VulDetector: Detecting Vulnerabilities Using Weighted Feature Graph Comparison

Lei Cui, *Member, IEEE*, Zhiyu Hao, Yang Jiao, Haiqiang Fei, and Xiaochun Yun

***Abstract*— Code similarity is one promising approach to detect vulnerabilities hidden in software programs. However, due to the complexity and diversity of source code, current methods suffer low accuracy, high false negative and poor performance, especially in analyzing a large program. In this paper, we propose to tackle these problems by presenting VulDetector, a static-analysis tool to detect C/C++ vulnerabilities based on graph comparison at the granularity of function. At the key of VulDetector is a weighted feature graph (WFG) model which characterizes function with a small yet semantically rich graph. It first pinpoints vulnerability-sensitive keywords to slice the control flow graph of a function, thereby reducing the graph size without compromising security-related semantics. Then, each sliced subgraph is characterized using WFG, which provides both syntactic and semantic features in varying degrees of security. As for graph comparison, we take full usage of vulnerability graph and patch graph to improve accuracy. In addition, we propose two optimization methods based on analysis of vulnerabilities. We have implemented VulDetector to automatically detect vulnerabilities in software programs with known vulnerabilities. The experimental results prove the effectiveness and efficiency of VulDetector.**

***Index Terms*— Vulnerability detection, code similarity, weighted feature graph.**

I. Introduction

OVER the last decades, the number of released open-source software programs has increased rapidly. Github, the popular open-source software platform, reports that it holds more than 100 million projects [1]. Meanwhile, the software evolves continuously, e.g., OpenSSL released over 200 versions, which further extends the number of programs. Due to careless operations (e.g., low coding quality) and intended operations (e.g., code clone) upon development as well as insufficient security testing, there exist vulnerabilities in software. Vulnerability denotes a weakness, defect or security bug in a software. It makes the software facing attack threats, including information leak, remote control, and denial of service [2]–[6]. Generally, the number of vulnerabilities keeps paces with the number of code, implying that a large number of vulnerabilities hide in software programs.

Code similarity is one promising approach to detect currently undiscovered vulnerabilities in a testing program (i.e., a program being tested) when some vulnerabilities are given [4], [5], [7]–[10]. It firstly transforms the testing code and vulnerabilities into an intermediate representation and then employs comparison algorithms to find matches. The proposed studies fall into four categories according to how the source code is transformed.[1] 1) Text-based method [4], [11]–[13], which directly compares the raw source code. This method is easy to find identical code, however, it fails to find highly similar code even with small variations in identifiers or types. 2) Token-based method [14]–[17], which divides each code line into a sequence of tokens according to the lexical rules and then compares the token sequence. It is more robust over small code variations than text-based method. 3) Tree-based method [9], [18]–[20], which represents the code with syntactic structures such as parse tree or abstract syntax tree (AST), and then finds similar code by tree matching. It can express syntactic similarity and thus allows large code changes. The problem is that it yields many false positives due to the abstraction of identifiers. 4) Graph-based method [5], [10], [21], [22], which characterizes source code with graphs (e.g., program dependency graph (PDG) or control flow graph (CFG)), in which the nodes denote statements or identifiers while the edges represent control or data dependencies. This method provides more precise semantical information than syntactic similarity. Of them, the graph-based method can better characterize semantic information of source code, e.g., control flow and data dependency. Thus, it can identify semantically similar functions even who exhibit larger lexical and syntactic variance and thus has the potential to discover more vulnerabilities [5], [10], [21].

A. Problems

Despite the advance, however, the graph-based approaches struggle to improve both efficiency and accuracy, which are critical for real-world vulnerability detection. On one hand, it is known that many graph matching algorithms, such as subgraph isomorphism and graph monomorphism, are NP-complete, which limits the graph-based method to small graphs [23]. Unfortunately, the graphs of vulnerability functions are always large. E.g., our analysis shows that 46.8% of

Manuscript received August 14, 2020; revised October 31, 2020; accepted December 7, 2020. Date of publication December 28, 2020; date of current version January 13, 2021. This work was supported in part by the National Natural Science Foundation of China under Grant 61972392 and Grant 62072453 and in part by the Youth Innovation Promotion Association of the Chinese Academy of Sciences under Grant 2020164. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Kai Zeng. *(Corresponding author: Zhiyu Hao.)*

Lei Cui, Zhiyu Hao, and Xiaochun Yun are with the Institute of Information Engineering, Chinese Academy of Science, Beijing 100093, China (e-mail: cuilei@iie.ac.cn; haozhiyu@iie.ac.cn; yunxiaochun@iie.ac.cn).

Yang Jiao and Haiqiang Fei are with the School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China, and also with the Institute of Information Engineering, Chinese Academy of Science, Beijing 100093, China (e-mail: jiaoyang@iie.ac.cn; feihaiqiang@iie.ac.cn).

Digital Object Identifier 10.1109/TIFS.2020.3047756

[1]We focus on C/C++ vulnerabilities at the granularity of function.

OpenSSL vulnerability CFGs have more than 50 nodes. On the other hand, although CFG can express semantical characteristics, some syntactic information such as declarations and statements are missing, thereby hurting accuracy. PDG and its variants (e.g., CPG, code property graph [24]) can provide more information by extending CFG with data dependency or AST, yet they are several times larger than CFG and thus imposes significant overhead. As a result, exiting graph-based approaches sacrifice either performance [10], [22], [25], [26] or accuracy [21], [27], [28]. More importantly, syntactic similarity and semantical similarity don't guarantee to find vulnerabilities, since the vulnerable code take only a small fraction of the entire function. Without taking this into account, it may miss the detection of vulnerabilities (false negatives) when the vulnerable snippets are similar yet the rest are not, or falsely identify vulnerabilities (false positives) when the most of two functions are similar yet no vulnerable code exists.

B. Our Approach

In this paper, we present VulDetector, a weighted feature graph-based approach capable of detecting vulnerabilities efficiently and effectively. **First**, VulDetector determines a set of sensitive keywords by analyzing vulnerabilities and then pinpoints the potentially vulnerable statements in the testing function where keywords are matched. Rooted from one statement, it slices the naïve CFG of function by only reserving potentially vulnerable snippets. **Second**, VulDetector embeds the AST expressions of source code into a vector of syntactic features and assigns it to each node, so that the formalized WFG can express both semantic and syntactic features effectively. In addition, it set varying weights to each node, where the weight denotes the sensitivity of the node to the pinpointed statement. Therefore, WFG can characterize code features in varying degrees of security. **Third**, for a potentially vulnerable graph, it should be more similar (in the view of vulnerability) to the known vulnerability graph than the patch graph. Inspired by this, VulDetector exploits both vulnerability graph and corresponding patch graph to improve graph similarity computation.

We have implemented a system capable of automatically collecting vulnerabilities and detecting vulnerabilities in software programs. In addition, several optimization methods based on analysis of vulnerabilities are proposed to reduce the search space and consequently enables efficient vulnerability detection in a large dataset. The experimental results show that VulDetector outperforms existing approaches in terms of precision, recall, and accuracy with comparable efficiency in detecting vulnerabilities. By two case studies, VulDetector is proved to be useful in real-world scenarios.

C. Contributions

To summarize, our contributions are as follows.

- First, a vulnerability-sensitive keywords selection method. It helps to locate the potentially vulnerable snippets and slice the raw graph into smaller subgraphs while preserving security-related information. Thus, it allows large code variance in insensitive snippets of two functions. Moreover, a smaller graph reduces the overhead of graph comparison without compromising accuracy.
- Second, a weighted feature graph (WFG) model. Compared to exiting graph models such as CFG, CPG, and ACFG, WFG provides both syntactic and semantical information by combing control dependency, data dependency and syntax together with a relatively small graph. In addition, it characterizes nodes with varying degrees of vulnerability sensitivity, enabling graph comparison to be more focused on potentially vulnerable ones.
- Third, optimization methods based on the analysis of vulnerabilities. In detail, a similarity computation method that compares two subgraphs exploiting both vulnerable code and patched code, and two optimization methods that exclude functions that are unlikely to be vulnerable.
- Finally, a static-analysis system and a set of experiments to evaluate VulDetector. The source code is available in https://github.com/leontsui1987/VulDetector.

The rest of the paper is organized as follows. The next section gives a brief overview of vulnerability detection approach using code similarity. Section III presents the basic design of our proposed method using weighted feature graph comparison. Section IV introduces the implementation details followed by the system evaluation in Section V. Then we present the previous related work in Section VI. Finally, we discuss the limitation of VulDetector in Section VII and conclude our work in Section VIII.

II. Background

Code similarity is one simple approach to detect vulnerabilities. The basic idea is to test whether a testing program contains code that is highly similar to a given vulnerable code. Generally, vulnerability detection using code similarity goes through three main steps. First, the testing program is divided into smaller comparison units, which can be at any level of granularity, such as file, class, function, basic block, or statement. Second, the unit is transformed into an intermediate representation that can effectively characterize the original code, meanwhile facilitating comparison. Finally, the transformed units are fed into a comparison algorithm where transformed units are compared with known vulnerable units to find matches. The matched units will be treated as potential vulnerabilities and reported for further confirmation. It is proved to be useful in detecting vulnerabilities [4], [5], [9], [10], [17], [18]. For example, Kim *et al.* show that their proposed tool VUDDY detected 144,496 vulnerable functions in more than 14 billion lines of code [4]. Li *et al.* detect 4 vulnerabilities which were not reported in the national vulnerability database [5]. The reason behind is that many vulnerabilities may share code that are identical in syntax and semantics. For example, a software developer may copy code from an open-source program. If the copied code is vulnerable, then the new code variant is prone to be vulnerable.

Generally speaking, there exist four types of code variants according to variance in syntax and semantics [7]. 1) Type-1. Identical code fragments except for variations in whitespace,

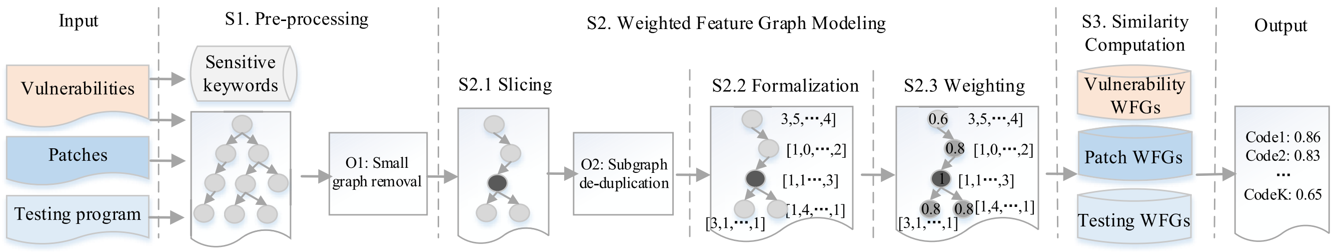


Fig. 1. Overview of VulDetector architecture.

layout, and comments. 2) Type-2. Syntactically identical fragments except for variations in identifiers, literals, and types. 3) Type-3. Copied fragments with further modifications such as changed, added or removed statements. 4) Type-4. Code fragments that perform the same computation but are implemented by different syntax.

Many studies abstract the source code so that various code are identical at a high level, e.g., tokens, trees and graphs, which helps to find Type-1 and Type-2 variants effectively [4], [11], [12], [14]–[16]. However, Type-3 and Type-4 variants are more difficult to detect because their differences in syntax and semantics can lead to large variance even at the high level [5], [9], [10], [18], [20]. Graph-based approach is promising since the graph structure can better characterize syntax and semantics of raw code. Unfortunately, the size of graph is generally large. Take the small function shown in Figure 3 as an example, the size of CFG, PDG is 6 and 10 respectively and reaches up to 35 for CPG. Due to the high computational complexity of graph matching algorithms, such a large graph would impose significant time overhead, especially when analyzing a large program. In addition, syntactic and semantical similarity do not guarantee to detect vulnerabilities, since the vulnerable code takes a little fraction of the entire function. Thus, how to detect Type-3 and Type-4 vulnerabilities effectively and efficiently is still challenging in real-world scenarios.

III. Basic Design

The main purpose of VulDetector is to analyze whether or not a testing software program contains vulnerabilities effectively and efficiently. Figure 1 illustrates the overview architecture of VulDetector. As we can see, in S1, the collected vulnerabilities are firstly preprocessed to acquire a set of vulnerability-sensitive keywords. Meanwhile, vulnerabilities, patches and testing program are divided into functions, each of which is transformed into a CFG. Then, these naïve CFGs are modelled as weight feature graphs through S2. In detail, with vulnerability-sensitive keywords, a CFG is sliced into one or many smaller subgraphs, which only reserves nodes and statements that have control and data dependencies with the matched keywords (S2.1). Then, WFG extends the subgraph by i) assigning each node with a vector of syntactic features (S2.2) and ii) assigning nodes with varying weighs to denote their importance to vulnerability (S2.3). Finally, the vulnerability WFGs, patch WFGs, and testing WFGs will be compared to identify vulnerabilities (S3). Note that the WFGs can be persistently stored for future use, i.e., detecting vulnerabilities in newly released programs, or re-analyzing old programs when new vulnerabilities are published. The following sections will describe the key ideas in VulDetector.

A. Vulnerability-Sensitive Keywords

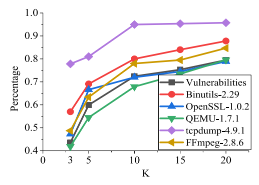
The vulnerability-sensitive keywords denote the identifiers that are more likely involved in vulnerabilities. As suggested in Checkmarx [29], some library functions are highly suspected to cause security issues, such as *memcpy*, *strcpy* for CWE199,[2] *malloc*, *free* for CWE399. Many user-defined functions and operations are also prone to security threats, e.g., *OPENSSL_malloc* in CVE-2014-0160, CVE-2016-2182, CVE-2016-2842, etc. A simple way to determine keywords is to collect frequently appeared identifiers in known vulnerabilities. Unfortunately, it would choose keywords overmuch. Moreover, many identifiers are domain-specific, e.g., *OPENSSL_malloc*, *OPENSSL_free* in OpenSSL, and *QEMU_alloc*, *QEMU_free* in QEMU, which limits its compatibility in analyzing different programs. To mitigate these problems, we propose to identify a few representative and generalized keywords. The basic idea is to generalize the identifiers extracted from the Diff file, compute the importance of the generalized identifiers, and finally select keywords by their importance.

1) Identifier Generalization: The Diff file, produced by the difference between vulnerable code and patched code, denotes the code changes involved in fixing a vulnerability. Thus, we use Diff file to approximate the really vulnerable code and extract identifiers from them. It can be observed that many identifiers denote the same kind of object in semantics. For example, *buf*, *buffer*, *dynbuf*, and *sigbuf* indicate a buffer area, while *len*, *inp_len*, and *length* denote the length of an object. Intuitively, *buf* and *len* are more generalized than *dynbuf* and *inp_len*. Inspired by this, we first employ the K-means clustering algorithm to cluster these identifiers. Then, we manually determine representative words (e.g., buf and len) and insert wildcard around each word. In this way, $*buf*$ and $*len*$ will be produced as generalized keywords. Note that here we only determine single identifiers and thus miss statements such as *if check* and *pointer arithmetic*. We will discuss the limitation in Section VII.

[2]CWE(Common Weakness Enumeration) denotes the underlying law of CVEs, and it can be considered as the category. Each CVE (Common Vulnerabilities & Exposures) is associated with one vulnerability.

CWE20	CWE119	CWE399
len	len	len
file	buf	buf
mac	data	frag
key	size	page
index	payload	free

(a) Top-5 keywords.



(b) Percentage with varying K.

Fig. 2. Performance of keywords.

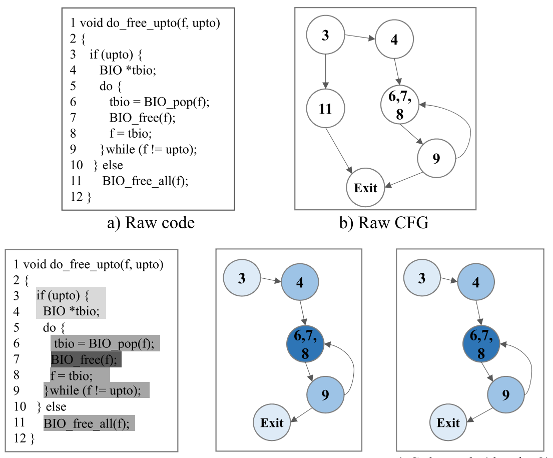
2) TF/DF Based Keyword Importance: To measure the importance of keywords, we borrow the idea of TF/IDF (Term Frequency/Inverse Document Frequency) [30] that reflects how important a word is to a document in a corpus. However, instead of using IDF, we use DF (Document Frequency) since it indicates how widely distributed a keyword is in a set of vulnerabilities. Combined with TF which measures how frequently a keyword occurs in vulnerabilities, the importance of a keyword is computed by $TF * DF$. Finally, the most important keywords will be selected as vulnerability-sensitive keywords. Note that the keywords are a little different for various categories of vulnerabilities (i.e., CWEs). Therefore, we determine top-K (K denotes the number of selected keywords) keywords for each CWE and treat the union as the final set. Figure 2(a) lists Top-5 keywords of three CWEs in OpenSSL and Linux vulnerabilities. E.g., CWE119 denotes *improper restriction of operations within the bounds of a memory buffer* and CWE399 denotes *improper management of system resources*, both of which are related to memory operations. Therefore, *buf*, as the expression of memory data, appears frequently.

Figure 2(b) compares the percentage of functions containing keywords to all functions when various K is set. As can be seen, when the number of keywords (i.e., K) is 10, about 72% of functions in OpenSSL-1.0.2 contain at least one keyword. This implies that most program functions can be sliced into smaller units. For the rest, the entire function will be compared upon detection.

B. Sensitive Keyword-Based Graph Slicing

VulDetector represents source code of a function with a CFG. As mentioned before, the size of CFG is large and thus imposes significant comparison overhead. In a vulnerability function, the really vulnerable snippet often takes a small fragment. Therefore, it is desired to split the entire graph into a smaller one without compromising security semantics. To achieve this, we propose a sensitive keyword-based slicing method to remove irrelevant code lines and graph nodes.

1) Slicing Code Lines: It employs the idea of data dependency to find identifiers and statements that are directly or indirectly related to the matched identifier in the source code. Specifically, given a keyword and a function, it first pinpoints the root line where the sensitive keyword is located. Specifically, given a sensitive keyword and a function, it first locates the identifier that matches the sensitive keyword, and then pinpoints the root line where the sensitive identifier is located. In this root line, other identifiers are considered to be directly



```
1 void do_free_upto(f, upto)
2 {
3   if (upto) {
4     BIO *tbio;
5     do {
6       tbio = BIO_pop(f);
7       BIO_free(f);
8       f = tbio;
9     }while (f != upto);
10  } else
11    BIO_free_all(f);
12 }
```

a) Raw code

b) Raw CFG

```
1 void do_free_upto(f, upto)
2 {
3   if (upto) {
4     BIO *tbio;
5     do {
6       tbio = BIO_pop(f);
7       BIO_free(f);
8       f = tbio;
9     }while (f != upto);
10  } else
11    BIO_free_all(f);
12 }
```

c) Sliced code

d) Sliced CFG

e) Subgraph (depth=3)

Fig. 3. Slicing a CFG from Line 7 (*free* is located). The darker the color, the closer to the root line/node.

related to the matched sensitive identifier and need to be traced. Then, started from all these identifiers in the root line, it repeatedly traces related identifiers using breadth-first search (BFS) backward and forward until no new identifier is found. Backward slicing intends to traverse the parent lines that affect the root line, while forward slicing intends to traverse the child lines that are affected by the root line. All these lines are critical to represent the vulnerable snippet, i.e., why it is introduced and how it will affect the execution. Therefore, we slice the function both forward and backward. Finally, the code lines locating these visited identifiers constitute the sliced snippet for the given keyword.

2) Slicing Graph Nodes: It shares a similar way to how the code lines are sliced. More specifically, it traverses the raw CFG using BFS to visit the nodes related to the root node where the root line is located. Then, the visited nodes constitute a sliced subgraph. Observed that the subgraph may still be large even after slicing, the BFS traversal terminates when a specified graph depth is reached (5 by default) to ensure that the subgraph is relatively small.

Finally, a sliced subgraph is produced by intersecting the visited lines and visited nodes for the given keyword. Figure 3 shows a case of slicing the function of CVE-2015-1792 [31]. Since *free* is a sensitive keyword, line 7 (*BIO_free*) and line 11 (*BIO_free_all*) will be selected as root lines. Figure 3 shows how to slice a CFG from line 7. As can be seen, c) depicts the sliced fragment of code lines, d) shows the sliced fragment of graph nodes in which the node of line 11 is removed since it is neither directly nor indirectly related to the root node, and e) shows the final subgraph when the graph depth is set to 3. It's worth noting that the number of sliced subgraphs is related to the number of matched sensitive identifiers. According to our analysis, the OpenSSL-1.0.2 function produces an average of 10.6 subgraphs when K is 10. Since a larger number implies more comparisons, we will de-duplicate these subgraphs in Section IV.

C. Formalization Using AST Features

CFG and AST are two classical representations of source code. Generally, CFG focuses more on the flow change

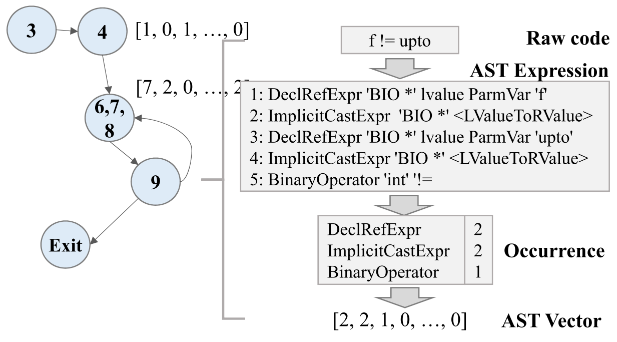

3
4
[1, 0, 1, …, 0]
[7, 2, 0, …, 2]
6,7,8
9
Exit
f != upto
Raw code
AST Expression
1: DeclRefExpr 'BIO *' lvalue ParmVar 'f'
2: ImplicitCastExpr 'BIO *' <LValueToRValue>
3: DeclRefExpr 'BIO *' lvalue ParmVar 'upto'
4: ImplicitCastExpr 'BIO *' <LValueToRValue>
5: BinaryOperator 'int' '!='
DeclRefExpr 2
ImplicitCastExpr 2
BinaryOperator 1
Occurrence
[2, 2, 1, 0, …, 0]
AST Vector

Fig. 4. Formalized graph using AST Features.

between code blocks and thus ignores the syntactic information insides the block, while AST is more focused on the syntax of source code. VulDetector formalizes the sliced subgraph using syntactic features to take advantages of the two. The key idea is to embed textual AST expression of a code block into a quantized vector and then assign it to the graph node. For example, LLVM defines *FunctionDecl* and *ValueDecl* for various declarations, *IfStmt* and *ForStmt* for various control statements in AST. Inspired by this, each dimension of the vector denotes a type in AST (e.g., *ValueDecl*, *IfStmt*), and its value measures the frequency the type appears in the AST expression of code block. As demonstrated in Figure 4, the expression in the text box denote the AST expression of code of line 9, i.e., *f!=upto*. In the expression, there exist two DeclRefExpr, two ImplicitCastExpr, one BinaryOperator, and 0 for other types. When preparing the AST vector, the value of the dimension is the number that the corresponding AST type appears in the expression. Assuming that DeclRefExpr, ImplicitCastExpr and BinaryOperator are the first three dimensions, then the produced AST vector is [2,2, 1, …, 0]. In this way, the formalized WFG model can characterize both semantic features with CFG and syntactic features using AST. Note that we use node as the unit to construct the AST vector. Therefore, for a node containing multiple statements, we merge the AST expressions of these statements, count the number of each AST type, and finally construct the vector.

One remaining problem is that the vector size is large. For example, LLVM provides more than 160 types in AST, so that each node would be expressed by a large vector, thereby introducing both storage cost and computational overhead. To mitigate this problem, we generate AST expressions from the vulnerabilities of OpenSSL and Linux, count the frequency of each type, and choose 32 most frequently used types as dimensions. In this way, a node is expressed by a 32 long AST vector. Consequently, each WFG is formalized by two matrices, i.e., a $P * P$ adjacency matrix which represents the graph structure where P refers to the number of nodes, and a $P * 32$ feature matrix in which each row denotes syntactic features of a node.

D. Graph Weighting

With slicing, the reserved code in the subgraph depend directly or indirectly on the root line containing the vulnerability-sensitive keyword. Nevertheless, the code contributes differently to the root line in the view of vulnerability. For example, a *len* identifier without being checked correctly may immediately result in information exposure on the subsequent *memcpy* operation (e.g., CVE-2014-3508), while it experiences multiple value operations before triggering a buffer over-read on *memcpy* (e.g., CVE-2014-0160). Therefore, it is preferred to model a weighted graph so that highly vulnerable code is more focused on comparison. Unfortunately, it is difficult to precisely assign weight to the node because of code complexity and vulnerability diversity. In this paper, we assume that the code closer to the root line (i.e., more related in control and data dependency) contributes more to the potential vulnerability.[3] Then, we assign varying weights to nodes. Note that a node may be close to the root node (i.e., high node weight) yet the associated code line is far from the root line (i.e., low line weight), and vice visa. Thus, we compute the weight of each code line and graph node separately and then combine the two as the final node weight.

1) Line Weight: As stated before, the sliced code lines are visited by BFS traversal from the root line. Let LW_0 denote the weight of the root line, then the visited code lines in the next layer (i.e., layer 2) are assigned with a decayed weight denoted by LW_2, which is calculated by $LW_0 * D$ where D indicates the decay ratio. Similarly, the lines in layer k are assigned with LW_k where $LW_k = LW_0 * D^{k-1}$.

2) Node Weight: The node weight is calculated following the way in line weight computation. The root node where the root line is located is assigned with NW_0, and the nodes in layer k are assigned with NW_k where $NW_k = NW_0 * D^{k-1}$.

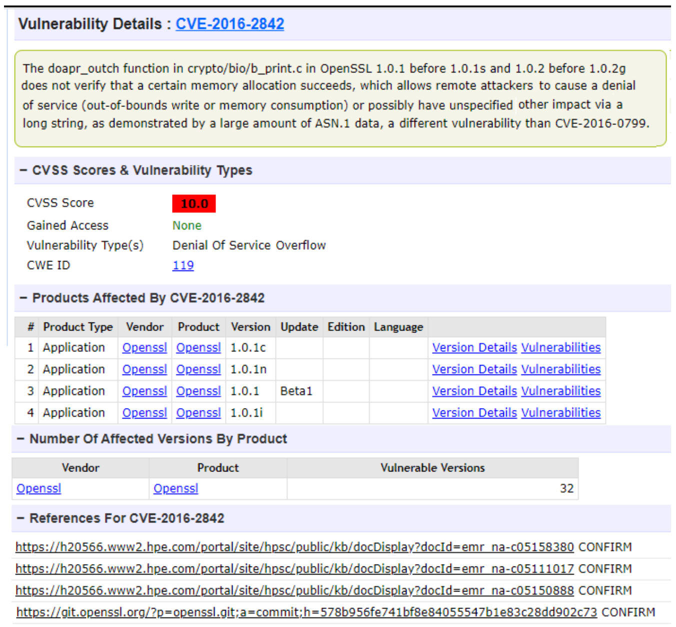
Finally, the weight of node is calculated by the production of line weight and node weight. E.g., for a node N_a in layer k and the code insides node in layer j, the final weight of node W_a is calculated by $NW_k * LW_j$. If the node contains multiple code lines, the maximum weight of code lines will be used for calculation, i.e., $W_a = NW_k * max(LW_j)$, $j = \{j_1, j_2, \ldots, j_n\}$ where j_i denotes the layer of ith code line. We by default set LW_0, NW_0 and D to 1, 1 and 0.85 respectively, and compare the performance with different settings in evaluation.

E. WFG Similarity

VulDetector intends to detect vulnerabilities by identifying the testing WFGs that are highly similar to known vulnerability WFGs. It's known that graph similarity is known as NP problem is hard to find the optimal solution. To this end, we employ the method proposed in Genius [28] which utilities bipartite graph matching to compute the similarity between two attributed control flow graphs (ACFGs). In Genius, ACFG extends CFG of binary function with binary-level attributes, such as No. of Instructions and No. of Calls. VulDetector is similar since it assigns a vector of syntactic features to each node. In addition, it sets varying weights to nodes. Therefore, for two nodes N_i and N_j, we replace $sim(N_i, N_j)$ in Genius with the weighted similarity $W_i * W_j * sim(N_i, N_j)$ to compute node similarity upon bipartite graph matching.

[3]The experimental results show that this assumption is held true in empirical.

In addition, we exploit the patch WFG to improve similarity computation. Specifically, the patched code involves code addition, deletion, or modification aiming to repair a vulnerability. Ideally, compared to the patched code P, a suspicious function is likely to be more similar to the unrepaired vulnerability code V. Inspired by this, if a testing WFG T is computed out to be highly similar to V, then it will be further compared with P. It is considered to be vulnerable only if $sim(T, V) > sim(T, P)$. This method can reduce false positives caused by Type-3/4 variants, e.g., a function that was vulnerable but now has already been fixed. The improvement of this effort will be shown in Section V.



Vulnerability Details : CVE-2016-2842

The doapr_outch function in crypto/bio/b_print.c in OpenSSL 1.0.1 before 1.0.1s and 1.0.2 before 1.0.2g does not verify that a certain memory allocation succeeds, which allows remote attackers to cause a denial of service (out-of-bounds write or memory consumption) or possibly have unspecified other impact via a long string, as demonstrated by a large amount of ASN.1 data, a different vulnerability than CVE-2016-0799.

– CVSS Scores & Vulnerability Types

CVSS Score	10.0
Gained Access	None
Vulnerability Type(s)	Denial Of Service Overflow
CWE ID	119

– Products Affected By CVE-2016-2842

#	Product Type	Vendor	Product	Version	Update	Edition	Language	
1	Application	Openssl	Openssl	1.0.1c				Version Details Vulnerabilities
2	Application	Openssl	Openssl	1.0.1n				Version Details Vulnerabilities
3	Application	Openssl	Openssl	1.0.1	Beta1			Version Details Vulnerabilities
4	Application	Openssl	Openssl	1.0.1i				Version Details Vulnerabilities

– Number Of Affected Versions By Product

Vendor	Product	Vulnerable Versions
Openssl	Openssl	32

– References For CVE-2016-2842

https://h20566.www2.hpe.com/portal/site/hpsc/public/kb/docDisplay?docId=emr_na-c05158380 CONFIRM
https://h20566.www2.hpe.com/portal/site/hpsc/public/kb/docDisplay?docId=emr_na-c05111017 CONFIRM
https://h20566.www2.hpe.com/portal/site/hpsc/public/kb/docDisplay?docId=emr_na-c05150888 CONFIRM
https://git.openssl.org/?p=openssl.git;a=commit;h=578b956fe741bf8e84055547b1e83c28dd902c73 CONFIRM

Fig. 5. The webpage of CVE-2016-2842 [35]. The page is tailored for clarity.

IV. Implementation Details

A. Overall Architecture

We implement VulDetector based on three open-source projects, Scrapy (a web spider) [32], LLVM (a compiler infrastructure) [33], and Clang (a C language analyze frontend of LLVM) [34]. It consists of four modules, Data Collector, Data Preparer, WFG Generator, and Vulnerability Detector. We first present how Data Collector collects data, mainly the vulnerabilities. Then, we describe the steps of vulnerability detection in a program in Algorithm 1, which involves Data Preparer, WFG Generator, and Vulnerability Detector.

Data Collector collects vulnerabilities by crawling CVE sites, e.g., CVE Details [36] within the paper. For each CVE webpage, the main information we want to acquire includes the involved function name, file name, affected versions of program, and patch (usually in external links). Ideally, all these information are available on the CVE webpage. As shown in Figure 5, we use some rules to extract the function name (e.g., a word is preceded by 'function' or contains '_') and file name (e.g., a word ended with '.c') from the description, acquire the list of affected versions from 'Products Affected' table, and obtain the patch from the external link (Step 1). Unfortunately, the function name or file name is not always provided in the description, e.g., CVE-2016-7052 misses the function name [37], and CVE-2016-7054 loses both of them [38]. To amend this issue, we resort to the patch where the involved file name and function name are probably located (Step 2). For the worst case that the links of patch are unavailable, we manually prepare the information by searching other web sites, e.g., stackoverflow (Step 3). For example, of the collected 203 OpenSSL CVEs, Step 1 provides 72 and Step 2 provides 53 more. The rest are manually collected by Step 3, which takes minutes per CVE. In addition, when the external links are unavailable, we treat the first released version after the vulnerability is patched as the patched version, from which the patched function is extracted. Meanwhile, Data Collector is responsible for downloading the source code of affected programs as well as testing programs for vulnerability detection.

Data Preparer takes charge of extracting the function from vulnerabilities, patches and testing programs. Specifically, given a tuple {program, file name, function name}, it employs *clang -Xclang -ast-dump* to dump the AST expression of the program. Then, it examines the function name following the keyword *FunctionDecl*. If it is the desired function, Data Preparer identifies the function's start line and end line from the AST expression, and finally extracts the function code from the file following these lines, as described in #2-#9 of Algorithm 1. The function code will be used to locate the sensitive root lines and slice the code lines by data dependency. In addition, Data Preparer is responsible for analyzing the collected vulnerabilities to determine the sensitive keywords by their TF*DF value, as described in §III-A.

WFG Generator is the key module in VulDetector. It first employs *scan_build*, a utility of Clang that parses a whole program [39], to dump control flow graphs for functions of the program. The control flow graph generated by *scan-build* consists of a set of basic blocks and their parent/child relationship. Thus, it locates the entry block and then builds the graph following the relationships, as mentioned in #10-#17 of Algorithm 1. Then, it slices the CFG started from the identified root line (#21-#24). After that, it examines the AST expression of code block and produces a vector of syntactic features. Meanwhile, it dumps the code line number to associate graph node and feature vector. Finally, it computes out the node weight for each node. In this way, a set of WFGs are produced for vulnerabilities, patches, and testing functions, as listed in #27-#29 of Algorithm 1.

Vulnerability Detector compares vulnerability WFGs with WFGs extracted from the testing program with the algorithm in §III-E. If a testing WFG is highly similar to a known vulnerability WFG, i.e., than a predefined threshold (e.g., 0.8), then the associated function will be reported for further investigation, as denoted in #30-#35.

B. Optimization Methods

We propose two optimization methods in VulDetector through the analysis of vulnerabilities and software programs.

Algorithm 1 Vulnerability detection in a program.

```
Input: $P$: Source code of program
$V_{wfg}$: A set of WFGs of known vulnerabilities
Output: $F_s$: Suspicious functions
1  $F$ = All functions of $P$
   /* Extract code body of functions */
2  $ASTs$ = AST expressions of $P$
3  for $f \in F$ do
4      $f_{ast}$ = AST expression of $f$ in $ASTs$
5      $f_c$ = extract_code_body($f_{ast}$, $P$)
6      if small_function($f_c$) then
7          remove($f$)
8      end
9  endfor
   /* Generate CFGs of functions */
10 $CFGs$ = CFG expressions of $P$
11 for $f \in F$ do
12     $BBs$ = Basic blocks of $f$ in $CFGs$
13     $f_{cfg}$ = generate_CFG($BBs$)
14     if small_graph($f_{cfg}$) then
15         remove($f$)
16     end
17 endfor
18 for $f \in F$ do
19     $SG$ = {} /* Slice into subgraphs */
20     $L$ = Sensitive root lines in $f_c$
21     for $l \in L$ do
22         $sg$ = slice($l$, $f_c$, $f_{cfg}$)
23         add_to_subgraph($SG$, $sg$)
24     endfor
25     deduplicate($SG$)
26     for $sg \in SG$ do
           /* Generate WFG */
27         $sg_{av}$ = compute_ast_vector($sg$, $f_{ast}$)
28         $sg_w$ = compute_node_weight($sg$, $f_c$)
29         $sg_{wfg}$ = generate_WFG($sg$, $sg_{av}$, $sg_w$)
           /* Compare with vulnerable WFGs */
30         for $v_{wfg} \in V_{wfg}$ do
31             $sim$ = compute_similarity($v_{wfg}$, $sg_{wfg}$)
32             if $sim \geq T$ then
33                 add_to_suspicious($F_s$, $f$, $sg$)
34             end
35         endfor
36     endfor
37 endfor
```

1) *Small Graph Removal:* Within the paper, a function (here function refers to the raw function before slicing) is considered as small if i) the number of code lines is ≤ 10 or ii) the associated raw graph size is ≤ 5. Table I reports the distribution of graph size and number of code lines respectively for functions of several programs and collected vulnerabilities. For example, for the raw CFGs of functions in OpenSSL-1.0.2 program, 39.6% of them have less than or equal to 5 nodes, 7.2% of them have more than 50 nodes. Meanwhile, for these functions, 39.6% of them contain no more than 10 lines of code, 5.9% of them contain more than 100 lines of code. As can be seen, small functions take a small percentage (1.4%) of vulnerability functions, yet they take a large fraction of program functions, e.g., 39.6% for OpenSSL-1.0.2 and 31.3% for Binutils-2.29. Intuitively, a small piece of code is unlikely to cause security issues due to its simple syntax and semantics. Therefore, the small functions in the testing program can be excluded directly for reducing computational overhead, as described in #6-#8 and #14-#16 of Algorithm 1. Also, the functions defined in header files can be excluded since almost all vulnerability functions are in source files.

TABLE I

DISTRIBUTION OF GRAPH SIZES AND FUNCTION LINES IN OPENSSL VULNERABILITIES AND PROGRAMS

Source	Graph Size			Function Line		
	≤ 5	5-50	> 50	$\leq$ 10	10-100	$>$ 100
Vulnerabilities	1.4%	51.8%	46.8%	2.7%	49.3%	48%
OpenSSL-1.0.2	39.6%	53.2%	7.2%	39.6%	54.5%	5.9%
Binutils-2.29	31.3%	54.8%	13.9%	25.2%	63.6%	11.2%
tcpdump-4.9.1	16.9%	62.2%	20.9%	17.4%	18.9%	13.7%
FFmpeg-2.8.6	31.5%	60.1%	8.4%	24.7%	67.3%	8.0%
QEMU-1.7.1	39.3%	57.4%	3.2%	51.8%	45.8%	2.4%

2) *Subgraph De-Duplication:* As stated before, a raw graph will generate several sliced subgraphs when many code lines contain keywords, e.g., averagely 10.6 subgraphs per raw graph of OpenSSL-1.0.2. For two root lines, if they lie in different root nodes, then the two subgraphs are structurally different and therefore both need to be tested. Otherwise, the two subgraphs are identical in structure yet exhibit difference in node attributes. For this case, only one line is employed to slice the graph. Specifically, we compute the weight of each line by the summary of TF*DF value of matched keywords in this line. Then, we choose the line with the maximum weight. Once if multiple lines have the same weight, one line is randomly selected as the root line. In this way, a large number of subgraphs can be de-duplicated, e.g., the number after de-duplication is reduced from 10.6 to 6.1 per raw graph of OpenSSL-1.0.2.

V. EVALUATION

A. Experimental Setup

1) *Hardware Configuration:* We conduct the experiments on one physical server configured with 6-core Intel Xeon CPU E5-2630 2.30GHz CPU, 64GB of RAM, and 1TB 7200 RPM WDC WD10EZEX.

2) *Dataset:* We collect the vulnerabilities of OpenSSL reported in CVE Details (203 CVEs in total by Dec 2019). Meanwhile, we download several programs as testing targets, including 194 versions of OpenSSL, Binutils-2.29, tcpdump-4.9.1, FFmpeg-2.8.6, QEMU-1.7.1.

3) *Approaches:* We compare VulDetector against 6 representative tools that cover text, token, tree and graph based code similarity methods. 1) NICAD, a text-based method [13].

TABLE II
DEFAULT PARAMETERS SET IN VULDETECTOR

Name	K	Graph depth	Initial weight	Decay ratio
Value	10	5	1	0.85

2) CPD, a token-based method [40]. 3) ReDebug, a syntax-based method [17]. 4) PDG, a variant of the graph-based method CBCD which compares two PDGs directly [10]. 5) VUDDY, an efficient hash-based method that intends to find exactly matched abstracted functions [4]. 6) vGenius, a variant of Genius [28]. Genius generates ACFG from binary code while vGenius targets source code.

The parameters in VulDetector is listed in Table II. In addition, we evaluate the results with varying graph depth and decay ratio in §V-D.

4) Performance Metrics: We measure the performance of vulnerability detection in terms of precision (p), recall (r), accuracy (a, which depicts an overall performance), and F1-score ($f1$, the harmonic mean of p and r) of various approaches. Let TP (true positive), FP (false positive), TN (true negative), FN (false negative) denote the number of correctly predicted vulnerabilities, falsely predicted vulnerabilities, correctly predicted benign functions, falsely predicted benign functions. The definition of these terms are as follows.

$$p = \frac{TP}{TP + FP} \tag{1}$$

$$r = \frac{TP}{TP + FN} \tag{2}$$

$$f1 = \frac{2 * p * r}{p + r} \tag{3}$$

$$a = \frac{TP + TN}{TP + FP + TN + FN} \tag{4}$$

Generally, a testing function is predicted as vulnerable if its similarity to vulnerability is higher than a specific threshold. Consider that it is unfair to set one threshold for different approaches, we set varying thresholds and report the results which achieve the best F1-score for these approaches.

B. Detecting Type-2/3 Vulnerabilities

The same function across different versions may change for many purposes including code optimization and bug fixing, and thus they reflect Type-2 and Type-3 variants. In this experiment, we compare vulnerability functions of OpenSSL-1.0.1a (72 CVEs, released on Apr 2012) with the same functions of 16 OpenSSL versions. Note that some functions in these versions are benign if they have been patched in newer versions or not yet affected in older versions.

1) Performance: Figure 6 compares the accuracy of different approaches for each version (for clarity, we report 6 versions released from Mar 2010 to Feb 2016). As we can see, all approaches perform well for OpenSSL-1.0.1. This is because this version is released closely before OpenSSl-1.0.1a, so many vulnerability functions of the two versions are exactly the same. For other versions that exhibit larger

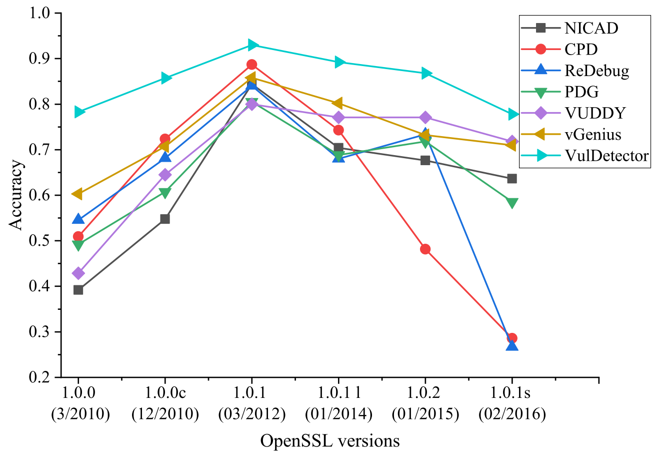


Fig. 6. Comparison of accuracy of each OpenSSL version (Release date is also reported).

variance, VulDetector outperforms other approaches. E.g., for OpenSSL-1.0.2 the accuracy achieves up to 0.87 for VulDetector while it is 0.68, 0.48, 0.73, 0.72, 0.77 and 0.73 for NICAD, CPD, ReDebug, PDG, VUDDY and vGenius respectively. The improvement comes from two factors. First, the WFG model only reserves snippets that are related to sensitive lines and thus enables comparison to be more focused on sensitive snippets. Therefore, it is hardly affected by changes (even large) in non-sensitive code, which yet impacts methods comparing the full function. Meanwhile, it characterizes these snippets both syntactically and semantically. Second, it exploits the patched code upon comparison to reduce false positives. Take OpenSSL-1.0.1s as an example in which many vulnerability functions have been fixed (released 4 years later after OpenSSL-1.0.1a), VulDetector achieves high accuracy (i.e., 0.78) in predicting vulnerable and benign functions. In contrast, the CPD and ReDebug approaches only reach an accuracy of 0.28 and 0.26 respectively. This is because they compute the similarity of the full body of functions. If the patched code only take a small part of the whole function, they will view the fixed function and vulnerable function as highly similar and thus regard it as vulnerable. In addition, vGenius, a graph-based method, performs well mainly because it is able to characterize the syntactic and semantical information of source code. Nevertheless, the accuracy is lower than that of VulDetector which focuses more on the vulnerable snippets instead of the whole function.

Figure 7 reports the average performance of 16 versions. As can be seen, the metrics of two naïve methods, i.e., NICAD (text-based) and CPD (token-based), are low, e.g., the precision is 0.49 and 0.52 respectively. This is mainly because of two reasons. First, they view the source code as text or sequence and thus ignore the syntax of code. Second, they compute the similarity of two full texts/sequences, so that the similarity between vulnerable snippets (if have) will be concealed by other snippets. ReDebug characterizes the syntax by formalizing the source code as trees and thus improves the performance. PDG and vGenius perform better than ReDebug because they use graphs to represent the source code which

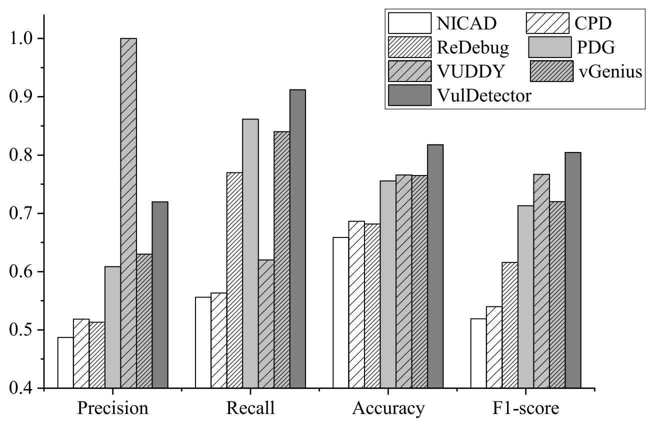


Fig. 7. Performance comparison of approaches.

TABLE III
PERFORMANCE ON DIFFERENT CWES

CWE	Description	F1-score	Accuracy
CWE399	Resource Management Errors	0.87	0.90
CWE20	Improper Input Validation	0.84	0.89
CWE200	Information Exposure	0.834	0.819
CWE125	Out-of-bounds Read	0.806	0.712
CWE119	Improper Restriction of Ops	0.785	0.808
CWE310	Cryptographic Issues	0.761	0.801
CWE189	Numeric Errors	0.737	0.786
CWE476	NULL Pointer Dereference	0.687	0.777

TABLE IV
NUMBER OF DETECTED CVES

NICAD	CPD	ReDebug	PDG	VUDDY	vGenius	VulDetector
4	5	10	9	0	11	26

help to characterize source code in both syntax and semantics. E.g., they achieve 0.71 and 0.72 in F1-score respectively compared to 0.62 of ReDebug. Our proposed VulDetector outperforms these approaches in these four metrics, i.e., it achieves 0.72 of precision, 0.91 of recall, 0.82 of accuracy, and 0.80 of F1-score. As described before, it characterizes the syntax and semantics using CFG with formalized AST vectors, and enables comparison to be more focused on vulnerable snippets. It's worth noting that VUDDY achieves 100% of precision because it intends to find exactly matched functions. However, it fails to detect Type-3 vulnerabilities and thus results in low recall, i.e., 0.62. The results prove that VulDetector is useful in detecting Type-2 and Type-3 vulnerabilities.

2) Performance on Different Vulnerability Types: We evaluate VulDetector on different CWEs to show how it performs on different types of weakness. We use the results of Figure 7 and recalculate f, r, a and $f1$ by CWEs to which 72 CVEs belong. Table III reports the F1-score and accuracy on 8 CWEs (the CWE whose CVE number is less than 5 is excluded). As we can see, VulDetector performs well on CWE399, CWE20 and CWE200, yet it performs relatively poor on CWE189 and CWE476. We attribute this to the difference in the ability of identifying sensitive keywords for various CVEs. Specifically, CWE399 and CWE200 are more likely to involve memory related operations, while CWE20 focuses on input validation. Therefore, the keywords such as *len*, *mem*, *buf*, *size* are more easily located. On the other hand, unless keywords are involved, VulDetector cannot identify statements and thus fails to identify numeric operations and pointer arithmetic. Consequently, the performance on CWE189 (Numeric Errors) and CWE476 (NULL Pointer Dereference) is poorer.

C. Detecting Vulnerabilities in Programs

In this experiment, we show whether VulDetector is effective in detecting new vulnerabilities in a different program. We use 72 CVEs in OpenSSL-1.0.1a, compare each vulnerability function with all functions of five testing programs, and report the Top-20[4] similar functions for each CVE as potential vulnerability functions. Of the reported 1440 (72*20) functions, if one exists in the published CVEs of five testing programs, then it is considered to be correctly identified. The testing programs are OpenSSL-1.0.2 (22 new CVEs since OpenSSL-1.0.1a), Binutils-2.29 (49 CVEs), tcpdump-4.9.1 (86 CVEs), FFmpeg-2.8.6 (6 CVEs), QEMU-1.7.1(40 CVEs).

[4]A large number implies heavy human effort to confirm the vulnerability. 20 is acceptable for manually analysis in real-world scenarios.

1) Performance: As shown in Table IV, VulDetector identifies 26 CVEs which outperforms other approaches, proving its effectiveness in detecting new vulnerabilities of software programs. E.g., ReDebug and vGenius correctly detects 10 and 11 vulnerabilities respectively, and VUDDY fails to detect vulnerabilities since there is no exact match. Table V lists several CVEs identified by VulDetector yet missed by the others. We use a case to explain why VulDetector works. CVE-2015-0205 in OpenSSL-1.0.1a involves function *dtls1_buffer_record* containing 59 lines of code, and CVE-2017-15225 in Binutils-2.29 involves function *_bfd_dwarf2_cleanup_debug_info* containing 93 lines of code. The two functions are implemented differently, thus, their similarity using token/syntax-based method is low. Suppose the function is transformed into CFG, then the two CFGs have 12 and 42 nodes respectively. Clearly, it is still difficult to regard them as highly similar using graph comparison. Therefore, the other 6 approaches fail to view the two functions as highly similar. Using VulDetector, since both functions contain the *free* keyword, i.e., *OPENSSL_free* and *free* respectively, two subgraphs of size 6 and size 7 are produced. Meanwhile, the AST expressions extracted from the code block contain similar types including *FunctionDecl*, *IfStmt*. Finally, the similarity between the two WFGs ranks 1st among similarities of all Binutils-2.29 functions and *dtls1_buffer_record*. Additionally, the WFG of *_bfd_dwarf2_cleanup_debug_info* is quite different from that of the patched *dtls1_buffer_record*, which further improves the confidence.

2) False Negative: Although VulDetector detects more vulnerabilities than other approaches, it still misses a lot of CVEs, e.g., 43 of 49 CVEs in Binutils-2.29 are not reported. We manually analyze 20 undetected CVEs and find that false negatives mainly fall into the following types. 1) No sensitive keywords (3 cases). As discussed before, VulDetector only

TABLE V
SEVERAL CVEs IDENTIFIED BY VULDETECTOR YET MISSED BY OTHER APPROACHES

CVE	Vulnerability Type	Program
CVE-2015-0290	DoS	OpenSSL-1.0.2
CVE-2017-15996	DoS, Overflow	Binutils-2.29
CVE-2017-15225	DoS	Binutils-2.29
CVE-2013-4537	Execute Code	QEMU-1.7.1
CVE-2018-14394	DoS	FFmpeg-2.8.6
CVE-2017-13045	Unknown	tcpdump-4.9.1

identifies a small set of sensitive keywords and cannot deal with statements or expressions. 2) Small function (1 case). VulDetector excludes small functions, which may contain vulnerabilities. 3) Function is context dependent (1 case). A CVE may involve multiple related functions, each of which contains a small piece of vulnerable snippet. 4) Not similar to known CVEs (15 cases). This is possibly because we use only 72 CVEs as the CVE database for detection, which takes a small piece of vulnerability space. We expect that a larger CVE database could help to detect more CVEs.

3) False Positive of Vulnerabilities: Generally, all the reported suspicious functions require manual investigation. Therefore, false positive (FP), which reflects the falsely predicted vulnerabilities, is vital for the practicality of the detection approach. The results in Figure 7 show that VulDetector achieves low FP in detecting Type-2/3 vulnerabilities. For example, 28% of the reported functions are FPs, while it is for 51.3%, 48.2%, 48.7%, 39.2% and 37.0% respectively for NICAD, CPD, ReDebug, PDG and vGenius. Although VUDDY causes no false positive, it is limited to detect Type-1/2 vulnerabilities. The improvement of VulDetector is mainly contributed to two aspects. First, VulDetector is more focused on the potentially vulnerable snippets instead of the entire function, thereby excluding function that is highly similar to the vulnerability in most code yet exhibits difference in the vulnerable snippets. Second, it exploits the patched code upon comparison to exclude benign or patched function which shares similar code with a vulnerability.

Nevertheless, the results in Table IV demonstrate that VulDetector still suffers high false positives in testing programs. For example, of the reported 1,440 (i.e., 72*20) functions, only 26 are confirmed as vulnerabilities. Observed that some functions in Top20 are not highly similar to vulnerability functions, we can only report function whose similarity to a vulnerability is larger than 0.85 (0.85 achieves the best F1-score in Figure 7). In this way, 872 (rather than 1,440) functions are reported without compromising the ability of vulnerability detection. We will explore to reduce false positives further in the future.

D. Performance Gain of Efforts

Here we measure the performance gained by different efforts in VulDetector, i.e., AST features (A), graph slicing (S), graph weighting (W), comparison using patch (P). Here, + denotes the combination of efforts, e.g., *A+S* combines

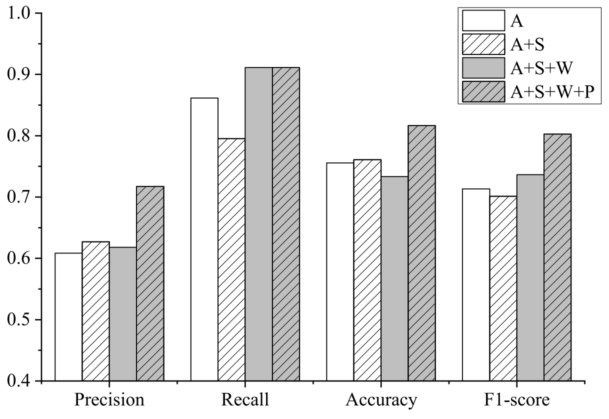


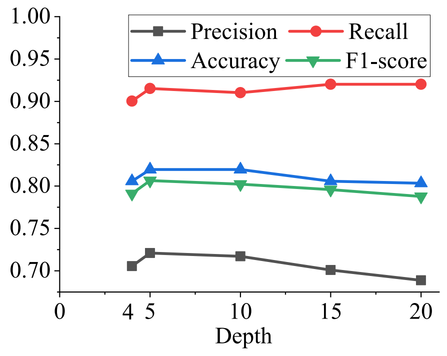
Fig. 8. Performance gain by proposed efforts.

AST features with graph slicing, while *A+S+W+P* denotes VulDetector using all efforts. We leverage these methods to compare vulnerability functions of OpenSSL-1.0.1a with same functions of 16 versions, and report the average results in Figure 8. In term of F1-score, *A+S+W* improves *A+S* by 4.0%, and *A+S+W+P* improves 8.2% further, proving the effectiveness of *W* and *P* respectively. Note that *A+S* performs a bit worse than *A* (i.e., using the whole CFG), because trimming codes causes information loss on the subgraphs and consequently hurts accuracy upon comparison. When coupled with graph weighting, *A+S+W* performs better than *A* due to its increasing focus on more vulnerable code.

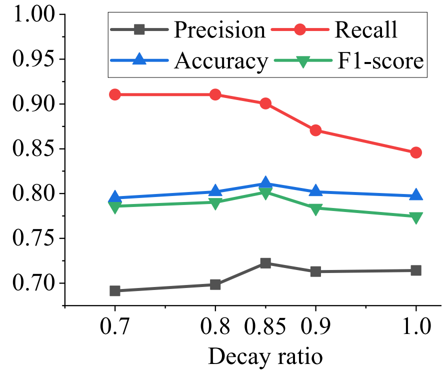
1) Varying Settings: We then measure the performance of VulDetector under varying settings, i.e., graph depth that dominates subgraph size upon slicing and decay ratio that determines node weight. As illustrated in Figure 9, VulDetector performs well with the default setting, i.e., the F1-score is 0.806 when depth is 5 and decay ratio is 0.85, both of which are higher than other settings. We use the case of graph depth to explain the reason. Generally, the subgraph grows with the increase of graph depth, e.g., it averagely takes 26.1% of the raw graph when depth is 3 and increases to 41.9%, 58.8% and 68.9% respectively when depth is 5 and 10, 20. A smaller subgraph will loss useful information, so that the similarity is amplified which eventually causes false positives. On the other hand, a larger one may contain useless information which diminishes similarities, thereby leading to false negatives. Thus, it is reasonable to determine a middle-size depth (e.g., 5) to achieve better performance.

E. Efficiency

We measure the efficiency in term of time cost. The process of VulDetector experiences three main steps, i.e., CFG extraction from source code, WFG formalization, and WFG comparison. The first two steps are executed only once, and then the generated WFGs can be stored persistently for repeated comparison. Table VI reports the cost of the first two steps. As we can see, the major cost is on CFG extraction, e.g., it takes 2h30m to extract 12,668 functions from OpenSSL-1.0.2. This is mainly because we employ Clang, which requires to parse the whole program for generating CFGs.



(a) Varying graph depth.



(b) Varying decay ratio.

Fig. 9. Performance of varying settings.

TABLE VI
TIME COST OF CFG AND WFG FORMALIZATION

	Extraction	CFG Cnt	Formalization	CFG Cnt
OpenSSL	2h30m	12,668	564s	3,617
Binutils	4h22m	6,699	523s	3,359

WFG formalization is relatively efficient, e.g., it processes 3,617 graphs of OpenSSL-1.0.2 in 563 seconds (other graphs are excluded since they are small or in the header files). This is because formalization involves lightweight operations, including keywords matching, BFS traversal (terminate when a specified depth is reached), feature extraction (AST expressions are generated upon CFG extraction), and weight assignment (performed on BFS traversal).

The comparison time is directly related to the size of representations, e.g., tokens and graphs. Here, we report the average time of comparing any two functions from OpenSSL. As reported in Table VII, VulDetector performs poorer than other approaches except PDG. The reason is as follows. The comparison time of two WFGs is small, i.e., 0.088 seconds, since the computation is performed on small matrices ($P * 32$). However, each function produces several WFGs, so that the total time of comparing two functions is multiple times larger. Fortunately, a large number of raw graphs can be excluded with the proposed optimizations, e.g., 71.4% of OpenSSL-1.0.2 and 49.8% of Binutils-2.29 shown in Table VI. Therefore, the total time cost for testing a program is comparable

TABLE VII
COMPARISON TIME OF TWO FUNCTIONS (SECONDS)

NICAD	CPD	ReDebug	PDG	VUDDY	vGenius	VulDetector
0.229	0.399	0.083	0.903	0.02	0.31	0.49

```
BIO_flush(bio);
do {
      tbio = BIO_pop(bio);
      BIO_free(bio);
      bio = tbio;
} while (bio != out);
```

Listing 1. Code snippet in OpenSSL.

with tree and graph based approaches. ReDebug and VUDDY are efficient mainly because they compute the hash value of the function and thus finds matches by hash lookup.

F. Use Cases in Real World

In this section, we will show that VulDetector is useful in real-world applications by two use cases.

1) Detecting Unknown Vulnerabilities: With VulDetector, we find several suspicious functions that are highly similar to known vulnerability functions. For example, Listing 1 denotes a code snippet of a OpenSSL function. It is highly similar to function *do_free_upto*(CVE-2015-1792, the code is shown in Figure 3a)), which allows remote attackers to cause a denial of service (infinite loop) by triggering a NULL value of a BIO data structure. It can be seen that *bio != out* would be enabled using a similar method, so that infinite loop is triggered. We will investigate these functions further and report them once they are confirmed as vulnerabilities.

2) Detecting Misinformation in CVE Description: In a CVE description, the list of affected program versions should be accurate, since it indicates which software is vulnerable. Unfortunately, due to the heavy effort to manually keep it up-to-date, many public reports contain erroneous information and miss the affected versions [41]. With VulDetector, we compare the vulnerability function with the same function in other versions of OpenSSL. Then, for a function with high similarity yet whose version is not listed in the description, we manually check it and finally find that the affected versions of 8 CVEs are inaccurate. E.g., CVE-2014-3507 [42] affects OpenSSL versions from 1.0.1i to 1.0.1u, and CVE-2015-1792 [43] affects OpenSSL versions from 1.0.2b to 1.0.2o, yet they are not listed in the CVE webpage.

VI. RELATED WORKS

We discuss several code similarity-based methods from four categories according to how the source code is transformed.

A. Text (String)

Text-based approach directly operates raw source code [4], [11]–[13], [44]. For example, Hunt *et al.* use longest common sequence for searching similar content in files [44]. Roy *et al.* propose NICAD to parse code text with potential clone extractor and standard pretty-printer and compare potential clones by clustering [13]. Johnson splits the source code into several fragments containing several code lines, computes hash value on each fragment, and then finds similar code by matching code lines having the same hash value [11]. Manber *et al.* use a similar method for comparison, the difference is that they select the keywords from the subsequences of code and compute the fingerprint of these keywords instead of the whole sequence [12]. Kim *et al.* propose VUDDY to abstract and normalize the code at the function level, compute the fingerprint for each function, and finally find similar code by performing hash lookup [4].

B. Token-Based

Token-based technique divides each code line into tokens according to the lexical rules and then compares the token sequence [14]–[17]. For example, Kamiya *et al.* extract a sequence of tokens through a lexical analyzer and transform code blocks into a regular form for matching code having similar meaning [14]. Sajnan *et al.* divide the code into smaller code blocks and represent each block with a bag of tokens. Then, they find similar blocks by querying from a partial inverted index [15]. Li *et al.* parse a program into token sequences and employ the frequent subsequence mining to find matched code [16]. Jang *et al.* propose ReDeBug to slide a window of N tokens and apply K independent hash functions to each window. Finally, they use the Bloom Filter method to find matches [17].

C. Tree-Based

Tree-based approach represents the code with syntactic structures such as parse tree or AST, and then finds similar code by tree matching [9], [18]–[20]. For example, Jiang *et al.* build ASTs for each file and extract characteristic vectors from the ASTs. Then, they compute the similarity between vectors by Euclidean distance to find similar code [18]. To improve the performance of tree matching, Baxter *et al.* compute hash value on nodes of ASTs and perform fast searching by hash lookup [19]. Koschke *et al.* build abstract syntax suffix trees to find syntactic matches in linear time and space [20]. Yamaguchi *et al.* extract patterns from ASTs and use latent semantics analysis technique to identify semantic matches [9].

D. Graph-Based

Graph-based method characterizes source code with CFG, PDG or CPG, in which the nodes denote statements or identifiers while the edges represent control or data dependencies [5], [10], [21], [22], [26], [27], [45]–[49]. For example, Leitao combines AST and call graphs to characterize source code in both syntax and semantics, and then designs a specialized algorithm to compare graphs [22]. Li *et al.* propose CBCD that uses isomorphism matching on PDGs to find similar code [10]. Gabel *et al.* map the graph to structured syntax and solve the tree similarity problem instead of the complex graph similarity problem. Thus, the proposed method can scale to big code while matching semantically similar code [21]. Qian *et al.* propose Genius to produce a ACFG for each binary function and then compare two ACFGs for detection [28]. Recently, many studies combine deep learning and graph in vulnerability detection [5], [25], [50]–[52]. For example, Grieco *et al.* propose a method that extracts a set of static and dynamic features and then employs random forest model to train a classifier [50]. Xu *et al.* use attributed CFG to represent the binary function and then train a Siamese network to compare two graphs [51]. Li *et al.* extract a set of features from vulnerabilities and patches, train a set of classifiers on features, and automatically choose the most suitable one when detecting vulnerabilities [53]. Li *et al.* slice the PDG to acquire a code gadget and transform each code gadget into a vector of symbolic representation. Then, they train a BLSTM neural network on vectors for vulnerability classification [5]. Wang *et al.* train a deep learning model on CPGs to detect command injections [25]. Li *et al.* propose a graph matching network model that directly trains on CFGs to compare graphs for vulnerability detection [52].

In this paper, we characterize the program function with graph. Compared to CFG, PDG and ACFG, the proposed WFG combines control dependencies and AST features without losing data dependencies, and thus well characterizes function in both syntax and semantics. Meanwhile, it sets vulnerability-sensitive weights to nodes, so that the comparison is more focused on potentially vulnerable snippets. In addition, VulDetector slices the raw CFG into smaller subgraphs and embeds heavy AST expression into a small size vector, so that WFG is much smaller than PDG, CPG and ACFG, which enables efficient graph comparison. Compared to deep learning methods which are limited to detect vulnerabilities of a few types (e.g., 2 CWEs in VulDeepecker [5]), VulDetector is more general and applicable to a wider range of CWEs.

VII. Discussion and Limitations

The experimental results show that VulDetector is effective and efficient in detecting vulnerabilities. However, there exist several limitations in VulDetector.

First, keywords selection. Within the paper, we determine the keywords from Linux and OpenSSL vulnerabilities which accounts for a small fraction of all vulnerabilities. Meanwhile, we focus on single identifiers and ignore statements or expressions such as *pointer arithmetic*, *if check*, *divide by zero*. In addition, synonym matching, such as *msg* and *message*, *cnt* and *count*, are not handled. Consequently, the selected keywords cannot cover all potentially vulnerable snippets, and may cause false negatives upon detection. Therefore, a more wise method is required to select keywords.

Second, the extension to other languages. Although the idea of VulDetector is language independent, the current version only focuses on C/C++ vulnerabilities mainly because it employs Clang to parse source code. To support other languages, a parser tool that generates CFGs and AST expressions is required, such as Soot for Java.

Third, system tuning. Some settings in VulDetector, mainly the graph depth and decay ratio, require human tuning. The default setting may not perform as well in other programs as it does in the paper. Thus, a tuning method that can adapt to other programs is needed.

VIII. Conclusion and Future Work

This paper presents VulDetector, a static-analysis tool to detect vulnerabilities based on graph comparison at the function level. VulDetector builds a WFG model to characterize function code syntactically and semantically in varying degrees of vulnerability sensitivity, and enables the comparison to be more focused on potentially vulnerable snippets. By a set of experimental results and two case studies, VulDetector is proved to be useful in detecting vulnerabilities.

Given the limitations discussed above, we will put more efforts in two directions in our future work. First, we plan to extend sensitive keywords to sensitive statements, so as to locate potentially vulnerable snippets more accurately. Second, we plan to support more programming languages in VulDetector to make it more useful in practice.

Acknowledgment

The authors would like to thank the reviewers of the IEEE Transactions on Information Forensics and Security.

References

[1] (2019). *Github*. [Online]. Available: https://github.com/

[2] (2016). *Heartbleed*. [Online]. Available: https://github.com/

[3] (2017). *Wannacry*. [Online]. Available: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

[4] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 595–614.

[5] Z. Li *et al.*, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[6] M. Zhang, X. de Carne de Carnavalet, L. Wang, and A. Ragab, "Large-scale empirical study of important features indicative of discovered vulnerabilities to assess application security," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 9, pp. 2315–2330, Sep. 2019.

[7] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009.

[8] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 187–196, Sep. 2005.

[9] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proc. 28th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2012, pp. 359–368.

[10] J. Li and M. D. Ernst, "CBCD: Cloned buggy code detector," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 310–320.

[11] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proc. Conf. Centre Adv. Stud. Collaborative Res., Softw. Eng.*, 1993, pp. 171–183.

[12] U. Manber, "Finding similar files in a large file system," in *Proc. USENIX Winter Tech. Conf.*, 1994, p. 2.

[13] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. 16th IEEE Int. Conf. Program Comprehension*, Jun. 2008, pp. 172–181.

[14] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[15] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 1157–1168.

[16] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *Proc. Operating Syst. Design Implement.*, 2004, p. 20.

[17] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire OS distributions," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 48–62.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, May 2007, pp. 96–105.

[19] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. Int. Conf. Softw. Maintenance*, Nov. 1998, pp. 368–377.

[20] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Proc. 13th Work. Conf. Reverse Eng.*, 2006, pp. 253–262.

[21] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*, 2008, pp. 321–330.

[22] A. M. Leitao, "Detection of redundant code using R^2D^2," *Softw. Qual. J.*, vol. 12, pp. 183–192, Mar. 2003.

[23] J. Chen, S. Lu, S. Sze, and F. Zhang, "Improved algorithms for path, matching, and packing problems," in *Proc. Symp. Discrete Algorithms*, 2007, pp. 298–307.

[24] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 590–604.

[25] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, "CPGVA: Code property graph based vulnerability analysis by deep learning," in *Proc. 10th Int. Conf. Adv. Infocomm Technol. (ICAIT)*, Aug. 2018, pp. 184–188.

[26] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 291–302, Aug. 2015.

[27] G. Upadhyaya and H. Rajan, "On accelerating source code analysis at massive scale," *IEEE Trans. Softw. Eng.*, vol. 44, no. 7, pp. 669–688, Jul. 2018.

[28] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 480–491.

[29] (2019). *Checkmarx*. [Online]. Available: https://www.checkmarx.com/

[30] S. Robertson, "Understanding inverse document frequency: On theoretical arguments for IDF," *J. Documentation*, vol. 60, no. 5, pp. 503–520, Oct. 2004.

[31] (2019). *Cve-2015-1792*. [Online]. Available: https://www.cvedetails.com/cve/CVE-2015-1792

[32] (2019). *Scrapy*. [Online]. Available: https://scrapy.org/

[33] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim.*, Mar. 2004, pp. 75–86.

[34] (2019) *Clang*. [Online]. Available: http://clang.llvm.org/

[35] (2016). *Cve-2016-2842/*. [Online]. Available: https://www.cvedetails.com/cve/CVE-2016-2842

[36] (2019). *Cve details*. [Online]. Available: https://www.cvedetails.com/

[37] (2016). *Cve-2016-7052/*. [Online]. Available: https://www.cvedetails.com/cve/CVE-2016-7052

[38] (2016). *Cve-2016-7054*. [Online]. Available: https://www.cvedetails.com/cve/CVE-2016-7054

[39] (2019). *Scan-Build*. [Online]. Available: https://clang-analyzer.llvm.org/scan-build.html

[40] (2019). *Copy-Paste Detector*. [Online]. Available: https://pmd.github.io

[41] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *Proc. USENIX Secur. Symp.*, 2019, pp. 869–885.

[42] (2014). *Cve-2014-3507*. [Online]. Available: https://www.cvedetails.com/cve/CVE-2014-3507/

[43] (2015). *Cve-2015-1792*. [Online]. Available: https://www.cvedetails.com/cve/CVE-2015-1792/

[44] J. W. Hunt and M. D. McIlroy, "An algorithm for differential file comparison," Bell Laboratories, New Providence, NJ, USA, Tech. Rep., 1976.

[45] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proc. 9th ACM Conf. Comput. Commun. Secur.*, 2002, pp. 217–224.

[46] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of software plagiarism by program dependence graph analysis," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 872–881.

[47] P. D. Schubert, B. Hermann, and E. Bodden, "PhASAR: An inter-procedural static analysis framework for C/C++," in *Tools and Algorithms for the Construction and Analysis of Systems*. 2019, pp. 393–410.

[48] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 10197–10207.

[49] H. Alasmary *et al.*, "Analyzing and detecting emerging Internet of Things malware: A graph-based approach," *IEEE Internet Things J.*, vol. 6, no. 5, pp. 8977–8988, Oct. 2019.

[50] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2016, pp. 85–96.

[51] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 363–376.

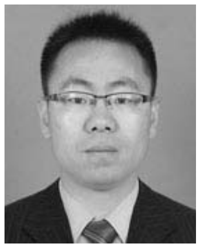
[52] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," 2019, *arXiv:1904.12787*. [Online]. Available: https://arxiv.org/abs/1904.12787

[53] 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 *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, 2016, pp. 201–213.

Lei Cui (Member, IEEE) received the Ph.D. degree in computer software and theory from Beihang University in 2015. He is currently an Associate Professor with the Institute of Information Engineering, Chinese Academy of Sciences. He has published over 30 papers in journals and conferences, including IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS (TPDS), IEEE TRANSACTIONS ON SERVICES COMPUTING (TSC), RAID, VEE, LISA, DSN, and *The Computer Journal*. His research interests include operating systems, system security, and system virtualization.

Zhiyu Hao received the Ph.D. degree in computer system architecture from the Harbin Institute of Technology in 2007. He is currently a Professor with the Institute of Information Engineering, Chinese Academy of Sciences. He has published over 40 papers in journals and conferences, including IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS (TPDS), ICPP, IEEE S&P, ICA3PP, and CLUSTER. His research interests include network security, system virtualization, and network emulation.

Yang Jiao received the master's degree from the Institute of Information Engineering, Chinese Academy of Sciences. Her research interests include network security and software security.



Haiqiang Fei received the master's degree from the Institute of Computing Technology, Chinese Academy of Sciences, where he is currently pursuing the Ph.D. degree with the Institute of Information Engineering. His research interests include network emulation and network security.

Xiaochun Yun received the Ph.D. degree from the Harbin Institute of Technology in 1998. He is currently a Full Professor with the Institute of Information Engineering, Chinese Academy of Sciences, China. He also works with the National Computer Network Emergency Response Technical Team/Coordination Center of China. He has authored more than 200 papers in refereed journals and conference proceedings. His research interests include network and information security.