MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components

Seunghoon Woo, Hyunji Hong, Eunjin Choi, Heejo Lee*
Korea University, {seunghoonwoo, hyunji_hong, silver_jin, heejo}@korea.ac.kr

Abstract

Vulnerabilities inherited from third-party open-source software (OSS) components can compromise the entire software security. However, discovering propagated vulnerable code is challenging as it proliferates with various code syntaxes owing to the OSS modifications, more specifically, internal (*e.g.*, OSS updates) and external modifications of OSS (*e.g.*, code changes that occur during the OSS reuse).

In this paper, we present MOVERY, a precise approach for discovering vulnerable code clones (VCCs) from modified OSS components. By considering the oldest vulnerable function and extracting only core vulnerable and patch lines from security patches, MOVERY generates vulnerability and patch signatures that effectively address OSS modifications. For scalability, MOVERY reduces the search space of the target software by focusing only on the codes borrowed from other OSS projects. Finally, MOVERY determines that the function is VCC when it matches the vulnerability signature and is distinctive from the patch signature.

When we applied MOVERY on ten popular software selected from diverse domains, we observed that 91% of the discovered VCCs had different code syntax from the disclosed vulnerable function. Nonetheless, MOVERY discovered VCCs at least 2.5 times more than those discovered in existing techniques, with much higher accuracy: MOVERY discovered 415 VCCs with 96% precision and 96% recall, whereas two recent VCC discovery techniques, which hardly consider internal and external OSS modifications, discovered only 163 and 72 VCCs with at most 77% precision and 38% recall.

1 Introduction

The growing number of open-source software (OSS) has made it possible for developers to reuse neat functionalities from reliable OSS projects [11, 12, 37]. At the same time, however, vulnerabilities propagated by the third-party OSS reuse can threaten the security of the entire system [14, 17, 48, 49]. Maintaining OSS components up to date can, in principle, be a solution to prevent such threats. However, a hasty component update can adversely affect the entire system (*e.g.*, backward compatibility problems [40, 44]), especially when developers reuse OSS projects with code or structural modifications [48].

Therefore, developers often backport upstream security patches [43] to prevent undesirable threats from vulnerabilities, instead of updating the entire component. To this end, they first need to identify vulnerable code in the component to be fixed [13, 43], for example, by leveraging vulnerable code clone (VCC) discovery techniques (*e.g.*, [17, 49]) and software composition analysis (SCA) techniques that identify reused components in a target program (*e.g.*, [9, 48]).

Unfortunately, the precise discovery of vulnerabilities from modified OSS components is becoming challenging; the main obstacle is the *syntax diversity* of vulnerable code, mostly caused by the following two types of OSS modifications.

- **Internal modification of the OSS:** The OSS source code frequently changes during *OSS version updates*. In this context, vulnerable codes can exist in various forms depending on the OSS version, and can be propagated to other software with various syntaxes.
- **External modification of the OSS:** Vulnerable code can be modified during the *OSS reuse process*, owing to the nature of the OSS ecosystem where the source code is often modified in the reuse process [48].

These two types of OSS modifications collectively impair the accuracy of discovering propagated vulnerable code, as their syntax can differ from that of the disclosed vulnerable code (*e.g.*, through public vulnerability databases [31]). To the best of our knowledge, none of the existing techniques are capable of precisely discovering vulnerable codes of various syntaxes present in the modified OSS components.

Limitations of existing techniques. Existing VCC discovery techniques (*e.g.*, [14, 17, 49]) do not consider internal modifications of the OSS and examine only external modifications in a limited fashion, thereby producing many false negatives. In particular, they can only discover VCCs containing code

*Heejo Lee is the corresponding author.

lines that were deleted from security patches, which are often omitted in VCCs within modified OSS components. In contrast, existing SCA tools (*e.g.*, [9,48]) determine the existence of vulnerabilities based only on the reused OSS version, producing false positives especially when the vulnerable code is patched through backporting or not reused. Incidentally, existing code clone detection techniques (*e.g.*, [35,42]) can be used to discover VCCs, but they are prone to produce false positives because they cannot distinguish vulnerable and patched codes that exhibit high code similarities in general [17, 49].

To overcome such shortcomings, we present MOVERY (MOdified Vulnerable code clone discovERY), a precise approach for discovering modified vulnerable code clones from modified OSS components.

Our approach. The key idea of MOVERY, which is outstandingly distinguishable from existing VCC discovery techniques, lies in generating an extensible signature that comprehends from the oldest to disclosed vulnerable functions.

MOVERY comprises the following two phases: (1) P1 for generating signatures, and (2) P2 for discovering VCCs.

In P1, MOVERY generates vulnerability and patch signatures using vulnerable and patched functions reconstructed from security patches. Specifically, MOVERY uses techniques called *function collation* and *core line extraction*. Unlike existing techniques that focus only on the disclosed vulnerable function, MOVERY additionally considers the oldest vulnerable function and collates them; this is for addressing internal OSS modifications, as the delta between the oldest and disclosed vulnerable functions summarizes the changes in the vulnerable code during OSS updates. Thereafter, MOVERY generates signatures by storing only the core vulnerable and patch code lines; the generated signatures can be used to discover VCCs without being affected by code changes other than core vulnerable and patch code lines (see Section 3.1).

In P2, MOVERY discovers the VCCs in the target software using the generated signatures. For scalability, MOVERY reduces the VCC search space by leveraging the concept of the existing SCA technique [48], which only inspects code parts borrowed from other OSS. For accuracy, MOVERY applies a selective abstraction technique to precisely discover VCCs with external OSS modifications. Finally, MOVERY confirms that the function in the target software is a vulnerable code clone when it matches vulnerability signatures and is distinctive from patch signatures (see Section 3.2).

Evaluation. Experimental results showed that MOVERY significantly outperformed existing VCC discovery techniques such as ReDeBug [14] and VUDDY [17]. For evaluation, we collected 4,219 Common Vulnerabilities and Exposures (CVE) patches from the NVD [31], including all the C/C++ CVEs that released their patches via Git [19, 47].

When we applied MOVERY, ReDeBug, and VUDDY on ten popular software selected from diverse domains, we observed that 91% of the discovered VCCs had a different syntax from disclosed vulnerable functions. In the experiments, MOVERY was able to discover VCCs at least 2.5 times more than existing techniques with much higher accuracy; this is because ReDeBug and VUDDY neither consider OSS modifications nor properly address the syntax diversity of VCCs, resulting in many false negatives. Specifically, MOVERY discovered 415 VCCs with 96% precision and 96% recall, meanwhile ReDeBug and VUDDY discovered 163 and 72 VCCs with at most 77% precision and 38% recall (see Section 5.1).

We further confirmed that MOVERY showed substantially better accuracy in discovering VCCs from modified components than MVP [49] (*i.e.*, a recurring vulnerability detection technique) and CENTRIS [48] (*i.e.*, an SCA technique): MVP, which does not consider internal OSS modifications, reported 184 false negatives (54% recall) for ten target software, while the CENTRIS-based VCC discovery approach yielded 272 false positives (51% precision) as it did not consider backported security patches (see Section 5.2 and Section 5.3).

Moreover, we demonstrated that MOVERY discovered VCCs from ten target software of various code sizes (*i.e.*, ranging from 212,672 to 14,489,534 C/C++ lines of code) within 200 s on average (for each software). This measured elapsed time is shorter than that of ReDeBug (298 s) and VUDDY (798 s), suggesting that MOVERY is sufficiently fast and scalable for practical usage (see Section 5.4).

Contributions. We summarize our contributions below.

- We present MOVERY, the first approach for precisely discovering VCCs in modified OSS components. The key technique is generating signatures that are capable of addressing the internal and external OSS modifications.
- We demonstrated that internal modifications of the OSS, which were hardly considered in existing techniques, can play a leading role in the syntax diversity of VCCs.
- Although most (91%) of the syntax of VCCs in modified components differed from the disclosed vulnerable functions, MOVERY was able to discover 415 VCCs in ten target software selected from diverse domains, with 96% precision and 96% recall.

2 Motivation

In this section, we clarify the target problem of MOVERY, and discuss the motivation for MOVERY with examples.

2.1 Problem statement

We focused on discovering propagated vulnerabilities in the modified OSS components. Suppose that a vulnerability is introduced in an OSS and is then patched. Let f_d be the disclosed vulnerable function (*e.g.*, through public vulnerability databases), f_p be the patched function, and f_o be the vulnerable function contained in an older version of the OSS, which has a different code syntax from f_d (see Figure 1).

Figure 1: Depiction of the vulnerability fix flow from the vulnerability introduction to the vulnerability patch.

Using these terms, we can classify the propagation of vulnerability caused by vulnerable OSS reuse into the following four categories (**C1** to **C4**, see Table 1):

Table 1: Classification of vulnerable function propagation.

Category	Description
C1	f_d is reused without code modification.
C2	f_d is reused with code modification.
C3	f_o is reused without code modification.
C4	f_o is reused with code modification.

Regardless of the category, all propagated vulnerable functions should be discovered and patched.

Technical challenges. The main technical challenge is addressing the syntax diversity of vulnerable code, mostly arises for the following two reasons: *internal* and *external modification of the OSS*. As the source code of an OSS frequently changes during OSS version updates (*i.e.*, internal modifications of OSS), the syntax of a vulnerable function can also be changed (*i.e.*, **C3** and **C4**). Moreover, developers often apply their own code patches to OSS components (*i.e.*, external modifications of OSS), and thus, a propagated vulnerable function can exist with various syntaxes (*i.e.*, **C2** and **C4**).

Such syntax diversity significantly impairs the accuracy of VCC discovery. Existing VCC discovery techniques (*e.g.*, [14, 17, 49]) can cover only **C1** and limited **C2**, thereby producing many false negatives. Typically, they define code lines deleted from the security patch as *vulnerable lines*, and determine a function as VCC only if the function contains all the vulnerable lines. However, the vulnerable lines in f_d they defined may not exist in f_o due to the OSS modifications; in this case, they fail to discover VCCs, producing false negatives.

2.2 Motivating examples

We introduce two VCC examples with different syntaxes from disclosed vulnerable codes; all are now fixed (*i.e.*, patched) by the development teams. What we want to emphasize here is that syntactically different VCCs can appear in practice owing to internal and external OSS modifications, and that these are difficult to discover precisely.

Example 1) We first introduce a memory allocation failure vulnerability that existed in LibZip before v1.3.0 (*i.e.*, CVE-2017-14107). Listing 1 shows the patch snippet for fixing the vulnerability. We confirmed that PHP[1], which was reusing vulnerable LibZip, backported the security patch in 2017; the patched function snippet in PHP is shown in Listing 2.

[1] `https://github.com/php/php-src`

Listing 1: A patch snippet for CVE-2017-14107 in LibZip.

```
index 3bd593b1..9d3a4cbb 100644
--- a/lib/zip_open.c
+++ b/lib/zip_open.c
@@ ... @@ _zip_read_eocd64 (...) {
  zip_cdir_t *cd;
  zip_uint64_t offset;
  zip_uint8_t eocd[EOCD64LEN];
...
      zip_error_set(error, ZIP_ER_SEEK, EFBIG);
      return NULL;
  }
-  if ((flags & ZIP_CHECKCONS)
       && offset+size != eocd_offset) {
+  if (offset+size > buf_offset + eocd_offset) {
+      /* cdir spans past EOCD record */
+      zip_error_set(error, ZIP_ER_INCONS, 0);
+      return NULL;
+  }
+  if ((flags & ZIP_CHECKCONS)
       && offset+size != buf_offset + eocd_offset) {
      zip_error_set(error, ZIP_ER_INCONS, 0);
```

Listing 2: The patched _zip_read_eocd64 function in PHP. Highlighted areas indicate the code parts that differ from the disclosed patched function. The vulnerable _zip_read_eocd64 function in PHP exhibited only 13% syntax similarity to the disclosed vulnerable function of LibZip.

```
static struct zip_cdir * _zip_read_eocd64 (...) {
  struct zip_cdir_t *cd;
  zip_uint64_t offset;
  const zip_uint8_t *cdp;
...
      _zip_error_set(error, ZIP_ER_SEEK, EFBIG);
      return NULL;
  }
  if (offset+size > buf_offset + eocd_offset) {
      /* cdir spans past EOCD record */
      _zip_error_set(error, ZIP_ER_INCONS, 0);
      return NULL;
  }
  if ((flags & ZIP_CHECKCONS)
      && offset+size != buf_offset + eocd_offset) {
      _zip_error_set(error, ZIP_ER_INCONS, 0);
```

Because PHP reused an older version of LibZip (released in 2013) with code modifications, the syntax of the vulnerable function in PHP was different from that specified in the disclosed patch for CVE-2017-14107. Considering the function as a set of code lines, when measuring Jaccard similarity [46] between the vulnerable function in PHP and the vulnerable function disclosed in LibZip, the measured similarity was 13%. Moreover, the PHP team could not directly apply the security patch owing to the syntax diversity. For these reasons, the PHP team modified and applied the disclosed security patch to their codebase, rather than updating the entire LibZip to a safe version (*i.e.*, v1.3.0 or later).

This suggests the need to discover VCCs with large code differences (in this case, 87%) from the disclosed vulnerable function. MOVERY, which considers the syntax of the oldest vulnerable function and uses only the core lines in signature generation, can discover even a VCC with such large code differences (details are explained in Section 3).

Listing 3: A patch snippet for CVE-2014-5461 in Lua 5.2.3.

```
index aafa3dca2..d02e11328 100644
--- a/ldo.c
+++ b/ldo.c
@@ -326,7 +327,13 @@ int luaD_precall (...) {
  Proto *p = clLvalue(func)->p;
- luaD_checkstack(L, p->maxstacksize);
- func = restorestack(L, funcr);
  n = cast_int(L->top - func) - 1;
+ luaD_checkstack(L, p->maxstacksize);
  for (; n < p->numparams; n++)
    setnilvalue(L->top++);
- base = (!p->is_vararg)? func + 1:
      adjust_varargs(L, p, n);
+ if (!p->is_vararg) {
+   func = restorestack(L, funcr);
+   base = func + 1;
+ }
```

Listing 4: A backporting patch for CVE-2014-5461 in Redis.

```
--- a/deps/lua/src/ldo.c
+++ b/deps/lua/src/ldo.c
@@ -276,3 +276,3 @@ int luaD_precall (...) {
  Proto *p = cl->p;
- luaD_checkstack(L, p->maxstacksize);
+ luaD_checkstack(L, p->maxstacksize + p->numparams);
  func = restorestack(L, funcr);
// No "base = (!p->is_vararg)? func + 1:..." code line
```

However, VUDDY [17], a scalable VCC discovery technique that only considers limited changes in the vulnerable function, cannot discover such VCCs. ReDeBug [14], a VCC discovery technique that considers nearby three (by default) lines of deleted and added code lines from the patch, also fails to discover such a VCC because the three lines immediately above the lines deleted from the patch (*i.e.*, lines #9, #10, and #11 in Listing 1) differ from those of the function in PHP (*i.e.*, lines #6, #7, and #8 in Listing 2).

Example 2) As an example of a VCC that does not include the code lines deleted from the patch, we introduce a buffer overflow vulnerability that existed in Lua[2] v5.1 through v5.2.2, which allows context-dependent attackers to cause a denial of service attack (*i.e.*, CVE-2014-5461). Listing 3 shows the patch snippet applied in Lua v5.2.3. We confirmed that Redis[3], which reused vulnerable Lua with code modifications, backported the security patch for CVE-2014-5461 in September 2020 (see Listing 4).

Specifically, we noted that the patch applied in Redis was different from that applied in the Lua repository. This is because Redis reused an old version of Lua (v5.1.5) where the syntax of the vulnerable function is different from what is disclosed, it was infeasible for the Redis team to apply the patch for CVE-2014-5461 as it was (*e.g.*, line #12 of Listing 3 did not exist in the reused code parts of Lua within Redis), and thus they followed the patch suggested on the official bug tracker of Lua[4].

[2] https://github.com/lua/lua
[3] https://github.com/redis/redis
[4] http://www.lua.org/bugs.html#5.2.2-1

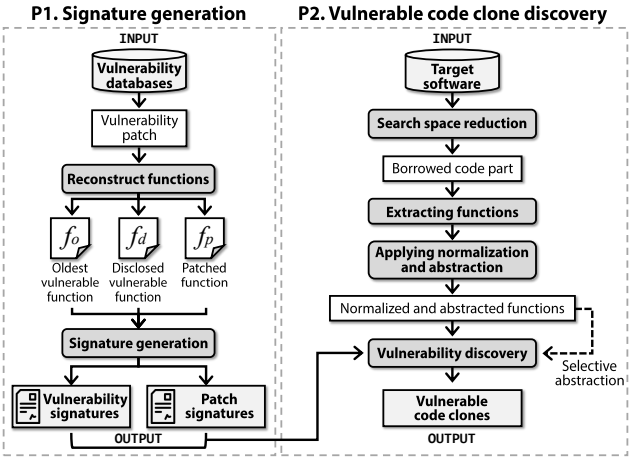


Figure 2: High-level overview of the workflow of MOVERY.

This case highlights the need to discover VCCs that were propagated with the syntax of older versions, which may not contain the code lines deleted from security patches. MOVERY makes this possible by collating vulnerable functions between the oldest and the disclosed vulnerable versions (see Section 3). However, existing VCC discovery techniques [14, 17, 49] fail to discover such a VCC when vulnerable lines they defined (*e.g.*, in this case, line #12 in Listing 3) are not contained in the VCC. In contrast, existing SCA techniques [9, 48] misinterpret that Redis still contains the vulnerability because a vulnerable version of Lua (*i.e.*, v5.1.5) is reused in Redis, despite the Redis team has backported the security patch.

3 Methodology of MOVERY

In this section, we describe the methodology of MOVERY. MOVERY comprises the following two phases: *signature generation phase* (**P1**) and *VCC discovery phase* (**P2**). In **P1**, MOVERY uses techniques called *function collation* and *core line extraction* to generate extensible signatures for addressing the syntax diversity of vulnerable codes. In **P2**, for scalability, MOVERY reduces the search space of the target software by focusing only on the code parts borrowed from other OSS. On top of that, MOVERY discovers a VCC in the target software: a function that is distinct from patch signatures and simultaneously matches the vulnerability signatures. The high-level workflow of MOVERY is shown in Figure 2.

3.1 Signature generation (P1)

Given a 3-tuple of functions, the oldest vulnerable function (f_o), the disclosed vulnerable function (f_d), and the patched function (f_p), MOVERY generates vulnerability and patch signatures (the method for collecting vulnerable and patched functions is introduced in Section 4.1).

Principles of signature generation. First, we decided that signatures should satisfy the following three principles to address the syntax diversity of vulnerable code clones.

(1) Minimization. The ability to address syntax diversity decreases as more code lines are included in the signature [49]. Hence, we need to generate a signature by collecting a minimal number of *core* code lines that are capable of discovering vulnerable code clones.

(2) Extensibility. Vulnerability signatures should be extensible (*i.e.*, from the oldest to the disclosed vulnerable functions) to address internal OSS modifications.

(3) Perceptibility. Signatures should be available to determine whether the environment in which the vulnerable code manifests (*e.g.*, control dependencies), is still preserved in the propagated vulnerable code.

Phase overview. For each vulnerability, MOVERY first examines the common code lines between f_o and f_d by collating them (*i.e.*, for extensibility). MOVERY then extracts only the following core code lines from f_o and f_d: *essential* and *dependent* vulnerable code lines (*i.e.*, for perceptibility). By gathering only the extracted essential and dependent vulnerable code lines (*i.e.*, for minimization), MOVERY generates a vulnerability signature. A similar process is applied to generating patch signatures; MOVERY analyzes f_p, extracts essential and dependent patch code lines, and then generates a patch signature by gathering them.

Function collation. Unlike existing VCC discovery techniques (*e.g.*, [14, 17, 49]) that focus only on code lines in f_d, MOVERY examines (f_o, f_d) pairs to generate extensible vulnerability signatures for addressing syntax diversity caused by internal OSS modifications. Because f_o and f_d contain the same vulnerability, we can infer that code lines (1) present in both f_o and f_d and (2) deleted from security patches would play an important role in manifesting the vulnerability (we further discuss this in Section 6).

Essential vulnerable and patch line extraction. To address syntax diversity caused by external OSS modifications, we only consider core lines in signature generation. To this end, we first define essential vulnerable and patch lines as follows.

> ◇ **Definition I. Essential vulnerable and patch lines.**
>
> We define *essential vulnerable lines* (E_V) as the code lines that are deleted in the security patch and included in both f_o and f_d. We then define *essential patch lines* (E_P), the code lines that are added in the security patch, but do not exist in both f_o and f_d.

The essential vulnerable (E_v) and patch lines (E_p) can be formally expressed as follows (let l be a source code line):

$$E_V = \{\, l \,|\, (l \in (f_d \setminus f_p)) \wedge (l \in (f_o \cap f_d)) \wedge (l \notin f_p) \,\}$$

$$E_P = \{\, l \,|\, (l \in (f_p \setminus f_d)) \wedge (l \notin (f_o \cup f_d)) \wedge (l \in f_p) \,\}$$

Listing 5: A patch snippet for CVE-2016-8654.

```
// Jasper_1.900.31/src/libjasper/jpc/jpc_qmfb.c
void jpc_qmfb_split_col (...) {
...
if (bufsize > QMFB_SPLITBUFSIZE) {
    if (!(buf = jas_alloc2(bufsize, sizeof(jpc_fix_t)))) {
        abort();
    }
}
if (numrows >= 2) {
-   hstartcol = (numrows + 1 - parity) > 1;
-   // ORIGINAL (WRONG): m = (parity) ?
                          hstartcol : (numrows - hstartcol);
-   m = numrows - hstartcol;
+   hstartrow = (numrows + 1 - parity) > 1;
+   // ORIGINAL (WRONG): m = (parity) ?
                          hstartrow : (numrows - hstartrow);
+   m = numrows - hstartrow;
    n = m;
    dstptr = buf;
    srcptr = &a[(1 - parity) * stride];
```

Listing 6: The oldest vulnerable function snippet for CVE-2016-8654. Highlighted areas indicate the code parts that differ from the disclosed vulnerable function.

```
// Jasper_1.900.1/src/libjasper/jpc/jpc_qmfb.c
void jpc_qmfb_split_col (...) {
...
if (bufsize > QMFB_SPLITBUFSIZE) {
    if (!(buf = jas_alloc(bufsize * sizeof(jpc_fix_t)))) {
        abort();
    }
}
if (numrows >= 2) {
    hstartcol = (numrows + 1 - parity) > 1;
    m = (parity) ? hstartcol : (numrows - hstartcol);
    n = m;
    dstptr = buf;
    srcptr = &a[(1 - parity) * stride];
```

As a working example, we introduce the CVE-2016-8654 case, a heap-buffer overflow vulnerability discovered in Jasper. Listing 5 shows the patch snippet based on Jasper v1.900.31, and Listing 6 shows the oldest vulnerable function snippet from Jasper v1.900.1. Among the code lines deleted from the patch (*i.e.*, lines #10 to #12 in Listing 5), only line #10 in Listing 5 exists in the oldest vulnerable function; therefore, this line belongs to E_v. Subsequently, the code lines added from the patch (*i.e.*, lines #13 to #15 in Listing 5) are included in E_p (*i.e.*, these lines belong to neither f_o nor f_d).

Dependent vulnerable and patch line extraction. Propagated essential vulnerable lines do not always guarantee that the vulnerable behavior is still maintained. To consider the environment where the vulnerability manifested, MOVERY examines dependent code lines. In particular, MOVERY pays attention to the code lines that have control or data dependencies with the essential vulnerable and patch lines, which have a major impact on vulnerability manifestation [3, 49].

> ◇**Definition II. Dependent vulnerable and patch lines.**
>
> We define *dependent vulnerable lines* (D_v) as the code lines that have control or data dependencies with the essential vulnerable lines and are included in both f_o and f_d. We then define *dependent patch lines* (D_p), the code lines in f_p that are dependent on the essential patch lines.

Let $x \mapsto_c y$ and $x \mapsto_d y$ indicate the control and data dependency of a code line x with y, respectively. Then, the dependent vulnerable code lines (D_v) and dependent patch code lines (D_p) can be formally expressed as follows:

$$D_v = \Big\{ l \,|\, \Big(l \in (f_o \cap f_d) \Big) \wedge \Big((l \mapsto_c l_v) \vee (l \mapsto_d l_v) \Big) \Big\}$$

$$D_p = \Big\{ l \,|\, \Big(l \in f_p \Big) \wedge \Big((l \mapsto_c l_p) \vee (l \mapsto_d l_p) \Big) \Big\}$$

where $l_v \in E_v$ and $l_p \in E_p$. For example, in the case of CVE-2016-8654, code lines that have control dependencies (*e.g.*, line #9 in Listing 5 and Listing 6) or data dependencies (*e.g.*, line #18 (#14) in Listing 5 (Listing 6)) with the essential vulnerable line are included in D_v.

Control flow code line extraction. Owing to the changes in control flows of the vulnerable function, the environment where the vulnerability executed may not be persisted in the cloned function; such differences in control flows can yield false positives in VCC discovery. Therefore, we decided to include vulnerable control flow code lines (F_v) into signatures.

> ◇**Definition III. Vulnerable control flow code lines.**
>
> We define *vulnerable control flow code lines* (F_v) as the conditional statements that directly related to the control flow from the entrance of the vulnerable function to the essential vulnerable code lines.

For example, in Listing 5 and Listing 6, three conditional statements (*i.e.*, lines #4, #5, and #9 in both listings) are passed before reaching the essential vulnerable lines (*i.e.*, line #10 in both listings). Similarly, for extensibility, only code lines common to f_o and f_d are considered when examining the control flow reaching the essential vulnerable lines. Consequently, lines #4 and #9 in Listing 5 and Listing 6 are defined as vulnerable control flow code lines.

Here, we consider the control flow only in vulnerable functions. If a new control flow is added through the security patch, the added lines (*e.g.*, conditional statements) are included in the essential patch lines; we decided that there was no need to include duplicate code lines in the patch signature.

Signature generation. Finally, MOVERY generates a vulnerability signature (S_v) and a patch signature (S_p) by gathering the previously extracted information.

$$S_v = (E_v,\, D_v,\, F_v)$$

$$S_p = (E_p,\, D_p)$$

One important thing is that MOVERY does not simply store code lines belonging to each element with their original syntax. Instead, MOVERY applies text-preprocessing to code lines, which can prevent false negatives caused by changes that do not affect the meaning of the vulnerable and patched code [17, 49]. Specifically, MOVERY utilizes the following two techniques: *normalization* and *abstraction*.

Listing 7: Example vulnerability signature for CVE-2016-8654.

```
Sv = (
   Ev =  [{ "norm": "hstartcol=(numrows+1-parity)>1;",
            "abst": "DVAL=(PARAM+1-PARAM)>1;" }
         ],
   Dv =  [{ "norm": "if(numrows>=2){",
            "abst": "if(PARAM>=2){" },
            "norm": "srcptr=&a[(1-parity)*stride];",
            "abst": "DVAL=&PARAM[(1-PARAM)*PARAM];" },
         ...],
   Fv =  [{ "norm": "if(bufsize>QMFB_SPLITBUFSIZE){",
            "abst": "if(DVAL>QMFB_SPLITBUFSIZE){" },
         ...]
)
```

- **Normalization.** Removing whitespaces and comments from each function and converting all the characters in the function to lower cases.
- **Abstraction.** Replacing every occurrence of parameters, variable names, variable types, and callee function names in each function with symbols `PARAM`, `DNAME`, `DTYPE`, and `FCALL`, respectively.

MOVERY first identifies all the code lines to be stored in the signatures. MOVERY then applies normalization to the given three functions (*i.e.*, f_o, f_d, and f_p), and stores the normalized form of the identified code lines into the signatures (*i.e.*, labeled as "`norm`"). Next, MOVERY applies both abstraction and normalization to the given three functions and then stores the output to the signatures (*i.e.*, labeled as "`abst`"). The reason for storing both normalized code lines in the form of abstraction applied/non-applied is to minimize false alarms that can occur because of the naive abstraction method (details are introduced in Section 3.2). Incidentally, MOVERY skips the normalized code line if the number of characters is less than 15, to prevent false alarm caused by general short code; we discuss this decision in Section 6. Listing 7 shows an example vulnerability signature for the working example.

3.2 Vulnerable code clone discovery (P2)

Given the target software (T), vulnerability signature (S_v), patch signature (S_p), and vulnerable functions (f_o and f_d), MOVERY discovers vulnerable code clones in T.

Phase overview. For scalable and precise VCC discovery, MOVERY utilizes two techniques: (1) search space reduction and (2) selective abstraction matching. First, MOVERY reduces the search space of T by focusing only on the codebases of reused OSS components, *i.e.*, MOVERY searches for VCCs only in "reuse" code regions. In addition, MOVERY uses selective abstraction matching between signatures and T to improve the discovery accuracy. Finally, MOVERY determines that a function in T is a vulnerable code clone when it (1) contains S_v, (2) does not contain S_p, and (3) has a syntax similar to the vulnerable functions (*i.e.*, f_o and f_d).

Reducing search space. When discovering VCCs in the target software T, searching the entire codebase of T is a burdensome task. Therefore, we decided to reduce the search

space by leveraging the concept of a state-of-the-art software composition analysis technique [48]. In particular, they segmented the codebase of software into the *borrowed code part* (*i.e.*, a part of the reusing third-party OSS) and the *application code part* (*i.e.*, the unique part of the software). Inspired by this approach, we focused only on the borrowed code part of T, which conceptually includes every OSS component.

Suppose MOVERY discovers a vulnerable code clone of the vulnerability V in T. To identify the borrowed code part of T, MOVERY first needs to collect the codebase of software C from which V originated. MOVERY focuses on the fact that there is at least one common function between T and C when C is reused in T [48]. The detailed process is as follows.

(1) First, MOVERY extracts all functions from T and C with their path information (*e.g.*, "`./src/file.c`").

(2) Next, MOVERY checks whether there is a common function (*i.e.*, a function that exists in common for both T and C, with exactly the same syntax) between C and T.

- **(2-1)** If there are common functions, MOVERY gathers the directory paths for every common function within T. The collected directory paths are considered as the borrowed code part of T.
- **(2-2)** If there is no common function, MOVERY determines that T does not reuse C, and subsequently, VCCs of V are not contained in T.

Using this method, MOVERY can only focus on the code where VCCs could exist. Moreover, MOVERY can skip the vulnerabilities originating in the OSS that are not reused in T; consequently, this method increases the VCC discovery scalability and performance (see Section 5.5).

Selective abstraction. MOVERY applied normalization and abstraction to all code lines in the signatures in **P1** to address code changes that preserve the semantics of functions. However, we noted that simple abstraction matching, as used in existing techniques, may impair the VCC discovery accuracy. For example, VUDDY [17] changes all variable names to the symbol `DVAL` at the time of abstraction, and then uses them for matching; if a security patch only changes variable names for fixing vulnerability (*e.g.*, Listing 5), VUDDY cannot distinguish vulnerable and patched functions.

Thus, we devised the *selective abstraction matching*. The main idea is as follows: if the abstracted syntaxes of the vulnerable and patched functions are the same, MOVERY considers only the normalized code lines in the signatures. To this end, MOVERY first applies normalization and abstraction to the entire code lines of f_d and f_p, respectively. If the syntaxes of f_d and f_p with abstraction are different, the abstracted code lines in the signatures are used for matching (*e.g.*, "`abst`" in Listing 7). However, if they are the same, this suggests that the security patch changes the part where the abstraction is applied; thus, MOVERY uses only the normalized code of the signatures in matching (*e.g.*, "`norm`" in Listing 7).

Discovering vulnerable code clones. Finally, MOVERY compares the vulnerability (S_v) and patch signatures (S_p) with the identified borrowed code part of the target software, to discover vulnerable code clones.

> ◇**Definition IV. Vulnerable code clone.**
>
> We define a function f in T is a *vulnerable code clone* if it satisfies the following conditions.
>
> - ***Cond 1)*** f should contain all code lines in S_v.
> $$\forall_{l \in S_v}.(l \in f)$$
> - ***Cond 2)*** f should not contain any code lines in S_p.
> $$\forall_{l \in S_p}.(l \notin f)$$
> - ***Cond 3)*** The syntax of f should be similar to f_o or f_d.
> $$(\texttt{Sim}(f, f_o) \geq \theta) \vee (\texttt{Sim}(f, f_d) \geq \theta)$$

MOVERY considers a function f as a VCC candidate if it contains every code line in S_v and does not include all code lines in S_p. To avoid false positives that occur when the code lines contained in signatures are very few and general, MOVERY considers the syntax similarity between f and vulnerable functions (*i.e.*, ***Cond 3***): MOVERY splits f and the vulnerable functions into sets of code lines, and then measures the Jaccard similarity [46] between them $\big($*i.e.*, $(|f \cap f_d| / |f \cup f_d|)$ and $(|f \cap f_o| / |f \cup f_o|)\big)$. Considering external OSS modifications, we set the threshold (θ) used for ***Cond 3*** to a small value (*e.g.*, 0.5). As a result, if a function satisfying the three conditions is discovered in the target software, MOVERY determines that the function is a vulnerable code clone.

In a general situation, all three conditions are considered. To cover more various types of vulnerabilities, MOVERY considers combinations of the conditions as follows:

(1) If no code line is added to the security patch, MOVERY uses only vulnerability signatures in VCC discovery (*i.e.*, considering ***Conds 1*** and ***3***).

(2) If the security patch does not contain deleted code lines, MOVERY uses only patch signatures in VCC discovery (*i.e.*, considering ***Conds 2*** and ***3***).

(3) If f_o does not exist (*e.g.*, the vulnerability only exists in a single OSS version), we determine that internal modification cannot occur, and MOVERY uses only f_d in the signature generation and in ***Cond 3***.

(4) If the code line sets of vulnerable and patched functions are the same (*e.g.*, when a security patch only changes the order of code lines), MOVERY records the order of the code lines of the vulnerable function. When a function f satisfying ***Conds 1*** and ***3*** appears, MOVERY determines that f is vulnerable only when the code line order of the vulnerable function is maintained in f.

Our experimental results demonstrate that MOVERY can also discover such VCCs with high accuracy; note that these four cases are not dominant (*i.e.*, less than 30%).

Table 2: Vulnerability dataset overview.

Category	Count (#)
Security patches	4,219
Disclosed vulnerable and patched function pairs	7,762
Oldest vulnerable functions	5,936

4 Implementation of MOVERY

In this section, we introduce the implementation of MOVERY, including the vulnerability dataset and the architecture.

4.1 Vulnerability dataset

Collecting security patches. We collected security patches by leveraging the method used in the previous approaches [19, 32]; we examined CVEs in the National Vulnerability Database (NVD) and checked if `git commit` URLs (*i.e.*, security patch commits) were included in the references. We then gathered security patches by crawling such patch commits from the corresponding Git repositories. We chose the C/C++ vulnerabilities as our initial targets, because code fragments reuse and modified OSS reuse are prominent in C/C++ software [17, 47–49]. Consequently, we collected 4,219 C/C++ security patches from the NVD (as of March 2021).

Reconstructing functions. We then reconstructed vulnerable and patched functions from the collected security patches. We focused on the header of a security patch, which provides file commits before and after applying the patch [17, 49]. For example, in Listing 3, "`aafa3dca2`" and "`d02e11328`" indicate the commit of the vulnerable and patched file, respectively. After accessing the vulnerable (*resp.* patched) file, we extracted every function containing code lines deleted (*resp.* added) from the patch as the disclosed vulnerable function f_d (*resp.* patched function f_p). Here we excluded (f_d, f_p) pairs that the code changes were not intended to fix vulnerabilities, such as comment or whitespace changes.

To reconstruct the oldest vulnerable function (f_o), we identified the *oldest vulnerable version* of the software from which the vulnerability originated, by referring to Common Platform Enumeration (CPE) [30] of the NVD. Since NVD may provide incorrect CPEs for some CVEs [47], we manually examined versions sequentially starting from the oldest version specified in the CPE: if a version containing a function with the same name as f_d is first detected, we determined that the version is the oldest vulnerable version. When software is maintained with parallel versioning (e.g., OpenSSL 1.0.0* and 1.0.1*), we infer the version order based on the version release date. We then accessed the oldest vulnerable version and extracted f_o that: (1) had the same name as f_d, and (2) existed in the path of the vulnerable file; we did not reconstruct f_o where the syntax of f_o was the same as that of f_d.

Consequently, we reconstructed *7,762 (f_d, f_p) pairs* and *5,936 oldest vulnerable functions* (see Table 2).

Dataset observations. We noted that internal modifications of OSS can play a leading role in the syntax diversity of vulnerable codes. To be specific, among the 5,936 vulnerable functions that exist in multiple OSS versions, we found the 4,623 cases (78%) where f_d and f_o had different syntaxes. When we measured the Jaccard similarity between every f_d and f_o pair, the average similarity score was 56%. The facts that f_o and f_d are different in many cases (*i.e.*, 78%) and that OSS internal updates change the vulnerable code syntax on average by 44%, suggest that internal OSS modifications should be considered in VCC discovery.

4.2 Architecture

MOVERY comprises the following three modules: (1) a *dataset collector*, (2) a *signature generator*, and (3) a *VCC discoverer*. The dataset collector gathers security patches and reconstructs vulnerable and patched functions. The signature generator, literally, generates vulnerability and patch signatures for every collected vulnerability. Finally, the VCC discoverer performs the actual vulnerable code clone discovery on the target software. Each module consists of 800 to 1,000 lines of Python code, excluding external libraries such as function parsers. The source code of MOVERY is publicly available at `https://github.com/wooseunghoon/MOVERY-public`.

Function parser and analyzer. To extract functions from vulnerable files, patched files, and the target software, we utilized universal Ctags [6], a precise and fast open-source function parser. In addition, we used the Joern parser [50] to examine control and data dependencies of functions for generating signatures. In particular, MOVERY generates a code property graph (*i.e.*, a graph that incorporates an abstract syntax tree, a control flow graph, and a program dependency graph) of the extracted vulnerable and patched functions using Joern parser; from the graph, MOVERY can obtain (1) the entire control flow of the function, and (2) dependencies between each code line, which are used to generate signatures.

5 Evaluation

In this section, we evaluate MOVERY. Section 5.1 investigates how accurately MOVERY discovers VCCs in modified components, compared to existing VCC discovery techniques (*i.e.*, VUDDY [17] and ReDeBug [14]). Section 5.2 and Section 5.3 compare MOVERY to the recurring vulnerability detection technique (*i.e.*, MVP [49]) and the SCA-based technique (*i.e.*, CENTRIS [48]), respectively, We evaluate the speed and scalability of MOVERY in Section 5.4, and introduce the efficacy of the search space reduction in Section 5.5. Finally, Section 5.6 presents a case study observed in our experiments. We ran MOVERY on a machine with Ubuntu 16.04, Intel Xeon Processor @ 2.40 GHz, 32GB RAM, and 6TB HDD.

Table 3: Target software overview.

IDX	Name	Version	#Line*	#Comp†	Domain
T1	FreeBSD	v12.2.0	14,489,534	47	Operating system
T2	ReactOS	v0.4.13	6,419,855	23	Operating system
T3	ArangoDB	v3.7.9	3,064,973	22	Database
T4	FFmpeg	n4.3.2	1,230,520	4	Multimedia processing
T5	OpenCV	v4.5.1	1,092,317	15	Computer vision
T6	Emscripten	v2.0.15	759,020	11	Compiler
T7	Crown	v0.42.0	723,372	20	Game engine
T8	Git	v2.31.0	293,467	5	Version control system
T9	OpenMVG	v1.6	262,610	8	Image processing
T10	Redis	v5.0.12	212,672	8	Database
Total	-	-	**28,548,340**	**190**	-

*: Counting only C/C++ code lines, †: The number of modified OSS components.

5.1 Accuracy of MOVERY

Target software selection. We selected target software based on the following three criteria to claim the generality of MOVERY: they (1) should be popular C/C++ OSS, (2) should contain a sufficient number of modified components, and (3) should not be biased toward any particular domain.

We first collected C/C++ repositories from GitHub that exhibit more than 1,000 stargazer counts, *i.e.*, a popularity indicator on GitHub. We then leveraged CENTRIS [48] to rank the collected software based on the number of modified OSS components. While examining the ranked software in descending order, we selected target software for which one or more VCCs were discovered by MOVERY.

Table 3 summarizes the selected ten target software programs. They were selected based on clear criteria, were obtained from diverse domains and had various code sizes (*i.e.*, ranging from 212,672 to 14,489,534 C/C++ lines of code), thus we decided that the selected target software can add generality to the experiments. Note that the cumulative code size of our dataset is 80 times larger than that used in VUDDY [17], and almost the same as the dataset used in MVP [49].

Methodology. We selected two VCC discovery techniques for accuracy comparison: ReDeBug [14] and VUDDY [17]. We applied MOVERY, ReDeBug, and VUDDY to ten target software and evaluated the VCC discovery results. We used ReDeBug and VUDDY with their default options by referring to their papers [14, 17]. We selected θ as 0.5 in MOVERY (related experiments are introduced at the end of this section).

To evaluate the accuracy, we used the following five metrics: true positives (TP), false positives (FP), false negatives (FN), precision ($\frac{\#TP}{\#TP+\#FP}$), and recall ($\frac{\#TP}{\#TP+\#FN}$). Because it is infeasible to find every vulnerable code in the target software, we cannot easily determine FNs. Therefore, we only measure indisputable FNs; for example, FNs of MOVERY are the VCCs (*i.e.*, not FPs) detected by VUDDY and ReDeBug but not discovered in MOVERY [17, 49]. TPs and FPs are determined by manual verification performed by two security analysts. We manually viewed the discovered VCCs and the environment where the vulnerability manifested to determine whether the discovered VCC is correct.

Listing 8: A patch snippet for CVE-2014-9669 in FreeType2.

```
1 tt_cmap12_validate (...) {
2   num_groups = TT_NEXT_ULONG( p );
3   if ( length > (FT_ULong)( valid->limit - table ) ||
4 -      length < 16 + 12 * num_groups                 )
5 +      /* length < 16 + 12 * num_groups ? */
6 +      length < 16                                   ||
7 +      ( length - 16 ) / 12 < num_groups             )
```

Listing 9: The oldest vulnerable function snippet for CVE-2014-9669. Highlighted areas indicate the code parts that differ from the disclosed vulnerable function.

```
1 // Extracted from FreeType2 v2.1.10 ("./src/sfnt/ttcmap.c").
2 tt_cmap12_validate (...) {...
3   num_groups = TT_NEXT_ULONG( p );
4   if ( table + length > valid->limit || length < 16 + 12 *
        num_groups )
```

Overview of accuracy measurement. Table 4 summarizes the VCC discovery results of MOVERY, VUDDY, and ReDeBug. The discovered 434 VCCs were found from 121 CVEs. Specifically, CVE vulnerabilities that were discovered in ReactOS contained many vulnerable functions, thus the number of discovered VCCs was considerable.

Owing to the OSS modifications, 396 VCCs (91%) existed in a different syntax to the disclosed vulnerable function. Nevertheless, MOVERY discovered 415 VCCs with 96% precision and 96% recall, whereas ReDeBug discovered 163 VCCs with 65% precision and 38% recall and VUDDY discovered 72 VCCs with 77% precision and 17% recall (Appendix A presents the modification types and Appendix B provides the severity and vulnerability types of the discovered VCCs).

Notably, the VCC discovery results of MOVERY included all VCCs discovered by VUDDY, and covered 144 (88%) VCCs discovered by ReDeBug, whereas VUDDY and ReDeBug were only able to cover 72 (17%) and 144 (35%) VCCs discovered by MOVERY, respectively.

FNs of ReDeBug and VUDDY. Existing techniques failed to discover many VCCs owing to the *OSS modifications*. ReDeBug did not discover 271 VCCs (62.4%) where the code lines deleted in the security patch did not exist, or the code syntax was modified beyond what ReDeBug could handle. VUDDY failed to discover 362 VCCs (83.4%), in which modifications occurred in code parts other than that VUDDY could address (*e.g.*, Type-3 clones). An example is CVE-2014-9669, an integer overflow vulnerability that exists in FreeType2 (see Listing 8). We confirmed that line #4 in Listing 8 (*i.e.*, defined as the vulnerable line by ReDeBug and VUDDY) existed in the oldest vulnerable function as a different syntax (*i.e.*, line #4 in Listing 9) because of the internal OSS modification, resulting in yielding FNs of ReDeBug and VUDDY.

FPs of ReDeBug and VUDDY. The reasons for FPs in ReDeBug were the lack of code normalization and the exclusion of function semantics. Security patch may contain non-security changes (*e.g.*, comments changes). Owing to the lack of code normalization, ReDeBug misinterpreted that the

Table 4: Accuracy of ReDeBug, VUDDY, and MOVERY in vulnerable code clone discovery.

Target software	#Discovered VCCs	ReDeBug					VUDDY					MOVERY				
		#TP	#FP	#FN	Precision	Recall	#TP	#FP	#FN	Precision	Recall	#TP	#FP	#FN	Precision	Recall
ReactOS	210	31	9	179	0.78	0.15	8	0	202	1.00	0.04	207	3	3	0.99	0.99
OpenCV	72	38	15	34	0.72	0.53	26	2	46	0.93	0.36	72	3	0	0.96	1.00
Emscripten	56	22	8	34	0.73	0.39	9	1	47	0.90	0.16	50	4	6	0.93	0.89
FreeBSD	33	25	44	8	0.36	0.76	6	16	27	0.27	0.18	27	4	6	0.87	0.82
Crown	23	22	2	1	0.92	0.96	14	2	9	0.88	0.61	23	2	0	0.92	1.00
OpenMVG	23	15	5	8	0.75	0.65	4	0	19	1.00	0.17	19	0	4	1.00	0.83
ArangoDB	6	4	1	2	0.80	0.67	2	0	4	1.00	0.33	6	2	0	0.75	1.00
FFmpeg	5	2	2	3	0.50	0.40	0	1	5	0.00	0.00	5	1	0	0.83	1.00
Redis	5	3	0	2	1.00	0.60	3	0	2	1.00	0.60	5	0	0	1.00	1.00
Git	1	1	1	0	0.50	1.00	0	0	1	N/A	0.00	1	0	0	1.00	1.00
Total	434	163	87	271	0.65	0.38	72	22	362	0.77	0.17	**415**	**19**	**19**	**0.96**	**0.96**

Table 5: VCCs that are hardly discovered by existing techniques.

Types	Description
T1	VCCs without code lines deleted in security patches.
T2	VCCs with various syntaxes derived from f_o.
T3	VCCs with heavy syntax change.

Listing 10: A patch snippet for CVE-2017-14039 in OpenJPEG.

```
1 static OPJ_BOOL opj_j2k_write_sot(opj_j2k_t *p_j2k, ...,
2                       const opj_stream_private_t *p_stream,
3                       opj_event_mgr_t * p_manager){
4     ...
5     OPJ_UNUSED(p_stream);
6 -   OPJ_UNUSED(p_manager);
```

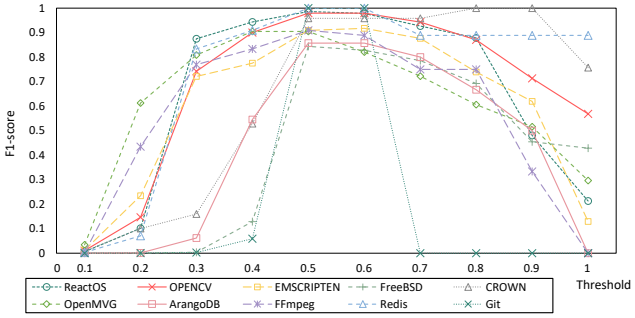


Figure 3: Experimental results for measuring efficiency of θ.

code to which non-security changes were not applied to be vulnerable. In addition, ReDeBug misinterpreted a patched function as vulnerable when the last few added code lines are the same as the code lines before the added code lines [17,49]. The FPs of VUDDY were caused by abstraction; if a security patch only fixes VUDDY's abstraction targets (*e.g.*, variable names), VUDDY cannot differentiate between vulnerable and patched functions, producing FPs.

***The accuracy of* MOVERY.** Table 5 summarizes the types of VCCs that were discovered by MOVERY but hardly discovered by ReDeBug and VUDDY. Among the 396 VCCs (TPs) discovered by MOVERY, 32 VCCs (8%) belonged to **T1**, and 221 VCCs (56%) showed a higher code similarity with f_o than that with f_d (**T2**); 166 of 221 VCCs exhibited that the code similarity with f_d was less than 50% (**T3**). Because MOVERY can address the modified OSS reuse, it could discover such VCCs. Using selective abstraction, MOVERY could eliminate 21 out of 22 FPs reported by VUDDY. MOVERY could discover 34 Type-2 [34] VCCs that ReDeBug could not discover.

However, MOVERY produced FPs when the abstraction was applied to similar code lines in a function. For example, after applying abstraction, both lines #5 and #6 in Listing 10 are converted to "`FCALL(PARAM);`". Hence, even if the patch was applied, MOVERY misinterpreted that the vulnerable line still existed, producing an FP. Another reason for FPs is syntactically similar functions. Suppose that security patches p and p' are applied to syntactically similar functions f and f', respectively. In such rare cases, MOVERY misinterprets that p' (*resp.* p) is not applied to f (*resp.* f'), producing FPs.

MOVERY reported FNs when semantically similar but syntactically changed code appeared (*e.g.*, for → while), or when the similarity between a VCC and the vulnerable function was less than θ. However, simply decreasing θ may yield more FPs; we believe that the current MOVERY approach offers a good balance of precision and recall, as MOVERY significantly outperformed existing techniques.

Threshold sensitivity. We used the θ value as 0.5 in the VCC discovery experiments. To measure threshold sensitivity, we evaluated each VCC discovery result of MOVERY while increasing θ by 0.1 from 0 to 1. Because it is infeasible to verify the numerous newly discovered VCCs, we measured the precision and recall with each θ based on the previously discovered 434 VCCs. In addition, we evaluated each θ by measuring the F1-score [45] $\left(\frac{2*\text{precision}*\text{recall}}{\text{precision}+\text{recall}}\right)$, as we focused on the balance between precision and recall.

Figure 3 presents the measurement results. We confirmed that the F1-score appeared the highest when θ was 0.5. For Crown and Emscripten, the F1-score was higher when θ is greater than 0.5; this is because the ratio of modified VCCs was small and thus FPs decreased at higher θ. Overall, the precision drops when θ is less than 0.5, while recall decreases when θ is greater than 0.6. Hence, we believe that θ of 0.5 maintains a good balance between high precision and recall.

One thing to note is that MOVERY can discover 15 VCCs that were not discovered in the previous experiments when it utilizes θ less than 0.5. In contrast, MOVERY still produces 4 FNs and 3 FPs even when θ is 0 and 1, respectively; we discuss this issue in Section 6 in detail.

Table 6: VCC discovery results of MVP and CENTRIS.

Name	#VCCs*	#TP	#FP	#FN	#Unique†	Precision	Recall
MVP	266	220	46	184	8	0.83	**0.54**
CENTRIS	553	281	272	152	37	**0.51**	0.65

*: #VCCs discovered for ten target software, †: #VCCs not discovered by MOVERY.

5.2 Comparison with MVP

We then compare MOVERY with MVP [49], a recurring vulnerability detection technique, to demonstrate that MOVERY is more effective in discovering VCCs from modified components. Since MVP is not available owing to the commercial issue, we implemented MVP based on their paper [49] with their default options.

Result analysis. When we applied MVP to ten target software, it discovered 220 VCCs (*i.e.*, TPs) while reporting 46 FPs and 184 FNs (see Table 6). It is noteworthy that MVP hardly discovered VCCs belonging to **T1** and **T2** (see Table 5), because MVP does not consider internal OSS modifications. MVP can only discover VCCs that contain all the code lines deleted from the security patch, and therefore it failed to discover 32 VCCs that do not contain such deleted code lines (**T1**). In addition, MVP failed to discover 142 VCCs with various syntaxes derived from f_o (**T2**). Specifically, variable types and caller function names in vulnerable functions are frequently changed during internal OSS modifications; MVP does not consider such changes, thereby yielding FNs. The remaining FNs were caused by semantically similar but syntactically changed code and threshold issues, similar to MOVERY.

MVP produced 46 FPs owing to the (1) syntactically similar functions and (2) the abstraction method. MVP misinterpreted a safe function as vulnerable when the function has a similar syntax to the vulnerable function but is semantically different (11 FPs). The remaining 35 FPs were caused by the abstraction method: if the security patch only fixes the abstraction targets of MVP (*e.g.*, variable names), MVP fails to differentiate between a vulnerable function and a patched function, producing FPs. Note that MOVERY could eliminate 31 of 35 FPs by utilizing the selective abstraction method.

Finally, MVP discovered 8 VCCs that MOVERY misses; all of these are VCCs with syntax similarities less than θ to the disclosed vulnerable functions; two of these VCCs were also not found in VUDDY and ReDeBug.

Our analysis results affirmed that MVP, which is not capable of addressing internal OSS modifications, may not be effective in discovering VCCs from modified components as it reported much more FNs (*i.e.*, 184 FNs) than MOVERY.

Threats to validity. Although we implemented MVP based on their paper, some functions may not be reproduced perfectly. In addition, the purpose of MVP is to detect recurring vulnerabilities and not to discover propagated vulnerable codes. Our intention is not to deny the effectiveness of MVP, but to demonstrate that MOVERY is more efficient in discovering VCCs from modified components.

5.3 Comparison with CENTRIS

Several existing approaches (*e.g.*, [9, 48]), including commercial tools such as Trivy [1] and Black Duck [38], have attempted to discover vulnerabilities contained in OSS components by clarifying reused OSS components and their versions. However, they do not seem to consider the modified OSS reuse, and further, they do not disclose detailed vulnerability discovery algorithms (even their algorithms frequently change). Hence, we decided to compare the VCC discovery results of MOVERY with CENTRIS [48], a recent SCA technique for identifying modified OSS components.

To investigate the vulnerabilities that affect the identified components, we utilized the "product search" and "version search" provided by CVE Details [7], *i.e.*, functionalities for providing CVEs that affect a given software name and version. For every component identified in each target software, we investigated CVEs affecting the component and then considered them the VCCs discovered by CENTRIS.

Result analysis. In our experiments, CENTRIS-based approach discovered 553 VCCs in ten target software, of which 272 (49%) VCCs were confirmed to be FPs (see Table 6). Such FPs occurred when: (1) the vulnerable code was not reused, or (2) the vulnerable code was patched through backporting (an example is introduced in Section 2.2).

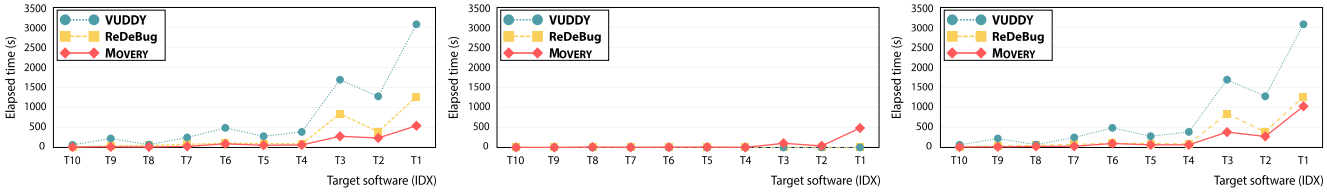
Furthermore, the CENTRIS-based approach failed to discover 152 (38%) out of 396 VCCs discovered by MOVERY, especially when (1) there is a vulnerability in a component that CENTRIS failed to identify, or (2) CENTRIS predicted incorrect version information. Finally, CENTRIS-based approach discovered 37 VCCs that were not discovered by MOVERY. We observed that all of them were the cases where security patches did not be released via Git; if these patches are added to our dataset, MOVERY can also discover them.

Consequently, we confirmed that discovering vulnerabilities simply based on the component name and version produced many false results, especially FPs (*i.e.*, 272 FPs), owing to the OSS modifications. This demonstrates that MOVERY considerably outperformed the SCA-based approach in terms of discovering vulnerabilities from modified components.

5.4 Speed and scalability of MOVERY

In this section, we evaluate the speed and scalability of MOVERY in VCC discovery. We classify the total time taking to discover VCCs into *signature generation* (*i.e.*, the elapsed time for generating signatures), *target preprocessing* (*i.e.*, the elapsed time for analyzing the target software), and *matching* times (*i.e.*, the elapsed time required to discover the VCCs). We measured all the times for MOVERY, ReDeBug, and VUDDY, and then compared the results.

Signature generation time. The signature generation time using the collected 4,219 security patches (see Table 2) was 2 h in VUDDY and 32 h in MOVERY; because ReDeBug



(a) Target preprocessing times. **(b) Matching times.** **(c) Total times (preprocessing + matching).**

Figure 4: Target preprocessing and matching time spent on target software with various sizes.

utilized the security patches as they were, there was no need for signature generation time. Unlike VUDDY, which simply extracts vulnerable functions, MOVERY required more time as it needed to reconstruct the oldest vulnerable functions and analyze the code line dependencies of functions; note that the signature generation is a one-time task.

Target preprocessing and matching times. Figure 4 shows the target preprocessing and matching times for the three tools. The results present the following three main observations.

(1) VUDDY requires the longest target preprocessing time.
(2) MOVERY requires the longest matching time.
(3) In total elapsed time, MOVERY requires the least amount of time and VUDDY requires the longest time (owing to its long preprocessing time).

To precisely discover VCCs, MOVERY used the finer granularity (*i.e.*, a set of code lines) that is slower than function units in VCC discovery [17], and further considered dependencies of code lines in matching, resulting in a longer matching time than ReDeBug and VUDDY. Nevertheless, (1) target preprocessing and matching could be performed within 200 s in MOVERY per target software (*i.e.*, the fastest among the three tools) and (2) the time was not significantly increased even when the lines of code of the target software varied from 213 K to 14.5 M (see Table 3), suggesting that MOVERY is sufficiently fast and scalable for practical use.

5.5 Efficacy of the search space reduction

To reduce the VCC search space, MOVERY focuses only on the borrowed code parts of the target software. Here we evaluate the efficacy of the search space reduction technique.

Scalability improvement. The cumulative number of directories and code lines in ten target software were 539,781 and 28,548,340, respectively. When considering only borrowed code parts, the number of directories decreases to 375,489 (70%), and the number of code lines to be scanned was reduced by 15,130,620 (53%), suggesting that MOVERY can skip approximately half of the software's codebase that does not need to be scanned, which ensures higher scalability.

Accuracy enhancement. In addition, we confirmed that focusing on borrowed code parts can reduce FPs in VCC discovery. Existing approaches search VCCs for the entire codebase; searching for "propagated" VCCs outside the "reuse" code regions makes existing approaches misinterpret a function, which has similar syntax but completely different semantics with vulnerability signature, as a VCC, thereby producing more FPs. Quantitatively, 3 FPs and 24 FPs of VUDDY and ReDeBug were discovered outside of the "reuse" code regions, respectively. MOVERY was able to reduce such FPs by considering only the vulnerabilities of the reused OSS and the code parts where the OSS exists.

5.6 Case study: Vulnerability in Git

MOVERY discovered that the fix for CVE-2019-9169 (*i.e.*, a heap-based buffer over-read vulnerability existed prior to Glibc v2.30) is not applied to the latest version of Git, which is one of the most popular version control systems. Since Git reused an older version of Glibc earlier than v2.27, the syntax of the VCC discovered in Git was quite different (*i.e.*, the syntax similarity was 65%) from that specified in the disclosed patch for CVE-2019-9169. Worse, we confirmed that this vulnerability can still cause memory leaks in the latest version of Git. We responsibly reported this to the Git team; they confirmed our report and replied that it will be addressed in a later task because they determined that this vulnerability does not presently pose a serious threat (we discuss responsible vulnerability disclosure in Section 6).

6 Discussion

Threshold setting. We used two threshold values: we skipped normalized code lines when the number of characters is less than 15, and used θ of 0.5 in the experiment. Since the related experiments of the latter case were introduced in Section 5.1, here we only discuss the former case.

Skipping short code lines was determined from our observations: when we manually inspected normalized code lines with less than 15 characters, more than 90% were observed as frequently appearing in non-vulnerable codes such as return statements, common-named variable declarations, parentheses, "else", and "continue" statements. If the vulnerability signature includes such a short code line, it could potentially lead MOVERY to produce FPs. When we slightly increased the character-length limit (*e.g.*, 16 or 17), we observed that the ratio of non-vulnerable code lines decreased. Therefore, we set the threshold value as 15 characters. Note that this threshold is dependent on the vulnerability dataset. If MOVERY is applied in a different vulnerability dataset, this threshold needs to be adjusted.

Use of the oldest vulnerable function. When generating signatures, MOVERY considers the oldest vulnerable function to address internal OSS modifications. In fact, MOVERY can use any older version's vulnerable function whose syntax differs from the disclosed vulnerable function; the reason for selecting the *oldest* one is to cover more internal modifications.

We introduce the rationale behind considering only the common code line between the oldest (f_o) and disclosed vulnerable functions (f_d). Suppose we want to extract essential vulnerable code lines (see Section 3.1) from the vulnerable functions. Such essential vulnerable code lines can (1) exist in both f_o and f_d, (2) not exist in both f_o and f_d, or (3) exist in either one. Let v_o be the oldest vulnerable version, and v_d be the latest (*i.e.*, disclosed) vulnerable version. Here we can make two inferences. First, if the essential vulnerable code lines do not exist in both f_o and f_d, then v_o and v_d should be excluded from the versions affected by vulnerability. Next, if the essential vulnerable code lines exist only in f_d and not in f_o, then v_o should be left out of the affected versions.

Based on these inferences, we can conclude that the essential vulnerable code lines are simultaneously included in f_o and f_d as long as the version information affected by the vulnerability includes both v_o and v_d. In this context, we can justify our approach as long as the NVD, which is our data source (see Section 4.1), provides correct CPE.

Vulnerability disclosure. We reported 14 triggerable VCCs (*e.g.*, using Proofs-of-Concept), which were discovered in the target software such as Git and OpenMVG, and in other popular OSS projects such as LibAV and LibGDX, to the development teams; it is noteworthy that 10 out of 14 VCCs were discovered only in MOVERY and not in previous VCC discovery techniques (*e.g.*, [14, 17, 49]).

(1) Vulnerability confirmed. Nine development teams confirmed our vulnerability reports, of which two of them were patched, and another two of them will be resolved.

(2) Under discussions. For the remaining five cases, we are still discussing or waiting for answers (*e.g.*, LibAV and OpenMVG), of which one pull request is pending.

Other VCCs that have not yet been successfully reproduced are on hold to report because we can hardly receive a response from the development teams even if reported. We will not disclose any VCCs until a security patch is applied, and we plan to trigger such VCCs with the help of a collaborator or refer to the related approaches (*e.g.*, [18, 27]) for triggering a propagated vulnerability; if a VCC is successfully reproduced, we will immediately report it to the development team.

Limitations. MOVERY leverages some assumptions that can limit its application. First, MOVERY can discover VCCs when the source code for the target software is available; if the control and data dependencies of functions in a binary can be accurately investigated, we expect that the methodology of MOVERY may be applied to binary-level VCC discovery.

Second, NVD may provide incorrect CPEs for some CVEs [8, 47], which could impair the accuracy of reconstructing older vulnerable functions. Although we manually verified the CPEs (see Section 4.1), this is inefficient for practical use, and if the correct version range (CPEs) of a vulnerability is provided, the effectiveness of MOVERY will be improved.

Third, MOVERY considers vulnerabilities within functions and cannot discover VCCs whose patches are out of functions (*e.g.*, inter-functional or C preprocessor-dependent vulnerabilities). Because MOVERY considers a function as a basic unit, it cannot address C preprocessor-dependent vulnerabilities, but we are considering addressing inter-functional vulnerabilities by including the correlation information of such inter-functions (*e.g.*, inter-function data flows) into signatures.

Last, even if we set θ to extremes (*i.e.*, 0 and 1), MOVERY still produces 4 FNs and 3 FPs; FNs were caused by VCCs with similar semantics but syntactically changed code, and FPs were produced owing to the syntactically similar functions with different patches applied (see Section 5.1). If MOVERY can handle Type-4 clones and if abstraction is not used, such FNs and FPs can be reduced, respectively. However, this may rather compromise the scalability and accuracy of MOVERY. Therefore, we retain the current MOVERY approach, which showed much superior performance than the existing approaches, and leave solving FPs and FNs that are independent of θ for future work.

7 Related Work

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

Code clone detection techniques. There are numerous techniques attempting to detect code clones (*e.g.*, [4, 10, 15, 16, 24, 28, 29, 33–36, 41, 42]). However, their concern is not discovering a *vulnerable* code clone; because such techniques do not consider the vulnerability characteristics, they yield many FPs when applied to VCC discovery [17, 49].

Software composition analysis techniques. Several techniques attempt to identify OSS components in the target software (*e.g.*, [2, 9, 21, 25, 39, 48, 51, 52]); some of these can be applied to discover known vulnerabilities in OSS components. For example, Duan *et al.* [9] proposed OSSPolice to identify 1-day security vulnerabilities from the libraries of an Android application. OSSPolice utilized constant features to extract the versions of libraries, and determined if vulnerable versions were used in the target Android application. Zhan *et al.* [51] proposed ATVHunter to precisely detect versions of third-party libraries by using the control flow information of an Android application. Using the identified versions, ATVHunter verified whether the target application contained known security vulnerabilities. However, if developers backport security patches to the vulnerability or do not reuse the vulnerable code, these techniques produce FPs (see Section 5.3). Hence, they are insufficient to solve our target problem.

VCC discovery techniques. Jang *et al*. proposed ReDeBug [14], which is a token-based VCC discovery approach using the slicing window technique. Kim *et al*. proposed VUDDY [17], a function-level scalable VCC discovery technique. Bowman *et al*. proposed VGRAPH [3], a CPG-based VCC discovery technique, which is more robust to code modification, especially for Type-3 code clones. Xiao *et al*. proposed MVP [49], a recurring vulnerability detection approach. By considering only the sliced code lines that are directly related to vulnerabilities, MVP can discover VCCs with low syntax similarity of disclosed vulnerable functions.

However, these existing techniques (1) cannot precisely discover modified VCCs caused by internal OSS modifications, and (2) can only discover VCCs with code lines that have been deleted from security patches, thereby showing low accuracy when applied to our target problem (see Section 5.1 and Section 5.2). Some other techniques attempted to discover vulnerable codes based on learning algorithms (*e.g*., [22, 23]) or to detect buggy codes (*e.g*., [20, 26]). They hold the promise of detecting general vulnerable or buggy code, however, they are not capable of precisely discovering VCCs propagated by modified OSS reuse.

8 Conclusion

Discovering propagated vulnerabilities from modified OSS components is a pressing issue, because unpatched vulnerabilities can pose a critical threat to the entire software. In regards to this, we present MOVERY, a precise approach that discovers VCCs from modified OSS components. Our experimental results affirmed that MOVERY significantly outperformed existing VCC discovery techniques in terms of VCC discovery accuracy. Equipped with VCC discovery results from MOVERY, developers can address potential threats caused by propagated vulnerabilities in modified OSS components, rendering a safer software ecosystem.

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, No.2022-0-01198 Convergence Security Core Talent Training Business, and No.IITP-2022-2020-0-01819 ICT Creative Consilience program).

Availability

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

References

[1] Aqua Security. Trivy: scanner for vulnerabilities in container images, file systems, and Git repositories, as well as for configuration issues, 2022. `https://aquasecurity.github.io/trivy/v0.22.0/`.

[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] Common Weakness Enumeration. 2021 CWE Top 25 Most Dangerous Software Weaknesses, 2021. `https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html`.

[6] Ctags. Universal Ctags, 2021. `https://github.com/universal-ctags/ctags`.

[7] CVE Details. The Ultimate Security Vulnerability Data source, 2021. `https://www.cvedetails.com`.

[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] 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.

[11] GitHub. *The GitHub Blog - Thank you for 100 million repositories*, 2018. `https://github.blog/2018-11-08-100m-repos/`.

[12] GitHub. *The 2020 State of the OCTOVERSE*, 2020. https://octoverse.github.com/.

[13] Red Hat. Backporting Security Fixes, 2021. https://access.redhat.com/security/updates/backporting.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] Jingyue Li and Michael D Ernst. CBCD: Cloned Buggy Code Detector. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 310–320, 2012.

[21] 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.

[22] 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.

[23] 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.

[24] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, volume 4, pages 289–302, 2004.

[25] 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.

[26] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Identifying Code Clones having High Possibilities of Containing Bugs. In *Proceedings of the IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 99–109, 2017.

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

[28] Ginger Myles and Christian Collberg. K-gram Based Software Birthmarks. In *Proceedings of the 20th ACM Symposium on Applied Computing (SAC)*, pages 314–318, 2005.

[29] 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.

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

[31] NVD. National Vulnerability Database, 2021. https://nvd.nist.gov/.

[32] 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 22nd ACM SIGSAC Conference on*

Computer and Communications Security (CCS), pages 426–437, 2015.

[33] 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.

[34] Chanchal Kumar Roy and James R Cordy. A Survey on Software Clone Detection Research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[35] 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.

[36] 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.

[37] Synopsys. *Open source security and risk analysis report (OSSRA)*, 2021.

[38] Synopsys. Black Duck Software: Software Composition Analysis, 2022. `https://www.blackducksoftware.com/`.

[39] 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.

[40] TechRepublic. *Backward compatibility issues can derail your development efforts*, 2001. `https://www.techrepublic.com/article/backward-compatibility-issues-can-derail-your-development-efforts/`.

[41] Tijana Vislavski, Gordana Rakic, Nicolás Cardozo, and Zoran Budimac. LICCA: A Tool for Cross-Language Clone Detection. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 512–516, 2018.

[42] 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.

[43] Wikipedia. Backporting, 2021. `https://en.wikipedia.org/wiki/Backporting`.

[44] Wikipedia. Backward compatibility, 2021. `https://en.wikipedia.org/wiki/Backward_compatibility`.

[45] Wikipedia. F-score, 2022. `https://en.wikipedia.org/wiki/F-score`.

[46] Wikipedia. Jaccard Index, 2022. `https://en.wikipedia.org/wiki/Jaccard_index`.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

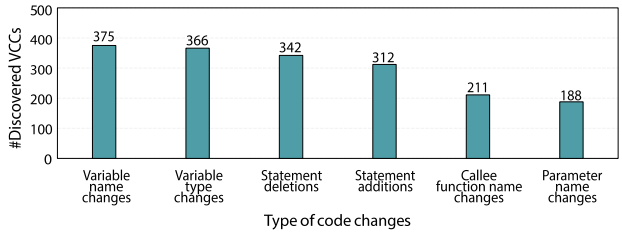
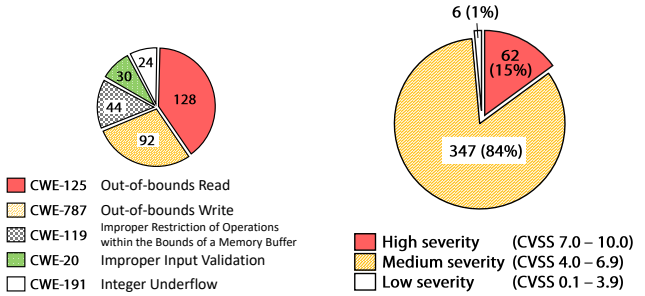


Figure 5: Types of code modifications in VCCs.



(a) Top 5 CWE distribution. **(b) CVSS distribution.**

Figure 6: CWE and CVSS distributions for the discovered VCCs by MOVERY.

Appendix A Code modification types of VCCs

We analyzed 415 VCCs discovered by MOVERY to better understand the types of code modifications. Figure 5 shows the analysis results; note that a VCC may contain multiple code modification types.

It is worth noting that the variable names and variable types of the discovered VCCs are mostly different from disclosed vulnerable functions. We confirmed that most of these cases were caused by internal OSS modifications. Approaches using the abstraction method such as MOVERY, VUDDY, and MVP could address this type of code modification.

Moreover, we confirmed that the cases, in which code lines of the disclosed vulnerable function were deleted or new code lines were added, accounted for more than 80% of the discovered VCCs. MOVERY, considering only the core lines of the vulnerable code, could respond to this code modification, but VUDDY, considering the syntax of the entire vulnerable function, hardly discovered VCC to which this code modification was applied.

All code modification types should be considered in VCC discovery. We confirmed that MOVERY's extensible and minimized signature generation made this possible while other existing approaches failed to handle some types of code modifications (as demonstrated in Section 5.1).

Appendix B Analysis for the discovered VCCs

In our experiments, MOVERY discovered 415 VCCs for ten target software. We analyzed the vulnerability types (*i.e.*, Common Weakness Enumeration, shortly CWE) and severity (*i.e.*, Common Vulnerability Scoring System, shortly CVSS) for the discovered VCCs. Figure 6 depicts the analysis results.

First, we confirmed that the 415 discovered VCCs belonged to 22 CWE groups; the distribution for the top five CWEs is shown in Figure 6a. The most frequently appeared type is "Out-of-bounds Read and Write" (53%); this vulnerability can lead to remote code execution and therefore requires extra attention. In addition, we confirmed that many vulnerabilities related to memory buffer (11%) and input validation (7%) appeared; the top four CWEs groups belong to the most dangerous vulnerability types in 2021 [5].

Next, we confirmed that most (84%) of the discovered VCCs had a medium severity, as shown in Figure 6b. Note that 15% of the propagated vulnerabilities had a high severity, which could pose a more critical threat to the entire software; it is noteworthy that the case study presented in Section 5.6 is a high-severity vulnerability. Especially high-risk vulnerabilities need to be discovered and patched more quickly, suggesting that there is in dire need of a precise VCC discovery tool, such as MOVERY.