EAI Endorsed Transactions
on Security and Safety

Research Article EAI.EU

Vul-Mirror: A Few-Shot Learning Method for Discovering Vulnerable Code Clone

Yuan He[1,2], Wenjie Wang[1], Hongyu Sun[3] and Yuqing Zhang[1,*]

[1]National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, Beijing, China
[2]School of mathematics and computer science, Dali University, Dali, China
[3]School of Cyber Engineering, Xidian University, Xi'an, China

Abstract

It is quite common for reusing code in soft development, which may lead to the wide spread of the vulnerability, so automatic detection of vulnerable code clone is becoming more and more important. However, the existing solutions either cannot automatically extract the characteristics of the vulnerable codes or cannot select different algorithms according to different codes, which results in low detection accuracy. In this paper, we consider the identification of vulnerable code clone as a code recognition task and propose a method named Vul-Mirror based on a few-shot learning model for discovering clone vulnerable codes. It can not only automatically extract features of vulnerabilities, but also use the network to measure similarity. The results of experiments on open-source projects of five operating systems show that the accuracy of Vul-Mirror is 95.7%, and its performance is better than the state-of-the-art methods.

Keywords: Vulnerability, few-shot learning, code clone, distance-metric.

Received on 25 May 2020, accepted on 09 June 2020, published on 10 June 2020

doi: 10.4108/eai.13-7-2018.165275

[*]Corresponding author. Email: zhangyq@nipc.org.cn

1. Introduction

With the rapid development of the open-source communities, code reuse has become very popular. Code clone is a code fragment with the same or similar code as the source code [1], it is one of the common ways of code reuse. Code clone improves development efficiency, but it will lead to potential security problems. If vulnerable code is reused, these vulnerabilities will spread to other applications and endanger the security of all related systems. For example, the OpenSSL heartbeat vulnerability (cve-2014-0160) [2] affects web sites, web servers, operating systems, and software applications because the affected system either uses the entire OpenSSL library or clones parts of the library for its system use. So we need an automatic method that can accurately detect code clone vulnerabilities with a minimal level of human intervention in different programs.

Once the vulnerability is exposed, researchers can analyze the vulnerable codes and extract the corresponding vulnerability pattern manually. Then the researchers can discover vulnerable codes in different programs based on the extracted vulnerability patterns. With the help of the traditional program analysis methods such as symbol execution [3-4] and taint analysis [5-7], the semi-automatic methods can be realized for discovering vulnerable codes in programs, but these tools are based on human learning and expert knowledge is required.

In order to alleviate the work of human intervention, researchers proposed many methods [8-11] to discover the vulnerable codes clone by calculating the similarity of the vulnerable codes and target codes. These methods are based on the assumption that similar code has the same vulnerability, which is tenable in most cases, so the method based on code comparison is feasible to discover vulnerable codes when two codes are the same or highly similar. But we know that most of the vulnerabilities are caused by a few codes, and there is a gap between code similarity and vulnerabilities. We need to extract the vulnerability characteristics and make a more accurate comparison to discover vulnerable code clones.

In this paper, we find that the process of discovering code clone vulnerability is very similar to that of face recognition. So we consider the discovering vulnerable code clone as a code recognition task and propose a few-shot learning model for discovering vulnerable code clone. The model not only automatically extracts vulnerability features, but also uses the network to measure similarity. To meet the requirements of few-shot learning, according to different types of clones, we build a sample set of vulnerabilities. By training the few-shot learning model with multitasks, the model learns how to compare codes. When we test target codes, the model can output the vulnerability most similar to the target codes. Based on code similarities, combined with the characteristics of vulnerabilities, vulnerable code clones can be discovered by the model. The contributions of this paper are as follows:

We first use few-shot learning to discover vulnerable code clones and propose a novel method named Vul-Mirror, which considers both codes features and similarity of codes.

We analyze the relationships between different types of code clones and original codes and effectively construct a code clone dataset for few-shot learning.

We implement the prototype system on five popular operating system codes, and the experimental results show that Vul-Mirror can achieve much higher performance than the state-of-the-art methods.

The remainder of the paper is organized as follows. Section 2 reviews the related work. Section 3 presents the design of the system. Section 4 describes our experimental results. Section 5 discusses the advantages and disadvantages of the method, and we conclude the paper in section 6.

2. Related work

Traditional static and dynamic analysis methods also can discover vulnerable code clones, but they rely heavily on security experts. We mainly review the method that relies less on security experts, these methods can be divided into two types: pattern-based methods and similarity-based methods.

2.1. Pattern-based methods

Li [12] assumes most of the codes are correct and proposed a method named CP-Miner to find code clone errors. CP-Miner parses a program and compares the resulting token sequences using the "frequent subsequence mining" algorithm known as CloSpan [13]. PR-Miner [14] focuses on the clone of vulnerability patterns, not the codes. With frequent patterns, it can discover paired vulnerabilities which need to appear together, such as "lock" and "unlock", "malloc" and "free". These methods can find code clone vulnerabilities, but in many cases, the vulnerabilities do not meet the frequent pattern.

Yamaguchi [15] provides a method called Chucky to discover miss-check vulnerabilities. Chucky maps code to vector space and extracts API (Application Programming Interface) usage patterns by principal component analysis. If the candidate functions are similar to vulnerable codes with high order, it should be audited. Yamaguchi [16] exploits patterns extracted from the abstract syntax trees of functions to detect semantic clones. Yamaguchi [17] proposes a method for inferring search patterns for taint-style vulnerabilities in C code. These methods extract vulnerability patterns semi-automatically, and each of the methods can only discover one fixed pattern.

Deep learning can automatically extract sample features. Li [18] develops a deep learning-based vulnerability detection system called VulDeePecker, which can extract more than one patterns automatically. μVulDeePecker [19] is based on VulDeePecker which can not only judge whether the code is vulnerable but also decide the type of vulnerability. However, due to the lack of a large number of high-quality training samples, the methods based on deep learning have not been widely used. To solve the problem of the sample shortage, few-shot learning [24-27] is proposed, but it is not applied in the field of vulnerability discovering.

2.2. Similarity-based methods

SourcererCC [20] and CCFinder [21] are typical lexicon-based approaches that only consider the similarity in the lexical level of code fragments. Deckard [22] is a standard syntax-based approach that uses structured information to identify a kind of code clones. White [23] proposes a deep learning method to detect code clones. These techniques are aimed at detecting as many code clones as possible but not for finding security vulnerabilities accurately.

ReDeBug [11] can quickly find some unpatched code clones of Type-3. However, it can hardly be applied to Type-2 clones. VulPecker [9] takes the advantages of a variety of algorithms to calculate similarity. However, its comparison algorithms are limited, and it characterizes vulnerability with a predefined set of features that need to be specified manually. VUDDY [8] normalizes tokens by replacing variables, function names, etc. with fixed names, and the hash values of functions are used to search code clones. Shi H. [10] adopts deep learning to detect vulnerable code clones. The common disadvantage of the methods mentioned above is that they only use a single pre-defined metric to compare codes base on token-level or line-level, and the vulnerability characteristics are not fully considered. Our approach not only uses a deep metric to compute the similarity of codes but also combines different vulnerabilities features to find vulnerable code clones.

3. System design

When we write a new program or check the codes, we want to test whether the program employs the historical vulnerable codes. We can compute the similarity of two codes, then further confirm whether there is a vulnerable code in the candidate code clones or not. To compute similarity, the first method is to compare exposed vulnerability with all the target code (see figure 1 (a)), the second method is to compare one target code with all historical vulnerabilities (see figure 1. (b)). The first method is suitable for comparison with a few vulnerabilities. The second method is suitable for finding multiple vulnerabilities. We select the second method to compare codes. In this way, the process of vulnerable code detection is similar to image recognition. Therefore we can use the method of image recognition to solve the problem of code clone vulnerability detection

program codes	vulnerable codes
code-1	vul-1
code-2	vul-1
…	…
code-n	vul-1

(a) One vulnerability vs. all codes

vulnerable codes	program codes
vul-1	code-1
vul-2	code-1
…	…
vul-n	code-1

(b) One code vs. all vulnerabilities

Figure 1. Two methods of code comparison

Referring to the method of image recognition, we design a system named Vul-Mirror to detect code clone vulnerabilities based on few-shot learning. In the training phase, we train a few-shot learning model by code clone vulnerabilities. The model learns how to find which vulnerability is most similar to the clone code from multiple vulnerabilities. In the testing phase, the clone codes are replaced with the target codes. The trained model can identify which historical vulnerability is most similar to the target code and output the similarity value. Because some code clone is low similar to original codes, we need further verify its vulnerable nature based on the output of few-shot learning model. So we add the vulnerability verification process in the testing phase. The framework of Vul-Mirror is shown in Figure 2.

To realize Vul-Mirror, we break the task down into four individual tasks: building data set, data processing, designing a few-shot learning model, and identification vulnerability.

3.1. Building data set

A few-shot learning model needs to be trained by a data set that every class has one original sample and k similar samples such as omniglot and miniImagenet. There is no data set for discovering the vulnerability, so we need to build the data set. Code clones can be divided into four types [1]. We treat exposed vulnerable codes as the original codes and code clones as target codes. We found that patterns of vulnerable code clones are as follows:

Type-1 clone is an exact clone where either completely copies the source codes or adds some comments at best. In type-1 clone, there is no change of function codes, so it has the same vulnerability as the original codes (see Table 1 original code and Type-1 clone). A buffer overflow vulnerability exists in the original code, type-1 clone has same vulnerability as the original code.

Type-2 clone is a renamed clone, it modifies variables or function names. If the changed name has nothing to do with any vulnerability, the kind of clone has the same vulnerability as the original codes (see Table 1 Type-2 clone 1). Otherwise, code clone has differently vulnerable from the original codes (see Table 1 Type-2 clone 2).

Type-3 clone is a restructured clone, and statements are inserted or deleted based on the type-2 clone. If the statements are related to some vulnerability, the kind of

Table 1. Code clone and vulnerability

Original code	Type-1 clone
1 void foo (int src[],int dst[])	1 void foo (int src[],int dst[])
2 {	2 {
3 int sum = 0;	3 int sum = 0;
4 for (i = 0; i < len; i++);	4 for (i = 0; i < len; i++);
5 sum += arr[i];	5 sum += arr[i]; //sum
6 memcpy(dst, src, sizeof(src))	6 memcpy(dst, src, sizeof(scr))
7 }	7 }
Type-2 clone 1	**Type-2 clone 2**
1 void foo (int src[],int dst[])	1 void foo (int src[],int dst[])
2 {	2 {
3 int sum = 0;	3 int var = 0;
4 for (i = 0; i < len; i++);	4 for (i = 0; i < len; i++);
5 sum += arr[i];	5 var += arr[i];
6 memcpy(dst, src, sizeof(dst))	6 memcpy(dst, src, sizeof(src))
7 }	7 }
Type-3 clone 1	**Type-3 clone 2**
1 void foo (int src[],int dst[])	1 void foo (int src[],int dst[])
2 {	2 {
3 int sum = 0;	3 int var = 0;
4 for (i = 0; i < len; i++);	4 for (i = 0; i < len; i++);
5 sum += arr[i];	5 var += arr[i];
6 memcpy(dst, src, sizeof(src))	6 //delete line
7 printf("\%d ",sum);// add line	7 }
8 }	

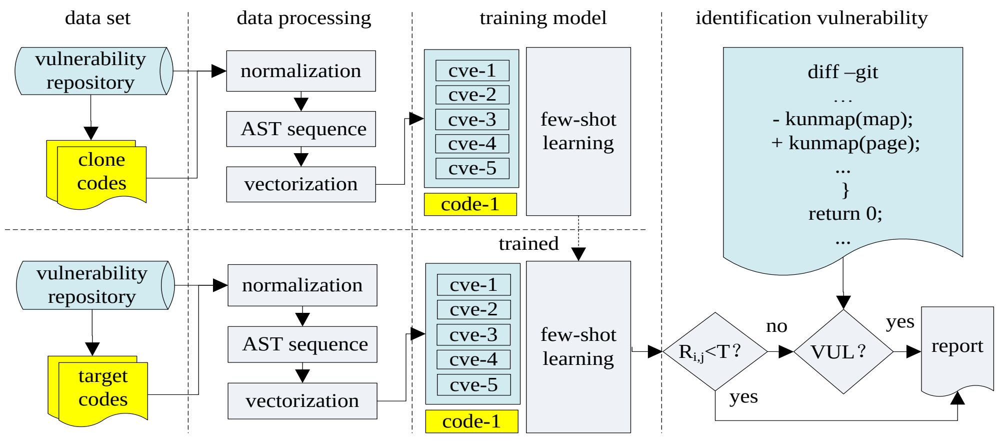

Figure 2. Overall framework of Vul-Mirror

clone has different vulnerabilities from the original codes (see Table 1 Type-3 clone 2). Otherwise, it has the same vulnerability as the original codes (see Table 1 Type-3 clone 1).

Type-4 clone is a semantic clone, it changes the statements but has the same functionality. In most cases, original codes have low similar to clone code. Therefore, it is difficult to determine the vulnerability directly.

According to the vulnerability pattern of code clone, we build the data set (code clone) by the following steps:

Firstly, according to the information of exposed vulnerabilities in the Common Vulnerabilities and Exposures (CVE), we download the vulnerability files and patch files from the open-source community and get the diff files of vulnerabilities and patches.

Secondly, we extract functions from vulnerable files. Because vulnerabilities usually in intra-function, we select the function as a unit to compare. When the vulnerable code spans multiple functions, we compare vulnerable code with the patch file and choose the function with the most rows changed.

Thirdly, based on the vulnerable functions, we generate the code clones using the heuristic method. For type-1 clone, code clones have the same vulnerabilities as the original codes, and we copy the original codes to get code clones. For type-2 clone, we copy the original codes and normalize variables, parameters, function-names, etc. as fixed symbols. For type-3 clone, we delete or insert some statements in functions based on the type-2 clone codes. For type-4 clone, create a code clone is difficult, so we discover it by similarity value and verification module.

We compare the modified code line with the patch. If the modified line same as the patch, replace it until there is no same code line with the patch. Finally, one class of vulnerability includes one original code and three code clones, all of the codes are vulnerable.

3.2. Data processing

The program code is different from the image, so we process the sample to adapt few-shot learning. The processing flow is as follows:

(i) Normalization. We remove the tokens that have nothing to do with the function of codes, such as comments, non-ASCII characters, and redundant whitespaces. We replace feature-independent prompts in codes to "str", such as a long string prompt statement in double-quotes. We normalize the numbers to "NUM1", "NUM2", and normalize variables as "VAR1", "VAR2".

(ii) Transforming code to abstract syntax tree (AST) sequence. AST can retain the most innovative information and remove the redundant information of source code, so we use AST to present functions. We first transform codes to AST, then transform AST into a token sequence by Deep-First Search.

(iii) Vectorization. To get a fixed-size vector, we split the AST sequence into many tokens and convert every token into a corresponding vector. We select 2,000 tokens (about 250 lines of C code) as the unit of the function. When the function tokens are less than 2,000, pad it with zero vectors. If the function length is longer than 2,000 tokens, intercept the corresponding number of lines of code. Then we use word2vec to complete the word embedding. After word embedding, each token is converted to a vector of 1*50 dimensions, and each function is converted into a vector matrix of 2000*50 dimensions.

3.3. Design few-shot learning model

There are many few-shot learning models to be used. According to our goal, we choose a 5-way 1-shot model. In every iteration step, an episode is formed by randomly selecting five classes from the training set with a labelled sample, as well as a fraction of the remainder of five classes' samples to serve as the query set. The features of the samples are extracted by the encoding module. The feature of five vulnerable codes and the clone code are combined, and the relationship value between them is calculated by the relation module. The relation score is a value from 0 to 1, 0 means two code is totally different, and 1 means precisely similar. Then we use MSE as the loss function of the network. The relation function and objective function are shown in equation (1), (2). We choose the CNN network for feature extraction and relationship comparison. The corresponding model is shown in Figure 3. The "code-1" is a query code of sample, having the highest similarity with the cve-3 in the vulnerable codes, so we consider that the "code-1" may contain the same vulnerability as cve-3.

$$R_{i,j} = g_{\emptyset}\left(c\left(f_{\varphi}(x_i), f_{\varphi}(x_j)\right)\right), i = 1,2,\dots C. \quad (1)$$

$$L_{\emptyset,\varphi} = arg\ \min_{\emptyset,\varphi} \sum_{i=1}^{m} \sum_{j=1}^{n} (R_{i,j} - 1(y_i - y_j))^2. \quad (2)$$

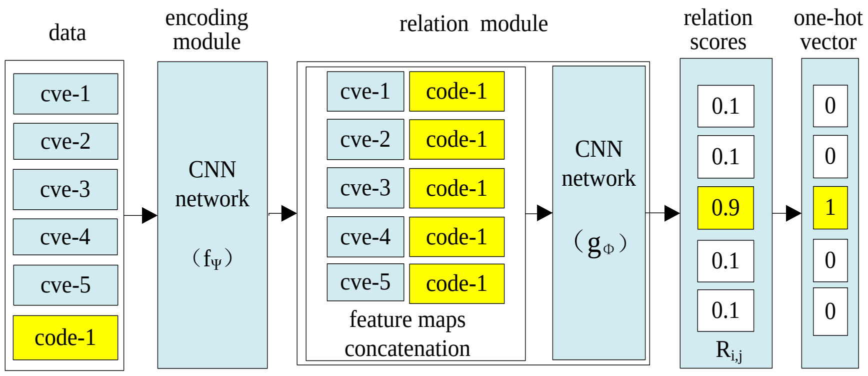

Figure 3. 5-way 1-shot model

3.4. Identification vulnerability

This process is added to confirm some vulnerability. If the vulnerability can be judged directly from the similarity such as type-1 cone, and some of the type-2 clones, this process can be avoided.

The few-shot learning model can output which vulnerability is most similar to the target code and its relationship value, which can be used to judge the code similarity. If the similarity of the two codes is too low, such as less than 50%, we think that the vulnerability is not a vulnerable code clone. Conversely, if the similarity of the two codes is high, such as greater than 95%, they have the same vulnerabilities. When we found a target code similar to vulnerable code with a score between 50% and 95%, we use patch (or diff file) of vulnerable code to check the target code. If the target code is more similar to the patch, we think the code has no vulnerabilities, vice versa, the target code is considered a vulnerable code. Our method is flexible, and we can select different thresholds manually according to the different situations.

4. Experiment

We perform our experiment with a large number of exposed vulnerabilities and conduct experiments on a machine running Ubuntu 16.04, with NVIDIA GeForce RTX 2070 GRU and Intel Xeon E5-2650 v4 CPU, 64 GB RAM, and 12 TB HDD.

In order to improve the similarity of code domain, we search exposed vulnerabilities of five operating systems and download the vulnerabilities and patches from the open-source community. Patches are used to verify candidate codes. According to the method discussed in the third section, we extract the vulnerability function from five operating system vulnerability files, construct the clone code of the vulnerability, and deal with the function code. The processed data set is used for training and testing the model. Table 2 summarizes the number of datasets. #CVE is the number of vulnerabilities, #Fun is the number of functions extracted from vulnerable files, #Patch is the number of functions extracted from patch files and use to train other models, #Clone is the number of generated clone codes. The dataset consists of 5,258 classes, one class includes one original vulnerable code and three clone codes. The dataset is randomly split into two parts, 80% for training and the remaining 20% for testing.

TABLE 2. Datasets used in experiment

Repository	#CVE	#Fun	#Patch	#Clone
FreeBSD	156	156	156	468
openSUSE	1,365	1,365	1,365	4,095
Linux kernel	1,299	1,299	1,299	3,897
OpenBSD	45	45	45	135
ubuntu	2,393	2,393	2,393	7,179
Total	5,258	5,258	5,258	15,774

We used the same metric as the description in [18], TP is the number of true positive samples that were correctly discovered as vulnerabilities, FP is the number of samples with false vulnerabilities discovered, FN is the number of samples with true vulnerabilities undetected, and TN is the number of samples with true non-vulnerable code detected. We use the widely used metrics Precision (P), Recall (R), False Positive Rate (FPR), False Negative Rate (FNR), and F1 Score (F1) to evaluate vulnerability detection systems. The ideal system neither misses vulnerabilities (FNR=0 and TPR=1) nor triggers false alarms (FPR=0 and P=1), which means F1=1.

To evaluate the efficacy and effectiveness, we compare against the various state of the art methods, VUDDY [8], VulPecker [9], and VulDeePecker [18]. All methods use the same vulnerability samples, and the results are shown in Figure 3.

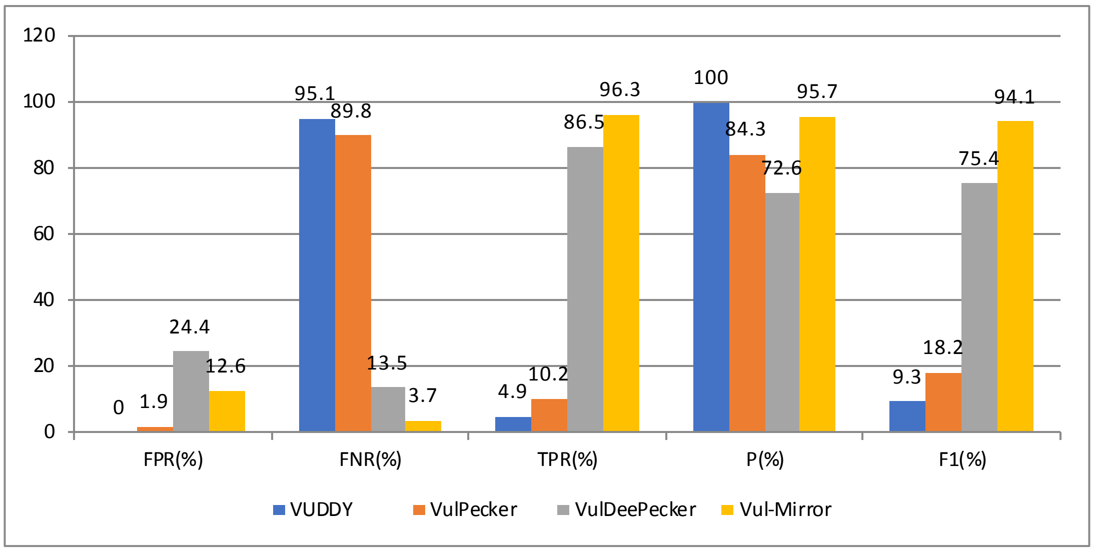

Figure 3. the results comparing to other methods

The results show Vul-Mirror achieves higher performance (F1=0.941). It misses fewer vulnerabilities (FNR=0.037 and TPR=0.963) and triggers less false alarms (FNR=0.126 and P=0.957). Because Vul-Mirror not only extracts features of vulnerabilities but also uses the network to measure the similarity of two codes. VulDeePecker also uses deep learning to extract features of codes, but the number of samples is small (only 5,258 vulnerabilities and 5,258 patches), which leads to the performance of VulDeePecker degradation. VulPecker can choose one algorithm from six algorithms to compare target codes with vulnerable codes according to different types of vulnerabilities, but it cannot extract features automatically. Imprecise features and limited algorithms reduce the effectiveness of VulPecker. VUDDY uses hash value to discover code clone vulnerabilities. VUDDY can discover type-1 and type-2 clones vulnerabilities and no trigger false alarms, but it hardly works in the case of that most of the samples are type-3 clone vulnerabilities. We use the end-to-end network, and it unifies feature extraction and relation calculation as a whole to identify vulnerable code clones, which is much more effective than the existing methods.

Because every sample in our data sets is vulnerability, we only need to find the most similar vulnerability to the cloned code, no need to validate candidate code. The model can achieve good performance, but due to the complexity and diversity of the code, it is still unable to recognize a few samples correctly.

In order to test the performance of the model in practice, we randomly select a new sample set that includes 13 code clones vulnerabilities and 417 non-vulnerable codes. The non-vulnerable code is different from all patch codes. We set the threshold to the high value (0.95), the middle value (0.65), and the low value (0.5) respectively. The test results of different thresholds are shown in Table 5.

Table 3. The result of the new sample

Threshold	FPR(%)	FNR(%)	TPR(%)	P(%)	F1(%)
0.95	0.00	84.62	15.38	100.00	26.67
0.65	1.93	15.38	84.62	57.89	68.75
0.5	6.24	0.00	100.00	33.33	50.00

From the result we know, when set high threshold (0.95), the model can get high precision (100%) but it misses some true vulnerabilities (FNR=84.6%). Because of the similarity between the target code and related vulnerability below the threshold, we think that samples are not vulnerable. Note that the model can identify which historical vulnerability is most similar to the target code.

So if we do not use threshold, the model can find these vulnerabilities.

When we set a middle threshold value (0.65), the model can obtain better comprehensive performance, but it still misses some vulnerabilities. When we set a lower threshold value (0.5), the model can find all vulnerabilities (TPR=100%). But some non-vulnerable samples are identified as vulnerabilities. When the similarity between the target code and one of the vulnerabilities is higher than the threshold value, the target code will be judged as vulnerability. The model triggers false alarms. By using the vulnerability verification process to confirm candidate target code, we can reduce false positives and false positives.

Experimental results show that our method can effectively detect code clone vulnerabilities, and our method can achieve good performance in practice.

5. Discussion

Few-shot learning is a hot topic. The model can reduce intra-class differences and increase inter-class differences of samples. Few-shot learning uses only a small number of samples, which alleviates the problem of lack of labelled samples. We use few-shot learning to detect code clone vulnerabilities and achieve good results. The results show that few-shot learning is suitable to solve the code clone issue. But at present, there are still some shortcomings in our method:

First, our method is based on few-shot learning, it only needs small samples, but it still requires every class of sample has some similar samples. We create the code clones (similar samples) set by the heuristic method. However, it does not necessarily mean that the heuristic method is always accurate in practice. How to effectively build a vulnerability data set is an interesting topic.

Second, different from code clones of detection, there is a gap between the similarity and vulnerability. When we test the non-vulnerable code, the performance of the model will decline. How to use a few-shot learning model to distinguish vulnerable codes and non-vulnerable codes correctly is worth studying.

Third, although our method can alleviate the problem of insufficient vulnerable samples in deep learning, it can only be used to discover vulnerable code clones at present, and it is difficult to detect vulnerabilities caused by other reasons. How to use few-shot learning to discover other kinds of vulnerabilities are the next topic.

Fourth, the similarity of different codes is different, so threshold adjustment is a complex process. It is our next work to give corresponding thresholds for different application scenarios to improve the practicability of the method.

6. Conclusion

In this paper, we propose Vul-Mirror to solve the problem of low accuracy in discovering vulnerable code clones. Vul-Mirror uses a few-shot learning model to extract code features and compare the relation of codes. It takes advantage of end to end network to implement fine-grained detection of similar codes. We use five common metrics to evaluate Vul-Mirror and conduct a comparative experiment on five open-source OS vulnerabilities datasets with three state-of-the-art methods. Experimental results show that Vul-Mirror is significantly better than other methods. We extend the application of few-shot learning, improve the efficiency of code clone vulnerability detection, and alleviate the lack of a large number of labelled data sets.

Acknowledgements.

We thank the anonymous reviewers for their comments that helped us improve the paper. This paper is supported by the National Natural Science Foundation of China (No.U1836210, No.61572460).

References

[1] Roy C K., Cordy J R. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools[C]// 2009 International Conference on Software Testing, Verification, and Validation Workshops. IEEE, 2009: 157–166

[2] Common vulnerabilities and exposures. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[3] Schwartz, E. J., Avgerinos, T., Brumley, D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA. IEEE,2010;317-331

[4] Cadar, C., Sen, K. Symbolic execution for software testing: three decades later. Communications of the ACM, 2013; 56(2), 82-90.

[5] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B. G., Cox, L. P., et al. Taintdroid: an information-flow tracking system for real time privacy monitoring on smartphones. Communications of the ACM, 2014; 57(3): 99-106.

[6] Rathi, D., Jindal, R. Droidmark: a tool for android malware detection using taint analysis and bayesian network. International Journal on Recent and Innovation Trends in Computing and Communication. 2018; 6(5): 71 - 76.

[7] Sergio Yovine and Gonzalo Winniczuk. Static taint analysis applied to detecting bad programming practices in android. Electronic Journal of SADIO, 2018; 17(1):35–53.

[8] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In 2017 IEEE Symposium on Security and Privacy, IEEE, 2017; 595–614.

[9] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016; 201–213.

[10] Heyuan Shi, Runzhe Wang, Ying Fu, Yu Jiang, Jian Dong, Kun Tang, and Jiaguang Sun. Vulnerable code clone detection for operating system through correlation-induced learning. IEEE Transactions on Industrial Informatics, 2019; 15(12):6551–6559.

[11] Jang. J., Agrawal A. and Brumley D. Redebug: finding unpatched code clones in entire os distributions. In 2012 IEEE Symposium on Security and Privacy, IEEE, 2012; 48–62.

[12] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on software Engineering, 2006; 32(3):176–192.

[13] Xifeng Yan, Jiawei Han, and Ramin Afshar. Clospan: Mining: Closed sequential patterns in large datasets. In Proceedings of the 2003 SIAM international conference on data mining, 2003; 166–177.

[14] James Newsome. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. Chinese journal of engineering mathematics, 2005, 29(5):720-724

[15] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, 2013; 12: 499–510.

[16] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In Proceedings of the 28th Annual Computer Security Applications Conference. 2012; 12: 359–368.

[17] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In 2015 IEEE Symposium on Security and Privacy, IEEE, 2015; 797– 812.

[18] Li, Z., Zou, D., Xu, S., Ou, X., Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. Network and Distributed System Security Symposium.2018;

[19] Zou, D., Wang, S., Xu, S., Li, Z., Jin, H. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. IEEE Transactions on Dependable and Secure Computing, 2019; 1-1.

[20] Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K. and Lopes, C. V. Sourcerercc: scaling code clone detection to big code. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, 2016; 1157-1168

[21] Kamiya T., Kusumoto S., Inoue K., CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Transactions on Software Engineering, 2002, 28(7):654-670.

[22] Jiang L. DECKARD: Scalable and accurate tree-based detection of code clones. Proc of Icse Minneapolis Mn Usa, IEEE, 2007; 96-105.

[23] White, M., Tufano, M., Vendome, C. and Poshyvanyk, D. Deep learning code fragments for code clone detection. the 31st IEEE/ACM International Conference. International Conference on Automated Software Engineering, ACM, 2016; 87–98.

[24] Gregory K., Richard Z., and Ruslan S. Siamese neural networks for one-shot image recognition. In ICML deep learning workshop, Lille, France, 2015. JMLR: W&CP volume 37

[25] Snell J., Swersky K., Zemel R S.., Prototypical Networks for Few-shot Learning. In Advances in Neural Information Processing Systems, 2017; 4077–4087.

[26] Sung F., Yang Y., Zhang L., et al. Learning to Compare: Relation Network for Few-Shot Learning. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018; 1199–1208.

[27] Vinyals O., Blundell C., Lillicrap T., et al. Matching Networks for One Shot Learning. In Advances in neural information processing systems, 2016; 3630–3638.