vectors to represent the relations between nodes due to the more information. The feature tensor of the CPG is defined as follows:

Definition 1. A *feature tensor* $T(G)$ is a 3rd-order tensor with the shape of $n \times n \times m$, where G is a code property graph containing n nodes $\{v_1, v_2, ..., v_n\}$ and $\forall t_{i,j,k} \in T(G)$ is set as follows:

$$t_{i,j,k} = \begin{cases} 1 & is\_match(relations(v_i, v_j), f(k)) \\ 0 & otherwise \end{cases}$$

where $f(k)$ denotes the corresponding feature at k index, $relations(v_i, v_j)$ represents the features of the relation between v_i and v_j and is_match return $true$ when certain conditions are satisfied between $f(k)$ and $relations(v_i, v_j)$.

It should be noted that the value of m in definition 1 is specific to the programming language. In this paper, m is set to 144 according to the characteristics of C/C++ language. We use $M(v_i, v_j)$ to denote the 144-dimensional vector $(t_{i,j,1}, t_{i,j,2}, ..., t_{i,j,144})$ where M can be regarded as a matrix whose rows and columns are composed of the CPG nodes. $M(v_i, v_j)$ can be divided into five different fields according to the features to be encoded:

$$M(v_i, v_j) = \{V_i, V_j, I_{ij}, AST_{ij}, CFG_{ij}\},$$

where $\{\cdot\}$ denotes the concatenation of vectors.

[3]https://joern.readthedocs.io/en/latest/index.html
[4]https://neo4j.com/

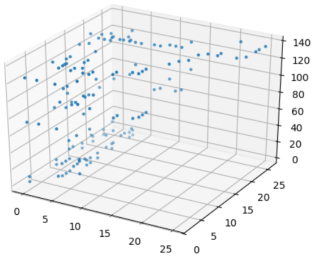

Figure 4: The feature tensor of the CPG in Figure 3. x axis and y axis both represent the CPG nodes. Along z axis are the 144-dimensional vectors which encode the relations between the nodes. The blue points represent 1 in the feature tensor.

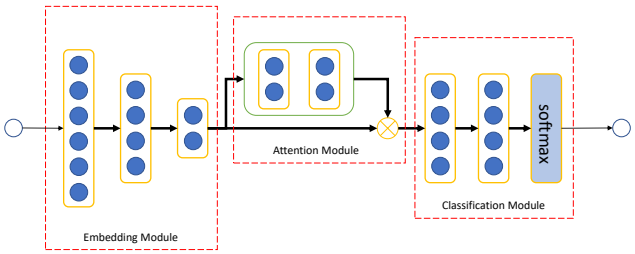

Figure 5: The structure of the neural network model in VulSniper.

Each field in $M(v_i, v_j)$ is a $|K|$-dimensional vector, where K is the set of the features to be encoded. Since each field encodes different features, their feature sets K is different. The details of each field are described as follows:

- V_i and V_j are 19-dimensional vectors and they encode the data types, modifiers or specific flags of v_i and v_j respectively. The feature sets of V_i and V_j are both $K = \{$*short, int, long, char, float, double, bool, const, static, *, void, unsigned, signed, struct, union, enum, function, constant, else*$\}$. The components in V_i (the same is true for V_j) can be expressed as follows:

$$t_{i,j,k} = \begin{cases} 1 & f(k) \in vartypes(v_i) \\ 0 & otherwise \end{cases},$$

  where $0 \leq k \leq 18$ and $vartypes(v_i)$ denotes the data types, modifiers or specific flags of v_i.

- I_{ij} is a 29-dimensional vector, which encodes the operator between v_i and v_j. The feature set of I_{ij} is $K = \{+, -, *, /, \%, \wedge, =, |, \&, \|, \&\&, <, >, <=, >=, ==, !=, + =, - =, * =, / =, \% =, \wedge =, |=, \& =, \ll, \gg, \ll=, \gg=\}$. The components in I_{ij} can be expressed as follows:

$$t_{i,j,k} = \begin{cases} 1 & f(k) = operator(v_i, v_j) \\ 0 & otherwise \end{cases},$$

  where $38 \leq k \leq 66$ and $operator(v_i, v_j)$ denotes the operator between v_i and v_j.

- AST_{ij} is a 57-dimensional vector, which encodes the parent-child relation between AST nodes. The elements in the feature set K of AST_{ij} are the 57 kinds of AST node types in the CPG (e.g., $Condition$ and $CallExpression$). The components in AST_{ij} can be expressed as follows:

$$t_{i,j,k} = \begin{cases} 1 & is\_parent(v_i, v_j) \text{ and } f(k) = type(v_j) \\ 0 & otherwise \end{cases},$$

  where $67 \leq k \leq 123$ and $is\_parent(v_i, v_j)$ return $true$ when there is a AST edge from v_j to v_i.

- CFG_{ij} is a 20-dimensional vector, which encodes the adjacency relation between CFG nodes. The elements in the feature set K of CFG_{ij} are the 20 kinds of CFG node types in the CPG (e.g., $ReturnStatement$). The components in CFG_{ij} can be expressed as follows:

$$t_{i,j,k} = \begin{cases} 1 & is\_adj(v_i, v_j) \text{ and } f(k) = type(v_j) \\ 0 & otherwise \end{cases},$$

  where $124 \leq k \leq 144$ and $is\_adj(v_i, v_j)$ return $true$ when there is a CFG edge from v_i to v_j.

We derive an encoded feature tensor for the CPG in Figure 3, and result is as shown in Figure 4. After that, because of the different length of the code, the above matrix M can have different sizes. However, static neural network requires the input to have the same size, so M needs to be adjusted to a same size. We define a *fixed size* which is set to 128, and then pad or cut the matrix M with different sizes to the *fixed size*. The details are as follows:

- When a matrix is smaller than the fixed size, we pad zeros at the end of the matrix.
- When a matrix is larger than the fixed size, we define a critical statement in the source code and cut off the code away from it. The critical statement are the statements that are highly vulnerable (e.g., sensitive API). The code far from the critical statement is less relevant to the vulnerability which means it is less important.

Through the above steps, we can obtain a feature tensor with a shape of $128 \times 128 \times 144$, which is used as the input to the neural network. The feature tensor greatly preserves the semantic information in the original program, which can be fully utilized by neural networks.

4 Attention-based Model

To overcome the difficulty of defining defect patterns in the static analysis methods, we use the neural network to automatically learn the patterns.

4.1 Overall Structure

Our neural network model is mainly composed of three modules: the embedding module, the attention module and the classification module. The embedding module transforms the feature tensor into a matrix composed of feature vectors of node by flattening and mapping. According to the idea of the soft attention, the attention module assigns attention weights to different nodes, which is implemented by bottom-up and top-down structures. The classification module at the end of

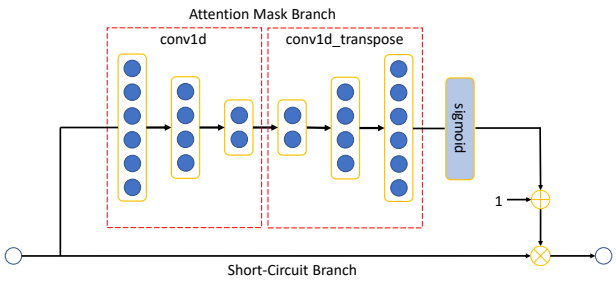

Figure 6: The structure of the attention module in VulSniper.

the network is composed of fully connected layers to summarizes the features. Finally, *softmax* is used to achieve the binary classification, which indicates whether the program has vulnerabilities. The structure of our neural network model is shown in Figure 5.

In the embedding module, we first use a fully connected layer to map the 144-dimensional vectors to a lower dimension $c = 6$. After that, we transform the shape of the feature tensor. Recall that $M(v_i, v_j)$ encodes the relations between v_i and v_j, the matrix $[M(v_i, v_1), M(v_i, v_2), ..., M(v_i, v_n)]$ describes the features of v_i. Therefore, we flatten the matrix to a vector which can be regarded as the feature vector of v_i. The length of the feature vector is $c \times n$ after the flattening. Finally, another fully connected layer is used to map the $c \times n$ vector to a lower dimension $d = 64$ for subsequent learning. All the parameters mentioned above are determined by our well-designed network structure.

4.2 Attention Module

The attention module in VulSniper can be regarded as a typical soft attention model. As shown in Figure 6, the attention module is divided into two branches: attention mask branch and short-circuit branch. The short-circuit branch directly takes the input $\boldsymbol{x}$ of the attention module as an output $S(\boldsymbol{x})$ without any other processing, which means $S(\boldsymbol{x}) = \boldsymbol{x}$. The attention mask branch learns the attention weights of different nodes through bottom-up and top-down structures. The size of the output $A(\boldsymbol{x})$ of the attention mask branch is the same with the input $\boldsymbol{x}$ of the attention module, so that $A(\boldsymbol{x})$ can be integrated with $S(\boldsymbol{x})$ by matrix dot production. In addition, before integrating $A(\boldsymbol{x})$ and $S(\boldsymbol{x})$, we normalize $A(\boldsymbol{x})$ with *sigmoid* and plus 1 to prevent the gradient from vanishing [Wang *et al.*, 2017]. The output of the attention module $O(\boldsymbol{x})$ can be summarized as follows:

$$O(\boldsymbol{x}) = (1 + A(\boldsymbol{x})) \cdot S(\boldsymbol{x}),$$

where $\boldsymbol{x}$ is the input of the attention module, and $\cdot$ denotes the matrix dot production.

As for the details of the attention mask branch, we use multiple one-dimensional convolution and the transpose of one-dimensional convolution to implement the bottom-up and top-down structure, respectively. With the one-dimensional convolution, the receptive field is gradually expanded to obtain surrounding and global information. Since the nodes in a certain AST subtree are adjacent in the feature tensor, the operation allows the nodes to perceive their located AST subtree. With the transpose of one-dimensional convolution, the high-level features are scaled to the same size as the input, so that the attention weights can be applied to the input.

Dataset	Test Cases	Bad Functions	Good Functions
CWE-119	7273	6586	9634
CWE-399	4124	3975	7854

Table 1: The amount of data in datasets. CWE-119 and CWE-399 refer to the SARD Buffer Error dataset and the SARD Resource Management Error dataset respectively.

5 Experiments

To evaluate the effectiveness of VulSniper, we conduct experiments and compare them with state-of-the-art models for fine-grained vulnerability detection.

5.1 Experiment Settings

Preparing data. SARD is a project maintained by NIST[5], which has a large number of production, synthetic, and academic security defects or vulnerabilities. The reason we use SARD as our data source is that the data in SARD has both vulnerable versions and non-vulnerable versions, which implies that it can effectively evaluate whether a model has good fine-grained vulnerability detection capabilities. In each test case, there is one bad function which contains a certain vulnerability and several good functions where the vulnerability in the bad function has been fixed. We use the syntactic analysis method to extract the bad and good functions in each test case. The amount of data finally obtained is shown in Table 1. VulSniper focuses on solving the problem of two certain types of vulnerability which are Buffer Error (i.e., CWE-119) and Resource Management Error (i.e., CWE-399). Some functions cannot be successfully extracted through the above method, resulting in a slight decrease in the number of bad and good functions. After that, all the extracted functions are encoded according to our encoding method, and the feature tensors of the bad and good functions are labeled as "1" (i.e., vulnerable), and "0" (i.e., non-vulnerable) respectively. In addition, we divide the datasets into a training set, a validation set, and a test set with a ratio of 6:2:2, which is similar to the common approaches.

Environment. We run our experiments on a machine with 32G RAM, 2T SSD and two Intel Xeon E7-4809 v4 CPUs operating at 2.10GHz.

5.2 Experiment Results

In this paper, we use five widely used metrics which are *false positive rate* (FPR), *false negative rate* (FNR), *true positive rate* (TPR), *precision* (P), and *F1-score* (F1) to evaluate the performance of VulSniper. FPR measures the proportion of false positive vulnerabilities in the entire population of non-vulnerable samples, FNR measures the proportion of false negative vulnerabilities in the entire population of vulnerable

[5]https://www.nist.gov/

Dataset	Tools	FPR (%)	FNR (%)	TPR (%)	P (%)	F1 (%)
CWE-119	Flawfinder	56.6	44.8	55.2	39.9	46.3
	RATS	68.7	31.3	68.7	40.5	51.0
	DeepSim	16.1	41.6	58.4	71.6	64.4
	VulSniper	6.42	26.2	73.8	88.7	80.6
CWE-399	Flawfinder	40.7	58.4	41.6	34.1	37.4
	RATS	33.9	63.8	36.2	35.0	35.6
	DeepSim	7.30	52.2	47.8	77.2	59.1
	VulSniper	8.49	32.7	67.3	80.4	73.3

Table 2: The metrics of the different tools on the two datasets.

samples, TPR measures the proportion of true positive vulnerabilities in the entire population of vulnerable samples, P measures the correctness of the detected vulnerabilities and F1-score takes both precision and recall rate into account, which can be regarded as their weighted average.

To demonstrate the effectiveness of VulSniper, we selected three tools as baselines to conduct comparative experiments on the SARD Buffer Error dataset and the SARD Resource Management Error dataset. The selected tools include Flawfinder[6], RATS[7], and DeepSim [Zhao and Huang, 2018]. Flawfinder and RATS are two widely used open source vulnerability detection tools. They do static searches based on lexical analysis. DeepSim is a deep learning model for measuring code similarity. There are two reasons for selecting DeepSim. On the one hand, it is very common to detect vulnerabilities by judging whether the code is similar to a known vulnerability code. On the other hand, since DeepSim is also based on neural networks, it can be used to compare the effectiveness of the network models. It should be noted that we have slightly adapted the above tools to get the best results on the above datasets. For example, Both Flawfinder and RATS can manually set the alarm threshold, so we experiment with different thresholds and present the best results.

Table 2 presents the experiment results on the SARD Buffer Error dataset and the SARD Resource Management Error dataset. As shown in Table 2, the performance of VulSniper is obviously better than the three baselines. VulSniper achieves a F1-score of 80.6% on the SARD Buffer Error dataset and a F1-score of 73.3% on the SARD Resource Management Error dataset, which shows it can detect vulnerabilities with few false positives and false negatives.

Meanwhile, we analyzed the reasons for the above results. Flawfinder and RATS work by using a built-in database of C/C++ functions with well-known problems and matching the source code text against those functions in database. However, they do not take the semantic features of the program into account, which leads to their lower F1-score. Although DeepSim utilizes control flow and data flow information, and also applies the neural network model, it does not make use of the attention mechanism, resulting in its poor performance on fine-grained vulnerability detection. As a comparison, VulSniper considers the above problems at the same time. Based on the fine-grained and semantic information, VulSniper takes advantage of the neural network model with the attention mechanism to well achieve the detection of fine-grained vulnerabilities.

[6]https://dwheeler.com/flawfinder/

[7]https://code.google.com/archive/p/rough-auditing-tool-for-security/

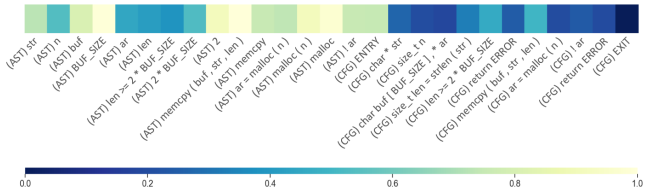

Figure 7: The visualization of attention weights for the feature tensor in Figure 4. The bright and dark colors represent higher and lower attention weights, respectively.

To better explore the effect of the attention mechanism in VulSniper, we visualize the normalized attention weights for the feature tensor in Figure 4 and the result is as shown in Figure 7. Since the type of vulnerability is Buffer Error, the attention is mainly on the area around the memory operation APIs such as *memcpy* and *malloc*. *BUF_SIZE* and *"2"* are also given sufficient attention, which indicates that the network has noticed the critical factors causing the vulnerability. In addition, it is not difficult to find that the attention of CFG nodes is generally lower than that of AST nodes, which shows that the network is more inclined to focus on finer-grained information. In general, the effect of the attention mechanism in VulSniper is similar to manually analyzing vulnerabilities, that is, first finding the sensitive operation and then examining if there are defects in the data associated with the operation, which demonstrates the effectiveness of introducing attention mechanisms to detecting fine-grained vulnerabilities.

6 Conclusion

In this paper, we propose VulSniper to solve the problem of low accuracy in detecting fine-grained vulnerabilities. VulSniper uses a well-designed encoding method to transform the source code into a feature tensor and takes advantage of attention neural networks to implement fine-grained detection of vulnerabilities. We use five common metrics to evaluate VulSniper and conduct a comparative experiment on the two benchmark datasets with Flawfinder, RATS and DeepSim. VulSniper achieves a F1-score of 80.6% and a F1-score of 73.3% on the SARD Buffer Error dataset and the SARD Resource Management Error dataset respectively, which is significantly better than state-of-the-art methods. Experimental results show that it is effective to improve the accuracy of detecting fine-grained vulnerabilities by utilizing the attention mechanism, which provides new insights for the future work.

Acknowledgments

This work is supported by National Key R&D Program of China (No.2018YFB0803600) and National Natural Science Foundation of China (No.61772507).

References

[Cadar *et al.*, 2008] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[Chen *et al.*, 2018] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[Feng *et al.*, 2016] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *CCS*, pages 480–491, 2016.

[Fu *et al.*, 2017] Jianlong Fu, Heliang Zheng, and Tao Mei. Look closer to see better: Recurrent attention convolutional neural network for fine-grained image recognition. In *CVPR*, pages 4476–4484, 2017.

[Grieco *et al.*, 2016] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using machine learning. In *CODASPY*, pages 85–96, 2016.

[Jaderberg *et al.*, 2015] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks. *CoRR*, abs/1506.02025, 2015.

[Karamcheti *et al.*, 2018] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. *CoRR*, abs/1808.08256, 2018.

[Li and Ernst, 2012] Jingyue Li and Michael D. Ernst. Cbcd: Cloned buggy code detector. In *ICSE*, pages 310–320, 2012.

[Li *et al.*, 2018] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *CoRR*, abs/1801.01681, 2018.

[Mnih *et al.*, 2014] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent models of visual attention. *CoRR*, abs/1406.6247, 2014.

[Neuhaus *et al.*, 2007] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *CCS*, pages 529–540, 2007.

[Perl *et al.*, 2015] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *CCS*, pages 426–437, 2015.

[Pham *et al.*, 2010] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Detection of recurring software vulnerabilities. In *ASE*, pages 447–456, 2010.

[Wang *et al.*, 2016] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *ICSE*, pages 297–308, 2016.

[Wang *et al.*, 2017] Fei Wang, Mengqing Jiang, Chen Qian, Shuo Yang, Cheng Li, Honggang Zhang, Xiaogang Wang, and Xiaoou Tang. Residual attention network for image classification. *CoRR*, abs/1704.06904, 2017.

[Wu and Yang, 2017] Jingzheng Wu and Mutian Yang. Lachouti: Kernel vulnerability responding framework for the fragmented android devices. In *ESEC/FSE*, pages 920–925, 2017.

[Wu *et al.*, 2015] Jingzheng Wu, Yanjun Wu, Zhifei Wu, Mutian Yang, Tianyue Luo, and Yongji Wang. Androidfuzzer: Detecting android vulnerabilities in fuzzing cloud. *Journal of Computational Information Systems*, 11(11):3859–3866, 2015.

[Wu *et al.*, 2017] Jingzheng Wu, Shen Liu, Shouling Ji, Mutian Yang, Tianyue Luo, Yanjun Wu, and Yongji Wang. Exception beyond exception: Crashing android system by trapping in "uncaughtexception". In *ICSE-SEIP*, pages 283–292, 2017.

[Xiao *et al.*, 2014] Tianjun Xiao, Yichong Xu, Kuiyuan Yang, Jiaxing Zhang, Yuxin Peng, and Zheng Zhang. The application of two-level attention models in deep convolutional neural network for fine-grained image classification. *CoRR*, abs/1411.6447, 2014.

[Yamaguchi *et al.*, 2011] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *WOOT*, pages 13–13, 2011.

[Yamaguchi *et al.*, 2012] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *ACSAC*, pages 359–368, 2012.

[Yamaguchi *et al.*, 2013] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *CCS*, pages 499–510, 2013.

[Yamaguchi *et al.*, 2014] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *SP*, pages 590–604, 2014.

[Yamaguchi *et al.*, 2015] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *SP*, pages 797–812, 2015.

[Yun *et al.*, 2016] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing api usages through semantic cross-checking. In *SEC*, pages 363–378, 2016.

[Zhao and Huang, 2018] Gang Zhao and Jeff Huang. Deepsim: Deep learning code functional similarity. In *ESEC/FSE*, pages 141–151, 2018.