VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis

Zhen Li[†§], Deqing Zou[†*], Shouhuai Xu[‡], Hai Jin[†], Hanchao Qi[†], Jie Hu[†]
[†] Services Computing Technology and System Lab, Big Data Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
[§]School of Computer Science and Technology, Hebei University, Baoding, 071002, China
[‡] Department of Computer Science, University of Texas at San Antonio, San Antonio, TX 78249, USA
deqingzou@hust.edu.cn

ABSTRACT

Software vulnerabilities are the fundamental cause of many attacks. Even with rapid vulnerability patching, the problem is more complicated than it looks. One reason is that instances of the same vulnerability may exist in multiple software copies that are difficult to track in real life (e.g., different versions of libraries and applications). This calls for tools that can automatically search for vulnerable software with respect to a given vulnerability. In this paper, we move a step forward in this direction by presenting *Vulnerability Pecker* (VulPecker), a system for automatically detecting whether a piece of software *source code* contains a given vulnerability or not. The key insight underlying VulPecker is to leverage (i) a set of features that we define to characterize patches, and (ii) code-similarity algorithms that have been proposed for various purposes, while noting that no single code-similarity algorithm is effective for all kinds of vulnerabilities. Experiments show that VulPecker detects 40 vulnerabilities that are not published in the *National Vulnerability Database* (NVD). Among these vulnerabilities, 18 are not known for their existence and have yet to be confirmed by vendors at the time of writing (these vulnerabilities are "anonymized" in the present paper for ethical reasons), and the other 22 vulnerabilities have been "silently" patched by the vendors in the later releases of the vulnerable products.

CCS Concepts

•Security and privacy → Software security engineering;

Keywords

Vulnerability detection, code similarity, vulnerability signature

*Corresponding author.

ACSAC '16, December 05-09, 2016, Los Angeles, CA, USA

DOI: http://dx.doi.org/10.1145/2991079.2991102

1. INTRODUCTION

It is difficult to patch all vulnerabilities in software systems because of code reuse, namely that a vulnerability may exist silently in multiple software programs without being adequately tracked. This is a manifestation of what may be called the *vulnerability prevalence* problem. Examples are abundant: the same webpage rendering engine is used by Safari and Chrome, the same Flash libraries are used by Adobe Reader and Adobe Air, and Adobe Reader may be included in printer drivers [22]. It is also not surprising to see unpatched code clones in operating systems [13].

The vulnerability prevalence problem cannot be solved simply by using multiple patch management mechanisms, because they often do not cover all vulnerability instances. While it may sound simple to track code reuse, it is actually unmanageable because of the large number of programs. This can be witnessed by the fact that despite the presence of 13 automated patching mechanisms, at least 86% median fraction of computers are not patched at the time exploits are available [22].

One solution to the vulnerability prevalence problem is to automatically identify all vulnerable executables in a computer. This turns out to be difficult. In this paper, we address an alternate problem:

> *Source code vulnerability detection problem*: Given a vulnerability and the source code of a target program, how can we automatically detect whether the program contains the vulnerability or not? If it does, what is the location of the piece of vulnerable code?

Our contributions. We address the *source code vulnerability detection* problem by presenting *Vulnerability Pecker* (VulPecker), a system for automatically detecting whether a program contains a given vulnerability or not. Such a solution needs to deal with two challenges. First, there is no readily available dataset for this kind of research. This is so despite that vulnerability-related databases, such as the *National Vulnerability Database* (NVD) [2] and *Open Sourced Vulnerability Database* (OSVDB) [3], have become publicly available. Though several studies [21, 24] have built databases for mapping *Common Vulnerabilities and Exposures number or identifiers* (CVE-IDs) to commits in software revision, where each commit contains a diff between the source code prior to the commit and the source code after the commit, these databases are also insufficient for our

purpose. Second, there is no single code-similarity algorithm that is effective for all kinds of vulnerabilities. For example, ReDeBug [13] can quickly find unpatched code clones at the OS-distribution scale of code bases, but can hardly be applied to code clones that involve variable name modifications, line additions or deletions, etc. Moreover, each vulnerability has its own characteristics that should be taken into consideration. For example, if the code representation of a code-similarity algorithm cannot distinguish a piece of vulnerable code from the patched piece of the corresponding code, the algorithm is not appropriate for dealing with this vulnerability because it can cause a false-positive. However, it is not known which code-similarity algorithms would be effective for which vulnerabilities.

We address the first challenge by building a *Vulnerability Patch Database* (VPD) and a *Vulnerability Code Instance Database* (VCID), which correspond to the C/C++ open source products that have some vulnerabilities according to the NVD. The VPD contains 19 products with 1,761 vulnerabilities related to 3,454 diff hunks, and the VCID contains 455 code reuse instances of vulnerability diff hunks. We have made the VPD and the VCID publicly available at https://github.com/vulpecker/Vulpecker. We plan to maintain the two databases to accommodate the new vulnerabilities published in the future. When used together with the NVD, a query with a CVE-ID allows one to get information about the patch and a number of code reuse instances.

We address the second challenge by designing algorithms to automatically select the code-similarity algorithm(s) that is effective for one specific vulnerability. Our contribution lies in the definition and use of vulnerability diff hunk features, especially the ones that are derived from vulnerability patches. These features allow us to systematically evaluate which code-similarity algorithms are effective for which vulnerabilities. We consider a number of code-similarity algorithms proposed in the literature, as well as the variants that we devise for the purpose of the present study.

We conduct experiments to systematically evaluate the effectiveness of VulPecker. Experiments show that it detects 40 vulnerabilities that are not published in the NVD. Among these vulnerabilities, 18 are not known for their existence and have yet to be confirmed by vendors at the time of writing[1], while the other 22 vulnerabilities have been "silently" patched by vendors in later releases of the affected products.

The remainder of the paper is organized as follows. Section 2 reviews the related work. Section 3 presents the design of VulPecker. Section 4 discusses the implementation of VulPecker. Section 5 describes our experimental results. Section 6 discusses the limitation of the present study. Section 7 concludes the present paper with a discussion on future research directions.

2. RELATED WORK

There are two approaches for vulnerability detection: using vulnerability patterns or using code similarity. The pattern-based detection approach typically requires multiple instances of the same or similar vulnerability before a pattern can be identified [9, 28]. Its usefulness is therefore limited. The code-similarity based detection approach only requires a single instance of vulnerability. It is based on the intuition that similar programs may contain the same vulnerability [23], which explains why it has been used for detecting software cloning/plagiarism [5, 26]. In what follows, we review the code-similarity based detection approach.

Code-similarity comparison algorithms can be characterized by three attributes: *code-fragment level*, *code representation*, and *comparison method*. The first two attributes describe the representation of a piece of code in an abstract way, while the third attribute describes how to compare the similarity between two pieces of code according to their representations. However, it is not known what code-fragment level and/or code representation would be effective for vulnerability detection.

Code-fragment level. A *code fragment* is the unit at which programs are compared. This means that for code-similarity comparison, code needs to be abstracted at a certain level of granularity. Five levels of code fragments have been proposed: patch-without-context, slice, patch-with-context, function, and file/component. At the *patch-without-context* fragment level, a fragment is obtained from the diff file by extracting the continuous lines prefixed by the "-" symbol (indicating the lines that should be patched). This granularity has been used for bug detection [18]. At the *slice* fragment level, a program is sliced based on its *Program Dependence Graphs* (PDG). Since slicing typically preserves the structure of PDG, the isomorphism between subgraphs indicates code similarity. This granularity has been used for clone detection [15]. At the *patch-with-context* fragment level, a fragment is obtained from the diff file by extracting the lines prefixed by the "-" symbol and the lines with no prefix. This granularity has been used for bug detection [19] and vulnerability detection [13, 17, 25]. At the *function* fragment level, a function is treated as an independent unit. This granularity has been used for vulnerability extrapolation [29, 30] and clone detection [12]. At the *file/component* fragment level, each file/component is treated as a unit. This coarse-grained granularity has been mainly used for vulnerability prediction [8, 23].

Code representation. Each code fragment can be represented via text, metric, token, tree, and graph. The *text*-based representation accommodates little syntactic or semantic information, and therefore is not appropriate for vulnerability detection. In the *metric*-based representation, a fragment is represented by a vector of features, which are then compared with each other [8, 23]. This representation is often used at the file/component fragment level. In the *token*-based representation, the source code is transformed into a sequence of "tokens" via compiler-style lexical analysis, and then the sequence is scanned for certain tokens. A token may correspond to a line of code [13, 17] or a component in a line [19]. This representation is often used for clone detection [12, 14], bug detection [19], vulnerability detection [13, 17], and vulnerability extrapolation [29, 30]. In the *tree*-based representation, a tree represents the syntactic structure of variables, constants, function calls, and other tokens in source code. This syntactic representation has been used for clone detection [16], vulnerability detection [25], and vulnerability extrapolation [30]. In the *graph*-based representation, a function is represented as a graph, where a node represents an expression or statement, and an edge represents control flow, control dependency or data dependency. This semantic representation has been used for clone detec-

[1]For ethical reasons, we do not give the detailed information about these vulnerabilities, but we can release the information to academic researchers for experimental repeatability.

tion [15, 20], bug detection [18], and vulnerability detection [25].

Comparison method. There are two kinds of comparison methods: vector comparison and approximate/exact matching. The *vector comparison* method first converts the representation of a vulnerability and the representation of a target program into vectors, and then compares these vectors for detecting vulnerabilities [12, 25, 29, 30]. The *approximate/exact matching* method searches the representation of a vulnerability in the code representation of a target program via containment [13, 17, 19], substring matching [16], full subgraph isomorphism matching [18], or approximate γ-isomorphism matching [20].

Finally, it is worth mentioning that *exploit signatures* are sometimes called vulnerability signatures, although they are actually used to recognize exploits [4, 6]. These signatures characterize the inputs that can be used to exploit vulnerabilities [6]. Exploit signatures are orthogonal to the vulnerability signatures we study in the present paper.

3. DESIGN

3.1 Overview

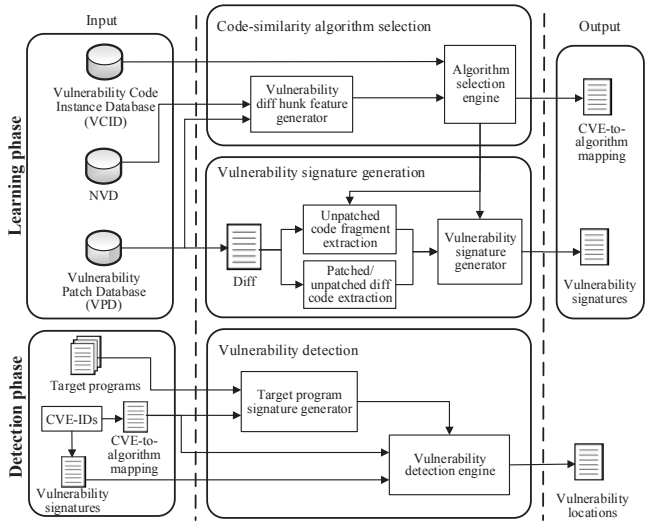


Figure 1: Overview of VulPecker: The learning phase selects code-similarity algorithm(s) that is effective for a vulnerability. The select algorithms in turn guide the generation of vulnerability signatures and the detection of vulnerabilities.

Figure 1 gives an overview of VulPecker. It has two phases: a learning phase and a detection phase. Before we elaborate on the modules described in Figure 1, let us discuss two high-level issues. First, which code-similarity algorithm(s) is effective for detecting which vulnerability? In answering this question, we analyze a set of candidate code-similarity algorithms by taking advantage of features describing vulnerabilities and patches. This analysis leads to a *CVE-to-algorithm mapping*, which maps a CVE-ID to the select code-similarity algorithm(s) that is effective for detecting the vulnerability.

Second, how should we generate and use vulnerability signatures? Recall that a code-similarity algorithm can be characterized by three attributes: *code-fragment level*, *code representation*, and *comparison method*. We observe that *code-fragment level* and *code representation* offer guidance for generating signatures of vulnerabilities as well as signatures of target programs. These signatures are then compared with each other for determining whether or not a target program has a vulnerability. If a vulnerability is found, the location of the vulnerable code fragment(s) is reported.

3.2 Defining vulnerability and code-reuse features

Since our focus is to use code similarity analysis to detect vulnerabilities, we need to define features to characterize vulnerabilities and code reuses.

Features for describing vulnerability diff hunks. Given a vulnerability and its patch, the vulnerability can be characterized by the vulnerability *diff*, which is composed of one or multiple *diff hunks*. Each diff hunk contains a sequence of lines of code, where each line is prefixed by a "+" symbol (addition), "-" symbol (deletion), or nothing. In order to define features to describe a vulnerability, it is sufficient to define features that describe these diff hunks. For a diff hunk, we define two sets of features: *basic features* and *patch features*. Table 1 summarizes these features. Basic features are the Type 1 features described in Table 1, including the unique CVE-ID, the *Common Weakness Enumeration Identifier* (CWE-ID) that represents the vulnerability type, product vendor, product affected, and vulnerability severity. Patch features are the Type 2–Type 6 features described in Table 1 and describe the code changes from the unpatched piece of code to the patched one. The five types of patch features are elaborated as follows.

- Non-substantive features: These features describe changes in whitespace, format or comment which have no impact on useful code.
- Component features: These features describe the changes of components in statements such as variables, operators, constants, and functions.
- Expression features: These features describe the changes of expressions in statements such as assignment expression, *if* condition, and *for* condition.
- Statement features: These features describe the changes of statements involving addition, deletion, and movement.
- Function-level features: These features describe the changes of functions or changes outside a function, such as macros and global variable definitions.

Features for describing code reuses. The term "code reuse" often means code cloning [5, 26], including exact clones, renamed clones, near miss clones, and semantic clones. For vulnerability detection, we are given a piece of code containing a vulnerability and a target piece of code that may or may not contain the same vulnerability, where the latter may or may not be an exact clone of the former. Note that we have already defined five types of patch features for describing vulnerabilities, namely Type 2–Type 6 described in Table 1. Since these features can already describe the "transformation" from an unpatched piece of vulnerable code to a patched piece of code, which may be seen as a sort of code reuse in a sense, we can naturally use these patch features to describe code reuses.

3.3 Preparing the input

Table 1: Vulnerability diff hunk features = basic features (Type 1) + patch features (Type 2–Type 6). The patch features are also used as code-reuse features.

Type	Description
1	Basic features
1-1	CVE-ID
1-2	CWE-ID
1-3	Product vendor
1-4	Product affected
1-5	Vulnerability severity
2	Non-substantive features
2-1	Changes in whitespace, format or comment
3	Component features
3-1	Variable name modification
3-2	Constant modification
3-3	Variable type modification
3-4	Function name modification
3-5	Function argument addition
3-6	Function argument deletion
3-7	Function argument modification
3-8	Variable declaration addition
3-9	Variable declaration deletion
3-10	Operator modification
4	Expression features
4-1	Assignment expression modification
4-2	*if* condition modification
4-3	*for* condition modification
4-4	*while* condition modification
4-5	*do while* condition modification
4-6	*switch* condition modification
5	Statement features
5-1	Line addition
5-2	Line deletion
5-3	Line movement
6	Function-level features
6-1	Entire function addition
6-2	Entire function deletion
6-3	Modification beyond the function

In the learning phase, there are three inputs: the NVD, VPD, and VCID. The NVD is a public database containing the basic information of vulnerabilities that can be uniquely identified by CVE-IDs. However, we need to build the VPD and VCID by ourselves. The VPD contains the mappings from CVE-IDs to diffs. Any vulnerability described in the NVD with an explicit vulnerability diff description is incorporated into the VPD. However, the NVD contains vulnerabilities that are caused by code reuse, but does not give any explicit diff description. For example, CVE-2015-0239 in the NVD states that a Linux kernel prior to version 3.18.5 contains a vulnerability in the `em_sysenter` function, and the NVD further gives a diff description of the vulnerability. However, the vulnerable pieces of code in these vulnerable kernels are not identical. More specifically, the vulnerable piece of code in kernel 3.18.1 indeed matches the vulnerable piece of code corresponding to the diff, but the vulnerable pieces of code in kernels 3.16.3 and 3.10.3 are not the exact clones of the vulnerable piece of code corresponding to the diff, where the latter two are actually different from each other as well. In this case, the vulnerable pieces of code in kernels 3.16.3 and 3.10.3 are treated as different code reuse instances, and are incorporated into the VCID with each described by its respective code-reuse features. These databases collectively allow us to achieve the following. Given a CVE-ID, we can extract the basic features of a vulnerability from the NVD, the patch features from the VPD, and the code reuse instances of the vulnerability from the VCID.

In the detection phase, the inputs are CVE-IDs, the target programs of interest, the *CVE-to-algorithm mapping*, and *vulnerability signatures*, where the last two are the output of the learning phase.

3.4 Code-similarity algorithm selection

Extracting vulnerability diff hunk features. For each vulnerability diff hunk, we extract the basic features of a vulnerability from the NVD. From the VPD, we obtain the diff corresponding to the vulnerability in question. After splitting the diff into possibly multiple diff hunks, we can extract the patch features as follows. First, non-substantive features and function level features can be extracted directly from the diff hunks. Second, the three other types of patch features (i.e., Types 3–Type 5 in Table 1), can be extracted via a sequence of edit actions from the unpatched piece of code to the patched one using, for example, the gumtree algorithm [11]. Finally, the feature vector of a vulnerability diff hunk is composed from the basic features and patch features.

Code-similarity algorithm selection engine. This module is to determine which code-similarity algorithm(s) is effective with respect to which vulnerability, where "effective" means that the higher the similarity between a piece of a target program and a piece of vulnerability code, the higher the chance the target program contains the vulnerability in question. For this purpose, we propose Algorithm 1, which has three steps as highlighted in Figure 2 and is elaborated below.

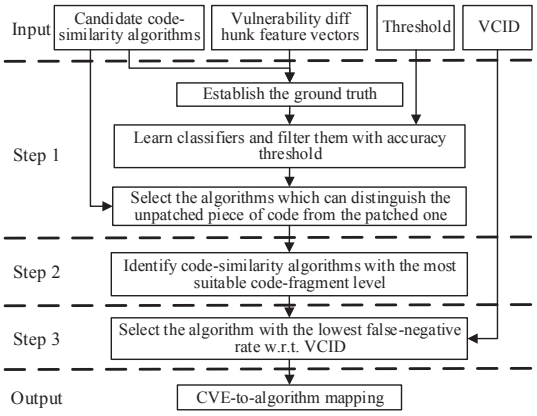


Figure 2: Illustration of Algorithm 1

First, we observe that a good algorithm should be able to distinguish the unpatched piece of code from the patched one. In order to evaluate whether or not a candidate algorithm can achieve this goal, we use the vulnerability diff hunks in the VPD to establish the ground truth. For each vulnerability diff hunk described by a feature vector $F_k \in F$ as described in Table 1 and each candidate code-similarity

Algorithm 1 Code-similarity algorithm selection

Input: A set of candidate code-similarity algorithms $S = \{s_1, \ldots, s_n\}$; a set of vulnerability diff hunk feature vectors $F = \{F_k\}_k$, where $F_k = (f_{k,1}, \ldots, f_{k,m}, f_{k,m+1}, \ldots, f_{k,m+n})$ and m is the number of features described in Table 1; the VCID; threshold τ

Output: CVE-to-algorithm mapping $\mathcal{M} = \{(F_k, s)\}_k$ where $s \in S$ is the most suitable code-similarity algorithm for diff hunk F_k

1: $S' \leftarrow \emptyset$
2: **for** each $F_k \in F$ **do**
3: $S'(F_k) \leftarrow \emptyset$
4: **for** each $s_i \in S$ **do**
5: set $f_{k,m+i} \leftarrow 1$ if s_i treats the patched piece of code corresponding to F_k as invulnerable (i.e., *not* similar to the unpatched piece of code) and $f_{k,m+i} \leftarrow 0$ otherwise
6: **end for**
7: **end for**
8: partition F horizontally into $F^{(1)}$ and $F^{(2)}$
9: **for** each $s_i \in S$ **do**
10: consider $F^{(1)}(i) = \{(f_{k,1}, \ldots, f_{k,m}, f_{k,m+i})\}_k$ as the projection of $F^{(1)}$ w.r.t. s_i
11: use machine learning to learn a classifier from $F^{(1)}(i)$ and denote the classifier's accuracy by a_i
12: **if** $a_i \geq \tau$ **then**
13: $S' \leftarrow S' \cup \{s_i\}$
14: **end if**
15: **end for**
16: **for** each $F_k = (f_{k,1}, \ldots, f_{k,m}, f_{k,m+i}) \in F^{(2)}$ **do**
17: $S'(F_k) \leftarrow S'$
18: **for** each $s_i \in S'$ **do**
19: **if** $f_{k,m+i} = 0$ **then**
20: $S'(F_k) \leftarrow S'(F_k) \setminus \{s_i\}$
21: **end if**
22: **end for**
23: **end for** {a pair $(F_k, S'(F_k))$ for each $F_k \in F^{(2)}$}
24: **for** each $F_k \in F^{(2)}$ **do**
25: **for** each $s_i \in S'(F_k)$ **do**
26: determine the most suitable code-fragment level for F_k as the one that leads to the smallest doa as described in text
27: **if** the fragment level used by s_i does not match the most suitable code-fragment level for F_k **then**
28: $S'(F_k) \leftarrow S'(F_k) \setminus \{s_i\}$
29: **end if**
30: **end for**
31: **end for** {a pair $(F_k, S'(F_k))$ for each $F_k \in F^{(2)}$}
32: $\mathcal{M} \leftarrow \emptyset$
33: **for** each $s_i \in S'(F_k)$ **do**
34: use the code reuse instances in VCID that are suitable for s_i (see text for details) to evaluate the false-negative rate of s_i, denoted by $\mathsf{fn}(s_i)$
35: **end for**
36: **for** each each $F_k \in F^{(2)}$ with associated $(F_k, S'(F_k))$ **do**
37: $s \leftarrow \{s_i : s_i \in S'(F_k) \wedge \mathsf{fn}(s) = \min_{s' \in S'(F_k)}\{\mathsf{fn}(s')\}\}$
38: $\mathcal{M} \leftarrow \mathcal{M} \cup \{(F_k, s)\}$
39: **end for**
40: **return** $\mathcal{M}$

algorithm s_i, we obtain the result on whether or not s_i treats the patched code as vulnerable (i.e., treating the vulnerable piece of code and the corresponding patched piece of code as similar or not). The code-similarity algorithm result is recorded as a class label $f_{k,m+i}$ (Lines 2-7). We partition horizontally $F = \{F_k\}_k$ into $F^{(1)}$ and $F^{(2)}$ so that $F^{(1)}$ will be used to learn a classifier and $F^{(2)}$ will be further used to select code-similarity algorithms. For each s_i, we use the projection of $F^{(1)}$ on the diff hunk features and the label of s_i, namely $F^{(1)}(i) = \{(f_{k,1}, \ldots, f_{k,m}, f_{k,m+i})\}_k$ to learn a classifier. For a learned classifier with respect to code-similarity algorithm s_i, its *accuracy*, denoted by a_i, is defined as $a_i = v/u$, where u is the total number of testing samples and v is the number of testing samples that are correctly classified by the classifier. For a given threshold τ, if $a_i \geq \tau$, namely that the classifier is accurate enough, then s_i is added to the set S' of code-similarity algorithms that will be considered further (Lines 9-15). For a candidate code-similarity algorithm $s_i \in S'$, if the classifier treats the patched piece of code corresponding to $F_k \in F^{(2)}$ as invulnerable, then s_i will be considered further. Otherwise, s_i is eliminated because it cannot detect the vulnerability correctly or it is unsuitable for F_k (e.g., a classifier using the code-fragment level of patch-without-context cannot be applied to a diff hunk involving no line deletion). This screening of candidate code-similarity algorithms corresponds to Lines 16-23.

Second, we identify the most suitable code-fragment level for a diff hunk F_k. This corresponds to Lines 24-31 in Algorithm 1. For this purpose, we introduce the concept of *core code fragment*, which is the piece of code that is directly related to the vulnerability in question. For example, consider *strcpy*$(dest, src)$, which is used to copy a string from one address to another. If the boundaries of *src* and *dest* are not checked, one can encounter a buffer overflow. In this example, the core code fragment only contains the statement *strcpy*$(dest, src)$ and the preceding statements that involve the operation of the arguments *dest* and *src*.

In order to search for the most suitable code-fragment level, we represent an unpatched piece of code via all of the code-fragment levels mentioned above, namely patch-without-context, slice, patch-with-context, function, and file/component with increasingly coarse granularity. Among these code-fragment levels, we observe that the slice code-fragment level can be naturally used to represent the core code fragment of a vulnerability. It would be ideal to ask a human expert to localize the precise position of a vulnerability segment, but this approach is too costly. As an alternative, we automatically treat the lines that are deleted by the patch as the location of the vulnerability; this approximation is also used by [24]. Having approximated the location of a vulnerability, we treat a slice as an approximation of a core code fragment.

Since an unpatched code fragment at a finer code-fragment level is contained in an unpatched code fragment at a coarser code-fragment level, we use the difference between the numbers of lines of code at two different code-fragment levels to indicate the difference between the two representations of the same vulnerable piece of code at the two different code-fragment levels. This leads to the *degree of approximation* metric, or **doa** for short, which measures the degree of approximation between the core code fragment cc corresponding to the representation of an unpatched piece of code at the slice code-fragment level and the unpatched code fragment cf corresponding to the representation of the same unpatched piece of code at one of the five code-fragment levels mentioned above. In principle, we have

$$\mathbf{doa}(cc, cf) = \frac{|\ell_{cf} - \ell_{cc}|}{\ell_{cc}} \quad (1)$$

where ℓ_{cc} and ℓ_{cf} denote the number of lines of core code fragment cc and code fragment cf, respectively. The closer the $\mathbf{doa}(cc, cf)$ is to 0, then the smaller the difference is between cc and cf, and the better the cf is at representing the vulnerability. Note that as we elaborate below, we can identify the minimum $\mathbf{doa}(cc, cf)$ without computing ℓ_{cc}, whose actual value cannot be calculated precisely because

of the use of approximate code slice.

Since we use the slice code fragment to represent the core code fragment, the code-fragment level of the core code is finer than the code-fragment level of patch-with-context (pc) but coarser than the fragment level of patch-without-context (pwc). The code-fragment level of slice sc can be little finer or coarser than the code-fragment level of the core code because the lines of code deleted by the patch are considered as lines of vulnerable code. The code-fragment level of slice sc is finer than the code-fragment level of patch-with-context (pc), and coarser than the code-fragment level of patch-without-context (pwc). Therefore, $\mathbf{doa}(cc, sc) < \mathbf{doa}(cc, pc)$ and $\mathbf{doa}(cc, sc) < \mathbf{doa}(cc, pwc)$. For a diff hunk F_k, if the slice code fragment can be extracted (i.e., the diff hunk having lines of code that are prefixed by the "-" symbol), the slice code-fragment level has the smallest **doa**. Otherwise, the code-fragment level of patch-with-context has the smallest **doa** because the representation at the code-fragment level of patch-without-context cannot be extracted from the unpatched piece of code corresponding to the diff hunk in question. The code-fragment level with the smallest **doa** is the most suitable because the unpatched piece of code is represented best at this code-fragment level.

Third, we use the VCID to further evaluate the false-negative rate of s_i that passes the previous rounds of screening, where $s_i \in S'(F_k)$ at the end of executing line 31 in Algorithm 1. For each candidate code-similarity algorithm s_i, we exclude the code reuse instances from the VCID according to the following two conditions. First, a code reuse instance in the VCID cannot be applied to s_i. For example, for a diff hunk with no "-" prefix symbol, a code-similarity algorithm that uses the code-fragment level of patch-without-context cannot be used to recognize the code reuse instances for this diff hunk [18]. This means that the code reuse instance cannot be applied to s_i and therefore should be eliminated. Second, if two code reuse instances have the same code-reuse features and are both detected by s_i as either vulnerable or invulnerable at the same time (i.e., s_i cannot tell them apart), one of them is eliminated. After filtering the code reuse instances that are unsuitable for s_i, we use the suitable code reuse instances in the VCID to evaluate the false-negative rate of s_i. If a code reuse instance of a vulnerable piece of code in the VCID is not detected as vulnerable, a false-negative occurs. The false-negative rate is defined $\mathsf{fn} = r/t$, where t is the the total number of code reuse instances that are suitable for s_i, and r is the number of code reuse instances that caused false-negatives. For a specific F_k, the code-similarity algorithm with the lowest false-negative rate is selected. This corresponds to Lines 33-39 in Algorithm 1.

3.5 Vulnerability signature generation

The vulnerability signature can be generated in two steps. First, we extract the patched/unpatched diff code and the unpatched code fragment corresponding to a vulnerability. The patched/unpatched diff code can be directly extracted from the diffs according to the VPD. Specifically, we can obtain the unpatched diff code by extracting the lines prefixed by a "-" symbol and the lines with no prefix, and the patched diff code by extracting the lines prefixed by a "+" symbol and the lines with no prefix. We can extract the unpatched code fragment from the source code of the vulnerable software according to each diff hunk and the code-fragment level used by the code-similarity algorithm that has been selected for the diff hunk.

Second, for each diff hunk, we preprocess and represent the patched/unpatched diff code and unpatched code fragments obtained at the previous step. The preprocessing usually involves whitespace, format, and comment processing. Note that it is possible that the code statements given in the diff hunk may be incomplete (e.g., missing the "}" at the end of an `for` structure), the missing part needs to be extracted from the unpatched piece of code. This operation is necessary for tree-based or graph-based code-similarity algorithms. Then, the preprocessed version of the patched/unpatched diff code and the unpatched code fragment are represented according to the code representation used by the code-similarity algorithm selected for the diff hunk in question. The results are the vulnerability signatures which will be used in the vulnerability detection stage.

3.6 Vulnerability detection

For a given diff hunk, the CVE-to-algorithm mapping already gives information on the code-similarity algorithm as well as its code-fragment level, code representation, and comparison method. Given a diff hunk and a target program, after the preprocessing of target programs involving whitespace, format, and comment processing, the target program signature can be generated by the code representation used by the code-similarity algorithm selected for the diff hunk. The vulnerability detection engine uses the comparison method of the code-similarity algorithm selected for the diff hunk to search for the vulnerability signature from the target program signatures. If the vulnerability signature is found, the location of the vulnerability in the target program is reported.

4. IMPLEMENTATION

4.1 Creating the VPD and VCID

Selecting product vulnerabilities. For the purpose of the present study, we need to select product vulnerabilities according to the following three constraints. First, a product should be open source because VulPecker analyzes source code. Second, a product should be programmed in C/C++ because most code-similarity algorithms we adopt or adapt deal with products written in C/C++. Third, a product should have a series of versions and contain many publicly disclosed vulnerabilities (e.g. at least 10 vulnerabilities) with the corresponding diffs. This constraint eases the process of selecting diffs according to the regularity of patches.

The NVD was collected in Dec. 2015, containing 73,510 vulnerabilities for 30,432 products. A screening based on the constraints mentioned above leads to 19 products: Linux kernel, Firefox, Thunderbird, Seamonkey, Fixfox_esr, Thunderbird_esr, Wireshark, Ffmpeg, Apache Http Server, Xen, OpenSSL, Qemu, Libav, Asterisk, Cups, Freetype, Gnutls, Libvirt, and VLC media player. Table 2 shows how the constraints screen the 30,432 products into 19, from which we build a VPD of 1,761 vulnerabilities with 3,454 diff hunks and a VCID of 455 code reuse instances. This means that many diff hunks do not have code reuse instances. The creation of the VPD and VCID is elaborated below.

Creating the VPD. For each select product, we search for CVE-IDs. For each CVE-ID, an important step is to

Table 2: Screening of vulnerable products

Step	Filter condition	#Products	# Vulnerabilities
1	NVD (Dec. 2015)	30,432	73,510
2	# Vulnerabilities $\geq$ 10	1,046	40,604
3	C/C++ Open source w/ versioning	62	7,168
4	Diff availability	19	1,761

identify the correct patch link from many reference links given in the NVD. The vulnerabilities of a product that has its patch submission and release platform (e.g., a defect tracking system or version control system) are often well documented in the NVD. For these products, we create a crawler to automatically identify the patch link corresponding to a CVE-ID. However, we find that multiple patch links may correspond to a single vulnerability, in which case we need to determine whether or not the diff from a patch link is specifically for the CVE-ID in question. To resolve this issue, we use the following two heuristics. First, a diff is incorporated if the names of the vulnerable files and functions from the text summary corresponding to the CVE-ID in question are the same as the names of the patched files and functions. This is because the vulnerable file and function are exactly the ones that should be patched in most cases. Second, a diff is incorporated if the corresponding webpage of the patch link mentions the CVE-ID in a place such as the title or commit message.

In order to check the accuracy of the heuristics we use, we take 10% of the select diffs as a random sample and manually examine them. We find that every instance in the sample is correct. However, the preceding heuristics may miss some mappings from the CVE-IDs to the diffs. In order to obtain more comprehensive mappings, we manually check the CVE-IDs that do not have diffs according to the heuristics mentioned above. This manual analysis is tedious, but we have no alternatives. A manual analysis is often encountered in a scenario that violates both of the heuristics mentioned above.

Creating the VCID. For the vulnerabilities incorporated into the VPD, we collect their code reuse instances at the function level from the different releases of the same product or from different product. Note that the vulnerable product releases listed in the text summary of CVE-ID are usually described as a scope. However, it can happen that some releases within the scope are actually patched already. For example, CVE-2011-1170 states that "Linux kernels before 2.6.39" are vulnerable, but Linux kernel 2.6.38.3 is already patched. We use the following heuristics to exclude the releases that have been patched. Let α and β respectively be the number of lines of the unpatched code and the number of lines of the patched code according to the diff hunk in question, and γ be the number of matched lines of code in a product release. If $\gamma/\alpha > \gamma/\beta > 0.8$, where 0.8 is the given threshold of code similarity, a function is treated as a code reuse instance.

After obtaining the code reuse instances, we add the code-reuse features to the VCID. Since code reuse instances in the VCID focus on code reuses within functions, the code-reuse features are Type 2–Type 5 described in Table 1. We use the gumtree algorithm [11] to obtain the sequence of edit actions from an unpatched piece of code to its corresponding piece of code in the code reuse instance. We derive code-reuse features from the sequence of edit actions.

4.2 Code-similarity algorithm selection

We now discuss the instantiations of Algorithm 1 from two aspects.

Candidate code-similarity algorithms. Table 3 lists the candidate code-similarity algorithms, some of which are variants of the algorithms reviewed in Section 2. Here we highlight the following issues. First, we exclude the algorithms that operate at the file/component fragment level, because they are usually used for vulnerability prediction and may not be applicable when vulnerabilities do not appear in a high frequency. Second, for the code-similarity algorithms that do not utilize the concept of code-fragment level, we select the code-fragment level with the smallest **doa**. Third, we consider the code representation of the CP-Miner algorithm [19] with different mappings of variable name and constants for the sake of comprehensiveness. The CP-Miner algorithm maps variables with different data types to the same token, and maps all constants to another token (Token-component-1). The following three variants are also considered: variables of different data types are always mapped to the same token, and constants are not mapped (Token-component-2); variables of the same data type are mapped to the same token, and all constants are mapped to another token (Token-component-3); variables of the same data type are mapped to the same token, and constants are not mapped (Token-component-4). In Table 3, "Token-component-{1, 2, 3, 4}" respectively indicates the four mappings mentioned above. Fourth, existing tree-based or graph-based code-similarity algorithms usually do not deal with the mapping of statement components, and therefore cannot cope with code reuses with identifier renaming. To resolve this issue, we add six hybrid algorithms that incorporate tree/graph-based code-similarity algorithms and four Token-component mappings mentioned above. For some graph-based code-similarity algorithms, the *Abstract Syntax Tree* (AST) is also extracted to attain the token-component mapping.

Using *Support Vector Machines* (SVM) for classifier learning. SVM is a popular supervised machine learning method. We use the open-source tool LibSVM [7] for our purpose. We first convert the vulnerability diff hunk features into numeric data and normalize each attribute to the range [0, 1] while treating all attribute values as non-negative. We take 70% vulnerabilities in each product according to the VPD (i.e. $F^{(1)}$) to learn classifiers. The classifier aims to distinguish the patched piece of code from the unpatched piece of code with respect to a same diff hunk. We use the RBF kernel, which maps the feature vectors to a high-dimensional space for handling the nonlinear relation between the class labels and the attributes. We perform a 10-fold cross-validation on $F^{(1)}$ to pick the best values corresponding to the penalty and kernel parameters.

To explain the decision of the classifier, we adopt the leave-one-out method to discover the important features. That is to say, each time we choose one of the vulnerability diff hunk features, set the value of the feature across the entire testing data to be the same, and obtain the accuracy of the classifier when apply to the modified test set. After repeating this process for each feature, we compare the resulting accuracies corresponding to the modified test data with the accuracies corresponding to the original test set. This allows us to determine which features make bigger contributions to the classifier.

Table 3: Candidate code-similarity algorithms

No.	Code-fragment level	Code representation	Comparison method	Application	References
1	Function	Token-frequency	Range-queries of metric tree	Vulnerability detection	[12]
2		Token-API node	PCA and cosine similarity of vectors	Vulnerability extrapolation	[29]
3		API-subtree		Vulnerability extrapolation	[30]
4	Patch-with-context	Token-line	Containment	Vulnerability detection	[13, 17]
5		Token-component-1		Bug detection	[19]
6		Token-component-{2,3,4}		Bug detection	Variants of [19]
7		AST-suffix	Substring matching	Clone detection	[16]
8		AST-suffix+Token-component-{1,2,3,4}		Clone detection	Variants of [16]
9		xAST	Manhattan distance of vectors	Vulnerability detection	[25]
10		xAST+Token-component-{1,2,3,4}		Vulnerability detection	Variants of [25]
11		xGRUM		Vulnerability detection	[25]
12		xGRUM+Token-component-{1,2,3,4}		Vulnerability detection	Variants of [25]
13		GPLAG	γ-isomorphic	Clone detection	[20]
14		GPLAG+AST+Token-component-{1,2,3,4}		Clone detection	Variants of [20]
15	Patch-without-context	PDG	Subgraph-isomorphic	Bug detection	[18]
16		PDG+AST+Token-component-{1,2,3,4}		Bug detection	Variants of [18]
17	Slice	PDG-slicing		Clone detection	[15]
18		PDG-slicing+AST+Token-component-{1,2,3,4}		Clone detection	Variants of [15]

4.3 Vulnerability signature generation

In this subsection, we use CVE-2015-0834, which is a vulnerability in Mozilla Firefox prior to version 36.0, as an example to illustrate the implementation details of vulnerability signature generation. Figure 3 shows one diff hunk of CVE-2015-0834 obtained from the VPD. After running Algorithm 1, code-similarity algorithm No.9 is selected. Correspondingly, the code-fragment level, code representation, and comparison method are respectively the *patch-with-context*, *xAST*, and *vector comparison using Manhattan distance*.

The vulnerability signature corresponding to the diff hunk described in Figure 3 is generated as follows. First, we extract the patched/unpatched diff code based on the prefix symbols "-" and "+" in the diff. Then, we collect the unpatched functions from the diff and source code of the vulnerable product versions as follows. We obtain the vulnerable product versions from the text summary of CVE-2015-0834 in the NVD (i.e., the versions prior to version 36.0), and use a Web crawler to download the source code of these versions. From the latest vulnerable product version to the oldest, we search for the first product version that contains exactly the lines of code prefixed by the symbol "-" and the lines of code with no prefix in the diff. We extract the unpatched code fragments at the code-fragment level of patch-with-context. Then we process the patched/unpatched diff code and the unpatched code fragments involving the whitespace, format, commenting processing, and the completion of the structures and statements of code fragments. Finally, we represent the patched/unpatched diff code and the unpatched code fragments by using xASTs, which are generated from the ASTs produced by the open source tool known as Joern [28].

```
@@ -485,16 +485,19 @@ PeerConnectionImpl::ConvertRTCConfigurat
      ...
1:    if (!(isStun || isStuns || isTurn || isTurns)) {
2:      return NS_ERROR_FAILURE;
3:    }
4: +  if (isTurns || isStuns) {
5: +    continue; // TODO: Support TURNS and STUNS (Bug 1056934)
6: +  }
7:    nsAutoCString spec;
8:    rv = url->GetSpec(spec);
      ...
```

Figure 3: A diff hunk corresponding to CVE-2015-0834

4.4 Vulnerability detection

We use Thunderbird 24.8.0 as an example of target program to illustrate the vulnerability detection process with respect to CVE-2015-0834, including its diff hunk described in Figure 3. The target program signatures are generated as follows. From the CVE-to-algorithm mapping, we obtain the select code representation xAST for the diff hunk of CVE-2015-0834 described in Figure 3. After the preprocessing of target programs involving whitespace, format, and comment processing, we generate the signatures of Thunderbird 24.8.0 in terms of representation xAST. Then, we use the detection engine to detect whether Thunderbird 24.8.0 contains the vulnerability. Since the select comparison method is the vector comparison using the Manhattan distance, we convert the vulnerability signature and target program signatures into vectors. Moreover, a target program is deemed vulnerable if it contains at least a signature that is closer to the vulnerability signature than the patched code signature. The target program satisfying these two requirements, namely passing the code-similarity detection and containing a signature that is closer to the vulnerability signature, are considered vulnerable. In the case the target program is found to be vulnerable, the locations of the vulnerable piece of code in the target program are determined.

5. EXPERIMENTAL RESULTS

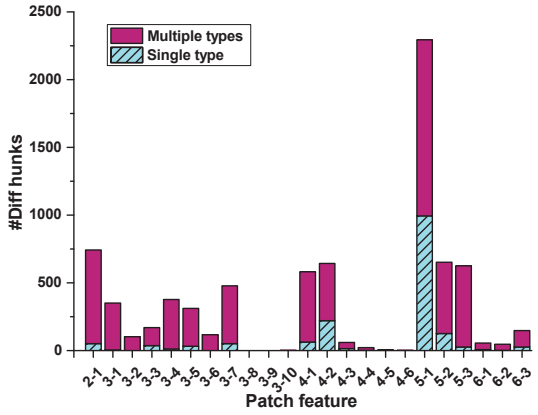
The effectiveness of code-similarity algorithms can be evaluated via standard metrics, such as *precision*, *recall*, and *F-measure* metrics [27]. Let TP be the number of true vulnerabilities detected (true-positives), FP be the number of false vulnerabilities detected (false-positives), and FN be the number of true vulnerabilities undetected (false-negatives). The metric $precision = \mathsf{TP}/(\mathsf{TP}+\mathsf{FP})$ reflects the correctness among the detected positives. The metric $recall = \mathsf{TP}/(\mathsf{TP}+\mathsf{FN})$ reflects the completeness of the detected positives. The overall detection effectiveness can be reflected by $F\text{-}measure = 2 \cdot precision \cdot recall/(precision + recall)$.

5.1 Learning the CVE-to-algorithm mapping and vulnerability signatures

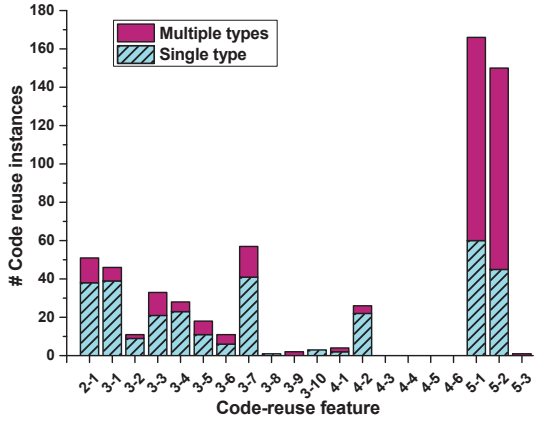
Distribution of patch features and code-reuse features. Figure 4(a) depicts the distribution of the patch features of diff hunks described in the VPD. We observe that many diff hunks have patch feature Types 2-1, 4-1, 4-2, 5-1, 5-2, and 5-3, while few diff hunks have patch feature Types 3-8, 3-9, 3-10, 4-5, and 4-6. Moreover, nearly 25% of the diff hunks have a single type of patch features, with Types

5-1, 4-2, and 5-2 being the top three types. For diff hunks with a single type of patch features, the particular type is probably the main factor in determining the classifier for selecting a particular code-similarity algorithm. For diff hunks with multiple types of patch features, one or multiple types may have contributed to the determination of the classifier for selecting a particular code-similarity algorithm. Deeper characterization on the roles played by the types of patch features is left to future investigation.

Figure 4(b) depicts the distribution of the code-reuse features obtained from the VCID. We observe that the distribution is similar to what is shown in Figure 4(a), with two exceptions. One exception is that the number of code reuse instances corresponding to code-reuse feature Types 4-1 and 5-3 is respectively smaller than its counterpart with respect to the diff hunks. The other exception is that the number of code reuse instances with a single type of code-reuse features is greater than the number of diff hunks with a single type of patch features. Since many vulnerability diff hunks in the VPD do not have code reuse instances in the VCID, we evaluate the precision and recall of code-similarity algorithms using VPD and VCID separately.



(a) Distribution of patch features



(b) Distribution of code-reuse features

Figure 4: Distribution of patch features and code-reuse features according to the VPD and VCID, respectively

Comparison among code-similarity algorithms. We took 70% of the vulnerabilities in each product according to the VPD to learn classifiers and the rest as the testing data for selecting code-similarity algorithms. When we set the accuracy threshold of classification model τ to 0.85 or 0.9, we found that the code-similarity algorithms achieving an accuracy lower than τ were algorithm No.13 as well as its variants and algorithm No.15 as well as its variants. When we set τ to 0.95, nearly half of the algorithms were selected. In order to select more algorithms with a high accuracy, we set τ to 0.9 in the following experiments. The accuracy of select SVM classifiers, as described in Algorithm 1, for existing code-similarity algorithms and their variants was 94.6% on average.

As described in Algorithm 1, if the code-similarity algorithm treats a patched piece of code obtained from the VPD as vulnerable, a false-positive occurs. If a code reuse instance obtained from the VCID is not detected as vulnerable, a false-negative occurs. Also as described in Algorithm 1, the VPD is used for evaluating false-positives, while the VCID is used for evaluating false-negatives. Therefore, for each code-similarity algorithm, we computed the precision based on the VPD, the recall based on the VCID, and the F-measure based on both the precision and the recall.

Since code-similarity algorithms typically use a similarity threshold, we need to know how to determine its value. Figure 5 shows the F-measure of code-similarity algorithm No.9 and its variants with respect to various similarity thresholds. We observe that 0.6 is a better choice than the default threshold of 0.8 recommended by code-similarity algorithm No.9 [25]. For algorithm No.9 with threshold 0.6, its variants do not achieve a higher F-measure. Indeed, threshold 0.6 leads to the highest F-measure for the four variants of algorithm No.9, namely algorithms No.10-{1, 2, 3, 4}. Therefore, we set the threshold for algorithm No.9 and its variants to 0.6. For other algorithms, we found that their thresholds leading to the highest F-measure respectively coincided with their default threshold values. For example, for code-similarity algorithm No.11 and its variants, threshold 0.8 led to the highest F-measure, which coincided with the default threshold value of algorithm No.11.

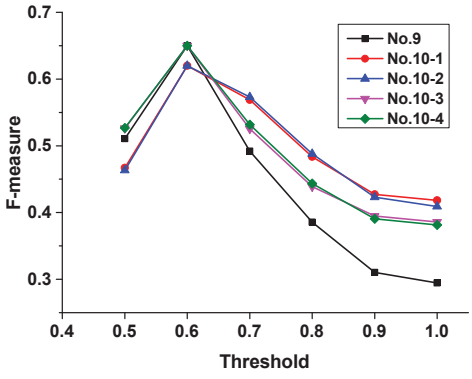


Figure 5: F-measures of code-similarity algorithm No.9 and its variants (No.10-{1,2,3,4})

Figure 6 shows the precision, recall, and F-measure of code-similarity algorithms with respect to the testing data. For the candidate code-similarity algorithms, we computed their precision and recall according to the testing data in the VPD and the code reuse instances in the VCID. The data unsuitable for a code-similarity algorithm were also included

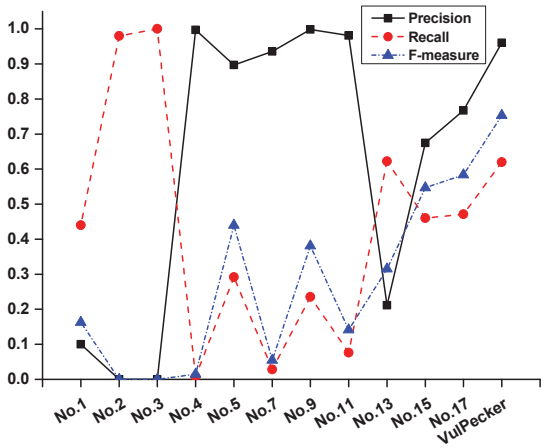
when computing the precision and recall because the algorithms dealt with all vulnerabilities. Figure 6(a) shows that the F-measure of the algorithms adopted from the literature is 0.24 on average and the best algorithm is code-similarity algorithm No.17 with a F-measure of 0.58. For the variants of code-similarity algorithms that we propose in the present paper, we consider three examples: code-similarity algorithms No.5, No.9, and No.17. As shown in Figure 6(b), algorithm No.9 and its variants (with threshold 0.6) lead to similar F-measures, and the variants of algorithm No.5 and the variants of algorithm No.17 exhibit a similar pattern in terms of their F-measures. This means that some token mappings can improve the F-measure, because the recall measure plays a more important role than the precise measure. Among all of the variants of algorithms No.5, No.9, and No.17, algorithm No.18-2 leads to the highest F-measure 0.66.

For VulPecker, the precision depends on the accuracy of the SVM classifier, and the recall is the average recall of the select algorithm for each diff hunk. The recall of the select code-similarity algorithm was calculated based on the code reuse instances in the VCID, except the instances that were ruled out by the code-similarity algorithm as described in Algorithm 1. Figure 6 shows that VulPecker has a F-measure that is 18% higher than the best existing code-similarity algorithm (algorithm No.17) and 10% higher than the best variant of the adopted code-similarity algorithms (algorithm No.18-2). VulPecker tends to select the code-similarity algorithm with a high F-measure, while no single code-similarity algorithm is suitable for all vulnerabilities.

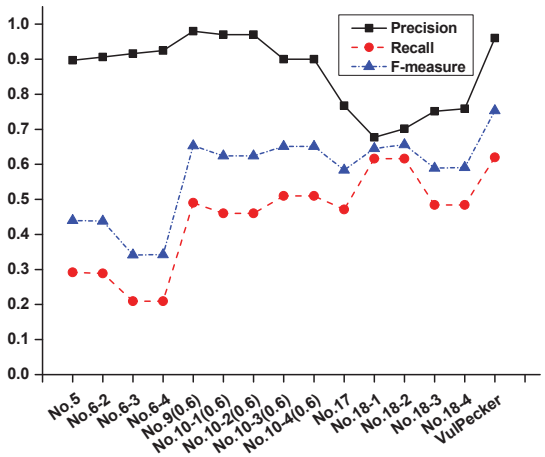
Explanation of the code-similarity algorithm selection results. We now explain the code-similarity algorithm selection results of Algorithm 1. For step 1 of Algorithm 1, VulPecker used learned SVM classifiers to select a set of code-similarity algorithms that could distinguish unpatched pieces of code from the patched ones. The result showed that no single basic feature made a significant contribution to the classifiers.

For diff hunks with a single type of patch feature, the single type probably determined the code-similarity algorithm that could distinguish between patched and unpatched pieces of code. For diff hunks with patch feature Type 5-1 involving only line addition, the set of code-similarity algorithms that could distinguish unpatched piece of code from the patched one consistently excluded the algorithms that used the code-fragment level of patch-without-context or slice, such as algorithms No.15-No.18. This was reasonable because these algorithms were based on the deleted lines according to the diff hunks. For diff hunks with patch feature Type 6-1 involving whole function addition, the algorithms depending on the AST or PDG (algorithm No.7-No.18) were all excluded because there were no unpatched functions. For diff hunks with patch feature Types 3-5, 3-6, and 3-7 involving only function argument addition, deletion, or modification, the algorithms depending on the PDG (algorithm No.13-No.18) were all excluded because they neglected the function arguments and generated false-positives. For diff hunks with patch feature Type 6-3 involving only modification beyond the function, the algorithms depending on the AST or PDG (algorithm No.7-No.18) were excluded because there were no functions for the AST or PDG.

After running step 2 and step 3 in Algorithm 1, we obtained the select code-similarity algorithms of VulPecker.



(a) Existing code-similarity algorithms



(b) Variant of existing code-similarity algorithms

Figure 6: Comparing existing code-similarity algorithms and their variants against VulPecker via three metrics: precision, recall and F-measure

Among them, code-similarity algorithm No.5 and its variants, algorithm No.9, algorithm No.17 and its variants were selected by most vulnerabilities. Vulnerabilities with diffs that involved function addition or modifications going beyond the function were more likely to choose algorithm No.5 or its variants. Vulnerabilities with diffs that only involved line addition in the function were more likely to choose algorithm No.9. Vulnerabilities with diffs that contained more than line addition were more likely to choose algorithm No.17 or its variants. Whether to select their variants, and if so selecting which variant, largely depended on the F-measure.

5.2 Detecting vulnerabilities in products

Now we report the results of using the learned vulnerability signatures and CVE-to-algorithm mapping to detect vulnerabilities in products. We selected 246 vulnerabilities that were published between 2013 and 2015 for three products, namely Firefox, Ffmpeg, and Qemu. The task is to determine whether a target product contains one or more of these vulnerabilities. The 40 vulnerabilities that are detected by VulPecker and are not published in the NVD are

listed in Appendix Table A.1. Among these vulnerabilities, 18 are not known for their existence and have yet to be confirmed by the respective vendors at the time of writing. For the 18 unpatched vulnerabilities, we anonymize their CVE-ID, vulnerability publish time, and vulnerability location for ethical reasons. The other 22 vulnerabilities have been "silently" patched by product vendors after 7.3 months on average since the vulnerabilities were published. We manually checked and confirmed these 22 vulnerabilities.

We used a virtual machine with Intel Core 2.5GHz processor and 3GB of RAM running CentOS 6.0-64bit for the experiments. For the select code-similarity algorithms listed in Table A.1, we take the code-similarity algorithms using the subgraph-isomorphic comparison method as examples to show the time overhead incurred by the vulnerability detection procedure, because these algorithms are usually considered more time-consuming than the others. In our evaluation, we adopted several optimizations to reduce their time overhead, such as the exclusion of a large number of irrelevant edges and nodes, and breaking a big graph into small ones [18]. Take the target project Libav 10.1 (29.6MB) as an example, our goal is to detect whether it contains vulnerabilities CVE-2014-8547 (via algorithm No.18-1), CVE-2013-7011 (via algorithm No.18-1), CVE-2013-3674 (via algorithm No.17), and CVE-2013-0851 (via algorithm No.18-2). The detection time corresponding to these four vulnerabilities was respectively 508.11s, 128.14s, 81.77s, and 141.44s. It is clear that the detection time depends on the size of target project, the selection of the code-similarity algorithm, and the complexity of graphs for slice code fragment.

In what follows, we elaborate two vulnerabilities that have been silently patched.

CVE-2015-0834. Mozilla Firefox prior to version 36.0 contains an information leak/disclosure vulnerability in the WebRTC subsystem, which "*makes it easier for man-in-the-middle attackers to discover credentials by spoofing a server and completing a brute-force attack within a short time window*" [1]. This vulnerability is originally reported for Mozilla Firefox. However, our study shows that this vulnerability also exists in Thunderbird 24.8.0 and the other versions prior to Thunderbird 38.0.1. Figure 3 shows the diff hunk of CVE-2015-0834. VulPecker selects code-similarity algorithm No.9 for this diff hunk, owing to the fact that the diff hunk only involves line addition in the function.

CVE-2014-2894. Qemu prior to version 2.0 has a numeric errors vulnerability, which "*allows local users to have unspecified impact via a SMART EXECUTE OFFLINE command that triggers a buffer underflow and memory corruption*" [1]. However, this vulnerability is reported only for Qemu. Our study shows that the very vulnerability also exists in Xen 4.4.0 and the other versions prior to Xen 4.4.3. VulPecker selects code-similarity algorithm No.18-2 for detecting this vulnerability (only one diff hunk), owing to the fact that the diff hunk only involves constant modification in the function.

6. LIMITATIONS

The present study has several limitations. First, our experiments focus on C/C++ open source products. While the methodology underlying VulPecker is language agnostic, experiments need to be conducted to analyze target programs written in other languages, such as Java and Python.

Second, the VPD and VCID databases need to be improved. For creating the VPD, we use heuristics to automatically find the diffs for given CVE-IDs. Our manual examination on a random sample of 10% of the vulnerabilities shows that the heuristics lead to accurate results, which however does not necessarily mean the heuristics are always accurate. It is important to test a bigger sample to assure their accuracy. For creating the VCID, the approach we use to obtain code reuse instances may be unnecessarily restrictive. More experiments need to be conducted to accept or reject this hypothesis.

Third, the ultimate goal of our research is the following: Given a vulnerability, how can one determine whether or not the vulnerability exists in any program of the entire software stack of a computer (assuming source code is available)? This raises the scalability issue which needs to be investigated.

Fourth, the methodology appears to be specific to the detection of vulnerabilities at the source code level. It is an important research problem to detect, automatically and effectively, whether a piece of binary code has a given vulnerability or not.

7. CONCLUSION

We have presented VulPecker, a system for automatically detecting whether a program contains a given vulnerability or not. VulPecker leverages features that we define to characterize vulnerabilities and patches. Experimental results show that VulPecker detects 40 vulnerabilities that are not published in the NVD. Among these vulnerabilities, 18 are not known for their existence and have yet to be confirmed by vendors at the time of writing, while the other 22 vulnerabilities have been "silently" patched by vendors when releasing a later version.

For future research, it is interesting to address the limitations mentioned above. It is also interesting to test whether VulPecker can detect the vulnerabilities that are intentionally inserted by systems like LAVA [10].

Acknowledgments

We thank the anonymous reviewers for their comments that helped us improve the paper. We thank Marcus Pendleton for proofreading the paper. This paper is supported by National Basic Research Program of China (973 Program) under grant No.2014CB340600. Shouhuai Xu is supported in part by NSF Grant #1111925 and ARO Grant #W911NF-13-1-0141. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] CVE. http://cve.mitre.org/.

[2] National Vulnerability Database. https://nvd.nist.gov/.

[3] Open Sourced Vulnerability Database. http://www.osvdb.org.

[4] A. Aydin, M. Alkhalaf, and T. Bultan. Automated test generation from vulnerability signatures. In *Proc. ICST*, pages 193–202. IEEE, 2014.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[6] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proc. SP*, pages 2–16. IEEE, 2006.

[7] C. C. Chang and C. J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):1–27, 2011.

[8] I. Chowdhury and M. Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[9] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *Proc. ACSAC*, pages 269–278. IEEE, 2006.

[10] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-scale automated vulnerability addition. In *Proc. SP*, pages 110–121. IEEE, 2016.

[11] J. R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus. Fine-grained and accurate source code differencing. In *Proc. ASE*, pages 313–324. ACM, 2014.

[12] F. Gauthier, T. Lavoie, and E. Merlo. Uncovering access control weaknesses and flaws with security-discordant software clones. In *Proc. ACSAC*, pages 209–218. ACM, 2013.

[13] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding unpatched code clones in entire OS distributions. In *Proc. SP*, pages 48–62. IEEE, 2012.

[14] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[15] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Static Analysis*, pages 40–56. Springer, 2001.

[16] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proc. WCRE*, pages 253–262. IEEE, 2006.

[17] H. Li, H. Kwon, J. Kwon, and H. Lee. A scalable approach for vulnerability discovery based on security patches. In *Applications and Techniques in Information Security*, pages 109–122. Springer, 2014.

[18] J. Li and M. D. Ernst. CBCD: Cloned buggy code detector. In *Proc. ICSE*, pages 310–320. IEEE, 2012.

[19] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[20] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proc. KDD*, pages 872–881. ACM, 2006.

[21] A. Meneely, H. Srinivasan, A. Musa, A. Rodriguez Tejeda, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *Proc. ESEM*, pages 65–74. IEEE, 2013.

[22] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *Proc. SP*, pages 692–708. IEEE, 2015.

[23] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proc. CCS*, pages 529–540. ACM, 2007.

[24] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proc. CCS*, pages 426–437. ACM, 2015.

[25] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Detection of recurring software vulnerabilities. In *Proc. ASE*, pages 447–456. ACM, 2010.

[26] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[27] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[28] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proc. SP*, pages 590–604. IEEE, 2014.

[29] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proc. WOOT*, pages 118–127. USENIX Association, 2011.

[30] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proc. ACSAC*, pages 359–368. ACM, 2012.

APPENDIX

Table A.1 summarizes the 40 vulnerabilities that are detected by VulPecker and are not published in the NVD, where the vulnerability publish time is the time the vulnerability was first published and found in another product, and the code-similarity algorithm is a sample of select code-similarity algorithms when a vulnerability diff has multiple diff hunks.

Table A.1: The 40 vulnerabilities detected by VulPecker in three target products, including 22 vulnerabilities that have been "silently" patched. For the 18 vulnerabilities that have not been patched, we anonymize their CVE-ID, vulnerability publish time, and locations in the target products.

Target product	CVE-ID	Vulnerability publish time	Vulnerability location in target product	Code-similarity algorithm (example)	1st patched version of target product	Date of 1st patched version
Thunderbird 24.8.0	CVE-2015-0834	2015/2/25	.../PeerConnectionImpl.cpp	No.9	Thunderbird 38.0.1	2015/6/11
	CVE-2015-****	2015/**/**	.../src/***	No.18-1	Unpatched	-
	CVE-2014-8643	2015/1/14	.../nsEmbedFunctions.cpp	No.18-1	Thunderbird 38.3.0	2015/9/29
	CVE-2014-1498	2014/3/19	.../src/nsCrypto.cpp	No.18-1	Thunderbird 31.0	2014/7/22
	CVE-2013-6167	2014/2/15	.../nsCookieService.cpp	No.9	Thunderbird 38.4.0	2015/11/24
Libav 10.1	CVE-2014-9604	2015/1/16	.../libavcodec/utvideodec.c	No.9	Libav 10.6	2015/3/11
	CVE-2014-****	2014/**/**	.../libavcodec/***	No.9	Unpatched	-
	CVE-2014-8547	2014/11/5	.../libavcodec/gifdec.c	No.18-1	Libav 10.6	2015/3/11
	CVE-2014-****	2014/**/**	.../libavcodec/***	No.18-1	Unpatched	-
	CVE-2014-****	2014/**/**	.../libavcodec/***	No.18-1	Unpatched	-
	CVE-2014-8541	2014/11/5	.../libavcodec/mjpegdec.c	No.18-1	Libav 10.6	2015/3/11
	CVE-2014-****	2015/**/**	.../libavcodec/***	No.18-3	Unpatched	-
	CVE-2014-2098	2014/3/1	.../libavcodec/wmalosslessdec.c	No.5	Libav 10.4	2014/8/18
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.18-3	Unpatched	-
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.18-1	Unpatched	-
	CVE-2013-****	2013/**/**	.../libavfilter/***	No.18-1	Unpatched	-
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.9	Unpatched	-
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.18-1	Unpatched	-
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.18-3	Unpatched	-
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.9	Unpatched	-
	CVE-2013-7011	2013/12/9	.../libavcodec/ffv1dec.c	No.18-1	Libav 10.4	2014/8/18
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.9	Unpatched	-
	CVE-2013-7008	2013/12/9	.../libavcodec/h264.c	No.18-1	Libav 11.1	2014/12/2
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.18-3	Unpatched	-
	CVE-2013-3674	2013/6/9	.../libavcodec/cdgraphics.c	No.17	Libav 10.4	2014/8/18
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.18-1	Unpatched	-
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.9	Unpatched	-
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.18-1	Unpatched	-
	CVE-2013-0851	2013/12/7	.../libavcodec/eamad.c	No.18-2	Libav 10.3	2014/8/4
	CVE-2013-****	2013/**/**	.../libavcodec/***	No.18-1	Unpatched	-
Xen 4.4.0	CVE-2014-2894	2014/4/23	.../ide/core.c	No.18-2	Xen 4.4.3	2015/8/25
	CVE-2014-5263	2014/8/26	.../usb/hcd-xhci.c	No.5	Xen 4.4.3	2015/8/25
	CVE-2013-6399	2014/11/4	.../virtio/virtio.c	No.9	Xen 4.4.3	2015/8/25
	CVE-2013-4534	2014/11/4	.../intc/openpic.c	No.18-1	Xen 4.5.0	2015/1/14
	CVE-2013-4533	2014/11/4	.../arm/pxa2xx.c	No.18-1	Xen 4.5.0	2015/1/14
	CVE-2013-4530	2014/11/4	.../ssi/pl022.c	No.5	Xen 4.5.0	2015/1/14
	CVE-2013-4527	2014/11/4	.../timer/hpet.c	No.5	Xen 4.5.0	2015/1/14
	CVE-2013-4151	2014/11/4	.../virtio/virtio.c	No.18-1	Xen 4.4.3	2015/8/25
	CVE-2013-4150	2014/11/4	.../net/virtio-net.c	No.9	Xen 4.4.3	2015/8/25
	CVE-2013-4149	2014/11/4	.../net/virtio-net.c	No.18-1	Xen 4.4.3	2015/8/25