VFDETECT: A Vulnerable Code Clone Detection System Based on Vulnerability Fingerprint

Zhen Liu[1,2], Qiang Wei[1,2], Yan Cao[1,2]
1.State Key Laboratory of Mathematical Engineering and Advanced Computing
2.China National Digital Switching System Engineering and Technological Research Center
Zhengzhou, China
lancelot1qaz@gmail.com

***Abstract*—Vulnerable code reuse in open source software is a serious threat to software security. However, the existing high-efficiency methods for vulnerable code clone detection have a large number of false-negatives when the code is modified, which results in limited application scenarios. In this paper, we present an innovative fingerprint model to describe the vulnerability code and propose VFDETECT, an efficient system to detect vulnerable code clones based on the fingerprints. Firstly, the fingerprint is constructed by applying hash function to appropriate code blocks in the diff which are preprocessed. Then, VFDETECT detects the vulnerable code clone by matching the preprocessed code blocks in target project with the fingerprint, which is mapped to a bitmap so that can be identified efficiently. VFDETECT could maintain better performance and acquire higher robustness under multiple code modification methods such as variable renaming, code sequence changing and redundancy inserting, which is difficult to achieve in existing research work. Our results in real-world datasets detection demonstrate that it is of practical values.**

Keywords—vulnerability fingerprint; code clone; vulnerability detection

I. INTRODUCTION

The rapidly growing demands for software lead to the increasing popularity of code reuse, including existing code templates and components. Open source software (OSS) has become one of the best solutions to improve both the efficiency and the quality of developing at the meanwhile reducing cost. However, a considerable number of vulnerabilities in OSS programs would naturally lead to lots of software vulnerabilities caused by code cloning, which poses a serious threat to the system security. The Black Duck Corp pointed out that about two-thirds of commercial applications have code with known vulnerabilities in it. It is predictable that the number of attacks based on the vulnerabilities in open source code will increase by 20 percent this year [1]. Therefore, research in the detection of vulnerable code cloning is of critical great importance in both software engineering and information security.

Methods of vulnerable code reuse detecting usually translate the target code into the intermediate representation, such as the parse tree or control flow graph, based on which, vulnerabilities caused by code clones could be found by matching the same structure or same feature with the known cases. A complex intermediate representation may help in increasing the accuracy but it will also lead to higher computational cost, while a high level of abstractions could benefit to the efficiency but lost the necessary factors of vulnerability. Consequently, how to balance the efficiency and accuracy under an acceptable cost, as well as dealing with common modification means in code cloning effectively, is of great research significance.

In this paper, we present an innovative fingerprint model to describe the vulnerability code and propose VFDETECT, an effective system to detect vulnerable code clones based on the fingerprints. By using the code block as granularity and applying preprocess, VFDETECT gets resilient to common modifications in cloned code. The hash function is used to simplify the fingerprint structure while the bitmap is introduced to increase detection efficiency. VFDETECT could maintain better performance and acquire higher robustness under multiple code modification methods, which is difficult to achieve in existing research work.

II. BACKGROUND AND RELATED WORK

It is pointed that code cloning usually accompanies with several code modification methods [2], such as comments modification, variable renaming, data type change, operators change, statements order change, code blocks order change, redundant code insert, and the equivalent transformation for the control structure. Based on a commonly accepted classification for code clone is proposed by Roy [3], we use the following definitions of the level of modifications in code clone.

Level1: Change the layout of code via modifying spaces and tabs, as well as edit the comments, where the code part is stayed unmodified.

Level2: Change the data type of the variable and the function return value, and rename the identifier and variable.

Level3: Add or delete a few code statement as well as modify expressions or function calls without changing the original function of the code.

Level4: Adjust the code structure without changing the semantics, such as changing the order of code blocks, equivalent transformation for the control structure.

The increase of the code modification level will directly result in the increase of the difficulty in code clone detection. Level1 and Level2 modification could be detected efficiently in the existing work while further modification will usually result in larger computational overhead. However, in reality, Level3 and Level4 modification are quite prevalent. Research for

978-1-5090-5363-6/17/$31.00 ©2017 IEEE

detecting them both accurately and efficiently has practical implications.

Many approaches have been proposed to detect the vulnerabilities brought by code clone. A system named ReDeBug is proposed by Jang et al. [4] , which uses the code lines as the granularity of detecting and find clone codes by using the sliding window algorithm as well as the bloom filter. ReDeBug has a great advantage in speed but also a higher false negative rate and the false positive rate under the common code modifications. A novel mechanism called CLORIFI is proposed by Li [5] for the buffer overflow vulnerability. It locates the known vulnerability code clones with n-token algorithm. With the help of the concolic test to verify the vulnerabilities so that it can reduce false positives. CLORIFI improved the accuracy of the detection with concolic test which will lead to a large consumption of resources. And it has a higher false negative rate due to the limit of the code clone detecting algorithm. A clone detection method based on a program characteristic metrics is proposed by Gan et al. [6]. The characteristic vector as well as matrix are constructed by traversing the parse tree and the vulnerability codes are located by performing cluster calculation. Though the time overhead is in a linear relationship with the amount of code be detected, the efficiency of this method is still in the need for an increase. Li et al. [7] proposed VulPecker, which combines multiple source code representation and similarity metric algorithms to detect the vulnerability of source code. By using the set of Token, AST, PDG and other intermediate representation, it express the characteristics of vulnerability code from multi-aspects. Machine learning is used to choose the best combination of the code representation and the detection algorithm. It supports a variety of vulnerabilities and the code modifications methods. However it is unsuitable for large-scale detection because of the limitation of efficiency. Kim et al. [8] proposed VUDDY, a highly efficient method for detecting vulnerable code cloning, which is achieved by leveraging function-level granularity and a length-filtering technique that reduces the number of signature comparisons. But it does not support common code modification methods such as word order modification and redundant code insertion, which causes its limitation in practice.

III. Design of Vulnerability Fingerprint

In order to detect the vulnerable code clones, we introduced a new fingerprint model to abstract the vulnerability. Existing researches has pointed out that the vulnerability can be characterized by the diff files in the patch [4, 7, 8]. The diff file consists of one or more diff hunks, and each diff hunk is a sequences of lines of code with special mark. Lines start with the symbol "+" represent the code added and the "-" represent the deleted, while no symbol indicating no modification. VFDETECT constructs the vulnerability fingerprint model based on the diff files in the patches of known vulnerabilities.

A. Granularity Selection

The selection of detection granularity has important consequences for detection effect. Using the single line of code as the basic unit can reflect neither the critical factors which cause the vulnerability, nor the context in which is necessary for the vulnerability to trigger off. For example, CVE-2015-7550 is a race condition vulnerability in the Linux kernel before the version 4.3.4, which is fixed only by merely changing the order of statements within the function *keyctl_read_key* in *security/keys/keyctl.c*. The sequence of the lines of code may lost if using the single line of code as the granularity, which will lead to a narrow applicability for our method.

There are often more than one essential conditions for the vulnerability, and the key codes that trigger it may be scattered across multiple functions or multiple source files. Choosing the single function as the basic unit for detection will not be able to cover all the necessary elements. Take CVE-2016-3134, an integer overflow vulnerability of the *netfilter* subsystem in the Linux kernel through 4.5.2 for example. It patched the function *arpt_do_table* in *arp_tables.c* and function *get_entry* in *ip_tables.c* under the directory *net/ipv4/netfilter/* to fix this vulnerability. If the design of the fingerprint extract code feature only from a single function or a single file, it will not be able to cover all the necessary requirements for the cause of the vulnerability. So an appropriate granularity is expected to be able to extract all the features from scattered codes.

To meet all the needs of granularity above, VFDETECT selects the code block as the basic unit of the feature extraction in the vulnerability fingerprint. The size of block is dynamically determined based on the window size of the sliding window algorithm which will be used in later detect. This granularity allows us to cope with the impact of code modification methods, as well as keep the key elements and the necessary context for vulnerability which is important for improving accuracy.

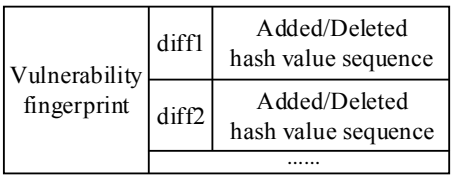

Fig.1. Structure of the vulnerability fingerprint

The structure of the vulnerability fingerprint used by VFDETECT is shown in Fig.1. For each diff file in the patch for a certain vulnerability, we select all the changes in the code as well as necessary part of its context, split them into code blocks with same size. These blocks are classified into two categories: the addition and the deletion. The feature values for each code block are calculated by hash function and be attracted together to form the feature sequences for each category. The feature sequences of all diff files in the same patch constitute a complete fingerprint aiming at detecting the corresponding vulnerability cause by code clones.

B. Code Preprocessing

Code preprocessing is an effective means of eliminating the influence brought by modification in code clones, such as format modification, editing comments. What's more, by formatting the name of variable and function calls, we could eliminates possible confusion brought by renaming. As the first step of preprocessing, we convert all the code to lowercase, remove extra spaces, tabs, line breaks, and all the comments. Change the indent style into Lisp style. Then preprocess as the following:

- *Function and parameter name substituting:* Gather all the formal parameters by parsing the function declarations, and replace every occurrence in the

function body with the symbol **_PARAM**. Also replace the function name in the declaration with the symbol **_FUNCDEC**.

- *Variable identifier substituting:* Replace all the variables defined inside the function with the symbol **_DATA**.
- *Data type substituting:* Replace all the data type declarations which are declared in the ISO C standard and the custom data structures with the symbol **_TYPE**. Since the data structures reflect the code characteristics to some extent, the keyword "*struct*" in the variable declaration will be reserved in order to distinguish it from the normal data type. Members declared in the custom data structures will not be replace for the consideration of incomplete code segment paring. Also the modifier such as "*signed*" or "*unsigned*" is unreplaced because of the signedness of a variable has an important effect on some vulnerabilities such as integer overflow.
- *String constants substituting:* Replace string constants with the symbol **_STR**. The string parameters that contain the format characters such as "%s" in all print-type functions will not be replaced, in consideration of possible format string vulnerabilities.
- *Function call substituting:* Replace each function call with the symbol **_FUNCTION**. Function or API calls are importantly linked to the causing and triggering of vulnerabilities. By substituting the name of calls, we can keep the detail of its usage and the parameters values, which are helpful characteristics in clone detection.

Listing 1: Part of the code in patch for CVE-2016-6198

```
int vfs_open(const struct path *path, struct file *file,
                    const struct cred *cred) {
– struct dentry *dentry = path->dentry;
– struct inode *inode = dentry->d_inode;
+ struct inode *inode = vfs_select_inode(path->dentry, file->f_flags);
– file->f_path = *path;
– if (dentry->d_flags & DCACHE_OP_SELECT_INODE) {
–       inode = dentry->d_op->d_select_inode(dentry, file->f_flags);
–       if (IS_ERR(inode))
–               return PTR_ERR(inode);
– }
+ if (IS_ERR(inode))
+               return PTR_ERR(inode);
+ file->f_path = *path;
  return do_dentry_open(file, inode, NULL, cred);}
```

Listing 2: Preprocessed output for code in Listing 1

```
_TYPE _FUNCDEC(const struct _TYPE *_PARAM, struct _TYPE
                    *_PARAM, const struct _TYPE *_PARAM) {
– struct _TYPE *_DATA = _PARAM->dentry;
– struct _TYPE *_DATA = _DATA->d_inode;
+ struct _TYPE *_DATA = _FUNCTION(_PARAM->dentry,
                                _PARAM->f_flags);
–  _PARAM->f_path = *_PARAM;
– if (_DATA->d_flags & dcache_op_select_inode) {
–       _DATA = _DATA->d_op->d_select_inode(_DATA,
                                _PARAM->f_flags);
–       if (_FUNCTION(_DATA))
–               return _FUNCTION(_DATA);
–       }
+       if (_FUNCTION(_DATA))
+               return _FUNCTION(_DATA);
+       _PARAM->f_path = *_PARAM;
  return _FUNCTION(_PARAM, _DATA, NULL, _PARAM); }
```

Take CVE-2016-6198, a denial of service vulnerability that exists in the Linux kernel before 4.5.5, as example, the patch file is related to *fs/namei.c* and *fs/open.c*. Part of the diff in the patch and its preprocessed result are shown in the Listing 1 and the Listing 2.

C. Fingerprint Generating

The vulnerability fingerprint consists of feature sequences extracted from the diff files in the patch. VFDETECT obtains the feature sequences and generates the vulnerability fingerprint by the following steps:

- Determine the difference between the vulnerability code and the patch according to the diff files, select an appropriate number of lines s as the size of the code blocks, which will also be used as the sliding window size in later detecting. Usually s takes 4.
- From the code after preprocessing, select a s-lines block of code deleted in the diff file as $block_D$, select a s-lines block of code added in the diff file as $block_A$. Fill these s-lines blocks with context while the code deleted or added is less than s lines, or separate into several consecutive blocks of which the last block is filled with context if more than s lines.
- Use the hash function to compute the eigenvalues of each $block_D$ and $block_A$. Marking type of the eigenvalues with "+" on behalf of the added and "-" the deleted. Assemble eigenvalues into the added feature sequences and the deleted feature sequences respectively.
- Perform the above steps for each diff files in the patch for this vulnerability. Then the vulnerability fingerprint is formed by all the feature sequences in each diff files.

In CVE-2016-6198, there are two diffs in the patch. Both the added and the deleted could be divided into two blocks in the first diff and only one block for the added in the other diff after preprocessing. After the hash function we can get the vulnerability fingerprint as shown in Fig.2:

CVE-2016-6198	diff1	+39ee72f2ad4e5a80
		+9574bdb85a3c60cd
		–51939b316a9dce4e
		–d81f25c67b26163d
	diff2	+1f20126211b1c8a6
		–a40ec87a443a8dba

Fig.2. Vulnerability fingerprint for CVE-2016-6198

IV. The Framework of VFDETECT

Fig.3 gives an overview of the detecting processes for VFDETECT. Vulnerability fingerprint database (VFD) is built in advance to serve as the basis of detection. The processes of parsing and preprocessing as well as applying hash function for the target codes are critical steps before fingerprint identification.

A. Parser Generation

The parser is an important component of code lexical analysis and parsing, which is the basis of preprocessing and feature extraction. Although the parsers integrated in compilers

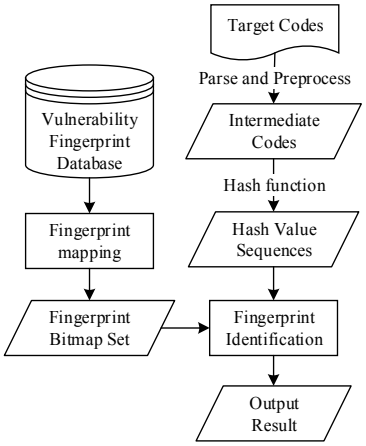

Fig.3. Overview for the framework of VFDETECT

such as llvm and gcc may be more mature and accurate, the efficiency is not satisfactory enough. What matters more is, under most circumstances, code fragments existing in the vulnerability databases are not complete and compilable, and so are the code to be detected. This calls for a parser with high robustness. For the sake of both efficiency and accuracy, VFDETECT uses the open source parser ANTLRv4 [9] to build the parser. Benefiting from the utilization of island grammars, we generate a robust C/C++ code parser that could work without complete header files even an available build environment. It will continue parsing disregarding syntax errors, which makes it feasible for vulnerable code analyzing and vulnerability fingerprint generation. Based on the parser, the preprocess goes and output the intermediate code for hash function.

B. Hash Function

Hash function is used in our research to abstract the code blocks both in vulnerability fingerprint generating and detecting. An appropriate hash algorithm is required to improve the efficiency and accuracy of the method. The hash algorithm is supposed to meet the principle that it could avoid the hash-collision and be as high-efficiency as possible. Since the commonly-used algorithms in text similarity measuring such as *Simhash* may lead to hash-collision in our approach, we use MD5 with the 8-byte output as our hash function, which is efficient enough and space-saving.

Algorithm 1 ***Hash value generating algorithm***

Input: Function f which is preprocessed;
Window size $size_{win}$
Output: Hash value sequence seq_f for function f

$seq_f \leftarrow \emptyset$
//Get the total lines of function f
$line \leftarrow 0$
for each row **in** f
  $line \leftarrow line + 1$
for each row' **in** f
  //Get the code block in current window
  $trunk \leftarrow f[row, row + size_{win}]$
  $hash \leftarrow \text{HASH}(trunk)$
  $seq_f \leftarrow seq_f \cup \{hash\}$
  if $row' + size_{win} > line$ **then**
    break
  end if
end for
return seq_f

The sliding window algorithm is used in calculating hash values for the input codes after parsing and preprocessing. The size of the sliding window $size_{win}$ is determined by the block size using in the vulnerability fingerprint. The working process of hash function is described in Algorithm 1, of which the output is seq_f, a sequence of hash values that can be used directly in fingerprint identifying.

C. Vulnerability Identification

Vulnerability fingerprint identification is a key step also the last step in vulnerable code clones detecting. After selecting the set of fingerprints for detecting in VFD, we mapping it into a set of bitmap, in which each bit represent the existence of corresponding hash value. The use of bitmap will simplify the process of fingerprint identifying.

Given the sequence of hash values seq_f as the output of the hash function, VFDETECT traverse seq_f and check if each values is in the vulnerability fingerprints. If it does exist, the corresponding bit in the bitmap is set to 1, else 0. We will also attach the tags, consist of file name and function, to this bit so that we could be able to tracking the position where the clones exist. Two examples are shown in Fig.4.

	Hash value	Bitmap	Tag
diff1	+39ee72f2ad4e5a80	0	
	+9574bdb85a3c60cd	0	
	–51939b316a9dce4e	1	fs/open.c - vfs_open
	–d81f25c67b26163d	1	fs/open.c - vfs_open
diff2	+1f20126211b1c8a6	0	
	–a40ec87a443a8dba	1	fs/namei.c - vfs_rename

(a) Fingerprint bitmap that meet the requirement for vulnerability existing

	Hash value	Bitmap	Tag
diff1	+39ee72f2ad4e5a80	1	fs/open.c - vfs_open
	+9574bdb85a3c60cd	1	fs/open.c - vfs_open
	–51939b316a9dce4e	0	
	–d81f25c67b26163d	0	
diff2	+1f20126211b1c8a6	1	fs/namei.c - vfs_rename
	–a40ec87a443a8dba	0	

(b) Fingerprint bitmap that does not meet the requirement for vulnerability existing

Fig.4. Fingerprint bitmap samples of the vulnerable code clone and the patched code clone

For each vulnerable code clone, we assume that none of the codes added in diffs is existing while all the codes deleted in diffs is exactly existing, indicating it unpatched. It shows up in the bitmap that all the values signed with "+" should be set to 0 and all the values signed with "-" should be set to 1. We also limit the filename in the Tag for each bit which is in the same diff should be equal, so that can reduce the false-positives brought by large scale codes. By checking the bitmap we could identify the vulnerability code and locate its position efficiently. Two samples are shown in Fig.4. The one in (a) meets all the requirement will be regarded as one piece of vulnerable code clone, the other in (b) will not. This means that VFDETECT can detect vulnerable code clones as well as variants with modifications, not requiring any supplementary procedure.

V. Evaluation and Analysis

We evaluate the effectiveness of VFDETECT from both the accuracy and efficiency, by comparing VFDETECT with other up-to-date method of vulnerable code clones detecting. In our research, we build up the dataset by selecting famous open source software projects from Git code repositories as the source of vulnerable codes, including Apache Http Server, Linux kernel, FFmpeg, Firefox, Openssl, Qemu ,etc. From CVE we get the detail information and patch files of the vulnerabilities in these open source software projects during the past three years, based on which we built the VFD. Information on fragile code sources is shown in Table 1.

TABLE I. Sources of the Dataset

Project Name	Vulnerability Number	Diff Number
Apache Http Server	15	18
Linux kernel	65	79
Ffmpeg	25	34
Firefox	25	33
openssl	20	24
qemu	15	17

A. Accuracy Evaluation

By applying the modifications from Level1 to Level4 to the vulnerable code, we build up a vulnerability code clones library (VCCL) as the test dataset for VFDETECT. We evaluate the effectiveness by standard metrics such as precision, recall, and F-measure metrics.

Let ***TP*** (true-positives) represent the number of true cases detected in the test set, ***FP*** (false-positives) represent the number of false cases detected in the test set, and ***FN*** (false-negatives) represents the number of cases that undetected. Then the metric *precision* = ***TP*** / (***TP*** + ***FP***) reflects the correct rate for results of positives. The metric *recall* = ***TP*** / (***TP*** + ***FN***) reflects the completeness of the detected positive. The overall detection effect can be measured by: *F-measure= 2 × precision × recall / (precision + recall)*. The higher scores indicating the better detection effect. Since the tools designed for general cloning detection such as CP-Miner [10] and SourcererCC [11] have a relatively high false positive rate while detecting vulnerable code clones, which the patched benign functions are often similar to the old, vulnerable version of itself. So we choose VUDDY and ReDeBug, which are exactly designed for detecting vulnerable code clones, as a control to measure the effectiveness of VFDETECT.

TABLE II. Detecting Effect Under Different Modifications

Modification Approach	ReDeBug	VUDDY	VFDETECT
Data type change	×	√	√
Variable rename	×	√	√
Insert redundancy	√	×	√
Code blocks reorder	√	×	√

We selected 70 code clone instances with different levels of code modification from the VCCL, which covers 15 real vulnerabilities. Calculate the F-measure of each tools after detecting, the test results are shown in Fig.5.

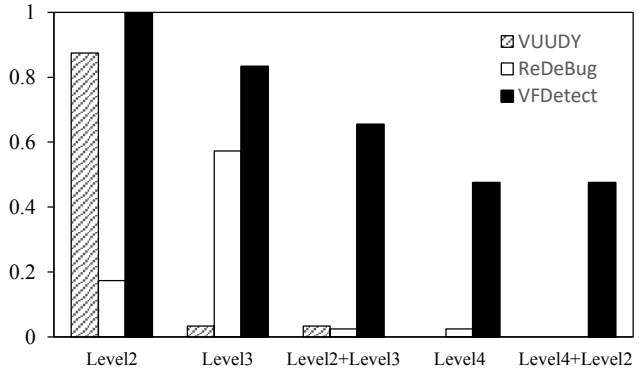

Fig.5. *F-measure* for the result of each tool in detecting under different level of code modifcaiton.

Fig.5. shows the F-measure of the three tools according to the testing data. It's obvious that VFDETECT has significantly higher accuracy under different of modification levels. We discovered major drawback of the compared techniques by analyzing the false-negatives cases of each tool. VUDDY requires an exactly equal numbers of tokens and the same sequence for each functions after abstraction, which makes it difficult to identify clones with redundant code inserting or deleting code that is not related to key point of vulnerability. ReDeBug is unable to recognize the code of which data types are changed or variables are renamed, this limits its application under the Level2 modifications. VFDETECT shows good robustness against different kinds of code modification due to the use of vulnerability fingerprint. The results for the different code modification methods in vulnerable code cloning according to the analysis of the results are shown in Table 4. The "√" is on behalf of the ability to detect under this kind of changes, the "×" means that it can't handle this change.

B. Efficiency Evaluation

In order to evaluate the efficiency of VFDETECT, we select 10 objects of different sizes for testing. The test cases, of which the size is varying from 0.15K LoC to 13.4K LoC, consist of real-world programs selected from the famous open source software projects we collected. Our experiments are processed on the server of Ubuntu 16.04, with a 2.40 GHz Intel Xeon E5 processor, 64 GB RAM, and 6 TB HDD. Figure 6 shows the time VFDETECT spent. It can be seen that there exists a rough linear rise for the time spent with the lines of input, which makes it practical in the detection of large-scale code.

VFDETECT does detection with the pre-generated vulnerability fingerprints in VFD, which require traversing through the target code for only once. By using the bitmap for the fingerprints, we judge the potential vulnerability after detection instead of looping traverses in VFD during the detection. This make VFDETECT more efficient which leads to almost linear rise for the time spent with the line of code input.

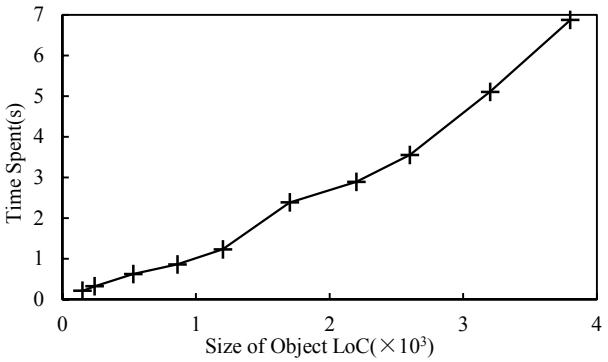

Fig.6. Time spent in detecting different size objects

C. Vulnerable Code Clone Case

Vulnerable code reuse is frequently occurring in practice, making it difficult to fix up all the vulnerabilities. While applying VFDETECT to the source code of QEMU, we have found a vulnerable clone of CVE-2009-2768 in Linux Kernel which will lead to a null pointer dereference and cause denial of service. The vulnerable function *load_flat_shared_library* in *qemu/linux-user/flatload.c* is described in Listing 3, which is written based on *linux/fs/binfmt_flat.c* to build the *bFLT binary loader*. Though it requires strict conditions to trigger, there still exist the risk. The vulnerable code in the Linux Kernel has long been fixed, however it is not applied to the code in QEMU, leaving it unpatched till now. What's more, other projects such as *s2e* who reuse the code of *bFLT binary loader* module also suffer from the same problem. This case indicates that our approach is effective and of practical value.

Listing 3: Part of the code in patch for CVE-2016-6198

```
678 static int load_flat_shared_library(int id,
        struct lib_info *libs)
691   if (IS_ERR(bprm.file))
692     return res;
693
694   res = prepare_binprm(&bprm);
695
696   if (res <= (unsigned long)-4096)
697   res = load_flat_file(&bprm, libs, id, NULL);
698   if (bprm.file) {
699     allow_write_access(bprm.file);
700     fput(bprm.file);
701     bprm.file = NULL;
702   }
703   return(res);
```

VI. Conclusion and Future Work

In this paper, we designed a fingerprint model for vulnerable code clones detecting and proposed VFDETECT, which is an approach to discover the vulnerable code clones based on the vulnerability fingerprint. By selecting the code block as the basic unit of the detecting, VFDETECT could be able to detect the codes after modifications from Level1 to Level4, which is difficult to achieve for existing research works. The use of the hash function and bitmap in fingerprint identification makes it efficient enough in large scale object detecting. The experimental results show that VFDETECT can detect the code variant under modification with a good accuracy. And the time cost is rough linear with the size of the code. Numerous vulnerable code clones has been found in practice, which shows that it has a fairly good practical values.

This work can be extended in following directions: Firstly, the hash function used in VFDETECT could simplify further, which will help in increasing detection rate. Then, the false-positives can be reduced by combining our approach with Fuzzing, which is also a possibly good route to validate whether the positives are real vulnerabilities. Moreover, the vulnerability fingerprint database can be enriched and optimized for a better performance.

Acknowledgment

We thank the anonymous reviewers for their comments to improve the quality of the paper. This work was supported by National Key Technologies Research and Development of China (Grant No.2016YFB0800203).

References

[1] M Korolov. (2017,JAN,17). Report: Attacks based on open source vulnerabilities will rise 20 percent this year[Online]. Avilable: http://www.csoonline.com/article/3157377/application-development/report-attacks-based-on-open-source-vulnerabilities-will-rise-20-percent-this-year.html

[2] E.L.Jones, "Metrics based plagiarism monitoring," in Proceedings of the 17th Consortium for Computing Sciences in Colleges, vol.16, ACM, 2001, pp.1–8.

[3] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Science of Computer Programming, vol. 74, no. 7, pp. 470–495, 2009.

[4] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: finding unpatched code clones in entire os distributions," in Proceedings of the Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012, pp. 48–62.

[5] H. Li, H. Kwon, J. Kwon, and H. Lee, "CLORIFI: software vulnerability discovery using code clone verification," Concurrency and Computation: Practice and Experience, pp. 1900–1917, 2015.

[6] S.Gan, X.Qin, Z.Chen, and L.Wang, "Software vulnerability code clone detection method based on characteristic metrics," Journal of Software, vol.26(2), pp. 348−363, 2015 (in Chinese)

[7] 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. ACM, 2016, pp. 201–213.

[8] S.Kim, S.Woo, H.Lee, H.Oh. "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," the 38th Symposium on Security & Privacy. IEEE, 2017.

[9] "ANTLR, ANother Tool for Language Recognition," http://www.antlr.org/, accessed: 2017-6-15.

[10] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," Software Engineering, IEEE Transactions on, vol. 32, no. 3, pp. 176–192, 2006.

[11] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: scaling code clone detection to big-code," in Proceedings of the 38th International Conference on Software Engineering. ACM, 2016, pp. 1157–1168.

[12] F.Yamaguchi, A.Maier, H.Gascon, and K.Rieck, "Automatic Inference of Search Patterns for Taint-Style Vulnerabilities," in Proceedings of the 36th Symposium on Security & Privacy. IEEE, 2015, pp.797-812.

[13] H.Perl, S.Dechand, M.Smith, D.Arp, F.Yamaguchi, and K.Rieck, "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits," in Proceedings of the Sigsac Conference on Computer and Communications Security. ACM, 2015, pp.426-437.

[14] A.Chandran, L.Jain, S.Rawat, and K.Srinathan, "Discovering Vulnerable Functions: A Code Similarity Based Approach," in Proceedings of the International Symposium on Security in Computing and Communication. Springer, 2016. pp.390-402.