V1SCAN: Discovering 1-day Vulnerabilities in Reused C/C++ Open-source Software Components Using Code Classification Techniques

Seunghoon Woo, Eunjin Choi, Heejo Lee*, Hakjoo Oh
Korea University, {seunghoonwoo, silver_jin, heejo, hakjoo_oh}@korea.ac.kr

Abstract

We present V1SCAN, an effective approach for discovering 1-day vulnerabilities in reused C/C++ open-source software (OSS) components. Reusing third-party OSS has many benefits, but can put the entire software at risk owing to the vulnerabilities they propagate. In mitigation, several techniques for detecting propagated vulnerabilities, which can be classified into version- and code-based approaches, have been proposed. However, state-of-the-art techniques unfortunately produce many false positives or negatives when OSS projects are reused with code modifications.

In this paper, we show that these limitations can be addressed by improving version- and code-based approaches and synergistically combining them. By classifying reused code from OSS components, V1SCAN only considers vulnerabilities contained in the target program and filters out unused vulnerable code, thereby reducing false alarms produced by version-based approaches. V1SCAN improves the coverage of code-based approaches by classifying vulnerable code and then detecting vulnerabilities propagated with code changes in various code locations. Evaluation on GitHub popular C/C++ software showed that V1SCAN outperformed state-of-the-art vulnerability detection approaches by discovering 50% more vulnerabilities than they detected. In addition, V1SCAN reduced the false positive rate of the simple integration of existing version- and code-based approaches from 71% to 4% and the false negative rate from 33% to 7%. With V1SCAN, developers can detect propagated vulnerabilities with high accuracy, maintaining a secure software supply chain.

1 Introduction

Open-source software (OSS) reuse has become an indispensable trend in software development [9, 36, 41]. In addition to the benefits of using existing functionalities, reusing OSS increases the reliability of software because OSS is often more exposed to verification by multiple parties.

* Heejo Lee is the corresponding author.

However, improper OSS reuse may compromise the security of the entire system (*e.g.*, vulnerability propagation [9, 14, 15, 34]). To obviate the threats imposed by unmanaged OSS reuse, many approaches have been proposed for detecting vulnerabilities in OSS components. These approaches are mainly classified into *version-* and *code-based* techniques.

- *A version-based approach* discovers vulnerabilities by identifying OSS versions [9, 36, 40]. It determines whether the versions contain known vulnerabilities such as Common Vulnerabilities and Exposures (CVE).
- *A code-based approach* detects propagated vulnerabilities by identifying codes that are syntactically or semantically similar to vulnerable code [12, 14, 34, 37].

Although resolving vulnerabilities is crucial for securing software, existing version- and code-based approaches easily produce false positives or negatives, resulting in impeding an effective vulnerability management process. This is primarily owing to the recent tendency of developers to typically reuse OSS with code or structural modifications [9, 36].

Limitations of existing approaches. Existing version-based approaches (*e.g.*, [9, 36, 40]) easily produce false positives because they fail to consider the impact of modified OSS reuse on vulnerability propagation, and thus misinterpret the unused vulnerable code as still being present in the target program. In contrast, existing code-based approaches (*e.g.*, [12, 14, 34, 37]) easily report false negatives when the syntax of the vulnerable code is modified. Moreover, because these approaches focus on a specific granularity (*e.g.*, function units), they miss vulnerabilities that exist outside the selected granularity.

Our approach. To overcome their shortcomings, we present V1SCAN, a novel approach for the precise and comprehensive discovery of 1-day security vulnerabilities propagated as a result of the reuse of C/C++ OSS components.

We devise a new way to combine version- and code-based approaches to take full advantage of their strengths and avoid weaknesses. The main idea of V1SCAN, which is significantly distinguishable from existing approaches, is to detect propagated vulnerabilities using the code classification techniques.

Given a target program, V1SCAN first detects vulnerabilities based on the identified OSS versions. To focus only on reused vulnerabilities, V1SCAN uses a *reused code classification* technique that divides reused OSS functions into three groups: `exactly reused`, `changed`, and `unused`. By filtering out unused and repaired vulnerable functions, V1SCAN reduces false alarms of the existing version-based approaches.

V1SCAN complements vulnerability detection results using a code-based approach. To expand vulnerability detection coverage, V1SCAN first clarifies vulnerability locations (`function`, `structure`, `macro`, or `variable`) using a *vulnerable code classification* technique, and then detects propagated vulnerabilities for each location. Here, V1SCAN only considers the core lines directly related to the vulnerability to counter code changes caused by modified OSS reuse, thereby addressing false negatives of existing code-based approaches.

Finally, V1SCAN confirms the propagated vulnerabilities in the target program by consolidating the results of the improved version- and code-based approaches.

Evaluation. For the experiment, we collected 4,612 CVE patches from the National Vulnerability Database (NVD) [22] and gathered Common Platform Enumeration (CPE) [21] on 137,892 CVE vulnerabilities. We evaluated V1SCAN using popular C/C++ software on GitHub.

When we applied the V1SCAN, MOVERY [34], and `VOFinder` [35] to the selected ten target programs, V1SCAN detected 50% more vulnerabilities (137 vulnerabilities) than the existing approaches, achieving a precision of 96% and a recall of 91% (see Section 5.1). It is remarkable that V1SCAN could reduce the false negative rate of MOVERY and `VOFinder` from 40% and 55% to 9%. Furthermore, V1SCAN could discover more propagated vulnerabilities than the combined results of existing version- [14] and code-based [36] approaches, while reducing the false positive rate from 71% to 4% and the false negative rate from 33% to 7% (see Section 5.2). Moreover, when we applied V1SCAN to 4,434 popular C/C++ software (ranging from 1K to 20M lines of code), V1SCAN could detect vulnerabilities within 20 seconds for each software (excluding preprocessing time), indicating that V1SCAN is sufficiently fast for practical use (see Section 5.3).

Contributions. We summarize our contributions below.

- We propose V1SCAN, a new approach for precisely detecting 1-day vulnerabilities in a target program. The key technical contribution is the effective integration of version- and code-based approaches by leveraging the classification of reused and vulnerable code.
- We examined the manner in which modified OSS reuse affects the vulnerability discovery process, and propose an effective solution for detecting vulnerabilities in modified C/C++ OSS components.
- V1SCAN discovered 137 vulnerabilities in ten target programs with 96% precision and 91% recall, thereby outperforming existing approaches.

2 Terminology and motivation

2.1 Basic terminology

We first clarify the following terms used throughout the paper.

- **OSS reuse.** This refers to the process of utilizing all or some of the OSS functions (*e.g.*, copying and pasting of functions from third-party OSS projects) [9, 36].
- **OSS component.** This refers to an entire OSS project or sometimes a portion of an OSS codebase that is reused in a target program [36].
- **OSS version.** We define that an OSS version follows the three-component semantic versioning notation with `major.minor.patch` by default [7].
- **1-day (or N-day) vulnerability.** This refers to a vulnerability that is known to responsible developers and for which a corresponding security patch has been released [9, 38]. These vulnerabilities can propagate to other software due to OSS reuse, compromising the security of the entire system.

2.2 Code classification

For precise vulnerability detection, we leverage the classification of reused and vulnerable code. This is because false positives and negatives may occur in vulnerability discovery if various types of reused and vulnerable code are not properly classified and considered (see Section 2.3). Figure 1 depicts the code classification defined in this paper.

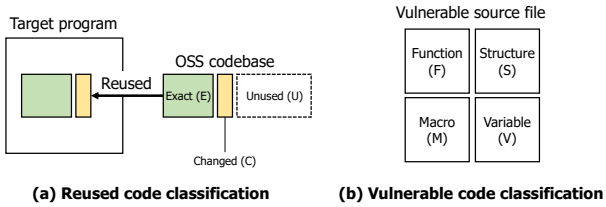
Reused code classification. When developers reuse an OSS project, they often modify the code of the original OSS project [9, 36]. We classified the types of OSS codes that were reused into the following three categories, according to the extent of modification of the code.

- **Exactly reused (E):** OSS code (*e.g.*, functions or files) is reused without any code changes.
- **Changed (C):** OSS code is reused with code changes.
- **Unused (U):** OSS code is not reused (not included in the codebase of the target program).

Vulnerable code classification. Security vulnerabilities can exist in various locations in the source code. Specifically, we consider the following four code locations.

- **Function (F)**. *e.g.*, `void main (...) {...}`
- **Structure (S)**. *e.g.*, `struct oss {...};`
- **Macro (M)**. *e.g.*, `#define ...`
- **Variable (V)**. *e.g.*, `const int ...;`

When we examined 4,612 C/C++ security patches, code patches applied to the selected four code locations accounted for over 97% (see Section 4.3). Therefore, we determined that considering these four code locations could cover C/C++ security patches comprehensively.



(a) Reused code classification **(b) Vulnerable code classification**

Figure 1: Depiction of the code classification.

2.3 Problem statement

In this paper, we focus on the problem in which existing version- and code-based approaches fail to precisely detect 1-day vulnerabilities in target software, leaving threats unmitigated and making the vulnerability management process ineffective. Two main issues with the existing version- and code-based approaches are summarized as follows.

- Version-based approaches produce many *false positives* because they do not consider the modification of reused vulnerable code.
- Code-based approaches report many *false negatives* because they do not consider code locations in which vulnerabilities exist.

First, although 57% of the C/C++ OSS codebase is changed or excluded from OSS reuse in practice [36], existing version-based approaches (*e.g.*, [9, 36, 40]) fail to consider the impact of modified reuse on vulnerability propagation. Even if the vulnerable code is excluded during OSS reuse, or is repaired by backporting the security patch, existing version-based approaches fail to filter out this and produce false alarms.

Moreover, version-based approaches can produce false results when component versions are misidentified. Because OSS codes often undergo modifications during the reuse process, it is challenging to map one version to the entire codebase of the reused component (*e.g.*, code from multiple OSS versions can be mixed) [36]. Leveraging a Software Bill of Materials (SBoM) [29] also presents challenges, as the current SBoM for C/C++ mostly lacks consideration for modified reuse; it maps a single version to an OSS component and can thus produce false results in vulnerability discovery.

On the other hand, existing code-based approaches (*e.g.*, [12, 14, 34, 37]) report many false negatives because they mainly focus on detecting propagated vulnerabilities within a specific granularity (*e.g.*, function units). Furthermore, they do not precisely identify propagated vulnerable code with various syntaxes (*e.g.*, propagated with code changes).

2.4 Motivating example

We attempted to discover 1-day vulnerabilities in ReactOS v0.4.13, which is a free Windows-compatible operating system. We used V1SCAN, version-based, and code-based approaches and then examined each vulnerability detection result. Table 1 summarizes the vulnerability detection results.

Table 1: Vulnerability detection results for ReactOS using the V1SCAN, version-based, and code-based approaches. We measured FNs by considering all TPs discovered in the three approaches as the ground truth.

Approach	#TP*	#FP†	#FN‡	Precision	R§
Version-based approach (V)	5	47	22	10%	19%
Version-based approach (manually correcting versions)	20	29	7	41%	74%
Code-based approach (C)	13	8	14	62%	48%
Union of the version- and code-base approaches ($V \cup C$)	15	55	12	21%	56%
V1SCAN	**26**	**1**	**1**	**96%**	**96%**

#TP*: #True positives, #FP†: #False positives, #FN‡: #False negatives, R§: TP detection rate (#TP / (#TP + #FN)).

Version-based approach. We used CENTRIS [36], a recent approach for precisely identifying modified OSS components. CENTRIS identified 23 C/C++ OSS components and versions thereof in ReactOS. Subsequently, we investigated the CVE vulnerabilities contained in the components using the CPE of NVD [21], which specifies the software affected by each CVE (details are introduced in Section 3.3).

Consequently, 10 components, including LibTIFF and MbedTLS, were identified as vulnerable, with 52 CVE vulnerabilities. However, when we manually examined these 52 CVEs, we observed that the version-based approach produced *many false positives*: 47 CVEs (90%) were false positives, whereas only five CVEs were identified correctly. Most false positives were the result of incorrect predictions of the component version (see Section 2.3).

Despite our attempt to rectify the problem by manually correcting the component versions, the version-based approach still produced 29 false positives. This is because the version-based approach misinterprets that the target program still contains unused vulnerable code or vulnerable code repaired by backporting security patches. For example, the ReactOS team resolved three CVE vulnerabilities contained in Libtirpc v0.1.11 by backporting the security patches, instead of updating the entire Libtirpc to a safe version. However, the version-based approach failed to filter out the resolved vulnerabilities, misconfirming that the three CVEs were still included in ReactOS (*i.e.*, producing false positives).

Code-based approach. We used VUDDY [14], a function-based vulnerable code detection approach that was specifically developed for the discovery of 1-day vulnerabilities. We tested VUDDY using 4,612 CVE patches (see Section 4.3) in abstraction mode, which makes VUDDY robust to changes in parameters, local variables, and called function names.

When we applied VUDDY to ReactOS, we confirmed that VUDDY reported many *false negatives*, which we attributed to the syntax diversity of the vulnerable code. VUDDY could discover only 13 (48%) of 27 vulnerabilities and failed to detect 14 vulnerable functions. Our measurement of the Jaccard similarity [32] (considering a function as a set of lines) be-

tween the disclosed vulnerable functions and the vulnerable functions that existed in ReactOS, indicated a similarity of less than 60% in the six cases. Moreover, because VUDDY is not designed to detect vulnerabilities that exist outside functions, it missed two vulnerabilities.

Finally, VUDDY produced eight false positives, all of which were caused by abstraction. In the case of a security patch that changes only its abstraction targets, VUDDY cannot distinguish between vulnerable and patched functions, thereby producing false positives.

V1SCAN. V1SCAN discovered 26 vulnerabilities in ReactOS while eliminating most false positives and negatives produced by the existing approaches. It is noteworthy that V1SCAN detected more vulnerabilities with significantly fewer false positives than the union results of existing approaches.

Here we introduce the LibTIFF v4.0.10 case, which is reused in ReactOS. V1SCAN could filter out vulnerabilities of LibTIFF that were not reused in ReactOS (*e.g.*, CVE-2020-35521). In addition, V1SCAN detected the CVE-2018-19210 vulnerability that was not discovered in the version-based approach, because the CPE provides an incorrect version (v4.0.9). Moreover, the CVE-2019-14973 vulnerability, in which the security patch modified outside functions, was not detected by the code-based approach but discovered in V1SCAN. Note that the aforementioned vulnerabilities have been patched in the latest version of ReactOS.

3 Design of V1SCAN

In this section, we describe the design of V1SCAN, which can precisely detect propagated 1-day vulnerabilities.

3.1 Overview

V1SCAN synergistically combines version- and code-based approaches to take advantage of their strengths and avoid their weaknesses. The key idea of V1SCAN, which is significantly distinguishable from existing version- and code-based approaches is that it classifies reused and vulnerable codes. Such code classification techniques allow V1SCAN to eliminate false positives of version-based approaches while addressing false negatives of code-based approaches.

Figure 2 depicts the workflow of V1SCAN, which comprises three phases: `classification` (**P1**), `detection` (**P2**), and `consolidation` phase (**P3**). In **P1**, V1SCAN classifies the reused codes of OSS components and the vulnerable code locations in the target program based on code classification (see Section 2.2). In **P2**, V1SCAN detects vulnerabilities contained in the target program by using improved version- and code-based approaches with code classification techniques. Finally, in **P3**, V1SCAN examines the two detection results and consolidates them to confirm the propagated vulnerabilities included in the target program.

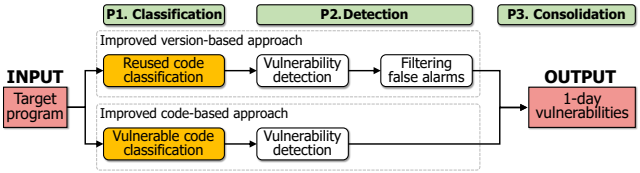


Figure 2: High-level overview of V1SCAN.

Design assumption and text preprocessing. We designed V1SCAN to detect vulnerabilities at the source code level. Although V1SCAN can operate with any granularity unit, we focus on the function units in the version-based approach, which are suitable for identifying reused OSS components [36].

Subsequently, V1SCAN extracts (1) all functions from every version of the OSS in our dataset (see Section 4.2), and (2) all functions of the target program, using a function parser (see Section 4.1). V1SCAN then performs text preprocessing on all extracted functions to normalize the code; it removes comments, linefeed, and whitespace from the function codes.

Moreover, V1SCAN uses locality sensitive hashing (LSH), which hashes similar input items into the same "buckets" with high probability [33]; therefore, LSH can be utilized to address code modifications in a flexible manner.

We applied LSH to all text-preprocessed functions in (1) the target program and (2) every version of OSS projects. Each LSH algorithm provides a functionality (Φ) to measure the distance between two inputs and the *cutoff* value (*i.e.*, threshold θ). Thus, the relationships in which the two functions (f_1, f_2) are *identical*, *similar*, or *different* can be defined as follows: let the output of Φ be an integer.

- **Identical**: If $\Phi(f_1, f_2) = 0$, f_1 and f_2 are identical.
- **Similar**: If $0 < \Phi(f_1, f_2) \leq \theta$, f_1 and f_2 are similar.
- **Different**: If $\Phi(f_1, f_2) > \theta$, f_1 and f_2 are different.

3.2 Classification phase (P1)

In this phase, V1SCAN classifies reused OSS functions and extracts code lines, which belong to the locations defined in the vulnerable code classification, from the target program.

3.2.1 Improved version-based approach

Component identification. The first step involved uncovering the OSS components in the target program, which, however, is difficult to precisely accomplish [9, 36]. Because the identification of components is beyond the scope of this paper, we decided to use an existing, well-implemented component identification tool. We took advantage of CENTRIS [36] because it can precisely identify OSS components that have been modified and its source code and dataset are publicly available. Using CENTRIS, we extracted the names of OSS components included in the target program. Note that CENTRIS is only concerned with component identification, and does not directly participate in vulnerability detection.

Prevalent version identification. We then need to identify the version of each reused component. However, precise identification of the version to which reused modified OSS components belong is challenging (see Section 2.3). Therefore, V1SCAN is designed such that it is not critically affected by the correctness of the identified versions.

Our approach is to first roughly detect vulnerabilities and then refine the results. In particular, we define *the prevalent version* of the OSS component as the version to which the functions contained in the reused OSS codebase most closely correspond. To identify the prevalent version, V1SCAN first compares all functions in the target program with those in every version of the OSS component. V1SCAN extracts the OSS functions that are exactly included in the target program, and then verifies the versions to which each extracted function belongs. The most frequently identified version is designated the prevalent version of the OSS component.

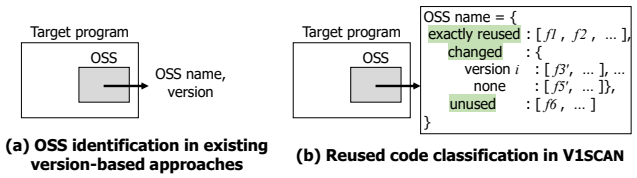
Reused code classification. Next, V1SCAN classifies functions reused in OSS components based on the reused code classification defined in Section 2.2. Because we consider the function unit in the version-based approach, V1SCAN classifies reused OSS *functions* based on code modifications.

When an OSS is reused in the target program, the functions in the original OSS can be in one of three states: (1) exactly reused, (2) changed, or (3) unused. Subsequently, V1SCAN examines the state of all the functions included in the prevalent version of the OSS component. We consider the function relationships defined in Section 3.1.

- **Exactly reused functions.** If a function in the prevalent version is identical to a certain function in the target program, it is considered reused in the target program.
- **Changed functions.** If a function f' that exists in the target program is similar to function f in the prevalent version, f is considered a changed function.
- **Unused functions.** Among all functions in the prevalent version, those that did not belong to the exactly reused or changed were included in the unused group.

The changed functions were further classified according to the version to which they belonged. To this end, V1SCAN compares f' with all functions in every version of the OSS. If f' belongs to any other version of the OSS and is not the prevalent version, f' is stored in the group under the name of the associated version. In contrast, if f' does not belong to any version of the OSS, it is stored in the "none" group. OSS functions that do not belong to the prevalent version but are contained in the target program, as detected in the prevalent version identification, are also included in the changed group with their respective version names.

Figure 3 depicts the procedure to classify reused code compared with OSS identification in existing version-based approaches. V1SCAN stores the hash value of each function obtained by applying the LSH algorithm (see Section 3.1).



f_i: a function in the prevalent version
f_i': a function in the target program similar to f_i.

Figure 3: Illustration of the differences between reused code classification and OSS identification in existing approaches.

3.2.2 Improved code-based approach

Vulnerable code classification. To increase the vulnerability detection scope, V1SCAN classifies the code locations where vulnerabilities exist, and then takes an effective vulnerability detection approach for each location. We considered the following four code locations (see Section 2.2): ① function, ② structure, ③ macro, and ④ (global and class) variable.

Code extraction. Subsequently, V1SCAN extracts the code lines belonging to the location in the target program. In fact, extracting functions, structures, macros, and variables from the target program can easily be performed using an existing C/C++ parser (*e.g.*, [5]). For more accurate vulnerability detection, V1SCAN further classifies the four locations into two groups based on the syntactic similarity: (functions and structures) and (macros and variables). Functions and structures have in common that multiple code lines are enclosed in curly braces, while macros and variables generally consist of only one or two code lines. V1SCAN extracts (1) LSH hash values (see Section 3.1) and (2) source code lines from all functions and structures in every C/C++ source file in the target program. We consider both hash values and lines of code to reduce false positives in vulnerability discovery (details are introduced in Section 3.3.2). For macros and variables, V1SCAN extracts only the corresponding source code lines.

3.3 Detection phase (P2)

V1SCAN detects propagated vulnerabilities in the target program, using improved version- and code-based approaches.

3.3.1 Improved version-based approach

Vulnerability detection. V1SCAN detects vulnerabilities in the target program by using the prevalent version. We used CPE [21] to construct a CPE database that maps the CVEs to the affected OSS versions (see Section 4.3). By leveraging the CPE database, V1SCAN identifies the CVE vulnerabilities included in the prevalent version of the OSS component. For example, the CPE of the Heartbleed vulnerability (CVE-2014-0160) specifies that the vulnerability exists in OpenSSL 1.0.1 to 1.0.1f. If OpenSSL is reused in the target program and its prevalent version is between 1.0.1 and 1.0.1f, V1SCAN determines that this vulnerability exists in the target program.

Collection of vulnerable and patched functions. V1SCAN then removes false positives from the version-based vulnerability detection results. To identify the resolved vulnerable functions, we refer to the vulnerable and patched functions of CVE vulnerabilities. To this end, we first collected the security patches for CVEs and then extracted the vulnerable and patched functions from these patches (see Section 4.3). Thereafter, text preprocessing is applied to all vulnerable and patched functions, after which the LSH algorithm is applied to generate hash values (see Section 3.1).

Filtering. Based on the classification of the reused code, V1SCAN checks for vulnerable functions and filters out false alarms from the vulnerability detection results. We consider the four notations listed in Table 2.

(1) Filtering unused vulnerable functions. The classification of reused code simplifies the filtering of the unused vulnerable functions. If f_v is included in the "unused" group (U), then V1SCAN determines that f_v is not reused in the target program.

$$\forall f_v \,|\, f_v \in U \rightarrow f_v \notin T \,(f_v \text{ is unused})$$

(2) Filtering resolved vulnerable functions. If developers resolve f_v by backporting security patches, this function has the same (or similar) code syntax as that of f_p belonging to the OSS versions that are safe from X. On this basis, if f_v is not included in "exactly reused" but has a similar function f'_v in the "changed" group (C), V1SCAN determines that f_v was either reused with code changes (still vulnerable) or resolved by backporting security patches. To extract only the latter cases, V1SCAN checks the version to which f'_v belongs.

(2-1) If the version is not included in the CPE of X, it can be determined that f'_v does not contain the vulnerability. To determine this more thoroughly, V1SCAN compares f'_v to both f_v and f_p. If the distance (Φ, see Section 3.1) between f_v and f'_v is greater than the distance between f_p and f'_v (*i.e.*, more similar to f_p than f_v), then V1SCAN considers f'_v has been resolved. This can be represented as follows; let v be the version to which f'_v belongs.

$$\forall f'_v \in C \mid \big(v \notin CPE(X)\big) \wedge \big(\Phi(f_p, f'_v) < \Phi(f_v, f'_v) \leq \theta\big) \rightarrow \; f_v \text{ is resolved}$$

(2-2) If f'_v does not belong to any version of OSS, deciding whether f'_v is vulnerable based only on the similarity to the vulnerable function and patch function may produce a false alarm. Therefore, we put it on hold and decide whether to filter this vulnerability through an improved code-based approach of V1SCAN (see Section 3.3.2).

Table 2: Defined notations.

Notations	Description
T	The target program.
X	The vulnerability to be examined.
f_v	The vulnerable function of X.
f_p	The patched function of X.

Here, a CVE vulnerability may (1) contain multiple vulnerable functions or (2) not include any vulnerable functions. In the former case, the propagation of only one vulnerable function may compromise the security of the entire system [12,14]. Therefore, if more than one vulnerable function remains after filtering, V1SCAN determines that CVE vulnerability has propagated to the target program. Conversely, in the latter case, V1SCAN determines CVE as the correct answer without filtering. This may yield false positives; however, this rarely occurs (*i.e.*, not found in our dataset) and arises from our decision that a small number of false positives would be more tolerable than missing vulnerabilities. Moreover, a security patch may alter the code beyond the scope of a function (*e.g.*, a macro). In this case, we put it on hold and verify it at the code level (see Section 3.3.2).

Finally, the vulnerability detection result of the version-based approach in which false positives are removed is obtained as the output of this phase.

3.3.2 Improved code-based approach

Vulnerability signature generation. V1SCAN generates vulnerability signatures for each collected CVE and uses them to detect propagated vulnerabilities. Vulnerability signatures are generated after identifying the location to which the code lines modified in the security patch belong to. For example, Listing 1 shows the patch snippet for CVE-2019-12904 in Libgcrypt. The snippet contains a patch for all four types of code locations: macro (lines #3 and #4), variable (lines #6 and #7), structure (lines #9 and #10), and function (lines #14 and #15). The locations of the code modified in the security patch are identified using an existing C/C++ parser (see Section 4).

Thereafter, V1SCAN uses different indexing methods for each code location. Listing 2 presents the indexing results for the patch code shown in Listing 1. All the code lines added and deleted in the security patch were stored. For functions and structures, the LSH hash value is also stored, as described in Section 3.2.2. We consider hash values for reducing false alarms in vulnerability discovery. If we were to detect vulnerabilities by considering only the code lines added or deleted in a patch, numerous false alarms could occur, particularly when short and general codes (*e.g.*, `else`) are modified in the patch [14, 34]. Therefore, we attempted to resolve this issue using the hash value of the entire function or structure. V1SCAN generates vulnerability signatures for all the collected CVE patches.

Listing 1: A patch snippet for CVE-2019-12904 in Libgcrypt.

```
//libgcrypt/cipher/cipher-gcm.c
...
+ #ifdef HAVE_GCC_ATTRIBUTE_ALIGNED
+ # define ATTR_ALIGNED_64 __attribute__ ((aligned (64)))
...
- static const u16 gcmR[256] = {
- 0x0000, 0x01c2, 0x0384, 0x0246, 0x0708, 0x06ca, 0x048c,
...
+ static struct {
+   volatile u32 counter_head;
...
  void prefetch_table(const void *tab, size_t len) {
...
-   for (i = 0; i < len; i += 8 * 32)
+   for (i = 0; len - i >= 8 * 32; i += 8 * 32)
```

Listing 2: Example vulnerability signature for CVE-2019-12904.

```
MACRO
 + #ifdef HAVE_GCC_ATTRIBUTE_ALIGNED
 + # define ATTR_ALIGNED_64 __attribute__ ((aligned (64)))

VARIABLE
 - static const u16 gcmR[256] = {
 - 0x0000, 0x01c2, 0x0384, 0x0246, 0x0708, 0x06ca, 0x048c,

STRUCTURE (HASH: 3A5F116800...)
 + static struct {
 + volatile u32 counter_head;

FUNCTION (HASH: BBC0994B88...)
 - for (i = 0; i < len; i += 8 * 32)
 + for (i = 0; len - i >= 8 * 32; i += 8 * 32)
```

Function and structure vulnerability detection. V1SCAN then detects the vulnerabilities propagated to the target program. To respond flexibly to code changes unrelated to a vulnerability, V1SCAN focuses on the core code lines that were added or deleted in the security patch [34, 37].

The process of detecting vulnerabilities in functions and structures comprises two steps.

S1. Hash comparison: First, V1SCAN compares the hash values of all functions (structures) in vulnerability signatures with those of the target program. If a hash value that belongs to both the signature and target program is detected, V1SCAN determines that the vulnerability has been propagated to the target program. Otherwise, if a similar hash value is discovered, V1SCAN moves to the next step, namely, line comparison.

S2. Line comparison: When a similar function (structure) is detected between the vulnerability signature and the target program, V1SCAN checks whether code deleted from (*resp*. added to) the patch was included (*resp*. was not included) in the function (structure) of the target program. If this is satisfied, V1SCAN determines that the target program contains the vulnerability.

Here, if we only consider the code syntax in the line comparison, as with certain existing approaches (*e.g*., [35, 37]), false positives or negatives may arise especially when the code modified in the patch is short and general. Assume that the code line "`else`" is added by the patch. This code line may exist in the vulnerable function. Hence, even if the vulnerable function is not patched, the function may be misinterpreted to be safe because it contains the code line added by the patch.

Therefore, we decided to consider the number of times deleted (added) code lines appear in the target function. Suppose that a security patch deletes the code line l_{del} and adds the code line l_{add} to the function f_v. Let the function after applying the patch be f_p. V1SCAN counts the number of times l_{del} and l_{add} appear in f_v and f_p, respectively. Thereafter, when a function f'_v similar to f_v is detected through hash comparison, V1SCAN counts the number of times l_{del} and l_{add} appear in f'_v. If the result is the same as that of f_v but differs from that of f_p, f'_v is considered a vulnerable function.

This method can distinguish between vulnerable and patched functions even when both deleted and added code lines exist in the target function. Therefore, V1SCAN can detect vulnerabilities with fewer false positives than existing simple code syntax checking approaches.

V1SCAN can detect exactly reused vulnerable functions (structures) with the aid of hash comparison and identify vulnerable functions (structures) whose code syntax has been changed using line comparison. This guarantees a higher vulnerability detection rate than the existing code-based approach that considers only hash comparison (*e.g*., [14]) and lowers the false positive rate compared with the existing approach that only considers line comparison (*e.g*., [12]).

Macro and variable vulnerability detection. To detect vulnerabilities in macros and variables (global and class variables), V1SCAN compares the lines of code containing macros and variables in vulnerability signatures to those in the target program. If all the lines of code containing macros (variables) deleted (*resp*. added) by the patch were included (*resp*. not included) in the target program, V1SCAN determines that the target program contains the vulnerability.

If the security patch did not add any lines of code, then only the deleted lines of code are used to check for vulnerability propagation. On the other hand, if the patch did not delete any lines of code, it would be premature to conclude that the vulnerability was propagated by solely relying on the fact that the lines of code added by the patch do not exist in the target program. Thus, V1SCAN omits the latter case to reduce false alarms. This could yield a small number of false negatives; however, this decision was made because the addition of lines of code containing a macro or variables by a security patch without removing any lines of code from all four vulnerable code locations rarely occurs.

The vulnerability detection result of the improved code-based approach in which false negatives are covered is obtained as the output of this phase. Among the vulnerabilities held by the version-based approach (*e.g*., existing outside the function), the vulnerabilities detected in the code-based approach are included in the detection result as correct answers, and all others are considered false positives and filtered out.

3.4 Consolidation phase (P3)

By consolidating the vulnerability detection results of the improved version- and code-based approaches, V1SCAN determines vulnerabilities that exist in the target program.

As explained in P1 and P2, V1SCAN reduces false positives and negatives in the version- and code-based approach. The false alarms produced by the version-based approach were removed through the reuse code classification-based filtering, and the false alarms of the code-based approach were prevented by using both the hash and line comparisons. Moreover, V1SCAN covers the false negatives of version- and code-based approaches by combining their results. Vulnerabilities included in less frequently identified versions (*i.e.*, not a prevalent version) can be covered by detecting them with the code-based approach of V1SCAN. In addition, by leveraging vulnerable code classification and focusing only on core vulnerable code lines, V1SCAN can address false negatives of existing code-based approaches. Therefore, we denote the union of the detection results of both approaches as the list of vulnerabilities contained in the target programs.

4 Implementation of V1SCAN

This section introduces the implementation of V1SCAN, including its architecture and the datasets.

4.1 Architecture

V1SCAN comprises the two modules: a *dataset collector* and a *vulnerability detector*. The dataset collector constructs the OSS and CVE datasets (see Section 4.2 and Section 4.3).

The vulnerability detector performs vulnerability discovery on the target program, which is implemented in approximately 1,800 lines of Python code excluding external libraries. We use CENTRIS [36] to identify the OSS components in the target program. For the parser and LSH algorithm, we utilize Ctags [5] and TLSH [23, 30]. Ctags is a regular expression-based parser, and has the advantage of scalability and detection accuracy; it can be used to precisely parse C/C++ source files without being affected by the code size of the target program. In addition, because Ctags provides the functionality to identify functions, structures, macros, and variables from source files, it can be effectively used to classify vulnerable locations. Next, we selected the TLSH [23, 30] as the LSH algorithm, which is both accurate and scalable. We used the similarity measurement function provided by TLSH as it is, and selected the cutoff value of 30 (see Section 3.1), which was observed to be the most efficient in their paper [23].

4.2 OSS dataset

The OSS dataset is utilized for identifying OSS components in the target program. We leveraged the OSS dataset provided by CENTRIS [36], which consists of all versions of the 10,241 popular C/C++ OSS projects on GitHub.

Table 3: CVE dataset overview.

Category	Count (#)
• **Security patches**	4,612
- Vulnerable and patched function pairs	7,675
- Vulnerable and patched structure pairs	106
- Vulnerable and patched macro pairs	221
- Vulnerable and patched variable pairs	598
• **CPE database**	
- Contained CVEs	137,892
- Collected vulnerable software	75,489
- Collected vulnerable versions	559,305

Since the dataset of CENTRIS was constructed in April 2020, we additionally collected versions generated after that of each repository (as of April 2022). As a result, the number of versions increased from 229,326 to 246,512 in V1SCAN.

4.3 CVE dataset

For each CVE, V1SCAN first collects the security patch, and then extracts necessary information (*e.g.*, vulnerable functions) from the patch. V1SCAN then constructs the CPE database, which is used in the improved version-based approach. Table 3 summarizes the CVE dataset.

Collecting security patches. We collected security patches of CVE vulnerabilities by leveraging existing approaches [11, 16, 35, 37]. We examined CVEs in the NVD and checked whether a "Git commit" URL was included in the references; this URL provides the code-level security patches. Hence, we can obtain the security patch commits for CVE vulnerabilities by crawling the URLs. As a result, we collected 4,612 C/C++ security patches from the NVD (as of August 2022).

Identifying vulnerable locations. Because V1SCAN uses vulnerable code classification, it is necessary to identify the location to which the code modified by the security patch belongs. Each security patch provides (1) a Git index and (2) code line numbers that include vulnerable and patched code.

We access the index to obtain the vulnerable and patched source files (*e.g.*, using the "git show" command). We then parse the source files using Ctags [5]. Using this output, we can obtain the start and end line numbers of each function, structure, macro, and variable. Finally, we identify the locations that contain the modified code lines in the patch. If the modified code lines belong to a function or structure, the entire code of the vulnerable and patched function (and structure) is extracted. If the modified code lines belong to a macro or variable, the exact lines of code are extracted.

As a result, we collected 7,675 vulnerable and patched function pairs, 106 structure pairs, 221 macro pairs, and 598 variable pairs. Over 97% of the code repaired by the patch was included in one of the four locations. The remaining 3% were code patches that very rarely appear, such as changing header file names. This confirms that considering these four locations for vulnerability detection is sufficiently comprehensive.

CPE database. To construct the CPE database, we extract CPEs of all CVEs registered in NVD using the NVD JSON feeds. We stored CPE in a dictionary, in which the vulnerable software (with version) as the key and the CVEs contained in the vulnerable software as the value; this dictionary becomes the CPE database. Consequently, we stored a total of 137,892 CVEs with 75,489 vulnerable software information (a total of 559,305 versions) in the CPE database (as of August 2022).

The issue here is that the names and versions of the OSS components detected in V1SCAN follow the form provided by GitHub (as with the CENTRIS dataset), whereas CPE uses its own form of vulnerable software and version names. For example, Linux kernel v5.15 is referred as "torvalds/linux, v5.15" on GitHub and "linux/linux_kernel, 5.15" on CPE. It is nearly infeasible and time-consuming to check all the tens of thousands of vulnerable software lists and internal versions. Therefore, we first examined the following software, and manually match the software and version names of GitHub and those of CPE: (1) OSS that was reused in the target programs selected for the experiments (see Section 5.1) and (2) frequently reused OSS projects mentioned in the CENTRIS paper (*e.g.*, Zlib, Lua, and GoogleTest). Information on the rest of the software will also be examined.

5 Evaluation

In this section, we evaluate V1SCAN based on the following three questions.

Q1. Detection accuracy: How precisely does V1SCAN detect vulnerabilities compared to the state-of-the-art vulnerability detection approaches? (Section 5.1)

Q2. Effectiveness: How effective is V1SCAN compared to existing version- and code-based approaches in vulnerability discovery? (Section 5.2)

Q3. Performance and scalability: How fast and scalable is V1SCAN in detecting vulnerabilities? (Section 5.3)

We ran V1SCAN on a machine with Windows 10, AMD Ryzen 7 3700X @ 3.60 GHz, 48GB RAM, and 1TB SSD.

Target software selection. To demonstrate the generality of V1SCAN, we selected target software based on the following three criteria: (1) popular C/C++ software, (2) containing a sufficient number of OSS components, and (3) including a considerable number of CVE vulnerabilities.

We examined GitHub [10], which is one of the most popular hosting services, as the first criterion. We collected C/C++ repositories that exhibited more than 5,000 stargazers (a popularity indicator) from GitHub and collected approximately 600 software programs. As a sufficient number of CVEs were required for accuracy measurement, we selected the version of each software that was released last year (closest to January 2021). For the second criterion, we ranked the 600 collected software according to the number of OSS components identified by CENTRIS. While examining the ranked software projects in descending order, we applied a version-based approach to each software program, identified the number of contained CVEs (with manual version correction), and selected the software that contained more than 10 CVEs (for the third criterion). Consequently, nine target software programs were selected. In addition, we included TizenRT[1], which contains many components and can represent industrial software.

Table 4: Target software overview.

IDX	Name	Version	#CVE†	#OSS	#C/C++ Line	#Star§	Domain
S1	Turicreate	v6.4.1	69	28	4,091,413	10.7K	Machine learning
S2	ReactOS	v0.4.13	67	23	6,419,855	10.8K	Operating system
S3	TizenRT	3.0_GBM	62	22	2,156,848	439	Operating system
S4	Aseprite	v1.2.25	53	12	846,500	17K	Animation tool
S5	FreeBSD	v12.2.0	30	47	14,489,534	6.4K	Operating system
S6	MongoDB	r4.2.11	28	13	2,822,534	21.5K	Database
S7	MAME	0228	24	26	4,541,014	5.8K	Emulator
S8	Filament	v1.9.9	16	16	1,295,918	13.8K	Rendering engine
S9	Godot	v3.2.2	16	21	1,298,228	48.1K	Game engine
S10	ArangoDB	v3.6.12	15	22	5,465,881	12.2K	Database
Total	-	-	380	230	43,427,725	147K	-

†: #CVEs discovered by the version-based approach, §: #Stargazers.

Table 4 summarizes the ten selected target programs. The target programs that were selected based on the clear criteria had various codebase sizes ranging from 846,500 to 14,489,534 C/C++ lines of code, and were obtained from diverse domains (operating systems, emulators, and databases). Therefore, we determined that the selected target programs could provide generality to V1SCAN during evaluation. Note that we chose target programs with a considerable number of vulnerabilities for a more impartial accuracy assessment. V1SCAN can detect vulnerabilities even in software with few vulnerabilities, without any limitations.

5.1 Accuracy measurement

Methodology. We compared V1SCAN with the state-of-the-art vulnerability discovery approaches: MOVERY [34] and `VOFinder` [35], which are capable of discovering vulnerable codes propagated with code modifications to some extent. We used their default settings by referring to their paper.

To evaluate accuracy, we used the following five metrics: true positive (TP), false positive (FP), false negative (FN), precision (P = #TP/(#TP + #FP)), and TP detection rate (R = #TP/(#TP + #FN)). Because it is nearly infeasible to detect all vulnerabilities in a target program, we cannot easily measure the FNs of the tested approaches. Therefore, we consider only indisputable FNs; for example, FNs in V1SCAN refer to the vulnerabilities detected by the other two approaches, but were not discovered by V1SCAN. The TPs and FPs were determined by manual analysis, and two security analysts examined all vulnerability detection results. We examined the results by referring to (1) the detected vulnerable code, (2) the security patch, (3) the NVD description, and (4) issue trackers that denote vulnerability.

[1] `https://github.com/samsung/tizenrt`

Table 5: Accuracy of V1SCAN, MOVERY, and VOFinder in vulnerability detection. Results with the highest precision and TP detection rate for each target program are highlighted in bold text.

Target program	CVEs*	V1SCAN					MOVERY					VOFinder				
		#TP	#FP	#FN	P†	R‡	#TP	#FP	#FN	P	R	#TP	#FP	#FN	P	R
Turicreate	36	32	1	4	**0.97**	**0.89**	22	5	14	0.81	0.61	22	2	14	0.92	0.61
ReactOS	29	26	1	3	**0.96**	**0.90**	24	3	5	0.89	0.83	18	4	11	0.82	0.62
FreeBSD	23	19	2	4	**0.90**	**0.83**	13	4	10	0.76	0.57	12	7	11	0.63	0.52
MongoDB	14	14	0	0	**1.00**	**1.00**	4	0	10	**1.00**	0.29	4	0	10	**1.00**	0.29
Filament	14	14	0	0	**1.00**	**1.00**	10	0	4	**1.00**	0.71	4	0	10	**1.00**	0.29
TizenRT	10	9	0	1	**1.00**	**0.90**	4	1	6	0.80	0.40	3	1	7	0.75	0.30
Aseprite	8	8	0	0	**1.00**	**1.00**	6	0	2	**1.00**	0.75	1	1	7	0.50	0.13
MAME	8	7	2	1	0.78	**0.88**	6	1	2	**0.86**	0.75	2	1	6	0.67	0.25
Godot	4	4	0	0	**1.00**	**1.00**	1	3	3	0.25	0.25	1	2	3	0.33	0.25
ArangoDB	4	4	0	0	**1.00**	**1.00**	0	0	4	N/A	0.00	0	1	4	0.00	0.00
Total	150	137	6	13	0.96	0.91	90	17	60	0.84	0.60	67	19	83	0.78	0.45

CVEs*: Total number of TPs detected by V1SCAN, MOVERY, and VOFinder, P†: Precision, R‡: TP detection rate.

Unlike VOFinder, which is compatible with our dataset, MOVERY offers datasets with their own preprocessing applied. Therefore, we additionally applied preprocessing to the vulnerability and patched functions we gathered (see Section 4.3), by referring to the MOVERY paper, *e.g.*, extracting control dependency graphs of the vulnerable and patched functions by using the Joern [39] parser.

Overall results. Table 5 summarizes the vulnerability detection results. We confirmed that 150 CVE vulnerabilities were discovered in ten target programs. Among them, 93 (62%) vulnerabilities were reused in the target program with code changes. Nevertheless, V1SCAN showed substantially better accuracy than the other approaches. It is noteworthy that V1SCAN could reduce the false negative rate of MOVERY and VOFinder from 40% and 55% to 9%; V1SCAN discovered 137 vulnerabilities from ten target programs, while existing approaches detected at most 90 vulnerabilities (see Figure 4).

FNs of existing approaches. In our experiments, MOVERY and VOFinder failed to detect many vulnerabilities, *i.e.*, they missed 60 and 83 vulnerabilities, respectively. We observed that there are two main reasons for this: when the propagated vulnerable code (1) underwent significant syntax change or (2) existed outside of functions.

MOVERY and VOFinder detect vulnerabilities by assuming that propagated vulnerable codes have similar syntax (*e.g.*, over 50% similarity) to the disclosed vulnerable codes, causing failure in discovering vulnerable codes with significant syntax changes. Specifically, VOFinder reported more false negatives compared to MOVERY as it was more limited in detecting heavily modified vulnerable codes. Additionally, they failed to detect seven vulnerabilities that exist outside of functions as they only consider function units.

FPs of existing approaches. MOVERY and VOFinder produced 17 and 19 false positives in vulnerability discovery.

MOVERY can detect vulnerabilities in which the code lines deleted from the patch do not exist, but this approach rather generated FPs; a function unrelated to the vulnerability, in which neither the codes added nor deleted from the patch were present, was falsely detected as vulnerable.

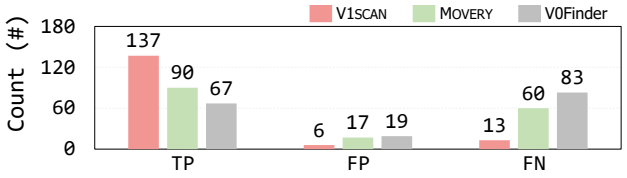


Figure 4: Vulnerability detection results of V1SCAN, MOVERY, and VOFinder.

In addition, when the code lines deleted in the patch exists in multiple locations in the vulnerable code, VOFinder misinterpreted that the vulnerability still remains even after the patch is applied, thereby generating false alarms.

Accuracy of V1SCAN. V1SCAN significantly outperformed the other approaches: V1SCAN could considerably reduce FPs of MOVERY and VOFinder, while discovering more TPs. It is noteworthy that V1SCAN showed high detection accuracy regardless of the number of vulnerabilities discovered in software; V1SCAN succeeded in detecting propagated vulnerabilities without false alarms even in Godot and ArangoDB, in which only a small number of vulnerabilities were discovered.

Although V1SCAN detected vulnerabilities precisely in most cases, it reported several false results (6 FPs and 13 FNs). The six FPs were caused by similar code logic in the OSS components [1, 37]. If a function similar to one that was repaired in the security patch existed in the target program, V1SCAN misinterpreted the reused function as being vulnerable even though it was safe. Because the possibility of actual abuse was negligible, we determined these to be FPs.

FNs in V1SCAN were reported when the syntax of the reused vulnerable function differed vastly from that disclosed by the NVD (*e.g.*, syntax similarity of less than 15%). Additionally, unlike MOVERY, V1SCAN only targets vulnerabilities with code lines deleted from the patch. This is to prevent false positives that may occur when considering only the code lines added in the patch, but this leads to two FNs. Further-

Table 6: Accuracy comparison between the improved version-based approach of V1SCAN and CENTRIS.

Target program	V1SCAN (version-based)			CENTRIS		
	#TP	#FP	#FN	#TP	#FP	#FN
Turicreate	23	0	3	22	24	4
ReactOS	20	0	1	5	47	16
TizenRT	7	0	0	7	33	0
Aseprite	3	0	4	7	24	0
FreeBSD	10	0	2	7	19	5
MongoDB	6	0	0	6	9	0
MAME	1	0	1	2	23	0
Filament	3	0	0	3	10	0
Godot	4	0	2	2	8	4
ArangoDB	1	0	0	1	10	0
Total	78	0	13	62	207	29

Table 7: Accuracy comparison between the improved code-based approach of V1SCAN and VUDDY.

Target program	V1SCAN (code-based)			VUDDY		
	#TP	#FP	#FN	#TP	#FP	#FN
Turicreate	13	1	0	9	9	4
ReactOS	21	1	0	13	8	8
TizenRT	7	0	0	5	5	2
Aseprite	8	0	0	3	0	5
FreeBSD	12	2	0	10	0	2
MongoDB	10	0	0	3	3	7
MAME	7	2	0	1	0	6
Filament	13	0	0	4	3	9
Godot	1	0	0	1	0	0
ArangoDB	4	0	0	0	2	4
Total	96	6	0	49	30	47

more, V1SCAN failed to detect OSS vulnerabilities when the reuse relationship was unclear. For example, vulnerable codes that were inherited from the Linux Kernel existed in FreeBSD. V1SCAN failed to identify the reuse relationship between the Linux Kernel and FreeBSD, resulting in three FNs.

5.2 Effectiveness of V1SCAN

Next, we evaluate the effectiveness of V1SCAN compared to existing version- and code-based approaches. In the experiments, CENTRIS [36] and VUDDY [14] were leveraged for the version- and code-based approaches, respectively.

Methodology. We first evaluated how the improved version- and code-based approaches of V1SCAN enhance vulnerability detection accuracy over the baselines individually (*i.e.*, CENTRIS and VUDDY). Thereafter, we assessed the effectiveness of V1SCAN by comparing its vulnerability detection results with the combined results (union) of CENTRIS and VUDDY. We used the vulnerability dataset (4,612 CVE vulnerabilities) collected in Section 4.3, and used the same target programs (Table 4) and metrics introduced in Section 5.1. As in the previous experiment, TPs and FPs were established through manual analysis. Finally, when comparing two approaches here, if a vulnerability detected by one approach goes undetected by the other approach, it is considered a false negative of the latter. Therefore, in subsequent experiments, FNs are may differ from that introduced in Section 5.1.

Effectiveness of the improved version-based approach. The improved version-based approach of V1SCAN was substantially better accurate than CENTRIS (see Table 6). It discovered 78 vulnerabilities in ten target programs without producing any false alarms by using reuse code classification. V1SCAN reported 13 FNs that occurred when a vulnerability contained in the less frequent version of the OSS (not a prevalent version) was reused in the target program, or because it was accidentally filtered out during the filtering process. Note that some of these FNs can be overcome with the improved code-based approach of V1SCAN.

CENTRIS discovered only 62 CVE vulnerabilities while producing 207 false alarms (*i.e.*, a precision of 23%). When CENTRIS fails to predict the correct component versions, it would fail to detect vulnerabilities contained in the reused components (FNs). Additionally, it produced many FPs when (1) the version prediction failed (103 FPs), and (2) a component was reused with code modifications (97 FPs). In the latter case, 81 vulnerabilities were not reused in the target program and 16 vulnerabilities were repaired by backporting security patches. The remaining seven FPs occurred because the CPE provided an incorrect OSS version in that a version without a vulnerable function was referred to as vulnerable [8, 35].

Effectiveness of the improved code-based approach. The improved code-based approach used by V1SCAN detected twice as many TPs as VUDDY (see Table 7).

It is noteworthy that V1SCAN covered all vulnerabilities detected by VUDDY. Specifically, V1SCAN could detect seven CVE vulnerabilities that existed outside of functions by using the vulnerable code classification technique. Moreover, V1SCAN could discover modified vulnerable codes by composing a signature with core code lines related to vulnerabilities. V1SCAN produced six FPs owing to the similar code logic in OSS components, as explained in Section 5.1.

In contrast, VUDDY reported 47 FNs (a 51% TP detection rate). It failed to detect most vulnerabilities when the propagated vulnerable code underwent syntax changes. In addition, VUDDY could not detect vulnerabilities that exist outside of functions (*e.g.*, macro vulnerabilities) because it only considered function units.

Furthermore, VUDDY produced 30 FPs, mainly because of the abstraction method (see Section 2.4). When the security patch changes only the abstraction targets (*e.g.*, parameters and local variable names), VUDDY cannot differentiate between vulnerable and patched functions, resulting in an FP being reported. Finally, if the code of the function to which the security patch is applied is extremely short and general (*e.g.*, a single line return statement), VUDDY mistakenly detects lines of code that are similar but safe as being vulnerable.

Table 8: Accuracy comparison between V1SCAN and the combined (union) results of CENTRIS and VUDDY.

Target program	V1SCAN					Union				
	#TP	#FP	#FN	P	R	#TP	#FP	#FN	P	R
Turicreate	32	1	2	0.97	0.94	30	32	4	0.48	0.88
ReactOS	26	1	1	0.96	0.96	15	55	12	0.21	0.56
TizenRT	9	0	0	1.00	1.00	9	38	0	0.19	1.00
Aseprite	8	0	4	1.00	0.67	10	24	2	0.29	0.83
FreeBSD	19	2	0	0.90	1.00	13	19	6	0.41	0.68
MongoDB	14	0	0	1.00	1.00	9	11	5	0.45	0.64
MAME	7	2	1	0.78	0.88	3	23	5	0.12	0.38
Filament	14	0	0	1.00	1.00	5	13	9	0.28	0.36
Godot	4	0	2	1.00	0.67	3	8	3	0.27	0.50
ArangoDB	4	0	0	1.00	1.00	1	12	3	0.08	0.25
Total	137	6	10	0.96	0.93	98	235	49	0.29	0.67

Comparison with the combined results. Finally, we compared the vulnerability detection results of V1SCAN to the combined results of CENTRIS and VUDDY. Table 8 summarizes the experimental results. Despite combining the results of CENTRIS and VUDDY, V1SCAN could detect 40% more vulnerabilities than the combined results while eliminating more than 97% of FPs. The code-based approach used by V1SCAN could address several FNs of the version-based approach of V1SCAN; by synergistically consolidating the two results, V1SCAN discovered more vulnerabilities than either approach in a precise manner.

In summary, each of the improved version- and code-based techniques used in V1SCAN showed higher accuracy than existing version- and code-based approaches. Even combining the results of existing approaches, V1SCAN showed significantly better detection accuracy. The comprehensive approach of V1SCAN, which systematically overcomes the problem of the many FPs and FNs generated by the respective existing approaches, can be considered highly effective.

5.3 Performance and scalability

The performance and scalability of V1SCAN were measured using the dataset built by CENTRIS [36], which revealed that 4,434 popular GitHub C/C++ software programs reused one or more OSS components. We measured the elapsed time required for V1SCAN to detect vulnerabilities in the 4,434 software programs (ranging from 1K to 20M code lines), excluding the time required for identifying OSS components using CENTRIS (averaged 2 minutes per target software) and collecting CVE datasets (total 2 hours). The average codebase size of 4,434 software programs was 300K lines of code.

However, manually matching the GitHub and CPE names of all the OSS components detected in the 4,434 software would be a tedious and error-prone task (see Section 4.3). Therefore, we only considered OSS projects that had already been verified (mapped) in Section 5.1. Note that such OSS projects could cover more than 80% of the components detected in the 4,434 software programs.

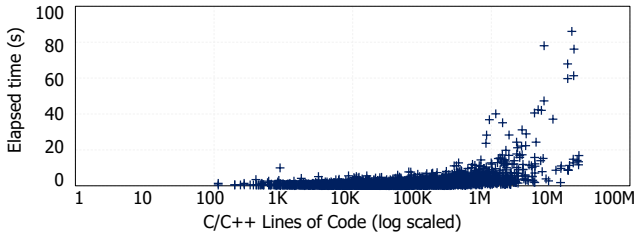


Figure 5: Elapsed time for vulnerability detection in 4,434 popular C/C++ software programs with various code sizes.

Figure 5 illustrates the time measurement result. We observed that V1SCAN detected vulnerabilities in most of the target programs (99%) within 20 seconds. The target program that required the longest time was FreeBSD (86 seconds), which contained many OSS components and had a large codebase size. Overall, the time V1SCAN required for vulnerability detection was proportional to the codebase size of the target program, but several factors, such as the number of components and vulnerabilities contained in the prevalent versions also affected the elapsed time.

Compared to the existing code-based approaches (*e.g.*, [12, 14, 34, 37]), which took an average of 10 s to detect vulnerabilities in a target program, it is noteworthy that V1SCAN did not differ significantly in terms of the time required, even though it used both the improved version- and code-based approaches and showed much higher detection accuracy. In summary, the discovery of vulnerabilities within 20 seconds in 99% of the target programs (even with large codebase sizes) suggests that V1SCAN delivers high performance combined with scalability and is suitable for practical use.

6 Discussion

Practicality of V1SCAN. We applied V1SCAN to the *latest* versions of various software programs to evaluate its practicality. To this end, we randomly selected 100 GitHub C/C++ repositories with over 1,000 stargazers, including the selected target programs in Section 5.1 and other popular software programs (*e.g.*, Git and Linux kernel).

Consequently, V1SCAN detected 73 vulnerabilities, of which 14 were successfully reproduced. The failure to reproduce the vulnerability was due to a compilation error, a failure to call the vulnerable function, or a publicly unavailable proof of concept. We reported all the cases in which vulnerabilities were reproduced to the responsible development teams. It is worth noting that the Common Vulnerability Scoring System (CVSS) for 12 of the 14 reproducible vulnerabilities is "high."

We introduce an example of the vulnerability discovered in LibGDX. V1SCAN detected that the latest version of LibGDX, which is a popular cross-platform game development framework, used a vulnerable version of the STB library. Because the vulnerabilities contained in STB can potentially lead to a remote code execution attack, a patch is urgently re-

quired. We reported two vulnerabilities to the LibGDX team in March 2022. They confirmed our report and immediately applied the security patches to their codebase.

V1SCAN also detected vulnerabilities in other software, including OpenMVG, FreeBSD, and Disque. We reported the vulnerabilities to the respective development teams; the FreeBSD team responded that they would fix the vulnerability in the subsequent versions. Patch requests are currently pending for the four cases, despite our multiple reports. Nonetheless, we observed that V1SCAN could be effectively used for vulnerability detection in real-world software programs.

Limitations. V1SCAN makes several assumptions that limit its application.

First, V1SCAN can be applied only when the source code of the target program is available.

Second, although V1SCAN outperformed the existing approaches, it may fail to detect several vulnerabilities. A typical example other than those introduced in Section 5.1 is the vulnerability in which CPE is incorrectly provided and, at the same time, reused with significant code changes. To address this problem, we can devise a CPE verification technique (*e.g.*, [6, 8, 35]) or apply a more relaxed code-based approach. However, the former is beyond the scope of this paper, and particular caution is needed, as the latter may produce more FPs. Additionally, while V1SCAN can handle the syntax diversity of vulnerable codes, it may not detect propagated vulnerable functions that do not contain the code lines deleted in the patch. This is a result of our decision to avoid potential FPs (Section 5.1), but it may result in the reporting of several FNs.

Third, V1SCAN may not be able to filter code vulnerabilities resolved by security patches applied by the developers themselves. To solve this problem, we are considering applying semantic analysis to vulnerable and patched codes.

Finally, the accuracy of V1SCAN can be affected by the performance of external tools. For example, CENTRIS can incorrectly identify components, causing V1SCAN to produce FPs. In addition, FPs and FNs may be generated owing to the selected function parser and LSH algorithm (see Section 5.1). If this becomes an issue, V1SCAN can be plugged into other tools to enhance the vulnerability discovery accuracy.

7 Related work

Software composition analysis. Several approaches have attempted to detect third-party OSS components reused in target programs (*e.g.*, [2, 9, 17, 20, 28, 36, 40, 41]); some of these can be used to detect vulnerabilities. Woo *et al.* [36] presented CENTRIS to identify the modified OSS components in a target program. It significantly reduces false alarms in component identification by considering only the unique parts of the OSS through code segmentation. Duan *et al.* [9] proposed OSSPolice to identify 1-day vulnerabilities from the third-party libraries of an Android application. It utilizes constant features to extract library versions, and determines whether vulnerable versions were used in the target application. Zhan *et al.* [40] proposed ATVHunter to precisely detect versions of third-party libraries using the control flow graph of the application. Based on the identified versions, they verified whether the target application contained known vulnerabilities.

However, their goal was to precisely identify the components, and they did not fully consider OSS modifications for vulnerability detection. Therefore, they may produce many false alarms in our problem situations (see Section 5.1).

Vulnerable code detection. Several approaches have attempted to discover vulnerable code in target programs (*e.g.*, [3, 12, 14, 34, 37]). Jang *et al.* [12] proposed ReDeBug, a token-level vulnerable code clone detection approach that uses a sliding window technique. Kim *et al.* [14] presented VUDDY, a function-level vulnerable code clone detection approach that can scalably detect 1-day vulnerabilities. Bowman *et al.* [3] proposed VGRAPH, a code property graph based vulnerable code detection approach that is robust to code modification. Xiao *et al.* [37] presented MVP, a recurring vulnerability detection approach, which can discover vulnerable code that recurs with different syntax. Woo *et al.* [34] proposed MOVERY, an approach that can precisely detect modified vulnerable code clones.

Although their goals are similar to ours, they did not consider vulnerable code whose code has changed significantly owing to the modified OSS reuse, which led to the production of false negatives when applied to our target problem. Also, they cannot counteract vulnerabilities that exist outside of the selected granularity (see Section 5.1 and Section 6). Other approaches have attempted to detect vulnerabilities based on learning algorithms (*e.g.*, [18, 19]). These approaches can be applied to discover general vulnerable code; however, they are not applicable for detecting 1-day vulnerabilities.

Code clone detection. Many approaches attempted to detect code clones (*e.g.*, [4, 13, 24–27, 31]). However, as verified in the previous studies (*e.g.*, [14, 37]), they generate numerous false alarms when applied to detect vulnerable code. Hence, they cannot be directly applied to solving our target issue.

8 Conclusion

Detecting known security vulnerabilities in third-party OSS components is the first step toward achieving secure software. In this regard, we presented V1SCAN, which is a precise approach for detecting 1-day vulnerabilities from modified OSS components in a target program. V1SCAN significantly outperformed existing approaches in terms of 1-day vulnerability discovery accuracy by consolidating version- and code-based approaches, along with the reused code classification and vulnerable code classification techniques. Equipped with the vulnerability detection results of V1SCAN, developers can mitigate the potential risks imposed by modified third-party OSS reuse, rendering a safer software ecosystem.

Acknowledgment

We appreciate the anonymous shepherd and reviewers for their valuable comments to improve the paper. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government Ministry of Science and ICT (MSIT) (No.2022-0-00277, Development of SBOM Technologies for Securing Software Supply Chains, No.2022-0-01198, Convergence Security Core Talent Training Business, and IITP-2023-2020-0-01819, ICT Creative Consilience program).

Availability

The source code and datasets of V1SCAN is publicly available at GitHub: `https://github.com/wooseunghoon/V1SCAN-public`.

References

[1] Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2025–2040, 2021.

[2] Michael Backes, Sven Bugiel, and Erik Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 356–367, 2016.

[3] Benjamin Bowman and H Howie Huang. VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 53–69, 2020.

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

[5] Ctags. Universal Ctags, 2023. `https://github.com/universal-ctags/ctags`.

[6] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. Facilitating Vulnerability Assessment through PoC Migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3300–3317, 2021.

[7] Alexandre Decan and Tom Mens. What Do Package Dependencies Tell Us About Semantic Versioning? *IEEE Transactions on Software Engineering (TSE)*, 47(6):1226–1240, 2019.

[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 *Proceedings of the 28th USENIX Security Symposium (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 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2169–2185, 2017.

[10] GitHub. The world's leading software development platform, 2023. `https://github.com/`.

[11] Hyunji Hong, Seunghoon Woo, Eunjin Choi, Jihyun Choi, and Heejo Lee. xVDB: A High-Coverage Approach for Constructing a Vulnerability Database. *IEEE Access*, 10:85050–85063, 2022.

[12] Jiyong Jang, Abeer Agrawal, and David Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP)*, pages 48–62, 2012.

[13] 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.

[14] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, pages 595–614, 2017.

[15] Seongkyeong Kwon, Seunghoon Woo, Gangmo Seong, and Heejo Lee. OCTOPOCS: Automatic Verification of Propagated Vulnerable Code Using Reformed Proofs of Concept. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 174–185, 2021.

[16] Frank Li and Vern Paxson. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2201–2215, 2017.

[17] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. LibD: Scalable and Precise Third-party Library Detection in Android Markets. In *Proceedings of the 39th International*

Conference on Software Engineering (ICSE), pages 335–346. IEEE, 2017.

[18] 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.

[19] 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 25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[20] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *Proceedings of the 38th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 653–656, 2016.

[21] NVD. Common Platform and Enumeration (CPE), 2023. `https://nvd.nist.gov/products/cpe`.

[22] NVD. National Vulnerability Database, 2023. `https://nvd.nist.gov/`.

[23] 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.

[24] Chaiyong Ragkhitwetsagul and Jens Krinke. Siamese: Scalable and Incremental Code Clone Search via Multiple Code Representations. *Empirical Software Engineering*, pages 2236–2284, 2019.

[25] Chanchal K Roy and James R Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, pages 172–181, 2008.

[26] 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.

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

[28] Wei Tang, Du Chen, and Ping Luo. BCFinder: A Lightweight and Platform-independent Tool to Find Third-party Components in Binaries. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 288–297. IEEE, 2018.

[29] National Telecommunications and Information Administration. NTIA Software Component Transparency with SBOM (Software Bill of Materials), 2023. `https://www.ntia.doc.gov/SoftwareTransparency`.

[30] TLSH. Trend Micro Locality Sensitive Hash, 2023. `https://github.com/trendmicro/tlsh`.

[31] 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.

[32] Wikipedia. Jaccard similarity, 2023. `https://en.wikipedia.org/wiki/Jaccard_index`.

[33] Wikipedia. Locality-sensitive hashing, 2023. `https://en.wikipedia.org/wiki/Locality-sensitive_hashing`.

[34] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In *Proceedings of the 31st USENIX Security Symposium (Security)*, pages 3037–3053, 2022.

[35] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In *Proceedings of the 30th USENIX Security Symposium (Security)*, pages 3041–3058, 2021.

[36] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 860–872, 2021.

[37] 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 *Proceedings of the 29th USENIX Security Symposium (Security)*, pages 1165–1182, 2020.

[38] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch Based Vulnerability Matching for Binary Programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 376–387, 2020.

[39] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*, pages 590–604. IEEE, 2014.

[40] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 1695–1707. IEEE, 2021.

[41] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review. *IEEE Transactions on Software Engineering*, 2021.