V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities

Seunghoon Woo[†], Dongwook Lee[†], Sunghan Park[†], Heejo Lee[†]*, Sven Dietrich[‡]
[†]*Korea University, {seunghoonwoo, dongwook2014, sunghan-park, heejo}@korea.ac.kr*
[‡]*City University of New York, spock@ieee.org*

Abstract

Common Vulnerabilities and Exposures (CVEs) are used to ensure confidence among developers, to share information about software vulnerabilities, and to provide a baseline for security measures. Therefore, the correctness of CVE reports is crucial for detecting and patching software vulnerabilities.

In this paper, we introduce the concept of "Vulnerability Zero" (VZ), the software where a vulnerability first originated. We then present `V0Finder`, a precise mechanism for discovering the VZ of a vulnerability, including software name and its version. `V0Finder` utilizes code-based analysis to identify reuse relations, which specify the direction of vulnerability propagation, among vulnerable software. `V0Finder` constructs a graph from all the identified directions and traces backward to the root of that graph to find the VZ.

We applied `V0Finder` to 5,671 CVE vulnerabilities collected from the National Vulnerability Database (NVD) and popular Bugzilla-based projects. `V0Finder` discovered VZs with high accuracy of 98% precision and 95% recall. Furthermore, `V0Finder` identified 96 CVEs with incorrect information related to their respective VZs. We confirmed that the incorrect VZ causes prolonged patch updates of vulnerable software; the patch update of CVEs with the incorrect VZ information takes 2 years, while the patch update of CVEs with the correct VZ takes less than a year on average. Such incorrectly identified VZ hinders the objective of the CVE and causes confusion rather than "ensuring confidence" among developers. Our analysis shows that `V0Finder` can enhance the credibility of information provided by the CVEs.

1 Introduction

The growing number of software made it possible for developers to share their code with one another in the form of a public library and open-source software (OSS). This code-sharing culture provides high productivity, yet causes the transfer of vulnerable code [18, 26, 58]. To mitigate this issue, information about known vulnerabilities is shared through channels such as the National Vulnerability Database (NVD) [39] in the form of Common Vulnerabilities and Exposures (CVE).

Despite such collective efforts, mistakes can cause a transfer of a vulnerability from one software to another without any interruption, as the quality and correctness of these vulnerability reports are not guaranteed. Therefore, quality control of public vulnerability reports has become a major research interest in software security [2, 5, 8, 34, 36]. Of the many pieces of information in the public vulnerability report, we paid particular attention to the *origin* of the vulnerable software where the vulnerability is coming from. We coin this origin of vulnerable software as **Vulnerability Zero** (VZ), a reference to the medical term *patient zero* (or *primary case*) [56].

One of the main problems with current vulnerability reports is the lack of verifying the correctness of VZ information. A third-party OSS vulnerability in a software program is occasionally reported as the entire program's vulnerability. Such reports with the incorrect VZ lead other developers, who reuse the vulnerable third-party software, to unintentionally overlook the propagated vulnerabilities within their software, causing transfer of the vulnerability to other software, and delaying patch deployment. Thus, without proper verification of VZ correctness in CVEs, incorrect VZs may create confusion and hinder the core objective of the CVE.

In other research fields, *e.g.*, medical [57] and malicious software [15, 19], the importance of discovering a primary case has already been introduced. To the best of our knowledge, however, no existing approaches have attempted to discover the VZ of a *software vulnerability* and further to reveal its importance. Recent studies can be classified into three categories: (1) identifying *missing* information in the report to efficiently mitigate vulnerabilities [2, 3, 5, 36], (2) analyzing the *reproducibility* of vulnerabilities [14, 34], and (3) ensuring *consistency* between the vulnerability description and the affected software information [8]. However, if those studies are based on public vulnerability reports with an incorrect VZ, the credibility of their results can be challenged.

Although a VZ literally means the vulnerable software with the earliest birth date, a method that solely relies on the

*Heejo Lee is the corresponding author.

timestamp metadata can often fail to discover the VZ; this is because the reliability of the timestamp metadata is not always guaranteed. Since software code can be modified after its release, we cannot always determine that the earliest-released software among the vulnerable software is the VZ. Moreover, a code-generation time could be changed due to operations such as copying and pasting of source files [51]. Thus, this approach yields considerable false alarms (see Section 5.2).

To overcome such shortcomings, we propose a novel mechanism called `VOFinder` (**V**ulnerability **Zero Finder**), to precisely discover a VZ assisting in detecting and patching software vulnerabilities. To discover a VZ precisely, we face two main technical challenges: (1) addressing syntax-variety of a vulnerable code and (2) selecting proper features that can be utilized to discover a VZ in the large-scale software pool. The syntax of a vulnerable code introduced in the VZ could be modified when it propagates to other software or when the VZ is updated to a newer version. Hence, we need a technique to discover the VZ while being aware of the variety of vulnerable code. Furthermore, among the software containing vulnerable codes with various syntax, we need to select proper features that can pinpoint the correct VZ; as mentioned earlier, timestamp metadata alone is insufficient.

Our approach. To discover a VZ, `VOFinder` (1) detects a set of vulnerable software that contains a particular vulnerability, (2) identifies propagation directions of this vulnerability among the vulnerable software, and then (3) determines the VZ of the vulnerability by backtracing propagation directions.

First, for detecting vulnerable software, `VOFinder` uses function-level vulnerable-clone detection for best accuracy and speed [26,58]. To address the syntax-modified vulnerable clones, we utilize Locality Sensitive Hashing (LSH) [12,41] and the patch code of the vulnerability (see Section 3.1).

Next, `VOFinder` identifies propagation direction of the vulnerability among the detected vulnerable software. In particular, we pay attention to the *reuse relation* between the two vulnerable software, because the reuse relation can be used to indicate the vulnerability propagation directions: when S is reused in S' and they share the same vulnerable code, we can infer that the vulnerable code was propagated from S to S' (*i.e.*, $S \rightarrow S'$). Thus, `VOFinder` performs a *code-based analysis* between the two vulnerable software to identify a reuse relation, more specifically, factors used for code-based analysis include (1) source code of the software, (2) location of the source code, and (3) metadata files (see Section 3.2).

Accordingly, `VOFinder` constructs a *vulnerability propagation graph*, where nodes indicate the vulnerable software and edges represent the propagation directions. Then `VOFinder` discovers the VZ by finding the root of the constructed graph.

Evaluation and observations. We collected 5,671 CVEs from the NVD and popular Bugzilla-based projects, including all the C/C++ related CVEs that released patches via Git. For each CVE, we compared the VZ discovered by `VOFinder` with the Common Platform Enumeration (CPE) [40], which specifies the software affected by each CVE. As a result, `VOFinder` discovered VZs with 98% precision and 95% recall for the collected CVEs (see Section 5.1).

Further analysis demonstrates the importance and need for VZ discovery. From the experiment, we found that the public reports of 96 CVEs (1.7%) provided the incorrect VZ information. It is worth noting that they are rarely patched once these vulnerabilities are propagated; 64% of the latest version of software affected by the 96 CVEs still contain the propagated CVE, whereas the ratio is much lower (15%) for CVEs with the correct VZ information. To make matters worse, many popular software programs have been released with unpatched vulnerabilities due to the incorrect VZ information (see Section 6.1). Even if developers succeeded in detecting the CVE's vulnerability in their programs, the elapsed time for patching doubled in the case of the CVE with the incorrect VZ information, compared to the case with the correct VZ information (see Section 6.2).

This paper makes the following three main contributions:

- We coin the term VZ, which represents the software and its version where the vulnerability originated, and show the importance of VZ discovery for the first time.
- We present `VOFinder`, a precise mechanism to discover the VZ using a vulnerability propagation graph. It is shown that `VOFinder` provides high accuracy with 98% precision and 95% recall.
- From the 96 CVEs with the incorrect VZ information discovered by `VOFinder`, we show a significant impact of the incorrect VZ information on the affected software programs including the delay of patch updates.

2 Motivation

In this section, we clarify the terminology used in this paper, and then discuss the motivation for VZ discovery.

2.1 Basic terminology

Software. We consider software as a source code-level project (rather than binaries) that contains source files and functions because our mechanism is based on the source code comparison. Furthermore, we only focus on the software that is publicly managed by hosting services (*e.g.*, GitHub).

Vulnerable code. We define vulnerable code to be source code causing a vulnerability that has not been fixed, *i.e.*, unpatched. If a copy of vulnerable code exists in a particular software program, the software is defined as vulnerable irrespective of the exploitability of the copied vulnerable code. We then define vulnerability propagation to occur when vulnerable code in a software program (S) propagates to another software (S') via software forks or OSS reuse.

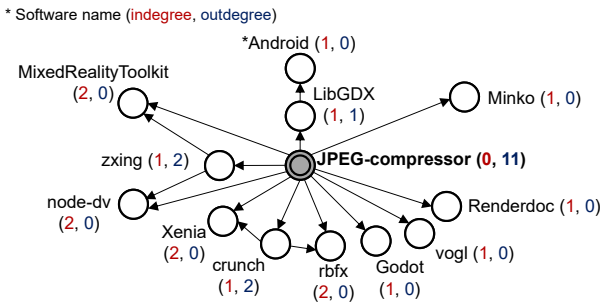


Figure 1: Illustration of the vulnerability propagation graph for CVE-2017-0700.

Vulnerability propagation graph. Since `VOFinder` discovers a VZ using the vulnerability propagation direction, we need a data structure that is able to show direction. In addition, as the propagation direction is one-way, there should be no cycle. Therefore, we derive a vulnerability propagation graph by leveraging a directed acyclic graph-based structure:

- *A vulnerability propagation graph* (G) is a directed acyclic graph, and is represented as G = (V, E), where V denotes a set of nodes (*i.e.*, vulnerable software) with indegree and outdegree labels for each vertex. For $v \in V$, the indegree of v is denoted as `IN`, and its outdegree is denoted as `OUT`. E denotes a set of edges ($E \subseteq V \times V$) and shows the vulnerability propagation directions.

Vulnerability Zero (VZ). VZ refers to the software and its version where a vulnerability *originated*. Note that a VZ is not the software that first *detected* or *reported* the vulnerability. Specifically, the VZ is regarded as the beginning of vulnerability propagation, and thus, the root of a vulnerability propagation graph is the VZ.

2.2 Problem statement

In this paper, we focus on the problem that occurs when the *index case* (*i.e.*, the first case discovered, and possibly believed to be the source) and the *primary case* (*i.e.*, the actual source) of a vulnerability are different. Let S be a software program and L be a third-party library, which is embedded in S. If a vulnerability is detected in L, it should be considered the vulnerability in L (*i.e.*, VZ). However, this vulnerability is occasionally reported as that of in P. For such case, the vulnerability that is fixed in P is usually not reported upstream L. Consequently, L not only remains unpatched, but also overlooked by developers of software, which is embedding L, furthering vulnerability propagation and hindering prompt detection and patches for the vulnerability.

The correctness of the VZ is important to software developers, as many parts of the vulnerability management process still rely on public vulnerability reports [1, 54]. However, the interests of existing approaches are far from this issue. Instead, they attempt to solve other problems by assuming that the VZ of a vulnerability was provided correctly (see Section 7). For instance, Dong *et al.* [8] identified the inconsistency between the CVE description and the CPE [40], which specifies a set of software affected by the CVE, but did not consider the possibility of the CPE containing an incorrect VZ.

Moreover, a method to scan vulnerabilities using static or dynamic tools (*e.g.*, fuzzing tools) cannot be a fundamental solution to the presented problem. As such method focuses only on the internal vulnerabilities of a particular software, to resolve the problem caused by incorrect VZ, it should be applied repeatedly to all affected software programs. Instead of such an inefficient solution, we need a one-stop solution that can detect the correct VZ for a vulnerability and notify all the affected software programs of the vulnerability once.

Technical challenges. Discovering a VZ is not a simple task mainly due to the following two technical challenges: (1) addressing syntax-variety of vulnerable code and (2) selecting proper features for discovering a VZ.

First, the syntax of a vulnerable code frequently changes while the software that contains the code is updated or when the code is reused in other software. When discovering a VZ, we should be aware that such a vulnerable code with various syntax exists in various software. Several vulnerable-code clone detection approaches can cope with syntax-changes [26, 58], but we cannot say that their techniques are still effective for VZ discovery.

Second, we need proper features to determine a vulnerable software as VZ. It is easy to assume that vulnerable software with the earliest birth date is the VZ. However, in practice, the source code of software may change after its release, and further, a code-generation time can also be changed easily owing to operations such as copying and pasting [51]; this approach often fails to discover the correct VZ (see Section 5.2). We need a method to discover VZ through a more appropriate feature than mere timestamp metadata.

2.3 A motivating example

We introduce the CVE-2017-0700 case, a remote code execution vulnerability reported by Android (see Listing 1), where the vulnerability originated in the JPEG-compressor[1] source code (*i.e.*, "jpgd.cpp" file), and not in the actual Android code (see Listing 3). The vulnerability propagation graph for CVE-2017-0700 is illustrated in Figure 1.

Listing 1: The description of CVE-2017-0700.

```
A remote code execution vulnerability in the Android
system ui. Product: Android. Versions: 7.1.1, 7.1.2.
Android ID: A-35639138.
```

Listing 2: The CPE of CVE-2017-0700.

```
cpe:2.3:o:google:android:7.1.1:*:*:*:*:*:*:*
cpe:2.3:o:google:android:7.1.2:*:*:*:*:*:*:*
```

[1] https://github.com/richgel999/jpeg-compressor

Listing 3: A patch snippet for CVE-2017-0700.

```
--- a/gdx/jni/gdx2d/jpgd.cpp
+++ b/gdx/jni/gdx2d/jpgd.cpp
@@ -2282,3 +2304,4 @@void jpeg_decoder::
     make_huff_table(int index, huff_tables *pH){..
for (l = 1 << (8 - code_size); l > 0; l--){
- JPGD_ASSERT(i < 256);
+ JPGD_ASSERT(i < JPGD_HUFF_CODE_SIZE_MAX_LENGTH);
+ JPGD_ASSERT(code < JPGD_HUFF_CODE_SIZE_MAX_LENGTH);
```

The software programs affected by this vulnerability can be classified into the following two groups:

(1) Software reusing Android: Because the vulnerability was reported by Android, software programs that reused the specified Android 7.1.1 or 7.1.2, or distributions of those Android versions, can easily resolve the vulnerability, *e.g.*, updating Android to the later version.

(2) Software reusing only JPEG-compressor: Software programs belonging to this group are not only uninterested in this vulnerability reported by Android but also barely attempt to determine whether their codebase is affected by the vulnerability. Thus, most of these software programs fail to detect and patch the vulnerability in a timely manner.

Surprisingly, all of the software shown in Figure 1, except Android, contained the vulnerable code up to their latest version. We succeeded in reproducing this vulnerability in the latest versions of three popular software by using a crafted image file as an input: JPEG-compressor, Godot[2] and LibGDX[3]. In other software, the failure to the reproduction was due to a compilation error and failure to call vulnerable functions. As soon as we reported this vulnerability, Godot and LibGDX patched it (July 2019); JPEG-compressor, which has not been developed any further in recent years, did not respond yet.

3 Methodology of V0Finder

In this section, we describe the methodology of `V0Finder`. The high-level workflow of `V0Finder` is depicted in Figure 2. `V0Finder` discovers the VZ by means of finding the root of a vulnerability propagation graph (see Section 2.1). Subsequently, `V0Finder` comprises the following three phases: *node discovery phase* (**P1**), *edge connection phase* (**P2**), and *root finding phase* (**P3**).

Let V be a set of vulnerable functions for a given CVE. In **P1**, `V0Finder` detects vulnerable software that contains a clone of V by using a vulnerable-clone detection technique. Let D represent the software dataset, S_i indicate a software program in D, and "‡" denote the software is vulnerable. Then the output of P1 is represented as follows:

$$P1\ (V, D) = \{S_1^‡, S_2^‡, ..., S_m^‡\}$$

In **P2**, for the given set of vulnerable software programs, `V0Finder` identifies reuse relations between every pair of

[2] https://github.com/godotengine/godot
[3] https://github.com/libgdx/libgdx

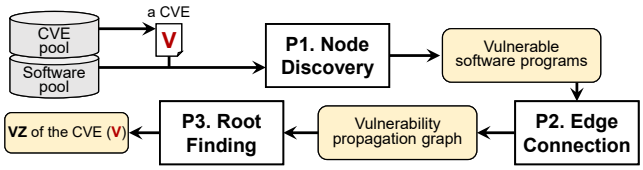


Figure 2: High-level workflow of `V0Finder`.

vulnerable software programs. When $S_i^‡$ is reused in $S_j^‡$, the relation between them is expressed as follows:

$$S_i^‡ \prec S_j^‡ \qquad (S_i^‡ \text{ is reused in } S_j^‡)$$

The identified reuse relations that exist between vulnerable software pairs are expressed as follows:

$$P2\ (\{S_1^‡, S_2^‡, ..., S_m^‡\}) = \{(S_1^‡ \prec S_2^‡), (S_1^‡ \prec S_3^‡), ...\}$$

Using this information, `V0Finder` constructs the vulnerability propagation graph for V, which is the output of P2. Here, the propagation paths are *reverse* directions of the reuse relations; this is because, if $S_i^‡$ is reused in $S_j^‡$ ($S_i^‡ \prec S_j^‡$), then the vulnerability has propagated from $S_i^‡$ to $S_j^‡$ ($S_i^‡ \rightarrow S_j^‡$).

Finally, in **P3**, `V0Finder` discovers the VZ of V by finding the root of the constructed graph.

3.1 Node discovery phase (P1)

Given a vulnerability patch as input, `V0Finder` reconstructs the vulnerable functions, and further detects vulnerable software using the vulnerable-clone detection technique. The detected vulnerable software programs become nodes in the vulnerability propagation graph.

Vulnerable-clone detection. For efficient VZ discovery, we value the following three factors in vulnerable-clone detection: robustness to syntax-modified clones, accuracy, and scalability. As the syntax of a vulnerable code introduced in a VZ can be modified when it propagates to other software, our vulnerable-clone detection scheme must be robust to the variable syntax. In fact, detecting vulnerable clones is not the core idea of this paper, and many relevant studies have already been conducted [18, 26, 30, 31, 42, 58]. Each of them has distinct advantages and disadvantages, *e.g.*, VUDDY [26] is scalable yet barely detects vulnerable clones with modified syntax, while MVP [58] detects syntax-modified clones well, but it is less scalable than other approaches.

Therefore, we decided to incorporate techniques specialized for VZ discovery while leveraging the advantages of existing vulnerable-clone detection techniques as follows:

- **Referenced features from existing techniques**:
  - → Function-level granularity [26, 58];
  - → Text-preprocessing [18, 26, 58];
  - → Function hashing [18, 26, 58].
- **Features specific to `V0Finder`**:
  - \+ Using Locality Sensitive Hashing (LSH) [12, 41];
  - \+ Detection mechanism for syntax-modified clones.

Since we selected function-level granularity as the basis, a software program and a CVE can be represented as a set of functions. Note that a patch for a vulnerability usually consists of several code block changes [29], meaning there can be multiple vulnerable functions causing a vulnerability. Using function-level granularity, the scale of comparison between entire software and vulnerability can be reduced to the comparison between a pair of functions set, which improves comparison performance and scalability [26, 48, 58].

Software and CVE pools. A software pool comprises a set of software source code, which becomes a search area for vulnerable-clone detection and contains the source code of all versions for each software program. The CVE pool contains a set of vulnerable codes and pieces of patch information as the source code form (details are explained in Section 4.1).

Detecting vulnerable software programs. Let s_j denote the version j of S_i. We define s_j as vulnerable when it contains a vulnerable clone from a set of vulnerable functions V, and this is determined by performing *preprocessing* and *comparison*.

Step 1. Preprocessing: Considering the syntax variety of vulnerable clones, we apply the following two preprocessing tasks to all the functions in the software and CVE pools: text-preprocessing and LSH [12, 41]. During text-preprocessing, the code part that maintains function semantics even upon introducing changes, such as spaces, newline characters, and comments, was removed from each function, and all the characters in the function were converted to lower cases [26, 58]. We then apply LSH to all text-preprocessed functions, which generates similar hashes for similar inputs and returns a low distance value when the two input hashes are similar [27, 41]. Unlike previous approaches, which store only the hash values, we store the preprocessed string value of each function, which will be utilized to detect syntax-modified vulnerable clones.

Step 2. Comparison: We then compare all the hashed functions of s_j and those of V by employing the comparison method provided by LSH [28]. As a result, we can obtain the distance (F) of all function pairs between s_j and V; this distance shows the syntactic difference between the source code of the two functions. Furthermore, LSH configures a *cut-off* value (q) [28], and the two inputs are similar if the F is less than or equal to q. Let f_s be a function in s_j and f_v be a function in V.

1) ***Exact clone***: If F is zero (*i.e.*, the syntax of both functions is exactly the same), this indicates that f_s is the unpatched vulnerable clone of f_v.
2) ***Modified clone***: If F is obtained as a value between zero and q ($0 < \mathsf{F} \leq \mathsf{q}$), this indicates that f_s is the modified clone of f_v. However, since the modification could be a patch for a vulnerability, an additional verification step must be performed.
3) ***No clone***: If F is greater than q, it means that f_s is not a clone of f_v.

To determine whether a modified clone is vulnerable, we utilize the vulnerability patch of V from the CVE. A patch consists of a set of *deleted* codes (*e.g.*, line 5 in Listing 3), which might be seen as vulnerable code, and a set of newly *inserted* codes (*e.g.*, line 6 and 7 in Listing 3). If f_s is the modified clone of f_v, and if all the deleted codes of the patch are contained in the preprocessed f_s without any of the inserted codes, we conclude that f_s is a vulnerable clone of f_v.

By applying these two steps to all the CVEs in the CVE pool, we can discover a set of vulnerable s_j containing vulnerable clones of V. Consequently, a software (S_i) becomes the node of the vulnerability propagation graph for V when at least one version (s_j) contains the vulnerable clone of V.

3.2 Edge connection phase

In this phase, `VOFinder` identifies vulnerability propagation directions among the detected vulnerable software.

As we discussed in Section 2.2, the method of tracking VZ by relying solely on the timestamp metadata is prone to false alarms. Therefore, rather than relying on such timestamp metadata, we focused on a relation between the two vulnerable software. More specifically, `VOFinder` identifies *reuse relations* between every vulnerable software pair. We represent the relation in which S_i is reused in S_j as $S_i \prec S_j$.

We particularly paid attention to the reuse relation as it can indicate the vulnerability propagation direction. If S_i is reused in S_j and they share the same vulnerable code, then we can infer that the vulnerable code has propagated from S_i to S_j, expressed as $S_i \rightarrow S_j$. Hence, the oldest ancestor-software of the vulnerability propagation, *i.e.*, the starting point of the propagation, becomes the VZ of the vulnerability.

If the dependency information is provided, *e.g.*, node package manager in JavaScript [38], we can simply parse and utilize the dependency information to identify reuse relations (*e.g.*, GitHub dependency graph [13]). However, some other languages such as C, do not provide any dependency information. Therefore, we devise a method to identify reuse relations without relying on the given dependency information.

Modified software reuse. `VOFinder` identifies reuse relations between the two vulnerable software by employing code-based analysis. However, in the case of $S_i \prec S_j$, the codebase of S_j may not be reused as it is. We thus define the two modification patterns that can arise from software reuse to efficiently identify reuse relations: (1) *code modification* refers to that S_i is reused in S_j with source code changes or when only part of the codebase is reused; (2) *structure modification* refers to that S_i is reused in S_j with structural changes, specifically, name (*e.g.*, file and function name changes) and location changes (*e.g.*, reused in a different directory) of reused code.

Key factors. Considering modified software reuse, we select three key factors utilized in reuse relation identification: (1) source code of the software, (2) location of the source code, and (3) a set of metadata files.

The source code of the software is used to identify reuse relations without code modification; specifically, `VOFinder` measures shared code (*i.e.*, common functions) ratio between two software. To identify a case in which software is reused with code modification, we use the location of the source code (*i.e.*, file paths) in the reused code. Finally, when the code and structure are simultaneously modified while being reused, we utilize the metadata files to identify reuse relations, which should be reused without modifications, *e.g.*, license files.

When `VOFinder` identifies the reuse relation, it compares all vulnerable version pairs between the two vulnerable software; if any reuse relation between the two versions is identified, we conclude that the reuse relation exists between the two software. Let s_v and s'_v denote the vulnerable versions of S_i and S'_i, respectively. We introduce how to identify the reuse relation between s_v and s'_v by using the three key features.

Shared code ratio-based identification. We define two notations, a and b for measuring the ratio of the shared code from the perspectives of s_v and s'_v, respectively:

$$\mathsf{a} = \frac{|s_v \cap s'_v|}{|s_v|} \quad \text{and} \quad \mathsf{b} = \frac{|s_v \cap s'_v|}{|s'_v|}$$

Here, $|s_v \cap s'_v|$ denotes the number of common functions between s_v and s'_v. Because a and b are a feature in identifying reuse relations without code modification, the number of hashed functions that are exactly the same between s_v and s'_v is measured to obtain $|s_v \cap s'_v|$. If s_v reuses s'_v without code modification, the entire codebase of s'_v should be contained in the s_v. Therefore, `VOFinder` determines that s_v is reusing s'_v when $\mathsf{a} < 1.0$ and $\mathsf{b} = 1.0$ (*i.e.*, $s'_v \prec s_v$ and $s'_v \rightarrow s_v$).

Source code location-based identification. The original code path of s'_v is included in the path of the reused code in s_v when s_v is reusing s'_v without any structural modification. `VOFinder` compares both file paths of the common functions between the two software by string comparison to determine which one belongs to the other. For example, the function containing the vulnerable code of JPEG-compressor is reused in Godot (CVE-2017-0700, see Section 2.3), and their paths are as follows (see Listing 4):

Listing 4: The path of the vulnerable code of CVE-2017-0700.

```
JPEG-compressor: "./jpgd.cpp"
     Godot     : "./thirdparty/jpeg-compressor/jpgd.cpp"
```

If the file path of a function (that belongs to both s_v and s'_v) in s'_v is included in that of s_v, we conclude that s_v reuses s'_v (*i.e.*, $s'_v \prec s_v$ and $s'_v \rightarrow s_v$).

Metadata file-based identification. If both code and structure are modified while being reused, `VOFinder` utilizes metadata files, which should be reused without modifications, to identify the reuse relation. Specifically, `README`, `LICENSE`, and `COPYING` files defined as metadata files, and the software reuse relation is identified based on whether these three metadata files are cloned into other software (refer to [16, 24]).

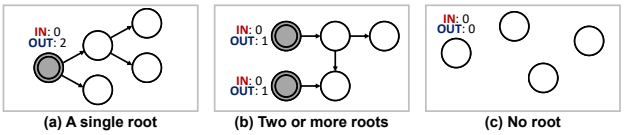


Figure 3: Depiction for the three root cases.

1. `VOFinder` first traverses the files in s_v and s'_v and extracts all metadata files with their path information.
2. `VOFinder` then checks whether each software has a metadata file in the root-source directory, and if it does, checks that the other software has exactly the same files in the other directory except for the root directory.

When comparing two metadata files, the entire content of each file is compared using string comparison. In addition, we determine whether a metadata file in the root directory of a particular software is its own metadata file; therefore, if the own metadata file of s'_v is cloned to s_v, then we can infer that s_v reuses s'_v (*i.e.*, $s'_v \prec s_v$ and $s'_v \rightarrow s_v$).

Tiebreaking. In both shared code ratio-based and metadata file-based identifications, $s'_v \prec s_v$ and $s_v \prec s'_v$ relations cannot be satisfied at the same time; however, this case can appear in source code location-based identification. Thus, we establish the tiebreaking rule: if the paths of a common function of s_v and s'_v are exactly the same, the reused relation is not estimated and no edge is constructed between them.

We assume that the software reuse follows the general code reuse convention. In particular, metadata files are data containing copyrights of the original software and must be specified when reused the software. If this code reuse convention is not followed, it is currently out of the `VOFinder` scope (these cases hardly appeared in experimental observations); covering even anomalous code reuse is left as a future work. Upon connecting all the nodes where the reuse relation exists, a vulnerability propagation graph is constructed for each vulnerability, and then we label the indegree and outdegree values of each node of the graph.

3.3 Root finding phase

Lastly, `VOFinder` discovers VZ by finding the root of the constructed vulnerability propagation graph. Based on the indegree (deg^-) and outdegree (deg^+) values specified for each node, the condition that a node v is a root is as follows:

$$\left(deg^-(v) = 0\right) \wedge \left(deg^+(v) > 0\right)$$

However, the constructed graph does not always contain a single root (see Figure 3); hence, we discover the VZ using different approaches depending on the number of graph roots:

1. ***Single root***: This case indicates that `V0Finder` pinpoints the VZ, *i.e.*, the root of the graph. Specifically, this graph is generated when the VZ is included in the software pool and `V0Finder` has successfully identified reuse relations between vulnerable software.
2. ***Two or more roots***: This case refers to `V0Finder` failed to pinpoint the VZ. If the VZ is not contained in the software pool or if `V0Finder` failed to identify some reuse relations, more than one root may appear.
3. ***No root***: This implies one of the following cases: (1) the VZ does not exist; (2) the vulnerability has not been propagated to other software; (3) `V0Finder` failed to identify reuse relations. First, in the cases of algorithmic vulnerabilities (*e.g.*, a vulnerability in cryptographic code), VZ discovery is infeasible because an algorithmic code has an ambiguous reuse relation; thus this case can be ignored. Next, if the vulnerability has not been propagated, there is only one node in the graph; we consider this node to be VZ. Last, `V0Finder` failed to identify reuse relations and thus there is no root in the graph, *i.e.*, a false negative of `V0Finder`.

To identify the VZ when a graph has *two or more roots*, we decided to take advantage of human intervention. Specifically, `V0Finder` examines the *common functions* of all roots, where each path of the common function contains strong hints about the VZ. For example, in the case of CVE-2019-12900, we found that the generated graph has three roots: GnuPG, PCSX2, and GR. `V0Finder` examined 58 common functions from the three software, of which, 56 were discovered under the path named "Bzip2" (*i.e.*, the VZ of CVE-2019-12900, which was not included in the software pool). Thus, we aim to solve the multi-root problem by providing common function information. This corner case currently requires human intervention and its automation is left for future work.

Lastly, if there is more than one vulnerable version in the discovered VZ, we determine the version the vulnerability first appeared to be the VZ of the vulnerability.

4 Dataset and Implementation of V0Finder

In this section, we introduce dataset construction, and then summarize the implementation of `V0Finder`.

4.1 Dataset

We introduce how to construct our dataset, *i.e.*, the CVE and software pools, for evaluation and further experiments.

CVE patch collection. To collect CVE patches, we leveraged the method used in Li *et al.* [29], where extracting a NVD referenced URL of a CVE that related to Git commits (*i.e.*, URL containing the term `github`, `gitlab`, `cgit`, or `gitweb`). Because these URLs provide the CVE patches in `diff` format, we were able to collect the CVE patches using a simple crawler developed using the `BeautifulSoup` library. We selected the initial vulnerability dataset for C/C++ vulnerabilities, which do not provide any dependencies unlike other languages (*e.g.*, a Gemfile in Ruby); thus, it is suitable to show that `V0Finder` is efficient even without any dependencies.

We then collected the additional CVE patches from Bugzilla for Android, Chromium, and Mozilla, where those three software are included in the top 10 software that reported the highest number of CVEs. To gather CVE patches from Android and Chromium, we adopted the method employed in VUDDY [26], which considers commits containing the keyword "CVE-20" in their log messages as CVE patches. In the case of Mozilla, a bug ID was assigned to each vulnerability in their Bugzilla, and we could disclose the corresponding patch in the commit history of their Git repository[4].

Table 1: CVE pool overview.

Collection source	#CVEs	#Vul. functions
NVD	3,246	12,587
Issue trackers		
Android	1,340	9,581
Chromium	366	5,807
Mozilla	719	4,598
Total	**5,671**	**32,573**

Table 2: Software pool overview.

Collection source	#Software programs
Popular software programs	10,241
CVE-registered software*	460
Total software programs	**10,701**
Total versions	**229,326**
Total lines of codes	**80 billion**

*: Software that has been reporting at least one CVE to NVD.

Vulnerable function reconstruction. Each collected patch contains the Git index and the line numbers including vulnerable code (*e.g.*, "-2282, 3" in Listing 3). To reconstruct vulnerable functions, we first accessed the index to obtain the corresponding vulnerable files (*i.e.*, using `git show` command), and then extracted the vulnerable functions that contain the vulnerable code lines using a function parser [26].

CVE pool construction. Using the method introduced in CVE patch collection, we collected *5,671 CVE patches* reported by 460 software including all the C/C++ CVEs that released their patches via Git. We extracted a total of *32,573 vulnerable functions* from the collected CVE patches in a source code form (see Table 1). On average, one CVE patch consists of six vulnerable functions; all the patches and vulnerable functions were collected in September 2020.

Software pool construction. First of all, we collected the previously mentioned 460 CVE-registered software which have

[4] https://github.com/mozilla/gecko-dev

reported at least one CVE. In addition, we decided to collect widely-reused popular software because the more frequently the software is reused, the more likely it is to propagate vulnerable code, *i.e.*, it has a high probability of being a VZ. To collect the software, we chose GitHub, which is one of the most popular version control systems [44], and collected C/C++ software with more than 100 stargazers [9], a popularity indicator available in GitHub. As a result, we collected *10,701* software (see Table 2), including OS (*e.g.*, Linux), databases (*e.g.*, Redis), and AI (*e.g.*, TensorFlow) related software in April 2020, with a total of 230K versions, 2.2 billion functions, and 80 billion lines of code (LoC).

4.2 Implementation

`VOFinder` comprises the following three modules: *pool construction*, *graph construction*, and *VZ discovery* modules. All the modules were written in Python, and the total length of the implementation is 1,500 lines of Python code.

Parsing and the LSH algorithm. We used universal Ctags [7] to extract functions in C/C++ source codes, which is a regular expression-based parser. Universal Ctags is managed as open-sources, and, therefore, their accuracy and speed are continually enhanced. Subsequently, we selected the LSH algorithm that was best suited for our mechanism. Among the several LSH algorithms available [27, 41, 45], we selected the TLSH algorithm[5], which is both accurate and scalable. Similarity detection using the TLSH algorithm resulted in less false positives with reasonable hashing and comparison speed; furthermore, compared to other algorithms, it was less influenced by the input size [28, 41]. Because TLSH was selected, we referred to [41], and a cut-off value (`q`) of 30 was selected, which is utilized in Section 3.1.

5 Evaluation

In this section, we evaluate `VOFinder`. Section 5.1 investigates how accurately `VOFinder` can discover VZs in practice by comparing the VZ discovery results of `VOFinder` with NVD CPEs, and Section 5.2 evaluates the efficiency of `VOFinder` by comparing it with the timestamp metadata-based approach. Section 5.3 demonstrates the effectiveness of the techniques utilized in `VOFinder`, and Section 5.4 measures the performance of `VOFinder`. We evaluated `VOFinder` on an Ubuntu server with a 2.40 GHz 8-core Intel Xeon Processor, 6TB HDD, and 32GB RAM.

5.1 Comparison with NVD

Methodology. As CPE in NVD specifies the software programs and their versions affected by a CVE, we compared the VZ discovery results of `VOFinder` using the corresponding CPEs. We conclude that our result is correct when it is contained in the CPE. If the discovered VZ is not included in the CPE, we analyze further to decide its correctness. It is possible that `VOFinder` and NVD simultaneously provide the incorrect VZ information for a CVE; however, the fact that the user's reports (*i.e.*, NVD information) and actual code-level VZ discovery (*i.e.*, `VOFinder` result) are the same suggests that these VZs are very likely to be correct.

By parsing the JSON feed obtained from the NVD, we extracted the (CVE, CPE) pairs. We then checked whether the name of the software and version of the discovered VZ were contained in the corresponding CPE. Notably, we were able to perform exact string comparisons for 60% of the discovery results, because the software and version names registered in the CPE were the same as those in GitHub. The remaining 40% exhibited differences from GitHub, *e.g.*, OpenSSL_1_0_0a on GitHub and 1.0.0a in CPE. Thus, we manually compared the discovery results of `VOFinder` and CPE of these CVEs with the help of simple regular expressions, *i.e.*, only numbers and the last alphanumeric characters were considered.

Finally, we introduce the metrics that are used in the accuracy measurement as follows:

- *True Positive (TP)*: The discovered VZ is correct;
- *False Positive (FP)*: The discovered VZ is incorrect;
- *True Negative (TN)*: VZ was not discovered and it does not exist (*e.g.*, algorithmic vulnerabilities);
- *False Negative (FN)*: VZ was not discovered but it exists.

Comparison results. We classified the VZ discovery results of `VOFinder` according to the number of roots in the graph.

(1) Single root cases: `VOFinder` generated a single root vulnerability propagation graph for 2,903 (51%) out of 5,671 collected CVEs. Among them, the discovered VZs for 2,807 CVEs were included in the corresponding CPEs, indicating that the CVEs have the correct VZ (we discuss the reliability of the evaluation in Section 8).

Next, there were 96 single root vulnerability propagation graphs where their roots were not included in the CPE. To verify the VZ from the discovery result of `VOFinder` and that of CPE, we used the following three validation methods:

1. ***Reviewing the code***: A code review was performed by checking whether the vulnerable code causing the vulnerability was included in the discovered VZ.
2. ***Referring to author sites***: The discovered VZ were verified by consulting statements by the software vendor and author sites regarding the origin of the vulnerability.
3. ***Reproducing the vulnerability***: If a proof of concept was publicly provided on a website such as Bugzilla or GitHub, it was utilized for reproducing the vulnerability in the discovered VZ. Otherwise, an attempt to reproduce the vulnerability was performed on our own.

[5] https://github.com/trendmicro/tlsh

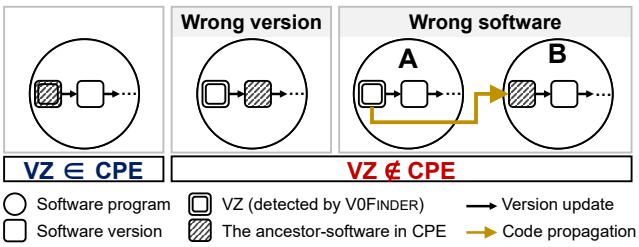


Figure 4: Classification of VZ.

We verified 21 discovered VZs by reproducing the vulnerability. In other cases, the failure to reproduce the vulnerability was due to compilation error, failure to call the vulnerable function, or the proof of concept not being publicly available. Among the remaining 75 cases, we verified 14 discovered VZs by referring to the software author sites. Lastly, for all other cases, we confirmed that the vulnerable code causing the vulnerability was contained in the discovered VZ. Consequently, we confirmed that all the discovered VZs for 96 CVEs were more accurate than those of CPEs (Appendix A shows the list of 96 CVEs, and the related information, *e.g.*, severity and type, for these 96 CVEs is given in Appendix B).

When we classified the 96 CVEs that have the incorrect VZ information into the *wrong version* and *wrong software* (see Figure 4), 50 CVEs have the wrong version, and the remaining 46 CVEs have the wrong software. The wrong version seems less dangerous than the wrong software, but it influences the developers considerably when addressing vulnerabilities. Thus, we decided not to make difference between the importance of the wrong version and the wrong software as both pieces of information are needed for efficiently patching vulnerabilities.

We have responsibly reported all the CVEs with incorrect VZ information that we successfully reproduced. First, it was reported to developers who could not detect the vulnerability due to the incorrect VZ information, and it further reported to the CVE Numbering Authority (CNA) to request changing the CPE. Most of the software authors that had not known the existence of the vulnerability immediately patched it through our reports. There even was a case where a new CVE ID was assigned to this issue (see Section 6.3). Lastly, some CNAs are requesting further details about our reports, while a few others are not currently responding to our requests, even after multiple reports and subsequent inquiries (quantitative figures about our reporting process are given in Appendix C).

(2) Multi-root cases: For the remaining 2,768 CVEs, excluding single root cases, the generated graphs for 52 CVEs have multiple roots. We confirmed that 20 cases occurred because the VZ was not included in the dataset (*e.g.*, when the VZ was not publicly available). The other 32 cases occurred because we failed to identify the software reuse relations; some software did not reuse metadata files while simultaneously modifying codes and structures of reused software. In these cases, we treated all the roots as the VZs of the CVEs and provided hints to predict correct but hidden VZs. As a result, we can identify the VZs for all CVEs by manual analysis with the hints. Nevertheless, these 52 CVEs are FPs.

Table 3: Results of accuracy measurement of `V0Finder`.

#CVEs	#TP	#FP	#TN	#FN	Precision (%)	Recall (%)
Total results:						
5,671	5,410	52	70	139	**99**	**97**
Excluding CVEs with a single node in the graph:						
3,164	2,903	52	70	139	**98**	**95**

(3) No root cases: We found that the generated graphs for the remaining 2,716 CVEs have no root. Among them, the graphs for 2,507 CVEs have only one node. Note that our software pool does not contain closed source software or commercial software. Because operating systems such as Android and browsers such as Firefox are often reused in commercial software rather than in other OSS, numerous graphs have only one root. Nonetheless, we confirmed that the only node in the graph is always contained in the corresponding CPE.

The generated graphs for the remaining 209 CVEs have more than one node but no root. 70 of them are algorithmic vulnerabilities, *i.e.*, TNs of `V0Finder`; for example, CVE-2019-13456 vulnerability was caused by the EAP-pwd handshake algorithm, and thus the VZ was not discovered. Finally, the graph for the remaining 139 CVEs contained multiple nodes but no edges, and they were not algorithmic vulnerabilities. These are cases where `V0Finder` fails to identify reuse relations, and are FNs of `V0Finder`.

(4) Graph statistics: There were 10 nodes and 24 edges in the generated graphs on average. The most complex graph consisted of 120 nodes and 1,942 edges (CVE-2015-4335), and 97% of the graphs with multiple nodes had less than 50 nodes. The average distance from the root node to a leaf node was one depth, and the longest distance was four depths.

In fact, CVEs with a single node in the graph can be considered TPs since the node is contained in the corresponding CPEs. However, we decided that these cases are not properly revealing the accuracy of `V0Finder` because the mechanism of `V0Finder` (*i.e.*, P2 and P3) is not applied. Therefore, we measured the accuracy separately for (1) the entire CVEs and for (2) the CVEs with multiple nodes in their vulnerability propagation graph. As a result, `V0Finder` showed 99% precision $\left(\frac{\#TP}{\#TP+\#FP}\right)$ and 97% recall $\left(\frac{\#TP}{\#TP+\#FN}\right)$ for the entire CVEs, and showed **98% precision** and **95% recall** for the CVEs with multiple nodes in their graph. The summarized results of the accuracy measurement are shown in Table 3.

5.2 Comparison with the timestamp metadata

To demonstrate that `V0Finder` is a more accurate approach than the timestamp metadata-based approach, we compared each VZ discovery result. After finding the vulnerable soft-

ware for a CVE in Section 3.1, the software with the earliest *release date* is determined as the VZ in the timestamp metadata-based approach. Since this approach can pinpoint one VZ for all collected CVEs, no FNs occur. Hence, we determined to focus more on its FPs.

Upon comparing the results between `V0Finder` and the timestamp metadata-based approach, the VZs for 264 CVEs were different. By analyzing the vulnerable code and CVE description, we confirmed that the 244 VZs obtained using the timestamp metadata-based approach were incorrect; the remaining 20 cases were the false alarms of `V0Finder`.

Result analysis. We affirmed that there are two main causes of false alarms in the timestamp metadata-based approach: (1) when the timestamp metadata is unintentionally changed or removed, and (2) when a new code is added to the software that has already been released (without updating its version).

Developers might decide that old versions of the software were no longer needed and therefore made them unavailable. In addition, developers often attempted to transfer their software repository to another hosting service. Due to such processes, the timestamp metadata was easily removed or changed; these unintended changes lead the timestamp metadata-based approach to yield false alarms. For example, in the case of CVE-2015-5221, `V0Finder` discovers that `Jasper` v1.900.1 (the oldest version of `Jasper` that managed by GitHub, released in 2016) is the VZ. The CVE description clearly indicates that `Jasper` is the VZ. However, if the VZ is discovered based on the release date, `ghostpdl` v8.52 (released in 2005) was discovered as the VZ. We confirmed that this situation happened because `ghostpdl` reused a much older version of `Jasper` that is currently not available.

In addition, there were cases that developers added a new code to the previously released version without version updating. As an example, we introduce the CVE-2014-8962 case. This vulnerability originated in Libflac v1.2.0 (released in July 2007). However, the timestamp metadata-based approach determined that the VZ is Praat v4.5.26 (released in May 2007). Praat developers have been continually modifying the code of the previous versions; unfortunately, they added the vulnerable version of Libflac to the previous version in 2014 for the purpose of reading FLAC audio files.

Comparing to the total number of collected CVEs (*i.e.*, 5,671), the number of false alarms of this approach may seem small (*i.e.*, 244). However, since the timestamp metadata still has the possibility that is changed or removed, its reliability will not be guaranteed in the future. From this perspective, we assert that the approach of `V0Finder` that discovers VZ with higher accuracy based on the reuse relation identification is more efficient than the timestamp-based approach.

5.3 Effectiveness of the utilized techniques

Here we evaluate the effectiveness of the techniques utilized in *node discovery* (P1) and *edge connection* (P2) phase.

Table 4: Node discovery phase statistics.

Detection target		#Detection results
(Unique) Vulnerable software programs		1,204
(Unique) Vulnerable versions		45,460
Vulnerable clones	*Exact clones*	249,702
	Modified clones	561,273

Table 5: Edge connection phase statistics.

Utilizing factor	#Identification results
Shared code ratio	5,004 (1.3%)
Path information (*i.e.*, location)	272,410 (72.7%)
Metadata files	97,392 (26.0%)

The node discovering technique. We detected vulnerable code clones based on the LSH and patch code for addressing syntax-modified clones. From the results, we confirmed that the modified clones were detected almost twice as much as the exact clones (see Table 4). Surprisingly, if we only considered exact clones, we failed to discover the VZ for 1,751 (31%) out of 5,671 CVEs. If a vulnerable code exists with significantly different syntax than that of originated in VZ, this cannot be detected with an exact or a limited modified clone detection method such as used in VUDDY (*e.g.*, robust to changes in only several parts such as variable names) [26]. This implies that our node discovering technique, capable of detecting modified clones, is effective for VZ discovery.

The edge connecting technique. We identified reuse relations between vulnerable software based on the shared code ratio, source code path information, and metadata files: only 1.3% of total reuse relations were identified based on the shared code ratio while 72.7% of them were identified by the source code path information (see Table 5). This means that most of the software was being reused *explicitly* (*i.e.*, under the path with the name of the software) in other software with code modifications. We further confirmed that the majority of the graph was constructed with a single root (see Section 5.1), including a case where only one node appears. This shows that `V0Finder` can clearly determine one software program closest to the VZ based on the identified reuse relations.

5.4 Performance of V0Finder

We focus more on accuracy than performance because the VZ discovery needs to be performed only once for a vulnerability. Regardless, we show that `V0Finder` can discover a VZ in a large software pool within a reasonable time.

Preparatory time. In our setup, it took 2 days to construct the CVE pool and 10 days to construct the software pool. This includes the time to clone all versions of a repository with the Git command and the time to extract all functions and apply the hashing after preprocessing the functions.

VZ discovery time. We measured the time required for each phase of `V0Finder`. It took 30 hours to identify vulnerable software programs that contain vulnerable clones of 5,671

Table 6: Success rate for the vulnerability detection.

Category	CVEs with the correct VZ	CVEs with the incorrect VZ
CVEs	3,068	96
Affected software*	10,523	1,000
Cases where the CVE was detected	8,994 (85%)	356 (36%)
Cases where the CVE was undetected	**1,529 (15%)**	**644 (64%)**

∗: The cumulative number of all nodes in the vulnerability propagation graph for all CVEs. Certain software can appear on multiple graphs.

CVEs over a base of 80 billion LoC. Thereafter, it took another 4 hours to construct the vulnerability propagation graphs and discover the VZ. On average, `V0Finder` took approximately 22 seconds to discover the VZ of one CVE, which is sufficient to discover VZs using a large-scale dataset.

6 Impact of VZ discovery

In this section, we analyze the impact of VZ discovery in detail. Specifically, we examine how the correctness of VZ influences prompt detection and patching vulnerabilities by answering the following two questions:

Q1. Are CVEs with the incorrect VZ more difficult to detect than CVEs with the correct VZ? (Section 6.1)

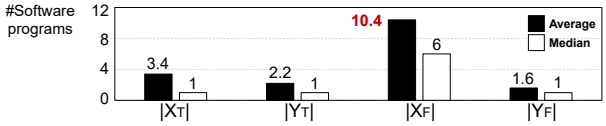
Q2. Do CVEs with the incorrect VZ cause longer patching times for the affected software? (Section 6.2)

6.1 Success rate of vulnerability detection

We analyze the correlation between the correctness of VZ and vulnerability detection. We suggest that vulnerability detection has failed if there was a vulnerable clone in the latest version of the affected software (at the time we collected it); we could measure this information from the output of P1. Table 6 summarizes results of the success rate for vulnerability detection. Note that we only considered CVEs with multiple nodes in their vulnerability propagation graph (*i.e.*, 3,164 CVEs) as CVEs with a single node have no affected software.

We confirmed that due to the incorrect VZ information, the developers were more likely to leave the vulnerability unattended: 64% of reused vulnerable codes were not detected and survived up to the latest version of each affected software. This ratio is much bigger than the cases of CVEs with the correct VZ, where only 15% of vulnerable clones were not detected. Notably, there exists a number of popular software that failed to detect the propagated CVE due to the incorrect VZ, such as Redis (45K GitHub stars, see Section 6.3) and Godot (33.8K stars). This leads to more serious threats: as popular software is reused in many other programs, its vulnerability has better chances of propagation to more programs.

Current vulnerability databases cannot resolve this problem effectively, because they do not provide sufficient information about the software affected by the CVE. To demonstrate this, we examine the following information for every CVE.



$|X_T|$: #Software affected by a CVE with the correct VZ detected by `V0Finder`;
$|Y_T|$: #Software affected by a CVE with the correct VZ provided by CPE;
$|X_F|$: #Software affected by a CVE with the incorrect VZ detected by `V0Finder`;
$|Y_F|$: #Software affected by a CVE with the incorrect VZ provided by CPE.

Figure 5: The number of software programs affected by the CVE according to the correctness of VZ.

- ***X*** : A set of software programs that contain a specific CVE detected by `V0Finder` (*i.e.*, all nodes of a graph);
- ***Y*** : A set of software programs that contain a specific CVE provided by the NVD CPE.

We measured the respective sizes of X and Y for CVEs with the correct VZ (*i.e.*, $|X_T|$ and $|Y_T|$) and with the incorrect VZ (*i.e.*, $|X_F|$ and $|Y_F|$). The results are as shown in Figure 5.

Notably, we confirmed that $|X_F|$ is much larger than $|Y_F|$ ($|X_F| = 6.5 \times |Y_F|$), which indicates many vulnerable software programs have not been traced in public vulnerability reports when a CVE with the incorrect VZ. Although a CPE is not intended to cover the entire vulnerable software, we discovered that the gap between $|X_F|$ and $|Y_F|$ is much larger than the gap between $|X_T|$ and $|Y_T|$ ($|X_T| \approx 1.5 \times |Y_T|$). This implies that CPEs are limited in providing affected software when the CVE with the incorrect VZ compared to that with the correct VZ. The experiment results, in turn, demonstrate the importance of the VZ in the context of vulnerability propagation and patching vulnerable software.

6.2 Elapsed time for vulnerability detection

We analyze the correlation between the correctness of a VZ and the elapsed time for vulnerability detection. We define three time-based metrics: vulnerability introduction time (t_i), vulnerability detection time (t_d), and CVE publication time (t_r); the example timeline is depicted in Figure 6.

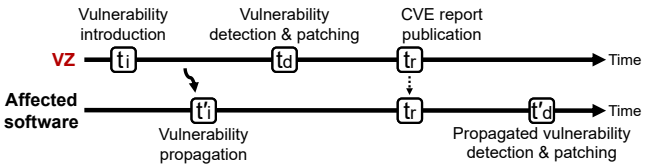


Figure 6: Illustration of vulnerability detection timeline. Note that t_d and t'_i can be later than t_r and t'_d can be earlier than t_r.

We consider the elapsed time for vulnerability detection from the perspective of *software* and *developers*, respectively. From the software perspective, the elapsed time for vulnerability detection is defined as the delta between vulnerability introduction and detection time (t_d - t_i). On the other hand, from

Table 7: Elapsed time measurement for CVE detection.

	Category	Ave. (days)	Med. (days)
$(t_d - t_i)$	VZ	365	142
$(t'_d - t'_i)$	Affected software with the correct VZ	524	371
$(t'_d - t'_i)$	Affected software with the incorrect VZ	**836**	508
$(t_d - t_r)$	VZ	167	0
$(t'_d - t_r)$	Affected software with the correct VZ	308	180
$(t'_d - t_r)$	Affected software with the incorrect VZ	**521**	305

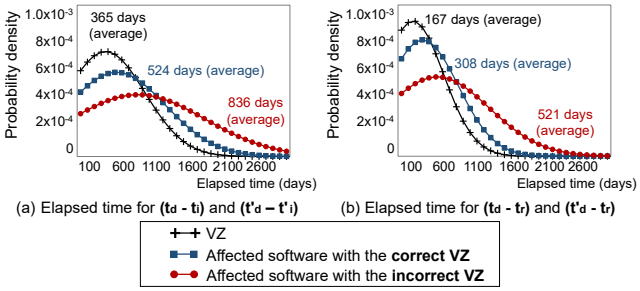


Figure 7: Normal distribution graphs illustrating the elapsed time for vulnerability detection.

the developer perspective, the elapsed time for vulnerability detection is defined as the delta between CVE publication and detection time $(t_d - t_r)$. We measure t_i and t_d by confirming the release date of the version in which the vulnerability first introduced and patched in a software program, respectively; the t_r for a CVE can easily be obtained from the NVD report.

Furthermore, we classify a vulnerability detection *behavior* into three types according to the *subject* of the detection: detection in (1) VZ, (2) software affected by a CVE with the correct VZ, and (3) software affected by a CVE with the incorrect VZ. We measured the elapsed time for vulnerability detection from both software and developer perspectives for each of these three subjects. The results are shown in Table 7.

The main quantitative finding is that if a CVE with the incorrect VZ is propagated to other software, those affected software programs required **312 more days** to detect the vulnerability than the case where a CVE with the correct VZ. CVEs with the incorrect VZ are difficult to detect unless software developers scan for vulnerabilities using static and dynamic tools, or receive reports from the security analysts. Consequently, the incorrect VZ information can broaden the attack surface of the affected software for extended periods.

Next, when a vulnerability is reported with a CVE ID, it is easier for developers to patch the vulnerability. Thus, the detection time $t_d - t_r$ (and, $t'_d - t_r$) was shorter than the detection time $t_d - t_i$ (and, $t'_d - t'_i$). In addition, even after CVE was published at t_r, we found that the VZ sometimes required additional time to patch the vulnerability, because some VZ authors released a patched software after CVE was published.

Another interesting result was that even if a CVE has the correct VZ, the affected software required more than 300 days to patch the propagated vulnerability. Presumably, developers seemed to be not urgent to patch a vulnerability when it is triggered only in the corner case because patching a vulnerability, *e.g.*, updating a vulnerable third-party software, needs considerable costs and efforts.

To investigate the overall distribution, we approximated the obtained result to a normal distribution. We selected the rank as 100 (days) and measured the standard deviation (s) for each of the six categories in Table 7, where all the average values (μ) have already been measured. We assume that the elapsed time for vulnerability detection follows a normal distribution with μ and s, and plot the normal distribution graphs. The probability density function (*i.e.*, $f(x)$) we used is as follows:

$$f(x) = \frac{1}{s\sqrt{2p}} e^{-(x-\mu)^2 / 2s^2}$$

The plotted graphs are illustrated in Figure 7. From the graphs, we intuitively confirmed that a propagated CVE with the correct VZ, overall, is patched almost a year after CVE is published. In addition, we discovered that when a CVE has the incorrect VZ information, the elapsed time for the affected software to detect the propagated CVE is longer than that of when a CVE with the correct VZ.

We summarize the two key points from our experimental results: (1) CVEs reported with the incorrect VZ took longer to detect the propagated vulnerabilities, and (2) the affected software programs are often unaware of the existence of CVEs due to the incorrect VZ. This demonstrates the importance and necessity of VZ discovery to enable developers to detect and patch propagated vulnerabilities in a timely manner.

6.3 Case study: Vulnerability in Redis

Redis is widely reused in-memory database. In 2015, a dangerous vulnerability (CVE-2015-8080) was discovered in Lua (VZ); however, it was reported as a vulnerability in Redis, which is reusing Lua, and only the Redis team patched the vulnerability in 2015. This is because the Redis team determined that vulnerability only affects their program as they incorporated Lua into their environment. Interestingly, in 2018, the Redis team updated Lua for security purposes, but kept the version of Lua with the vulnerable code. Even more seriously, by inserting a single line of proof of concept into Redis, an integer overflow vulnerability is reproduced, which can be extended to a denial-of-service attack. We reported this case to Redis and they accepted our patch request in February 2020. This resulted in a new CVE ID (CVE-2020-14147).

7 Related work

In this section, we introduce a number of related studies.

Detecting code clones. There are many approaches attempting to detect source code clones [4, 11, 18, 20, 21, 26, 30–33, 35, 37, 42, 43, 46–49, 53, 55]. However, these approaches do not consider discovering the VZ of software vulnerabilities, but only focus on detecting clones in the specific snapshot of a software program via code scanning.

Tracing the code history. Some approaches attempted to trace the history of the source code to improve software maintenance [10, 17, 19, 22, 23, 25, 50, 51]. However, tracing the history of source code is fairly different from detecting the history of a vulnerability; the way to clearly distinguish between vulnerable code and patched code is needed, but it is not discussed in those approaches.

Verifying reliability of vulnerability reports. Existing approaches that aimed to verify the reliability of public vulnerability reports can be divided into three categories: identifying *missing* information in the report [2, 3, 5, 36], analyzing the *reproducibility* of vulnerabilities [14, 34], and ensuring *consistency* of the report [8]. Some of them showed that the additional information in the reports can help counter vulnerabilities [2, 36] and bugs [3, 5]. Guo *et al*. [14] analyzed the characteristics of reports to efficiently fix bugs, and Mu *et al*. [34] showed that there was not sufficient information in the reports to reproduce the vulnerability in order to analyze the cause of the problem. Dong *et al*. [8] focused on the consistency of the data provided by the vulnerability databases; they attempted to detect inconsistencies between the affected software mentioned in the description of vulnerability reports and the corresponding CPE. Although these approaches addressed the reliability and consistency of the information provided by the public vulnerability reports, they did not handle our targeted issue, discovering the correct VZ for a vulnerability.

8 Discussion

Here we discuss several considerations related to `VOFinder`.

Inferring the main causes of an incorrect VZ. From our experiment results, we confirmed that there were two main causes for the occurrence of an incorrect VZ: (1) lack of a proper tool to discover the VZ and (2) reuse of third-party software with modification. Currently, there is no automated tool that is capable of discovering the VZ. Therefore, when software authors received a vulnerability report, they often registered the vulnerability as theirs without confirming the VZ. In particular, when receiving reports through Bugzilla, the trend of having CVEs with the incorrect VZ is prominent. In addition, a third-party software is often modified prior to reuse in other software. Thus, when a vulnerability is reported to software program developers, they easily misinterpret that the vulnerability only manifests in their program, even though the vulnerable code belongs to a third-party software.

Our suggestions. Based on our experiment results, we suggest that the task of finding the VZ of vulnerabilities, primarily a manual task, should be automated and accurately performed with a system such as `VOFinder`. In addition, `VOFinder` can be integrated with other techniques, such as Software Bill of Materials (SBoM) [52], which declares the components and pieces that the software was built with. We expect to enhance software security by (1) analyzing the components of a software through SBoM, and (2) identifying whether those components are affected by the VZ determined by `VOFinder` based on an existing or newly registered CVE.

Reliability of our evaluation. In our evaluation, we determined a VZ is correct if the VZ discovered by `VOFinder` for a certain vulnerability is included in the corresponding CPE. To validate this, we manually analyzed a subset of the VZ discovery results. Among the 2,807 CVEs where the CPE and the VZ discovered by `VOFinder` coincided, we randomly selected 20 CVEs per year from 2006 to 2020, *i.e*., 280 CVEs (10%) out of 2,807 CVEs. We then analyzed whether the VZ discovered by `VOFinder` is correct, especially by referring to the patch code and external links in the NVD reference. As a result, we confirmed that for all cases but one, the VZ discovered by `VOFinder` that are contained in the CPE is the correct VZ of each vulnerability. In one case, `VOFinder` and the CPE determined that `MySQL` was the VZ for CVE-2009-4484, but it was actually a vulnerability that originated in `YaSSL`, which is not included in the dataset because it does not satisfy our dataset collection criteria (*i.e*., the number of stars on GitHub is less than 100). Even if this error ratio (*i.e*., less than 0.4%) extends to the entire CVE dataset, we can verify the VZ discovered by `VOFinder` with more than 99% confidence. Therefore, we concluded that our assumption is valid that considering the VZ discovered by `VOFinder` is correct when it is contained in the CPE.

Use case: vulnerability reporting process. `VOFinder` can be adopted in the vulnerability reporting process. First, security analysts who find a new vulnerability can report it to a CNA after discovering the VZ using `VOFinder`. Next, when a CNA verifies the reported vulnerabilities, `VOFinder` can be applied to detect affected software as well as the VZ. In fact, we have contacted a CNA that is currently manually discovering the VZ for a vulnerability. Based on an initial discussion, `VOFinder` can be integrated into their workflow, such as discovering a VZ using `VOFinder` when verifying the vulnerability before assigning the CVE ID.

Future work. First, we will consider the vulnerability propagation arising from copying and pasting small pieces of code, *e.g*., provided by Q&A fora. To address them, features other than the reuse relation identification should be selected; we are investigating an extension technique to deal with such cases. Second, since the methodology of `VOFinder` can be applied to other languages, we will discover VZs for other languages. This will be conducted by constructing the CVE and software pool for the new languages.

9 Conclusion

As public vulnerability reports are widely utilized to resolve threats arising from software vulnerabilities, quality control of reports is emerging. In response, we presented `VOFinder`, a precise mechanism to discover the VZ of software vulnerabilities. We found that the current NVD provides the incorrect VZ for some CVEs, and by analyzing the adverse effects of

those CVEs in detail, we demonstrated the importance and necessity of VZ discovery. Equipped with VZ discovery results from `VOFinder`, developers can address software vulnerabilities potentially included in their software due to vulnerable code reuse (*e.g.*, via third-party software), more specifically, they can apply appropriate security patches.

Acknowledgment

We appreciate the anonymous reviewers for their valuable comments to improve the quality of the paper. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01697 Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security and IITP-2021-2020-0-01819 ICT Creative Consilience program).

Availability

The source code of `VOFinder` is available at https://github.com/wooseunghoon/VOFinder-public.

References

[1] Jorge Aranda and Gina Venolia. The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 298–308, 2009.

[2] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful... really? In *2008 IEEE International Conference on Software Maintenance (ICSME)*, pages 337–345, 2008.

[3] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM Conference on Computer-Supported Cooperative Work*, pages 301–310, 2010.

[4] Lutz Büch and Artur Andrzejak. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104, 2019.

[5] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 396–407, 2017.

[6] Common Weakness Enumeration. 2020 CWE Top 25 Most Dangerous Software Weaknesses, 2020. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.

[7] Ctags. Universal Ctags, 2020. https://github.com/universal-ctags/ctags.

[8] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX Security Symposium (USENIX Security)*, pages 869–885, 2019.

[9] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2169–2185, 2017.

[10] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. Some from here, some from there: Cross-project code reuse in github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, pages 291–301, 2017.

[11] Mohammad Gharehyazie, Baishakhi Ray, Mehdi Keshani, Masoumeh Soleimani Zavosht, Abbas Heydarnoori, and Vladimir Filkov. Cross-project code clones in GitHub. *Empirical Software Engineering*, pages 1–36, 2018.

[12] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. 99(6):518–529, 1999.

[13] GitHub. Securing software, together, 2021. https://github.com/features/security.

[14] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 495–504, 2010.

[15] Irfan Ul Haq, Sergio Chica, Juan Caballero, and Somesh Jha. Malware lineage in the wild. *Computers & Security*, 78:347–363, 2018.

[16] Shohei Ikeda, Akinori Ihara, Raula Gaikovina Kula, and Kenichi Matsumoto. An Empirical Study of README contents for JavaScript Packages. *IEICE TRANSACTIONS on Information and Systems*, 102(2):280–288, 2019.

[17] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. Where does this code come from and where does it go?-integrated code history tracker for open source systems. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 331–341, 2012.

[18] Jiyong Jang, Abeer Agrawal, and David Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *2012 IEEE Symposium on Security and Privacy (SP)*, pages 48–62, 2012.

[19] Jiyong Jang, Maverick Woo, and David Brumley. Towards automatic software lineage inference. In *22nd USENIX Security Symposium (USENIX Security)*, pages 81–96, 2013.

[20] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 96–105, 2007.

[21] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro 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.

[22] Tetsuya Kanda, Takashi Ishio, and Katsuro Inoue. Extraction of product evolution tree from source code of product variants. In *Proceedings of the 17th International Software Product Line Conference (SPLC)*, pages 141–150, 2013.

[23] Tetsuya Kanda, Takashi Ishio, and Katsuro Inoue. Approximating the Evolution History of Software from Source Code. *IEICE Transactions on Information and Systems*, 98(6):1185–1193, 2015.

[24] Georgia M Kapitsaki, Nikolaos D Tselikas, and Ioannis E Foukarakis. An insight into license tools for open source software systems. *Journal of Systems and Software*, 102:72–87, 2015.

[25] Naohiro Kawamitsu, Takashi Ishio, Tetsuya Kanda, Raula Gaikovina Kula, Coen De Roover, and Katsuro Inoue. Identifying source code reuse across repositories using LCS-based source code similarity. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 305–314, 2014.

[26] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, 2017.

[27] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.

[28] Amanda Lee and Travis Atkison. A comparison of fuzzy hashes: evaluation, guidelines, and future suggestions. In *Proceedings of the SouthEast Conference*, pages 18–25, 2017.

[29] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

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

[31] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[32] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. 4(19):289–302, 2004.

[33] Dmitry Luciv, Dmitrij Koznov, George Chernishev, Hamid Abdul Basit, Konstantin Romanovsky, and Andrey Terekhov. Duplicate finder toolkit. In *Proceedings of the 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 171–172, 2018.

[34] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security)*, pages 919–936, 2018.

[35] Ginger Myles and Christian Collberg. Detecting software theft via whole program path birthmarks. In *International Conference on Information Security*, pages 404–415. Springer, 2004.

[36] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 692–708, 2015.

[37] Manziba Akanda Nishi and Kostadin Damevski. Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software*, 137:130–142, 2018.

[38] NPM. Node Package Manager, 2020. https://www.npmjs.com/.

[39] NVD. National Vulnerability Database, 2020. https://nvd.nist.gov/.

[40] NVD. Common Platform and Enumeration (CPE), 2021. https://nvd.nist.gov/products/cpe.

[41] Jonathan Oliver, Chun Cheng, and Yanggui Chen. TLSH–a locality sensitive hash. In *Proceedings of the 2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13, 2013.

[42] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 426–437, 2015.

[43] Chaiyong Ragkhitwetsagul and Jens Krinke. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering*, pages 2236–2284, 2019.

[44] RhodeCode. Version Control Systems Popularity, 2016. https://rhodecode.com/insights/version-control-systems-2016.

[45] Vassil Roussev. Hashing and data fingerprinting in digital forensics. *IEEE Security & Privacy*, 7(2):49–55, 2009.

[46] Chanchal K Roy and James R Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, 2008.

[47] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina Lopes. Oreo: Detection of Clones in the Twilight Zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 345–365, 2018.

[48] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, 2016.

[49] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization. In *24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 654–659, 2017.

[50] Francisco Servant and James A Jones. Fuzzy fine-grained code-history analysis. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 746–757, 2017.

[51] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Incremental origin analysis of source code files. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 42–51, 2014.

[52] National Telecommunications and Information Administration. NTIA Software Component Transparency with SBOM (Software Bill of Materials), 2020. https://www.ntia.doc.gov/SoftwareTransparency.

[53] Tijana Vislavski, Gordana Rakic, Nicolás Cardozo, and Zoran Budimac. LICCA: A tool for cross-language clone detection. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 512–516, 2018.

[54] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs.Testers: A Comparison of Software Vulnerability Discovery Processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 374–391, 2018.

[55] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. CCAligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 1066–1077, 2018.

[56] Wikipedia. Index case (or patient zero), 2020. https://en.wikipedia.org/wiki/Index_case.

[57] Michael Worobey, Thomas D Watts, Richard A McKay, Marc A Suchard, Timothy Granade, Dirk E Teuwen, Beryl A Koblin, Walid Heneine, Philippe Lemey, and Harold W Jaffe. 1970s and 'Patient 0' HIV-1 genomes illuminate early HIV/AIDS history in North America. *Nature*, 539(7627):98–101, 2016.

[58] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *29th USENIX Security Symposium (USENIX Security)*, pages 1165–1182, 2020.

Appendix A Results of the incorrect VZs

We introduce a list of CVEs that provides the incorrect VZ, along with the correct VZ discovered by `VOFinder` for each CVE. Note that there was no case where the reproduction was succeeded only in the incorrect VZ provided by CPE; the reproduction either was succeeded in both the VZ discovered by `VOFinder` and that of CPE or was failed in both of them due to the reasons specified in Section 5.1.

∗ **CPE**: Show only one parent software among the CPE-registered software (determined by the result of `VOFinder`).

∗∗**VZ**: Discovered VZ using `VOFinder`.

† **Verification**: How to verify the VZ (see Section 5.1).

Table 8: **50 CVEs** with wrong **versions**.

CVE ID	CPE∗		VZ∗∗	Verification†
	Software	Version	**Version**	
CVE-2007-1320	Qemu	v0.8.2	v0.6.0	Code review
CVE-2008-4225	Libxml2	v2.7.2	v2.2.6	Code review
CVE-2009-3720	Expat	v2.0.1	v2.0.0	Code review
CVE-2012-1013	Krb5	v1.8	v1.7.1	Code review
CVE-2014-0185	PHP	v5.5.0	v5.3.4	Author sites
CVE-2014-3985	Miniupnp	v1.9	v1.8	Code review
CVE-2014-4344	Krb5	v1.10	v1.5	Code review
CVE-2014-8116	File	v5.20	v5.19	Code review
CVE-2015-3395	FFmpeg	v2.0.6	n0.10.8	Author sites
CVE-2016-3705	Libxml2	v2.9.3	v2.9.0	Reproduction
CVE-2016-8687	Libarchive	v3.2.1	v3.1.0	Code review
CVE-2016-8688	Libarchive	v3.2.1	v3.1.0	Reproduction
CVE-2016-9118	OpenJPEG	v2.1.2	v2.1.1	Code review
CVE-2016-9388	Jasper	v1.900.14	v1.900.1	Reproduction
CVE-2016-9535	Libtiff	v4.0.6	v4.0.5	Code review
CVE-2016-9573	OpenJPEG	v2.1.2	v2.1.1	Reproduction
CVE-2016-10269	Libtiff	v4.0.7	v4.0.0	Reproduction
CVE-2016-10270	Libtiff	v4.0.7	v4.0.0	Reproduction
CVE-2017-5225	Libtiff	v4.0.7	v4.0.0	Reproduction
CVE-2017-5601	Libarchive	v3.2.2	v3.0.4	Reproduction
CVE-2017-6420	ClamAV	v0.99.2	v0.99.1	Code review
CVE-2017-7407	Curl	v7.53.1	v6.5.0	Author sites
CVE-2017-7746	Wireshark	v2.0.0	v1.99.9	Code review
CVE-2017-9047	Libxml2	v2.9.4	v2.6.20	Code review
CVE-2017-9226	Oniguruma	v6.2.0	v5.9.6	Code review
CVE-2017-9227	Oniguruma	v6.2.0	v5.9.6	Code review
CVE-2017-9229	Oniguruma	v6.2.0	v5.9.6	Code review
CVE-2017-11462	Krb5	v1.8	v1.14	Reproduction
CVE-2017-14054	FFmpeg	v3.3.3	v3.1	Code review
CVE-2017-14152	OpenJPEG	v2.2.0	v2.1.1	Code review
CVE-2017-14164	OpenJPEG	v2.2.0	v2.1.1	Code review
CVE-2017-14169	FFmpeg	v3.3.3	n2.5	Reproduction
CVE-2017-14170	FFmpeg	v3.3.3	n2.6	Code review
CVE-2017-14222	FFmpeg	v3.3.3	n2.5.3	Code review
CVE-2017-14223	FFmpeg	v3.3.3	n2.6	Code review
CVE-2017-14502	Libarchive	v3.3.2	v3.1.900a	Code review
CVE-2017-17081	FFmpeg	v3.4	n2.3	Code review
CVE-2017-17480	OpenJPEG	v2.3.0	v1.1	Code review
CVE-2017-1000249	File	5_29	5_22	Code review
CVE-2018-5727	OpenJPEG	v2.3.0	v2.1.2	Code review
CVE-2018-5785	OpenJPEG	v2.3.0	v2.1.1	Reproduction
CVE-2018-16790	Libbson	v1.12.0	v1.9.0	Code review
CVE-2018-18088	OpenJPEG	v2.3.0	v2.2.0	Code review
CVE-2019-9718	FFmpeg	v4.1	n3.5	Code review
CVE-2019-12973	OpenJPEG	v2.3.1	v2.3.0	Reproduction
CVE-2019-13224	Oniguruma	v6.9.2	v6.4.0	Code review
CVE-2019-15681	Libvncserver	v0.9.12	v0.9.8	Code review
CVE-2019-19317	Sqlite	v3.30.1	v3.26.0	Code review
CVE-2019-1010239	cJSON	v1.7.8	v1.7.4	Reproduction
CVE-2020-7595	Libxml2	v2.9.10	v2.9.6	Code review

Table 9: **46 CVEs** with wrong **software programs**.

CVE ID	CPE∗	VZ∗∗		Verification†
	Software	**Software**	Version	
CVE-2006-5748	Firefox	JS engine	N/A	Code review
CVE-2009-0774	Firefox	JS engine	N/A	Code eview
CVE-2009-2466	Firefox	JS engine	N/A	Code eview
CVE-2009-2663	Firefox	Vorbis	v1.0.1	Code eview
CVE-2009-3379	Firefox	Vorbis	v1.0.0	Code review
CVE-2010-3176	Firefox	Vorbis	v1.0.0	Code review
CVE-2011-3026	Chrome	Libpng	v1.2.44	Author sites
CVE-2011-3045	Chrome	Libpng	v1.2.44	Author sites
CVE-2011-3439	iOS	Freetype	v2.4.5	Author sites
CVE-2011-3893	Chrome	FFmpeg	n0.5.8	Author sites
CVE-2013-0760	Firefox	Uchardet	v0.0.2	Code review
CVE-2013-0894	Chrome	FFmpeg	n0.10.6	Author sites
CVE-2013-6629	Chrome	Libjpeg	6b	Code review
CVE-2013-7226	PHP	Libgd	v2.1.0	Author sites
CVE-2014-0237	PHP	File	5_06	Author sites
CVE-2014-3710	PHP	File	5_16	Author sites
CVE-2014-6262	Zenoss_core	RRDtool	v1.5.0	Author sites
CVE-2015-2756	Debian Linux	Qemu	v2.3.0	Code review
CVE-2015-4335	Redis	Lua	v5.1	Reproduction
CVE-2015-5165	Xen	Qemu	v2.3.0	Code review
CVE-2015-8080	Redis	Lua	v5.3	Reproduction
CVE-2015-8865	PHP	File	4_26	Author sites
CVE-2016-1624	Chrome	Brotli	v0.1.0	Code review
CVE-2016-1626	Chrome	OpenJPEG	v2.0	Code review
CVE-2016-1968	Firefox	Brotli	v0.1.0	Code review
CVE-2016-2464	Android	Libwebm	v1.0.0.26	Code review
CVE-2016-2808	Firefox	Spidermonkey	v25	Author sites
CVE-2016-4477	Android	Wpa_supplicant	v0.4.0	Code review
CVE-2016-5152	Chrome	OpenJPEG	v2.0.1	Reproduction
CVE-2016-5257	Firefox	FFmpeg	n3.1	Code review
CVE-2017-0381	Android	Libopus	draft-09	Code review
CVE-2017-0386	Android	Libnl	v2.0	Code review
CVE-2017-0393	Android	Libvpx	v1.3.0	Code review
CVE-2017-0408	Android	JPEG-compressor	v0.1	Code review
CVE-2017-0553	Android	Libnl	v2.0	Code review
CVE-2017-0663	Android	Libxml2	v2.7.0	Code review
CVE-2017-0700	Android	JPEG-compressor	v0.1	Reproduction
CVE-2017-5056	Chrome	Libxml2	v2.2.6	Code review
CVE-2017-6983	iOS	Sqlite	v3.7.11	Code review
CVE-2017-13693	Linux Kernel	Acpica	20170629	Code review
CVE-2017-13695	Linux Kernel	Acpica	20170629	Code review
CVE-2018-5146	Firefox	Vorbis	v1.2.0	Reproduction
CVE-2018-5711	PHP	Libgd	v2.1.0	Reproduction
CVE-2018-6064	Chrome	V8	v5.1.219	Reproduction
CVE-2019-9278	Android	Libexif	v6.21	Code review
CVE-2019-17371	Libpng	Gif2png	v2.5.13	Reproduction

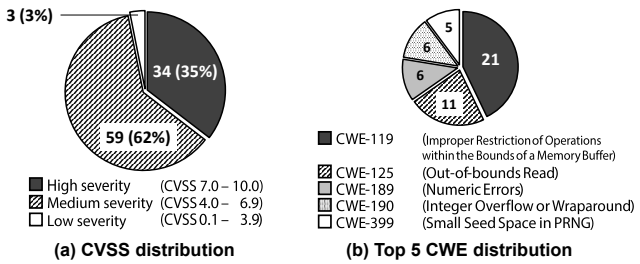


Figure 8: CVSS and CWE distributions for the CVEs with the incorrect VZ.

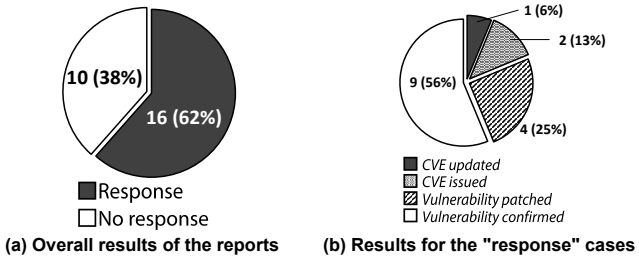


Figure 9: Depiction of the vendors' responses.

Appendix B Analysis for the mislabeled CVEs

`VOFinder` identified that 96 CVEs that have wrong information related to their respective VZs. We analyzed the severity (*i.e.*, Common Vulnerability Scoring System, shortly CVSS) and type for these CVEs (*i.e.*, Common Weakness Enumeration, shortly CWE); the analysis results are shown in Figure 8.

In terms of the severity, approximately a third of the total was a high-risk vulnerability (see Figure 8 (a)). Of course, when we successfully reproduce and report high-risk vulnerabilities, most of them were immediately patched (*e.g.*, CVE-2017-0700) or managed with a CVE ID assigned (*e.g.*, CVE-2020-14147).

In the perspective of the vulnerability type, as shown in the Figure 8 (b), the most frequently appeared type is the vulnerability related to the boundary of the buffer (CWE-119, *e.g.*, buffer-overflow vulnerability). In particular, CWE-119, CWE-125, and CWE-190 belong to the top 25 most dangerous vulnerability types in 2020 [6].

Appendix C Results of the vendors' responses

We classify our reporting process into two categories: *response* and *no response* (see Figure 9). Among the total of 26 reports, we have received responses from 16 vendors:

CVE updated (1 case). This is a case where the CPE was changed to the discovered VZ by `VOFinder`.

CVE issued (2 cases). This is a case where a new CVE ID has been assigned. Of course, the vulnerability was patched before it was issued.

Vulnerability patched (4 cases). This is a case where the vulnerability that existed in the latest version of the affected software due to the incorrect VZ was patched.

Vulnerability confirmed (9 cases). This is a case where the vendors confirmed the vulnerability but decided that the vulnerability was not critical (*e.g.*, a corner case); thus, they would not patch immediately but considered patching the later versions if applicable.

Lastly, for the no response cases, we are still trying to communicate with the corresponding vendors.