VMud: Detecting Recurring Vulnerabilities with Multiple Fixing Functions via Function Selection and Semantic Equivalent Statement Matching

Kaifeng Huang[*]
Tongji University
Shanghai, China

Chenhao Lu[†]
Fudan University
Shanghai, China

Yiheng Cao[†]
Fudan University
Shanghai, China

Bihuan Chen[†‡]
Fudan University
Shanghai, China

Xin Peng[†]
Fudan University
Shanghai, China

Abstract

The widespread use of open-source software (OSS) has led to extensive code reuse, making vulnerabilities in OSS significantly pervasive. The vulnerabilities due to code reuse in OSS are commonly known as vulnerable code clones (VCCs) or recurring vulnerabilities. Existing approaches primarily employ clone-based techniques to detect recurring vulnerabilities by matching vulnerable functions in software projects. These techniques do not incorporate specially designed mechanisms for vulnerabilities with multiple fixing functions (VM). Typically, they generate a signature for each fixing function and report VM using a matching-one-in-all approach. However, the variation in vulnerability context across diverse fixing functions results in varying accuracy levels in detecting VM, potentially limiting the effectiveness of existing methods.

In this paper, we introduce VMud, a novel approach for detecting Vulnerabilities with Multiple Fixing Functions. VMud identifies vulnerable function clones (VCCs) through function matching similar to existing methods. However, VMud takes a different approach by only selecting the critical functions from VM for signature generation, which are a subset of the fixing functions. This step ensures that VMud focuses on fixing functions that offer sufficient knowledge about the VM. To cope with the potential decrease in recall due to excluding the remaining fixing functions, VMud employs semantic equivalent statement matching using these critical functions. It aims to uncover more VM by creating two signatures of each critical function and matching precisely by contextual semantic equivalent statement mapping on the two signatures. Our evaluation has demonstrated that VMud surpasses state-of-the-art vulnerability detection approaches by 30.30% in terms of F1-Score. Furthermore, VMud has successfully detected 275 new VM from 84 projects, with 42 confirmed cases and 5 assigned CVE identifiers.

[*]K. Huang is with the School of Software Engineering, Tongji University, China.
[†]C. Lu, Y. Cao, B. Chen, X. Peng are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China.
[‡]B. Chen is the corresponding author.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690372

CCS Concepts

• Security and privacy → Software security engineering; **Vulnerability scanners**; **• Software and its engineering → Software libraries and repositories**.

Keywords

software vulnerability detection, recurring vulnerabilities, open source repositories

ACM Reference Format:
Kaifeng Huang, Chenhao Lu, Yiheng Cao, Bihuan Chen, and Xin Peng. 2024. VMud: Detecting Recurring Vulnerabilities with Multiple Fixing Functions via Function Selection and Semantic Equivalent Statement Matching. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690372

1 Introduction

In recent years, the open-source software (OSS) ecosystem has witnessed substantial growth [3, 14, 20], providing developers with a vast array of options for using free software. It has become a common practice for developers to adopt third-party OSS components to accelerate development and reduce costs [26]. Consequently, the vulnerability landscape has expanded beyond traditional proprietary software to encompass open-source software (OSS). The increasing vulnerabilities in open-source software have raised significant concerns, such as inaccurate vulnerability reports [6, 58], silent vulnerability fixes [64], and software supply chain attacks [7]. Attackers can exploit these vulnerabilities, affecting not only the open-source software itself but also its downstream users, thus amplifying potential financial gains and cyber chaos [1].

Notably, downstream software shares or reuse logic from upstream OSS components. As these OSS components evolve and downstream software undergoes customization, the code initially derived from OSS components diverges from the original one at both ends. Consequently, it leads to challenges in mitigating *Vulnerable Code Clones* (VCCs) [40]. When facing incompatible changes or customized OSS components, mitigating VCCs by replacing newer OSS components becomes infeasible or cumbersome, especially in languages like C/C++ that lacks a dominant package manager.

In this context, developers often resort to manually finding and backporting security patches [60] into downstream software. However, it is time-consuming and resource-intensive. Notably, 80% of attacks leveraged the vulnerabilities reported more than three years ago [2], indicating that the issue of VCCs has not been fully resolved.

Typically, developers employ vulnerable code clone (VCC) discovery techniques [25, 53, 54, 59] to identify reused components (such as files, functions, or fragments) in a target project program. They can also use learning-based methods for classifying vulnerable functions [5, 8, 41, 47, 65]. However, the effectiveness of vulnerability detection is often hindered in cases of *Vulnerabilities with Multiple Fixing Functions* (VM), where fixing patches span across multiple functions or files. Unfortunately, VM are inevitably prevalent in real world due to complex inter-procedural program logic, posing challenges for accurate VM detection.

Limitations of Existing Approaches. Existing research often addresses the detection of *Vulnerabilities with Multiple Fixing Functions* (VM) by dividing the task into matching vulnerability characteristics within individual functions (*i.e.,* matching-one-in-all approach). Clone-based techniques [25, 53, 54, 59] identify vulnerable code clones (VCCs) within functions, while learning-based approaches represent functions in abstract spaces and classify them as vulnerable or not [5, 8, 41, 47, 65]. However, these approaches assume equal importance for all fixing functions in a vulnerability, potentially leading to over-representation resulting in false alarms. Additionally, while software composition analysis (SCA) techniques [21, 53, 56, 57, 61] can detect VM, they are primarily designed for identifying reused OSS components, which may overlook specific vulnerability or fixing characteristics, resulting in false alarms.

Challenge. Naturally thinking, assigning all fixing equal importance is straightforward and may not accurately represent the actual characteristics of vulnerabilities. Some fixing functions may carry adequate information to represent a vulnerability, while others may not. However, it is challenging to measure and weigh the unique fingerprints of each fixing function to a vulnerability. The primary challenge lies in how to capture and distinguish the uniqueness of fixing functions and identify the critical ones for a vulnerability.

Our Approach. To tackle this challenge, we introduce VMud, a novel approach for Vulnerabilities-with-Multiple-Fixing-Function-Detection. Unlike traditional methods that treat all fixing functions equally, VMud prioritizes the fixing functions and selects the most critical functions for vulnerability representation (see Section 3.3.1). It involves matching the signatures [25, 59] of these critical functions. To cope with the potential decrease in recall due to excluding the remaining fixing functions, we employ semantic equivalent statement matching, which consists of program rephrasing [46] (see Section 3.3.2) and contextual semantic equivalent statement mapping (see Section 3.4.3). The program rephrasing aims to uncover more VM by creating another signature for the rephrased form of the critical functions, while contextual semantic equivalent statement mapping aims to match the composing statements precisely based on their contextual semantic equivalence.

The insight of the contextual semantic equivalent statement mapping stems from our observation that some syntactically identical statements serve different contextual purposes. Existing approaches add new context similarity constraints using syntactically equivalent statements, which does not correct the error from the essential. We claim that "statement equivalence" should be redefined not only from syntactic similarity, but also from their contextual similarity. To our best knowledge, VMud is the first to discriminate critical functions from fixing functions and consider both program rephrasing and contextual semantic equivalent statements to identify vulnerabilities with high quantity and quality.

Evaluation. We implemented VMud and generated vulnerability signatures from 810 CVEs. We conducted an evaluation using a dataset comprised of the top 1,000 C/C++ real-world projects selected from GitHub. Comparing VMud with state-of-the-art techniques, VMud achieves a f1-score of 0.84, outperforming the best state-of-the-art at 30.30%. VMud discovered 275 VM across 84 projects, with 42 of them confirmed by developers and 5 of them assigned with corresponding new CVE identifiers. We observed the robustness of VMud regarding the ixing function number in the VM. We conducted an ablation study to observe the contribution of key components in VMud. We also measured the threshold sensitivity and performance of VMud. The results demonstrate VMud's effectiveness and efficiency in identifying VM within real-world projects.

Contribution. We summarize our contributions as follows:

- We propose VMud, a novel approach to detect Vulnerabilities with Multiple Fixing Functions (VM) in real-world projects.
- We evaluate VMud on our ground truth collected from real-world C/C++ projects and demonstrate its effectiveness compared with the state-of-the-art approaches.
- VMud has discovered 275 vulnerabilities in 84 C/C++ projects. 42 are confirmed by developers, 5 has been assigned with CVE identifiers, demonstrating its effectiveness in real-world projects.

2 Motivation

We discuss the problem of Vulnerabilities with Multiple Fixing Functions (VM) and the motivation for our design of VMud.

2.1 Problem

Vulnerabilities with Multiple Fixing Functions (VM). We concentrate our goal on detecting recurring vulnerabilities with multiple fixing functions (VM) in the open source software (OSS). Specifically, developers need to apply fixing changes in vulnerable code to fix vulnerabilities and release new safe versions. The fixing action for vulnerabilities follows same routines as other actions in evolutionary software, such as general bug fixing, feature addition, and code refactoring. They usually record the corresponding code changes by Version Control Tools (VCS). As software becomes more complex, developers usually decouple program logic [45] to facilitate its flexibility. Thus, the code changes are complex, and distributed across multiple locations. e.g., classes, functions, etc. They may have changed patterns [62], tangled changes [19] (*i.e.,* unrelated or loosely related code changes), or systematic changes [24]. Taking Big-Vul [11], a widely used vulnerability detection dataset [44, 48, 63] for example, it comprises of 3,754 code vulnerabilities and 11,823 vulnerable functions collected from CVE database and CVE-related source code repositories. Averagely, fixing a vulnerability involves three fixing functions, indicating the VM is prevalent.

Table 1: Fixing Functions of `CVE-2021-32134` and FPs in the Target Project Matched by Vuddy, V1Scan and VMud

Fixing File	Fixing Functions	Vuddy	V1Scan	VMud
box_code_base.c	*mp4s_box_new*	1	1	0
box_code_base.c	*encs_box_new*	1	1	0
sample_desc.c	*gf_isom_sample_entry_init*	0	0	0
media.c	*Media_GetESD*	0	0	0

Fixing vulnerabilities exhibit various patterns such as change patterns, tangled changes, or systematic changes. Consequently, different fixing functions contribute unequally in terms of importance and characteristics to identify the vulnerability. However, existing research often treats each function equally [25, 53, 54, 59], triggering a vulnerability alarm if any of the functions matches. Applying uniform importance to all fixing functions would result in diverse effectiveness per function. *i.e.*, some functions may carry adequate information to represent a vulnerability, resulting in high accuracy, while others may not. Therefore, it is not reasonable to assign equal importance to all fixing functions. Li et al. [30] investigated inter-procedural vulnerabilities, where the statements to be patched and the statements triggering the vulnerability belong to different functions. Their approaches are based on a carefully designed list of vulnerability-trigger statements regarding popular CWE types, hindering its effectiveness in a wide range of vulnerability types.

Challenges. It is challenging to recognize the functions that contribute higher accuracy against the functions that contribute lower accuracy about one vulnerability. The key problem lies in measuring and weighing the unique fingerprints of each fixing function so that we can prioritize the functions. We refer to the recognizing task as *selecting critical functions* and filtering out less important functions. However, discarding the less important functions would result in a drop in recall as it would miss some vulnerabilities discovered by these functions. Therefore, we enhance the signature generation and matching by semantic equivalent statement matching to cope with the recall drop and improve our detection accuracy.

2.2 Motivation

(a) The Motivating Example of `CVE-2021-32134`. We illustrate our motivation using an CVE [18, 37] from the *gpac/gpac* project. It is a VM that involving four fixing functions across three files (see Table 1). We present the corresponding code and patches in Figure 1. Notably, although there are statement deletions in *mp4s_box_new* and *encs_box_new* (Figure 1(a)), these deletions do not alter the overall semantics as the deleted statements are reintroduced in *gf_isom_sample_entry_init* (Figure 1(b)). We employed state-of-the-art approaches, including Vuddy [25], MVP [59], Movery [54], and V1Scan [53] to detect VCCs regarding this vulnerability compared with our approach. Note that we also consider VCCs within the same but different versions of the project. We present two false positives generated by Vuddy and V1Scan from gpac/gpac[17] in Table. 1. We exclude MVP and Movery because they yield zero false positives. Comparatively, our approach VMud avoids false positives by exclusively considering *gf_isom_sample_en try_init* and *Media_GetESD* as critical functions using Critical Function Selection in Signature Generation (see Section 3.3.3).

```
GF_Box *mp4s_box_new(){
    ISOM_DECL_BOX_ALLOC(GF_MPEGSampleEntryBox, GF_ISOM_BOX_TYPE_MP4S);
    gf_isom_sample_entry_init((GF_SampleEntryBox*)tmp);
-   tmp->internal_type = GF_ISOM_SAMPLE_ENTRY_MP4S;
    return (GF_Box *)tmp;
}
GF_Box *encs_box_new(){
    ISOM_DECL_BOX_ALLOC(GF_MPEGSampleEntryBox, GF_ISOM_BOX_TYPE_ENCS);
    gf_isom_sample_entry_init((GF_SampleEntryBox*)tmp);
-   tmp->internal_type = GF_ISOM_SAMPLE_ENTRY_MP4S;
    return (GF_Box *)tmp;
}
```

(a) The *mp4s_box_new* and *encs_box_new* function in *box_code_base.c*

```
void gf_isom_sample_entry_init(GF_SampleEntryBox *ent){
+    ent->internal_type = GF_ISOM_SAMPLE_ENTRY_MP4S;
}
```

(b) The *gf_isom_sample_entry_init* function *sample_desc.c*

```
GF_Err Media_GetESD(GF_MediaBox *mdia, u32 sampleDescIndex, GF_ESD
**out_esd, Bool true_desc_only) {...
    switch (type) {
        case GF_ISOM_BOX_TYPE_MP4V:
+           if (entry->internal_type != GF_ISOM_SAMPLE_ENTRY_VIDEO)
+               return GF_ISOM_INVALID_MEDIA;
            ESDa = ((GF_MPEGVisualSampleEntryBox*)entry)->esd;
            if (ESDa) esd = (GF_ESD *) ESDa->desc; ...
    }...
    if (*out_esd == NULL) return gf_odf_desc_copy((GF_Descriptor
*)esd, (GF_Descriptor **)out_esd); ...
```

(c) The *Media_GetESD* function in *media.c*

Figure 1: A Motivating Example of Patches from *gpac/gpac* for Fixing `CVE-2021-32134`

Table 2: Number of True Positives (TP), False Positives (FP), False Negatives (FN) Changed Using Critical Function in MVP and V1Scan. (Precision$_\Delta$ Denotes the Precision in Changed TPs and FPs while Precision$_o$ Denotes the Original Precision without using Critical Functions)

Tool	Numbers Changed (#)			Precision$_\Delta$	Precision$_o$
	TP	FP	FN		
MVP	-39	-119	+39	0.24	0.81
V1Scan	-39	-18	+39	0.68	0.71

(b) Using Critical Functions in Existing Approaches. We utilize the critical functions generated by our apporach as vulnerability signatures and apply them to MVP [59] and V1Scan [53]. We did not apply Vuddy [25] and Movery [54] due to that both tools did not provide the matched vulnerable functions from the original VM in their generated output. The outcomes are detailed in Table. 2. The origial false positives generated by existing approaches are presented in Table 4. Comparatively, it can reduce 73.01% (119/163) and 38.30% (18/47) false positives in MVP and V1Scan, respectively when employing the critical function signatures. Although true positives decrease by 39 for both tools, the precision in Changed TPs and FPs are lower than the original precision for each tool. This underscores the effectiveness of our critical function selection in reducing false positives and enhancing precision, despite the trade-off resulting in more true positives converting to false negatives. To address the trade-off, we employ the program semantic-centered statement matching to improve recall in our approach. *i.e., Program Rephrasing and Contextual Equivalent Statements Map* (see Section 3).

3 Design of VMud

We first provide VMud's overview and clarify the terminology and definition. Then, we present the technical details of VMud.

3.1 Overview

VMUD differs from existing works in two aspects. On the one hand, existing works don't distinguish the different importance of functions in VM, hindering the effectiveness of detecting VM. We select the critical functions and particularly enhance the signatures for these functions. On the other hand, existing works have imperfect matchings of statements. The reason is that most approaches only consider the syntactical equality of the statements (*e.g.,* using =, == or `strcmp`). However, some syntactically identical statements serve different contextual purposes. Although existing approaches add new context similarity constraints using syntactically equivalent statements, it does not perfectly resolve the semantic equality of statements (*i.e.,* the equality of statements' surrounding contexts).

Differently, VMUD first prioritizes and selects critical functions among multiple fixing functions during Signature Generation (see Section 3.3). Then, VMUD designs a new semantic-centered statement matching technique (see Section 3.4) to redefine the equality of the statements from the target function and vulnerable functions. It includes generating the semantic enhanced signature (i.e., *See*) during signature generation (see Section 3.3.3), and matching the signature via contextual equivalent statements map during signature matchting (see Section 3.4.3). Additionally, VMUD leverages program rephrasing [46] (see Section 3.3) to create semantically equivalent transformations of the expression, thus creating new rephrased signatures for critical functions. It allows VMUD to filter out false positives introduced by less important functions and imperfect matching, and find true positives using rephrased signature.

Figure 2 illustrates the workflow of VMUD, comprising two main stages: Signature Generation (see Section 3.3) and VM Detection (see Section 3.4). In the Signature Generation stage, VMUD takes a VM patch commit as input and extracts the fixing functions. Subsequently, it identifies critical functions among these fixing functions. Then, the critical functions either undergo the Abstraction and Normalization directly or the Program Rephrasing [46] beforehand. Next, VMUD extracts the original and rephrased signatures of these critical functions. In the VM Detection stage, VMUD reduces the detection space by preprocessing the target project, similar to existing works[53, 54]. Subsequently, the preprocessed software progresses through the Signature Matching (I) and Program Rephrasing. If necessary, the rephrased program enters the Signature Matching (II). Finally, VMUD outputs the identified vulnerable code clones.

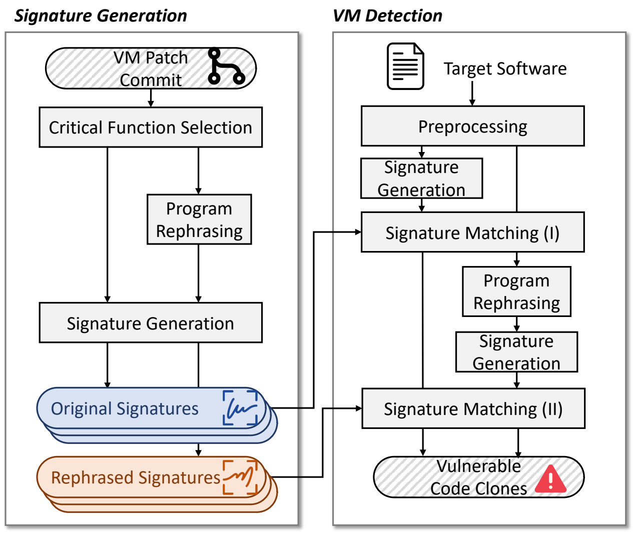


Figure 2: The Overview of VMUD

3.2 Terminology and Definition

- **Vulnerable Code Clones (VCCs):** The vulnerable code clones [25] are reused OSS components in the target project program that contain vulnerabilities. Typically, the granularity of VCCs can vary at the component, file, function, or fragment level. We focus on detecting the general type of vulnerabilities which has multiple fixing functions. Therefore, the granularity of VCCs in our context can be a component, multiple files, multiple functions, or multiple fragments in multiple functions.
- **Critical Function:** In VM, the different fixing functions contribute diverse contexts for the vulnerability. The critical functions should carry the most significant fixing information to represent the vulnerability. We denote the critical functions as $\mathcal{F}$.
- **Signature.** The signatures for functions $\mathcal{F}$ are denoted as $\mathcal{X}$. Each entry X in the $\mathcal{X}$ corresponds to the signature for a function $f \in \mathcal{F}$. X_{org} denotes the original signature and X_{rep} denotes the rephrased signature for f. We use X^P to denote the vulnerability signature and X^V to denote the patched signature. For a target project function, we use X^T to denote the signature. X is a 3-tuple $\langle Syn, Sem, SeE \rangle$ where Syn consists of the syntactic statements of the vulnerable (resp. patched) function, Sem represents the semantic relationships of the vulnerable (patch) function, See represents the Semantic Enhanced (See) signature of the vulnerable (resp. patched) function.
- **Rephrased Program.** Program Rephrasing is a type of Program Transformation [46]. It involves changing one program into another, a practice widely used in software engineering for tasks like compilation, optimization, refactoring, and program synthesis. Program transformation can be split into program translation, where source and target languages differ, and program rephrasing, where both languages are the same. In our case, we employ program rephrasing to unify the "idioms" in vulnerable code and the target project, thus improving syntactic consistency.

3.3 Signature Generation

We gather the patches, consisting of added and deleted lines, denoted as C_{add} and C_{del}, correspondingly, and the corresponding fixing commit ID of the VM from a given CVE. We generate the original signatures X_{org} and the rephrased signatures X_{rep} for the VM.

VMUD starts by giving an input as a CVE ID and traces patches to find the vulnerability-fixing commit in two ways. First, it searches the CVE ID in open-source software repositories' histories with vulnerability keywords and filters out irrelevant commits. The irrelevant commits include reverted commits, merging commits, etc. Second, it leverages the National Vulnerability Database (NVD) [38] reference pages for silent fixes (*i.e.,* commits without explicit CVE ID mentions). This database provides valuable metadata and referencing pages related to CVE fixes. Combining the two approaches, we gather the patches, consisting of added lines and deleted lines, which are denoted as C_{add} and C_{del}, respectively, as well as their corresponding fixing commit ID for the given CVE ID.

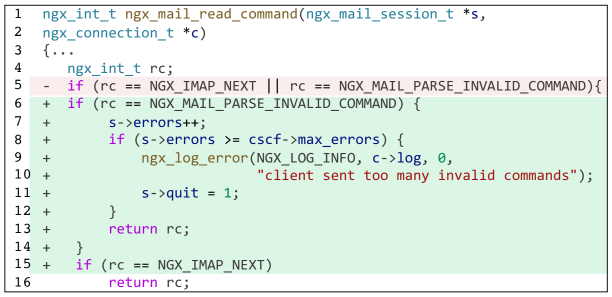
3.3.1 Critical Function Selection. VMUD obtains the files before and after change using the gathered commit ID. Then, it transforms the files into the Abstract Syntax Tree (AST). Using the added lines (C_{add}) and the deleted lines (C_{del}) for each file, VMUD locates the modified, added, and deleted functions, denoted as $\mathcal{F}^*$. To identify the critical functions, VMUD first generates a local call graph for the target project which is directly related to the vulnerabilities. Then, VMUD utilizes the PageRank algorithm [51], a widely recognized algorithm for assessing the significance of web pages in search engine rankings. This choice stems from the analogy between the structural characteristics of a call graph and the linkages among web pages, which are both represented as directed graphs. In the PageRank, a page has high rank if the sum of the ranks of its backlinks is high. Similarly, functions with higher caller numbers are likely to be central to vulnerability because they tend to receive more vulnerable paths, which are more representative of the VMs. Therefore, we identify critical functions using their connectivity over the surrounding call graph. The PageRank score $R(u)$ for a function is calculated as:

$$R(u)=c\sum_{v\in B_u}\frac{R(v)}{N_v}+cE(u) \quad (1)$$

where u denotes the function, B_u is the set of functions that calls to u, N_u is the number of outgoing links from u, c is a normalization factor. $E(u)$ is some vector over the functions as a source of rank. We use the implementation from networkx [36] where the calculation uses power iteration with a SciPy sparse matrix representation.

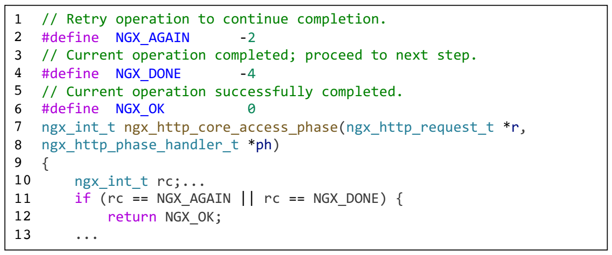
Instead of processing an entire call graph, the algorithm operates on a local graph centered around the changed functions. Li et al. [30] suggest that vulnerabilities often involve multiple functions spanning an average depth of 2.8 invocation levels. Accordingly, VMUD focuses on callers and callees of $\mathcal{F}^*$ within a three-level invocation depth. It captures the local graph, comprising invocation relationships up to three levels of invocation depth, both upstream and downstream. The algorithm outputs a list of significance scores for each function in the local graph. We use a threshold variable th_{pr} to filter out the functions from $\mathcal{F}$ with low significance scores and keep the rest as critical functions, denoted as $\mathcal{F}'$.

3.3.2 Program Rephrasing. Previous studies [25, 53, 54, 59] do not consider the VCCs in different syntactic forms yet equivalent in semantic meanings, leading to potential false negatives. To address this, we employ program rephrasing to enrich the syntactic diversity of vulnerability signatures. Program rephrasing, a form of program transformation [46], reshapes code while preserving its semantic meaning. This approach allows us to detect more VCCs manifesting in diverse syntactic structures. We primarily focus on the equivalent rephrasing in the C/C++ language. Specifically, we refer to the C11 language standard documentation[39], which is a comprehensive manual illustrating the syntaxes and grammar of C/C++ in C11. We search the documentation containing the "equivalent" keyword and find 160 locations mentioning the keyword. The three authors annotated and resolved disagreements. 27 were removed for not containing equivalent rules, e.g., explaining language grammar, comments equivalence. 133 instances were categorized into 15 types.



```
1   ngx_int_t ngx_mail_read_command(ngx_mail_session_t *s,
2   ngx_connection_t *c)
3   {...
4      ngx_int_t rc;
5   -  if (rc == NGX_IMAP_NEXT || rc == NGX_MAIL_PARSE_INVALID_COMMAND){
6   +  if (rc == NGX_MAIL_PARSE_INVALID_COMMAND) {
7   +        s->errors++;
8   +        if (s->errors >= cscf->max_errors) {
9   +            ngx_log_error(NGX_LOG_INFO, c->log, 0,
10  +                          "client sent too many invalid commands");
11  +            s->quit = 1;
12  +        }
13  +        return rc;
14  +    }
15  +    if (rc == NGX_IMAP_NEXT)
16           return rc;
```

(a) Patch Code for CVE-2021-3618



```
// Retry operation to continue completion.
#define  NGX_AGAIN      -2
// Current operation completed; proceed to next step.
#define  NGX_DONE       -4
// Current operation successfully completed.
#define  NGX_OK          0
ngx_int_t ngx_http_core_access_phase(ngx_http_request_t *r,
ngx_http_phase_handler_t *ph)
{
    ngx_int_t rc;...
    if (rc == NGX_AGAIN || rc == NGX_DONE) {
        return NGX_OK;
    ...
```

(b) A False Alarm in Target Project

Figure 3: Examples of Program Rephrasing

8 types (105 instances) are excluded as specific to library APIs, e.g., setbuf function is equivalent to setvbuf when using _IOFBF. We exclude goto equivalence (1 instance), primary expressions (6 instances), and pointers (5 instances) because of discouraged usage and their equivalence after abstraction. As a result, we identified four equivalence types (16 instances) summarized as follows:

- **Macros.** Existing works [25, 53, 54, 59] regard macro as either function calls, parameters or local variables. Given a statement s containing a macro, we rephrase it by expanding the macro at the statement. For example, the statement containing the macro `MAX(a,b)` would be rephrased to $a > b?a : b$.
- **Indirection Expression:** $*\&a$ and $\& * a$ can be rephrased to a.
- **Logical NOT Expression:** $!E$ can be rephrased to $0 == E$ and $!!E$ can be rephrased to E.
- **Comparison Expression:** $a >= b$ can be rephrased to $a > b || a == b$.

We transform the source code of the critical functions $\mathcal{F}'$ into the Abstract Syntax Tree (AST) representation. Then, based on the above rephrasing rules, we transform it into the semantically equivalent form according to the rules on the AST and convert it back to source code. The rephrased functions are denoted as $\mathcal{F}'_{rep}$.

Example 3.1. Figure 3 illustrates an instance of program rephrasing. In Figure 3(a), we depict a patch addressing the vulnerability in `ngx_mail_read_command` identified in `CVE-2021-3618`. Figure 3(a) presented a false alarm reported by VUDDY [25], MVP [59], MOVERY [54] and V1SCAN [53]. The vulnerable code snippet comprises a variable declaration statement (Line 4), an `if` statement (Line 5), and a return statement (Line 16). Notably, the `if` statement incorporates two comparison expressions, a frequently encountered construct, each involving a macro. Existing tools [25, 53, 54, 59] abstract these macros into local variables, resulting in false alarms. In contrast, our tool (VMUD) replaces the macros with their corresponding expressions. For instance, the macro `NGC_DONE` gets transformed into the number literal `-4`.

3.3.3 *Signature Generation.*

VMud generate the original signatures $\mathcal{X}_{org}$ for functions $\mathcal{F}'$ and rephrased signatures $\mathcal{X}_{rep}$ for $\mathcal{F}'_{rep}$. It consists of Program Slicing, Abstractions and Normalization, and Semantic Enhanced Signature Generation.

Program Slicing. VMud generates the Program Dependence Graph (PDG) for the critical functions $\mathcal{F}$. It utilizes the deleted lines C_{del} and added lines C_{add} and conducts forward and backward slicing on the PDG similar to MVP[59]. Specifically, VMud first perform backward slicing for C_{del} and C_{add} in the PDG. i.e., for any of the statement s in C_{del} and C_{add}, it scans backforwardly and collects any preceding statements that have data or control dependencies over s. Second, VMud conducts forward slicing C_{del} and C_{add} in the PDG. For an assignment statement, it collects the following statements of s on the PDG. For a conditional statement or function call statement, it searches backwardly for initial variable declarations of the variables used in s, and then collects the following statements on the PDG. For a return statement, it directly outputs s as it reaches the end of the PDG.

Abstraction and Normalization. The reused code may undergo parameter renaming, variable renaming, or code style changes. We abstract and normalize the original functions f_{org} as well as rephrased functions f_{rep}. For each statement in *Syn*, *Sem*, and *See*, we apply abstraction and normalization. We abstract function parameters, type declaration, local variables, function calls, and string literals in the code into "FPARAM", "DTYPE", "LVAR", "FUNCCALL", and "STRING", respectively[25]. In string literals, we retain format specifiers which have semantic meaning for specific placeholders. For example, we abstract "protos=%s" to "%s". We also remove comments, braces, spaces, and tab characters [59].

Semantic Enhanced Signature Generation. *Syn* is a list comprising of the code statements s in the function, denoted as $\langle s \rangle$. Each statement is a 2-tuple $\langle l, t \rangle$, where l is the line number and t is the statement's text. *Sem* includes the semantic relationship of the statements, denoted as a 3-tuple $\langle s_1, s_2, type \rangle$. The *type* can either be *control* or *data* dependency. Existing work [59] compares the similarity based on *Syn* and *Sem*. Differently, VMud creates Semantic Enhanced (SeE) signature *See* to obtain the Contextual Equivalent Statements Map (see Section 3.4.3).

To this end, VMud extract the semantic enhanced sets *See* for vulnerability signatures V and patched signatures P. Specifically, for each statement $s \in Syn$, we iterate the entries in *Sem*. For each entry $\langle s_1, s_2, type \rangle$, if s either equals s_1 or s_2, we put $\langle s_1, s_2, type \rangle$ into *See*. X_{src} can either be X_{org} or X_{rep}.

$$\begin{aligned} See = \{ \langle l, \langle s_i, s_j, type \rangle \rangle \mid \forall s.l \in X_{src}.Syn, \\ (s_i, s_j, type) \in X_{src}.Sem, if\ s.l \in \{s_i.l, s_j.l\} \} \end{aligned} \tag{2}$$

To accelerate the comparison, we leverage the MD5 hash to compute the hash value for each statement s after abstraction and normalization. Finally, we obtain $\langle Syn, Sem, See \rangle$ for vulnerable function V and patched functions P regarding the original signature X_{org} and the rephrased signature X_{rep}.

3.4 VM Detection

VMud detects the VM in the target project using X_{org} and X_{rep}.

3.4.1 *Preprocessing.*

We conduct preprocessing for efficient detection. Similar to search space reduction in Movery[54] and code classification of reused and vulnerable code in V1Scan[53], the purpose of the preprocessing is to reduce the search space, focus on relevant clones, and filter out the functions that are less likely to be code clones. We leverage SAGA[27], an effective clone detection tool. We conduct snippet-level clone detection with the minimum token of snippet set as 20 token by default to ensure high recall rates. Specifically, the length of the snippet is determined by the number of tokens. SAGA finds a cloning snippet if there is one snippet equal in functions from $\mathcal{F}_{src}$ and $\mathcal{F}_{tgt}$. Finally, we obtain the functions that may contain the VCCs, denoted as $\mathcal{F}_{tgt}$.

3.4.2 *Signature Generation.*

Given input as candidate functions from target project $\mathcal{F}_{tgt}$, we generate a signature for each function and its semantically equivalent functions, denoted as X_{tgt_org} and X_{tgt_rep}. For each X_{tgt_org} and X_{tgt_rep}, we generate a syntactic signature *Syn*, semantic signature *Sem*, and semantic enhanced signature *See* following same signature generation in Section 3.3.3.

3.4.3 *Signature Matching.*

The signature matching process comprises two phases (see Figure 2). In **Phase I**, the signatures X_{tgt_org} of target project functions are matched against the original vulnerability signatures X_{org}. In **Phase II**, the Rephrased target signatures X_{tgt_rep} are compared with rephrased vulnerability signatures X_{rep}. In each phase, VMud evaluates syntactic similarity ($X_{tgt}.Syn$ vs. $X^V_{src}.Syn$ and $X^P_{src}.Syn$) as well as semantic similarity ($X_{tgt}.Sem$ vs. $X^V_{src}.Sem$ and $X^P_{src}.Sem$), where X_{src} can be either X_{org} or X_{rep} and X_{tgt} can be either X_{tgt_org} or X_{tgt_rep}. We refine the similarity assessment using $X_{tgt}.See$, $X^V_{src}.See$, and $X^P_{src}.See$ (see Eq. 7). We incorporate the Contextual Equivalent Statement Map (CESM) to ensure that only contextually and semantically equivalent statements contribute to the similarity metric.

We run Phase I and Phase II under four conditions. The rationale is rooted in the principle that a signature of a VCC should be similar to the vulnerable function signature while different from the patched function signature. False negatives occur when the target function signature is different from the vulnerable function's signature. Therefore, Phase II is executed under the condition that the target function signature X_{tgt} aligns with the vulnerable function signature X^V_{org} while differing from the patched function signature X^P_{org}. We describe the conditions using the following rules:

- **Run Phase I**: If `sim`(X_{tgt_org}, X^V_{org}) is `true` and `sim`(X_{tgt_org}, X^P_{org}) is `false`, report a positive vulnerability alarm for the target function and skip **Phase II**.
- **Run Phase I**: If `sim`(X_{tgt_org}, X^V_{org}) is `true` and `sim`(X_{tgt_org}, X^P_{org}) is `true`, report a negative result for the target function, skip **Phase II**.
- **Run Phase I**: If `sim`(X_{tgt_org}, X^V_{org}) is `false`, then **Run Phase II**:
  - If `sim`(X_{tgt_rep}, X^V_{rep}) is `true` and `sim`(X_{tgt_rep}, X^P_{rep}) is `false`, report a positive vulnerability alarm for the target function.
  - If `sim`(X_{tgt_rep}, X^V_{rep}) is `true` and `sim`(X_{tgt_rep}, X^P_{rep}) is `true`, report a negative result for the target function.

The similarity function `sim` is defined in Eq. 3. Literally, it means that the signature of the target function should be syntactically (*i.e.,* `syn()`) and semantically (*i.e.,* `sem()`) similar to the source function (*i.e.,* V or P). Meanwhile, the deleted statements at the source function should exist in the target function (*i.e.,* `exist()`).

The similarity function `sim` is formalized in Eq. 3. It denotes that the target function's signature X_{tgt} should have both syntactic and semantic similarity to the vulnerability function X_{src}. Additionally, it also requires that any statements deleted in the source function should be present in the target function.

$$\begin{aligned}\texttt{sim}(X_{tgt}, X_{src}) = \texttt{syn}(X_{tgt}, X_{src}) \\ \& \ \texttt{sem}(X_{tgt},\ X_{src}) \\ \& \ \texttt{exist}(X_{tgt})\end{aligned} \tag{3}$$

The `syn()` function measures the syntactic similarity between the syntax signatures of $X_{tgt}.Syn$ and $X_{src}.Syn$. The `sem()` function evaluates their semantic similarity between $X_{tgt}.Sem$ and $X_{src}.Sem$. The `exist()` function verifies the presence of deleted statements S_{del} from X_{src} in the target project X_{tgt}. They are defined as follows:

$$\texttt{syn}(X_{tgt}, X_{src}) = 1 \text{ if } \frac{|\ \texttt{cesm.keys}\ |}{X_{src}.Syn} > th_{syn}, \text{ and 0 otherwise} \tag{4}$$

$$\texttt{sem}(X_{tgt}, X_{src}) = 1 \text{ if } \frac{Sem_{pft} \cap Sem_{pfv}}{Sem_{pfv}} > th_{sem} \text{ and 0 otherwise} \tag{5}$$

$$\texttt{exist}(X_{tgt}) = 1 \text{ if } \forall s \in S_{del}, s \in X_{tgt}.Syn \text{ and 0 otherwise} \tag{6}$$

The computation of `syn()` involves generating the Contextual Equivalent Statements Map (CESM) through the `CESM()` function, as defined in Eq. 7. This function produces a CESM output denoted as a tuple $\langle s_j, \langle s_i \rangle \rangle$, where s_j represents a statement in X_{src} and s_i denotes a statement in X_{tgt}. It implies that for a given statement s_j in X_{src}, there are several statements s_i in X_{tgt} that are contextually equivalent to s_j. Unlike traditional equality comparisons (*i.e.*, using = or ==), the `CESM()` function determines the equivalence between statements based on their contextual equivalence.

For any given $s_i \in X_{tgt}$ and $s_j \in X_{src}$, VMUD first obtains semantic relationships. *i.e.*, $X_{tgt}.See[s_i.l]$ for s_i and $X_{src}.See[s_j.l]$ for s_j. It then computes the intersection of these sets and divides it by $X_{src}.See[s_j.l]$. If the intersection ratio exceeds the threshold th_{ce}, VMUD considers the two sets as contextually similar, thus concluding that the s_j and s_i are contextually equivalent. `cesm.keys` denotes the keys of the `cesm` output (see `cesm.keys` in Eq. 4).

$$\begin{aligned}\texttt{cesm} = \texttt{CESM}(X_{tgt}, X_{src}) = \{\langle s_j, \langle s_i \rangle \rangle \mid \forall s_i \in X_{tgt}, s_j \in X_{src}, \\ \frac{X_{tgt}.See[s_i.l] \cap X_{src}.See[s_j.l]}{X_{src}.See[s_j.l]} > th_{ce}\}\end{aligned} \tag{7}$$

In computing `sem()`, we utilize partial functions, which are a subset of statements from X_{src} or X_{tgt}. Specifically, with the contextual equivalent statements identified from the target and source functions (see Eq. 7), our goal is to identify the subset of statements within the target function that exhibits the strongest semantic relationship with the source function. To this end, it involves the search problem where for each entry $\langle s_j, \langle s_i \rangle \rangle$, we need to find an optimal statement s_i from $\langle s_i \rangle$ for each s_j, such that the semantic relationship between X_{tgt} and X_{src} is maximized. We use the `argmax()` function denoted in Eq. 8.

$$argmax(\frac{Sem_{pft} \cap Sem_{pfv}}{Sem_{pfv}}) \tag{8}$$

Table 3: Statistics of Our Collected VM

Fixing Functions (#)	VM (#)	Percentage of VM
2	333	41.11%
3	146	18.02%
4	97	11.98%
5	56	6.91%
6	42	5.19%
7	32	3.95%
8	20	2.47%
9	10	1.23%
10	14	1.73%
>10	60	7.41%
Total	810	100.00%

Sem_{pfs} denotes the semantic signature of the partial function of the source function (see Eq. 9). Syn_{pft} and Sem_{pft} denote the syntactic and semantic of the partial function of the target function.

$$\begin{aligned}Sem_{pfs} &= \{\langle s_1, s_2, type \rangle \mid s_1 \ or \ s_2 \ in \ \texttt{cesm.keys}\} \\ Syn_{pft} &= \{s_i \mid \exists s_i \in \texttt{cesm}[s_j], \forall s_j \in \texttt{cesm.keys}\} \\ Sem_{pft} &= \{tup \mid tup.s_1 \ or \ tup.s_2 \in Syn_{pft}, \forall tup \in Sem_{tgt}\}\end{aligned} \tag{9}$$

4 Implementation

We implemented VMUD using 1,900 lines of Python code. To generate the call graph for the project before vulnerability fixing commits, we utilized Doxygen [9]. Due to Doxygen's scalability limit in analyzing large amounts of files, we prioritized neighboring source code files based on their path relativity to the changed files in the vulnerability fixing commit. Then, we limited Doxygen's analysis input to a maximum of 5,000 files. The threshold score of PageRank [51], denoted as th_{pr}, was set at 0.018 based on our sensitivity evaluation (see Section 5.5). We extracted patch statements using the *git show* command and leveraged Joern [42] to generate a comprehensive code property graph, including Abstract Syntax Trees, Control Flow Graphs, and Program Dependence Graphs. We employed the GCC compiler [13] for program rephrasing on macros.

To collect CVEs, we accessed the official API [38] in 2023, containing the data in the recent decade. We obtained corresponding fixing commits on 1,782 CVEs, filtering out 972 whose fixing functions are less than 2. As a result, we obtained 810 VM with 4,551 fixing functions. The statistics of the VM are presented in Table 3.

5 Evaluation

We evaluate VMUD based on the following research questions.

- **RQ1. Effectiveness Evaluation.** How is the effectiveness of VMUD compared to the state-of-the-art tools?
- **RQ2. Robustness Evaluation.** How is the robustness VMUD in detecting VM?
- **RQ3. Ablation Study.** How does each component contribute to VMUD's effectiveness?
- **RQ4. Threshold Sensitivity.** How do the thresholds contribute to the effectiveness of VMUD?
- **RQ5. Performance Evaluation.** How is the efficiency of VMUD?

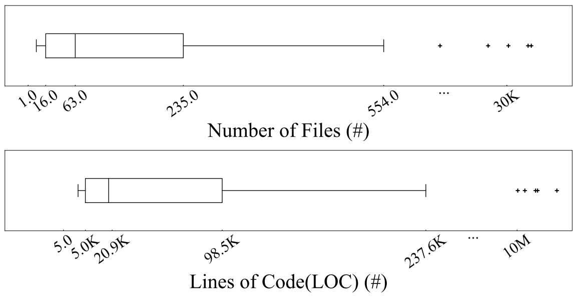


Figure 4: Number of Files and Lines of Code (LOC) on Our Evaluated Dataset

5.1 Setup

Target Project Selection. We established three criteria for project selection. Firstly, projects should be written in C/C++. Secondly, they should be popular in popularity and span diverse domains. Thirdly, they should be actively maintained. Subsequently, we collected 972 projects from GitHub by querying the Top 1000 popular projects ranked by their stars[16]. These projects encompass areas such as databases, operating systems, image processing, reverse development, etc. Figure 4 depicts the number of files and lines of code (LOC) across our selected projects using box plots.

Comparison Tools. We chose four clone-based tools, namely Vuddy [25], MVP [59], Movery [54], and V1Scan [53], recognized as state-of-the-art in VCC detection. Additionally, we included two learning-based tools, SySeVR [32] and DeepDFA [43], to assess their effectiveness compared with VMud. We used the default configurations outlined in the original papers for all selected tools.

Ground Truth Construction. To identify VM comprehensively, we executed all tools on the target projects. Subsequently, we conducted manual validation on all positive results to confirm the presence of VM. This process involved two authors independently classifying each positive result. *i.e.,* whether it is a true positive or a false positive. Any discrepancies were resolved through discussion, as well as a third author consulted if necessary. Two annotators and one mediator have over three years' expertise in security and C/C++ development. Two annotators have experiences in submitting CVEs previously. The validation process took 3 rounds. In the first and second rounds, they had 16 and 5 disagreements, steming from different understanding in vulnerability context and triggering conditions. We use Cohen's Kappa coefficient to measure agreement, and it reached 0.951. This process led to the creation of a ground truth dataset containing 329 confirmed true VM.

Configuration and Metrics. We used a machine equipped with a 2.10GHz Intel Xeon processor and 256GB of RAM. We employed true positives (TP), false positives (FP), false negatives (FN), precision, recall, and F1-Score [52] to measure the tools' effectiveness and accuracy. Specifically, we define one TP when the tool reports a vulnerability warning due to a match with at least one of its fixing functions of a VM. We record one FP when the tool reports false vulnerability warnings based on matching its fixing functions to one VM. We denote one FN when the project contains a VM, but the tool fails to identify any of its matching functions as vulnerable.

RQ Setup. To study **RQ1**, we ran all tools on our ground truth dataset (see Section 4). However, As Movery does not provide open-source code for signature generation, we intersected our collected CVEs with their dataset to ensure a fair evaluation, which resulted in a Diminished VM Dataset of 144 VM signatures. Therefore, Movery is not included in the comparisons shown in Figure 5 and Figure 6. Furthermore, we assessed each clone-based tool under two settings to explore the relationship between fixing functions. A superscript $*$ denotes a tool reporting a VM upon matching any fixing function signature (*i.e.,* the matching-one-function-in-all approach), while a superscript & denotes reporting a VM only when matching all fixing function signatures (*i.e.,* the matching-all-functions approach). Additionally, we compared two learning-based tools using precision and recall w.r.t vulnerable functions. To study **RQ2**, we observed VMud's robustness by evaluating its accuracy across VM with varying numbers of fixing functions. To investigate **RQ3**, an ablation study was conducted to assess VMud's effectiveness in critical function selection (w/o CFS), and semantic equivalence (w/o SE). The semantic equivalence includes program rephrasing (w/o PR), and contextual equivalent statement mapping (w/o CESM). Besides, we also created ablated versions in critical function selection by replacing PageRank with either pre-defined heuristics (*i.e.,* matching all changed functions in one of the invocation paths) and HITS algorithm (*i.e.,* a link analysis algorithm that usually used for rating web pages) [50]. **RQ4** involved evaluating the sensitivity of thresholds in VMud, while **RQ5** focused on its overall performance.

5.2 Effectiveness Evaluation (RQ1)

We evaluate the accuracy and effectiveness of VMud with the state-of-the-arts using our ground truth and analyze the inaccuracies. Additionally, we compare VMud against learning-based approaches.

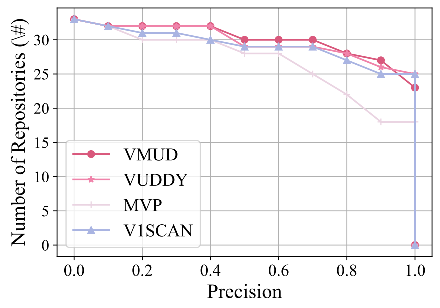
5.2.1 *Accuracy Results.*

Table 4 presents the effectiveness and accuracy results of VMud and the clone-based tools on our ground truth. Overall, VMud achieves a precision, recall, and F1-Score of 0.84. Comparatively, Vuddy* achieves a highest precision of 0.87 but only reaches a recall of 0.51. Movery* reaches a highest recall of 0.75 and an F1-Score of 0.66. Notably, VMud outperforms Movery with an F1-Score increase of 30.30%. Additionally, using the matching-all-functions approach (denoted by &) significantly boosts precision while lowering recall, which is impractical in most scenarios.

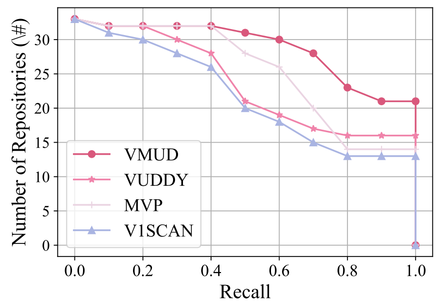
Accuracy Per Project We calculated the accuracies of each tool's results on every project containing VM. Figure 5 displays the distribution of projects based on precision, recall, and F1-Score thresholds. The metrics with NaNs (*i.e.*, either TP+FP=0 or TP+FN =0) are not included. Notably, VMud detects more projects with precision ranging from 0.2 to 0.9 compared to other tools. It reports 2 fewer projects than Vuddy and V1Scan with a precision of 1.0. Additionally, VMud demonstrates significant advantage across the evaluated projects in terms of recall and F1-Score, surpassing other tools in any recall or F1-Score values.

5.2.2 *Effectiveness Results.*

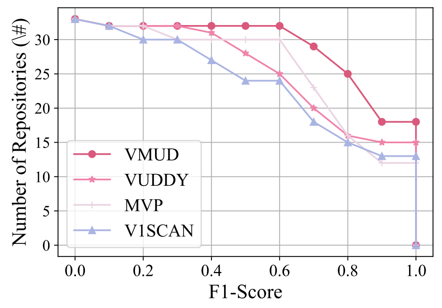
VMud identified 275 true VM across 84 projects where developers confirmed 42 VM from 25 projects, 28 VM from 20 projects have been resolved, and 14 VM from 5 projects await resolution in the next version. In contrast, Vuddy detected 168 true VM across 63 projects, MVP detected 226 true



(a) The Number of Projects w.r.t the Precision and Above

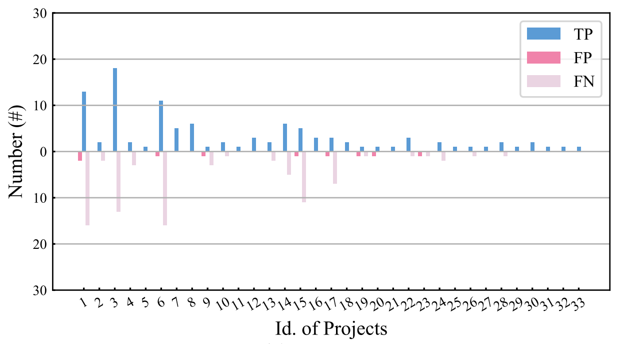


(b) The Number of Projects w.r.t the Recall and Above

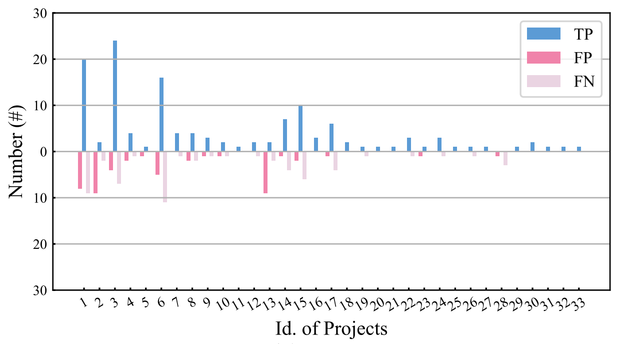


(c) The Number of Projects w.r.t the F1-Score and Above

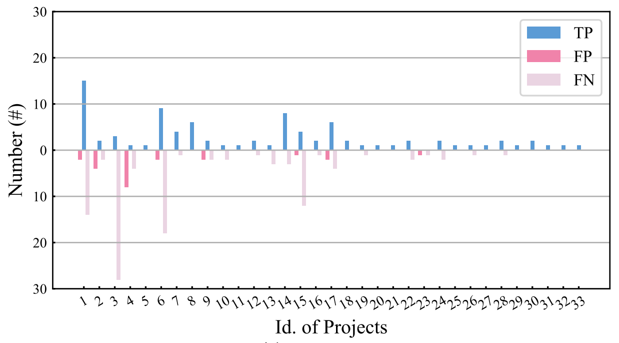
Figure 5: The Numbers of Projects w.r.t the Precision, Recall, and F1-Score and Above



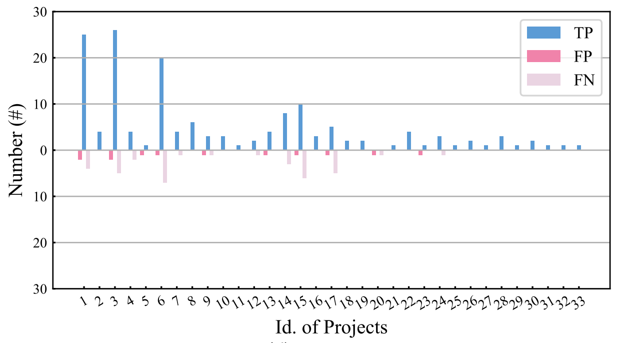
(a) Vuddy



(b) MVP



(c) V1Scan



(d) VMud

Figure 6: The Number of True Positives, False Positives, and False Negatives Detected by VMud and Clone-based Tools

VM across 74 projects, Movery detected 82 true VM across 36 projects, and V1Scan detected 113 true VM across 43 projects. Besides, we selected signatures of the critical functions identified by VMud, excluded signatures of non-critical functions, and applied them in those tools. The results are reported in Section 2.2(b). Table 5 details the projects with confirmed or resolved VM of VMud. Additionally, 5 VM are assigned with 5 CVE identifiers.

Effectiveness Per Project Figure 6 illustrates the number of true positives (TPs), false positives (FPs), and false negatives (FNs) detected by VMud and clone-based tools across projects with VM. To equally compare effectiveness per project, we choose projects that contain at least one TP, FP, or FN in the results across all four tools. The project Id. corresponds to the project names listed in the appendix table (see Table 9). We excluded Movery due to the diminished VM dataset. Specifically, VMud identifies more correct VM(i.e. identifies more true positives) than any other tool in 29 projects, accounting for 87.9% of the projects with VM. Moreover, VMud reports fewer false positives than other tools in 19 projects and fewer false negatives in 14 projects, representing 57.6% and 42.4% of the projects with VM, respectively.

5.2.3 Comparison with Learning-based Approaches. We conducted a comparative analysis of VMud's accuracy with DeepDFA [43] and SySeVR [32], two leading learning-based approaches. We directly utilized the trained models of DeepDFA and SySeVR on their respective datasets and used our ground truth as the testing dataset. Table 6 presents their results. The precision and recall metrics in their context are measured concerning reporting vulnerable functions, which differs from our definition. Therefore, we present VMud's precision and recall based on their definition to facilitate comparison, denoted by a superscript ^{F}. Consequently, SySeVR achieves

Table 4: Effectiveness and Accuracy Results of VMud and Clone-Based Tools (∗ denotes the Matching-one-function-in-all Approach and & Denotes the Matching-all-functions Approach)

Dataset	Tool	TP (#)	FP (#)	FN (#)	Precision	Recall	F1-Score
Complete VM Dataset	Vuddy*	168	25	161	**0.87**	0.51	0.64
	MVP*	226	163	103	0.58	0.69	0.63
	V1Scan*	113	47	216	0.71	0.34	0.46
	VMud *	275	51	54	**0.84**	**0.84**	**0.84**
	Vuddy&	47	2	282	0.96	0.14	**0.25**
	MVP&	48	3	281	0.94	0.15	**0.25**
	V1Scan&	32	1	297	**0.97**	0.10	0.18
	VMud &	59	2	270	**0.97**	**0.18**	**0.30**
Diminished VM Dataset	Movery*	82	56	27	0.59	0.75	0.66
	VMud *	90	10	19	**0.90**	**0.83**	**0.86**
	Movery&	17	3	94	0.85	0.16	0.26
	VMud &	29	1	80	**0.97**	**0.27**	**0.42**

Table 5: Target Projects with the Confirmed or Fixed VM

Target Project	TP (#)	FP (#)	FN (#)	Precision	Recall	Confirmed (#)	Fixed (#)
seemoo-lab/nexmon	4	0	2	1.00	0.67	4	0
panda-re/panda	4	0	1	1.00	0.80	4	0
rizinorg/rizin	6	0	1	1.00	0.86	3	3
Cisco-Talos/pyrebox	3	0	1	1.00	0.75	3	0
freebsd/freebsd-src	4	1	0	0.80	1.00	2	2
civetweb/civetweb	3	0	0	1.00	1.00	2	2
nodemcu/nodemcu-firmware	2	0	0	1.00	1.00	2	2
radareorg/radare2	2	0	0	1.00	1.00	2	2
gpac/gpac	2	0	0	1.00	1.00	2	2
catboost/catboost	2	0	0	1.00	1.00	2	0
TelegramMessenger/Telegram-iOS	4	0	0	1.00	1.00	2	2
ravynsoft/ravynos	4	0	0	1.00	1.00	1	1
mridgers/clink	1	0	0	1.00	1.00	1	1
eduard-permyakov/permafrost-engine	1	0	0	1.00	1.00	1	1
ntop/PF_RING	1	0	0	1.00	1.00	1	0
wireshark/wireshark	1	0	0	1.00	1.00	1	1
libretro/RetroArch	1	1	0	0.50	1.00	1	1
ApsaraDB/PolarDB-for-PostgreSQL	2	0	0	1.00	1.00	1	1
momotech/MLN	1	0	0	1.00	1.00	1	1
slact/nchan	1	0	0	1.00	1.00	1	1
Proxmark/proxmark3	1	0	0	1.00	1.00	1	1
sdlpal/sdlpal	1	0	0	1.00	1.00	1	1
openzfs/zfs	1	0	0	1.00	1.00	1	1
ImageMagick/ImageMagick	1	1	0	0.50	1.00	1	1
alibaba/tengine	1	1	0	0.50	1.00	1	1
Total	54	4	5	0.93	0.92	42	28

Table 6: Accuracy Results of Learning-Based Approaches

Tool	Precision^F	Recall^F	F1-Score^F
SySeVR	0.58	0.05	0.10
DeepDFA	0.68	0.81	0.74
VMud	0.83	0.79	0.81

a precision of 0.58 and a recall of 0.05, while DeepDFA achieves a precision of 0.68 and a recall of 0.81. In comparison, VMud achieves a precision of 0.83 a recall of 0.79, and an F1-Score of 0.81, marking a 7.5% F1-Score improvement over DeepDFA.

5.2.4 Inaccuracy Analysis. **False Positives Analysis** VMud generated 51 false positives, caused by two primary factors. Firstly,

Table 7: VMud's Robustness in Detecting VM with Multiple Numbers of Fixing Functions (GT Denotes the Number of VM and its Ratio in the Ground Truth Dataset)

Fixing Function (#)	GT (#, %)	TP (#)	FP (#)	FN (#)	Precision	Recall	F1-Score
2	152, 46.2%	134	15	18	0.90	0.88	**0.89**
3	38, 11.6%	33	11	5	0.75	0.87	0.80
4	45, 13.7%	35	8	10	0.81	0.78	0.80
5	22, 6.7%	19	4	3	0.83	0.86	0.84
6	12, 3.6%	9	0	3	1.00	0.75	0.86
7	19, 5.8%	18	5	1	0.78	0.95	0.81
8	5, 1.5%	5	3	0	**0.63**	**1.00**	0.77
9	4, 1.2%	1	0	3	**1.00**	**0.25**	0.40
10	4, 1.2%	3	1	1	0.75	0.75	0.75
>10	28, 8.5%	18	4	10	0.82	0.64	0.72
Total	329	275	51	54	-	-	-

some statements have limited semantic relationships, such as function invocations lacking input parameters (*e.g.,* `invoke()`) or return statements with literals (*e.g., return 0*). In these cases, matching semantic similarity is less effective than matching syntactic similarity. Consequently, VMud relies more heavily on syntactic similarity despite their contextual differences, resulting in 29 false positives. Secondly, for vulnerable functions already been patched, downstream developers proceeded with customizations by altering the patched statements. Typically, the patched statements are concise, comprising a small number of statements. Consequently, a single alternation can cause a significant drop in similarity with the original patched statements, leading to 22 false positives.

False Negatives Analysis. VMud encountered 54 false negatives, primarily due to three reasons. First, in the preprocessing (see Section 3.4.1), we utilized a fragment-level clone detection tool SAGA[27]. While this preprocessing helped eliminate numerous unnecessary detections while preserving VCCs in most cases, it resulted in 4 false negatives. Secondly, VMud labels a vulnerable function as safe if its signature matches both the vulnerable and patched functions from the VM. However, certain patches are trivial, leading to a significant resemblance in the signatures of vulnerable and patched functions. The high resemblance causes VMud to mistakenly categorize the vulnerable function as the patched, classifying the vulnerable function as safe, which results in 22 false negatives. Thirdly, similar to false positives caused by downstream customizations, there are instances where vulnerable functions have been patched, but downstream customizations have affected the vulnerable statements. It causes the functions to exhibit lower similarity to the vulnerable function signature, resulting in 28 false negatives.

5.3 Robustness Evaluation (RQ2)

To assess VMud's robustness, we examine its accuracy across VM with varying numbers of fixing functions. Table 7 depicts VMud's precision and recall concerning VM with the number of fixing functions. VMud achieves maximum precision and recall of 1.00 for VM and a minimum precision of 0.63 and recall of 0.25 for VM with different fixing function numbers. Nevertheless, certain categories may have a limited number of VM, which could impact the accuracy assessment. Overall, VMud maintains an F1-Score over 0.8 across fixing function numbers from 2 to 7, indicating its robustness for VM with different number of fixing functions.

Table 8: Results of the Ablation Study (CFS denotes the Critical Function Selection, SE denotes the semantic equivalence, PR denotes the Program Rephrasing, and CESM denotes the Contextual Equivalent Statement Matching)

	TP (#)	FP (#)	FN (#)	Precision	Recall	ΔPrecision	ΔRecall
w/o CFS	305	216	24	0.59	0.93	-0.25	0.09
w/o SE	187	44	142	0.81	0.57	-0.03	-0.27
w/o PR	249	46	80	0.84	0.76	-0.00	-0.08
w/o CESM	270	81	59	0.77	0.82	-0.07	-0.02
w/ Rules	210	71	119	0.75	0.64	-0.09	-0.20
w/ HITS	265	95	64	0.74	0.81	-0.10	-0.03

5.4 Ablation Study (RQ3)

To assess the impact of key components in VMUD, we conducted the ablation study. Table 8 summarizes the effectiveness results of the ablated versions. Notably, the Signature Matching in Phase II relies on the rephrased signature generated by Program Rephrasing. Therefore, in the ablated version VMUD w/o PR, both the Signature Matching in Phase II and Program Rephrasing are jointly removed. The ablated versions of VMUD show decreases in precision and recall compared to the original VMUD. Specifically, VMUD without Critical Function Selection experiences a precision drop of 0.25, with a 0.09 increase in recall. The increase in recall is due to the inclusion of recovered functions thus helping in VM detection. For VMUD without Program Rephrasing and Signature Matching in Phase II, the precision remains unchanged, and recall drops by 0.09. For VMUD without Contextually Equivalent Statements Map (CESM), the precision drops by 0.07, with a 0.02 decrease in recall. The results indicate that the three components of VMUD all contribute to the effectiveness of VMUD. For VMUD without semantic equivalence (*i.e.*, without program rephrasing and contextual equivalent statement mapping), it obtains a precision decrease of 0.03 and recall decrease of 0.27. Additionally, replacing PageRank in critical function selection with heuristic rules (w/ Rules) resulted in a 0.09 decrease in precision and 0.20 decrease in recall, while using the HITS algorithm (w/ HITS) resulted in a 0.10 decrease in precision and 0.03 decrease in recall.

5.5 Threshold Sensitivity (RQ4)

We conducted a sensitivity analysis to assess the impact of various thresholds (th_{pr}, th^{V}_{syn}, th^{V}_{sem}, th^{P}_{syn}, th^{P}_{sem}, th_{ce}) on VMUD's performance. Figure 7 illustrates how each threshold influences the recall, precision, and F1-Score of VMUD. The optimal performance for VMUD is achieved when the thresholds are set as 0.018, 0.7, 0.6, 0.3, 0.4, 0.6, respectively.

5.6 Performance Evaluation (RQ5)

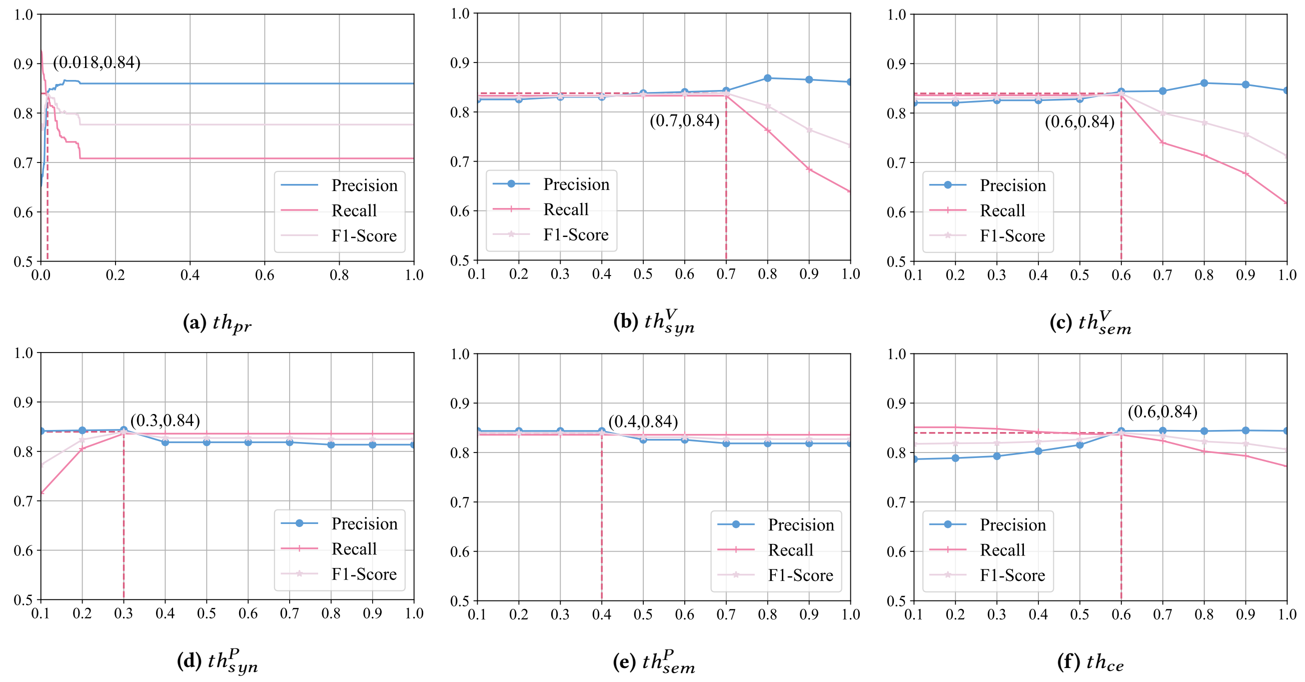
We evaluated the time cost of VMUD and several clone-based tools across a set of 972 target projects in VM detection. Figure 8 illustrates the time spent on VM detection in each project. On average, VMUD takes a median of 227 seconds to detect VM in a project, which is 173, 45, 142, 178 seconds longer than VUDDY, MVP, MOVERY, and V1SCAN. The time cost of VMUD primarily stems from call graph generation and semantic equivalent statement matching, which are computationally intensive. However, we consider this additional time cost acceptable given VMUD's effectiveness in detecting VM in real-world projects compared to other tools.

5.7 Discussion

Implication. We propose VMUD to discriminate fixing functions and leverage critical functions in VMUD. We utilize semantic equivalence to identify more VMs. VMUD achieves a precision, recall, and F1-score of 0.84, significantly outperforming state-of-the-art approaches. This result highlights the effectiveness of our proposed features. VMUD has successfully detected 275 new VM from 84 projects, with 42 confirmed cases and 5 assigned CVE identifiers. These outcomes demonstrate the effectiveness of VMUD. Our robustness evaluation shows that VMUD maintains consistent effectiveness regardless of the number of fixing functions in VMs. The ablation study indicates that critical function selection and semantic equivalence can significantly improve precision and recall. To the best of our knowledge, no existing works address both critical function selection and semantic equivalence. Our results demonstrate the potential of using critical function selection, which can guide future research further on critical function selection techniques and semantic-equivalent transformations and comparisons.

Threats. First, concerning our ground truth, it's constructed based on the manual analysis of detected vulnerabilities from VMUD and other clone-based tools. This approach may introduce bias as not all vulnerabilities may be detected by these tools, impacting the accuracy of our ground truth dataset. However, we have utilized existing tools comprehensively to cover a broad range of vulnerabilities. Second, in our effectiveness evaluation, we compared MOVERY on a smaller scale, which may not provide a complete comparison of precision and recall between MOVERY and other tools. Nonetheless, our evaluation of the small-scale dataset demonstrates VMUD's advantages over MOVERY fairly. Third, in our robustness evaluation, the distribution of VM concerning the number of fixing functions is uneven, with some categories having a small number of VM. This may not fully reflect VMUD's effectiveness in those categories. However, VMUD has consistently shown strong performance across most categories, highlighting its robustness.

Limitations. First, VMUD relies on several tools, limited by their accuracy and performance. For instance, Doxygen may generate incomplete call graphs, and Joern may produce incomplete CPGs, impacting critical function selection, signature generation, and signature matching. To address this, VMUD is designed to incorporate with other state-of-the-art tools to enhance its accuracy and performance. Second, VMUD is designed for detecting vulnerabilities with multiple functions. We plan to collect and generate signatures for vulnerabilities with a single function, and adopt program rephrasing and contextual equivalent statement map to observe the effectiveness of VMUD. Third, VMUD does not maintain the knowledge of code clone evolution or customizations in the target project, potentially resulting in false negatives or false positives if significant modifications occur. However, we implement semantic-centered statement matching to ensure consistent semantics even amid code changes. We also plan to obtain evolutionary knowledge on code clones to enhance the effectiveness of VMUD. Fourth, our semantic equivalence currently focuses on C/C++ and supports equivalent types based on documentation. We plan to expand our semantic equivalence transformations using a data-driven approach and extend support to additional programming languages.



(a) th_{pr} **(b)** th^{V}_{syn} **(c)** th^{V}_{sem}

(d) th^{P}_{syn} **(e)** th^{P}_{sem} **(f)** th_{ce}

Figure 7: Results on Threshold Sensitivity

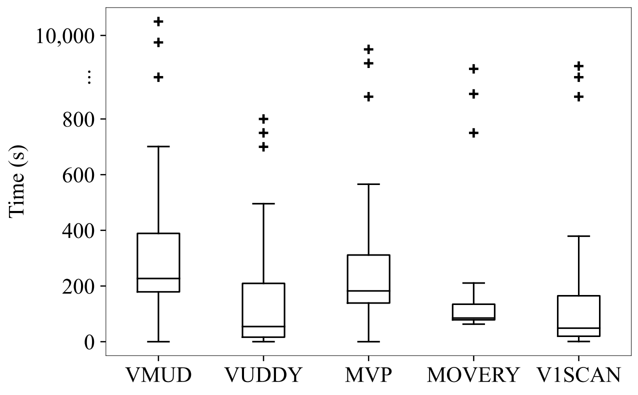


Figure 8: Results of Performance Evaluation

6 Related work

Vulnerability Detection within a Function. The function serves as the fundamental unit for vulnerability detection, as it is the most common granularity level. Existing research usually transforms the task of detecting vulnerabilities across an entire program into individual tasks within each function. Many approaches employ clone detection techniques to identify vulnerable code clones (VCCs) within each function. For instance, Kim et al. [25] introduced VUDDY, which identifies VCCs by matching their clone signatures. Xiao et al. proposed MVP [59], leveraging program slicing to extract partial functions and match patching clone signatures, thus reducing false positives in VCC detection. Woo et al. [54] used the knowledge of the oldest vulnerable function to overcome the syntax diversity of vulnerable code. They further proposed V1Scan [53] to filter out unused vulnerable functions, thus reducing false alarms. Feng et al. [12] utilizes multi-stage filtering and differential taint paths to achieve precise clone vulnerability scanning.

Additionally, learning-based approaches have been explored to classify vulnerable functions [5, 8, 41, 47, 65]. For example, Russell et al. [41] utilized CNN and DNN with Word2Vec embeddings for function-level code, while Zhou et al. introduced Devign [65] that encodes various function representations (*i.e.*, AST, CFG, DFG, NCS) into graph neural networks. Cui et al.[8] proposes a weighted feature graph (WFG) to compare functions in graph representation. Similarly, Wang et al. proposed GraphSPD [47] for multi-attributed graph convolution in vulnerability detection. However, for a vulnerability with multiple fixing functions, the vulnerability characteristics carried in each fixing function can vary depending on its surrounding context. Assigning equal importance to all fixing functions for a vulnerability may lead to over-representation, which brings false alarms. To this end, our approach applies critical function selection (see Section 3.3.1) to choose a subset of the fixing functions that are more significant, thus reducing false alarms.

Vulnerability Detection beyond a Function. Several studies have introduced methods for detecting vulnerabilities that go beyond the scope of individual functions. Software composition analysis (SCA) is instrumental in identifying vulnerabilities [21, 56, 57, 61]. By maintaining a database of vulnerable third-party libraries, SCA flags corresponding vulnerabilities based on instances of reused vulnerable libraries. However, it suffers from limitations in both precision and recall. This is because SCA primarily aims to characterize third-party libraries, which have a coarse-grained granularity compared to specific vulnerability characteristics.

Moreover, many approaches represent vulnerabilities at the file level or take the file-level program to analyze [23, 29, 35, 49]. Wi et al. [49] constructs a Control Flow Graph (CPG) for a PHP application and performs subgraph isomorphism analysis on it with a target

PHP snippet to detect vulnerabilities across functions. Similarly, Khodayari et al. [23] create a Hybrid Property Graph (HPG) for each JavaScript file and identify CSRF vulnerabilities using declarative traversals. Mirsky et al. [35] customize an LLVM compiler toolchain to generate enriched Program Dependency Graphs (PDGs) for input files and apply program slicing and graph neural networks to identify vulnerabilities and their CWE types. While these methods are effective at detecting vulnerabilities within a single file, they may overlook vulnerabilities that span multiple files. Inter-procedural vulnerability detection methods, on the other hand, can identify vulnerabilities where the statements to be patched and the statements triggering the vulnerability belong to different functions [30]. These methods utilize inter-procedural program slicing [32, 33] or employ inter-procedural analyses to detect various vulnerability types [4, 10, 15, 22, 28, 34]. However, they require prior knowledge of specific CWE types and sophisticated inter-procedural analysis. In contrast, our approach can detect vulnerabilities with multiple functions without prior knowledge of CWE types.

Vulnerability Abstraction and Normalization. Vulnerability abstraction and normalization aim to create an intermediate representation for comparing original vulnerabilities with their instances in the target program. Learning-based approaches perform implicit abstraction and normalization in high-dimensional spaces, which makes it hard to interpret the manifestation of abstracted and normalized vulnerabilities. In contrast, clone-based methods explicitly encode the abstraction and normalization. For instance, Li et al. [31] address missing code parts (e.g., missing }) by analyzing diff hunks from unpatched code pieces. Kim et al. [25] replace occurrences of parameter variables, local variables, data types, and function calls with symbols to adapt the intermediate representation for type-2 clones. The abstracted code is then normalized by standardizing formatting elements such as comments, spaces, tabs, and line feeds, and converting all characters to lowercase. Similar normalization techniques are also applied in Woo et al.'s work [53, 55]. Woo et al. [54] also used the knowledge of the oldest vulnerable function to diversify the representation of vulnerable code.

Xiao et al. [59] employ a similar approach to abstraction and normalization but exclude format strings from abstraction due to their association with certain vulnerabilities. Additionally, Cui et al. [8] propose preserving identifiers using the K-means algorithm and manual selection of representative ones, such as `buf` and `len`, as they are often involved in vulnerabilities. Mirsky et al. [35] utilize LLVM to generate a customized LLVM IR, reducing syntactic-level differences. However, these methods compare similarity based on the equality of abstracted and normalized statements without considering their contextual semantic equivalence. Our approach addresses this by employing semantic equivalent statement matching for vulnerability signature matching.

7 Conclusions

In this paper, we focus on the vulnerabilities due to code reuse in OSS, known as vulnerable code clones (VCCs) or recurring vulnerabilities. we introduce VMud, a novel approach for detecting "Vulnerabilities with Multiple Fixing Functions" (VM). VMud selects the critical functions from VM for signature generation which are a subset of the fixing functions. To deal with the potential decrease in recall due to excluding the remaining fixing functions, VMud employs semantic equivalent statement matching. It aims to uncover more VM by duplicating signatures of the critical functions and match precisely by contextual semantic equivalent statement mapping on the duplicated signatures. Our evaluation has demonstrated that VMud surpasses state-of-the-art vulnerability detection approaches by 17.6% in terms of F1-Score. VMud has successfully detected 275 new VM from 84 projects, with 42 confirmed cases and 5 assigned CVE identifiers.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 62372114, 62332005, 62402342), and "Sino-German Cooperation 2.0" Funding Program of Tongji University. This work was also sponsored by CCF-Huawei Populus Grove Fund.

References

[1] arcticwolf. 2024. *Beyond Sisense Navigating Rising Tide of Supply Chain Attacks.* Retrieved April 20, 2024 from https://arcticwolf.com/resources/blog/beyond-sisense-navigating-rising-tide-of-supply-chain-attacks/
[2] businessinsights. 2024. *10 Stats on the State of Vulnerabilities and Exploits.* Retrieved April 20, 2024 from https://www.bitdefender.com/blog/businessinsights/10-stats-on-the-state-of-vulnerabilities-and-exploits/
[3] Checkmarx. 2023. *9th Annual State of Software the Supply Chain.*
[4] Checkmarx. 2024. *Checkmarx.* Retrieved April 20, 2024 from https://checkmarx.com/
[5] Kenneth Ward Church. 2017. Word2Vec. *Natural Language Engineering* 23, 1 (2017), 155–162.
[6] Roland Croft, M Ali Babar, and Li Li. 2022. An investigation into inconsistency of software vulnerability severity across data sources. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering.* 338–348.
[7] crowdstrike. 2024. *Supply Chain Attacks.* Retrieved April 20, 2024 from https://www.crowdstrike.com/cybersecurity-101/cyberattacks/supply-chain-attacks/
[8] Lei Cui, Zhiyu Hao, Yang Jiao, Haiqiang Fei, and Xiaochun Yun. 2020. Vuldetector: Detecting vulnerabilities using weighted feature graph comparison. *IEEE Transactions on Information Forensics and Security* 16 (2020), 2004–2017.
[9] doxygen. 2024. *Doxygen.* Retrieved April 20, 2024 from https://www.doxygen.nl/
[10] Facebook. 2024. *Infer.* Retrieved April 20, 2024 from https://fbinfer.com/
[11] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories.* 508–512.
[12] Siyue Feng, Yueming Wu, Wenjie Xue, Sikui Pan, Deqing Zou, Yang Liu, and Hai Jin. 2024. {FIRE}: Combining {Multi-Stage} Filtering with Taint Analysis for Scalable Recurring Vulnerability Detection. In *Proceedings of the 33rd USENIX Security Symposium.* 1867–1884.
[13] gcc. 2024. *GCC.* Retrieved April 20, 2024 from https://gcc.gnu.org/
[14] GitHub. 2023. *The 2023 State of the OCTOVERSE.* Retrieved April 20, 2024 from https://octoverse.github.com/
[15] GitHub. 2024. *CodeQL.* Retrieved April 20, 2024 from https://codeql.github.com/
[16] GitHub. 2024. *GitHub API Query.* Retrieved April 20, 2024 from https://api.github.com/search/repositories?q=language:C&order=desc&per_page=1000&page=1
[17] gpac. 2024. *gpac repository.* Retrieved April 20, 2024 from https://github.com/gpac/gpac/
[18] gpac. 2024. *Patch Commit for CVE-2021-32134.* Retrieved April 20, 2024 from https://github.com/gpac/gpac/commit/328c6d682698fdb9878dbb4f282963d42c538c01
[19] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories.* 121–130.
[20] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empirical Software Engineering* 27, 4 (2022), 90.
[21] Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2023. Third-Party Library Dependency for Large-Scale SCA in the C/C++ Ecosystem: How Far Are We?. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis.* 1383–1395.
[22] Wooseok Kang, Byoungho Son, and Kihong Heo. 2022. TRACER: signature-based static analysis for detecting recurring vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.* 1695–1708.

[23] Soheil Khodayari and Giancarlo Pellegrino. 2021. {JAW}: Studying Client-side {CSRF} with Hybrid Property Graphs and Declarative Traversals. In *Proceedings of the 30th USENIX Security Symposium.* 2525–2542.

[24] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *Proceedings of the IEEE 31st International Conference on Software Engineering.* 309–319.

[25] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy.* 595–614.

[26] Rainer Koschke and Saman Bazrafshan. 2016. Software-clone rates in open-source programs written in C or C++. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Vol. 3. 1–7.

[27] Guanhua Li, Yijian Wu, Chanchal K Roy, Jun Sun, Xin Peng, Nanjie Zhan, Bin Hu, and Jingyi Ma. 2020. SAGA: efficient and large-scale detection of near-miss clones with GPU acceleration. In *Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering.* 272–283.

[28] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. 2023. A hybrid alias analysis and its application to global variable protection in the linux kernel. In *32nd USENIX Security Symposium.* 4211–4228.

[29] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining node.js vulnerabilities via object dependence graph and query. In *Proceedings of the 31st USENIX Security Symposium.* 143–160.

[30] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the Effectiveness of Function-Level Vulnerability Detectors for Inter-Procedural Vulnerabilities. (2024), 1–12.

[31] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications.* 201–213.

[32] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.

[33] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[34] microfocus. 2024. *Fortify.* Retrieved April 20, 2024 from https://www.microfocus.com/en-us/cyberres/application-security

[35] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. VulChecker: Graph-based Vulnerability Localization in Source Code. In *31st USENIX Security Symposium, Security 2022.*

[36] networkx.org. 2024. *Networkx Implementation of PageRank.* Retrieved May 25, 2024 from https://networkx.org/documentation/networkx-1.10/reference/generated/networkx.algorithms.link_analysis.pagerank_alg.pagerank_scipy.html

[37] NVD. 2024. *CVE-2021-32134 Detail4.* Retrieved April 20, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2021-32134

[38] nvd. 2024. *National Vulnerability Database.* Retrieved April 20, 2024 from https://nvd.nist.gov/feeds/json/cve/1.1/

[39] Openstd. 2024. *Open Std.* Retrieved April 20, 2024 from https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf

[40] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2010. Detection of recurring software vulnerabilities. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering.* 447–456.

[41] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th IEEE international conference on machine learning and applications.* IEEE, 757–762.

[42] ShiftLeftSecurity. 2024. *Joern.* Retrieved April 20, 2024 from https://github.com/ShiftLeftSecurity/joern

[43] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering.* 1–13.

[44] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An empirical study of deep learning models for vulnerability detection. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering.* 2237–2248.

[45] Friedrich Steimann, Philip Mayer, and Andreas Meißner. 2006. Decoupling classes with inferred interfaces. In *Proceedings of the 2006 ACM symposium on Applied computing.* 1404–1408.

[46] Eelco Visser. 2001. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science* 57 (2001), 109–143.

[47] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. GraphSPD: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy.* 2409–2426.

[48] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M Zhang, and Qing Liao. 2023. Vulnerability detection with graph simplification and enhanced graph representation learning. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering.* 2275–2286.

[49] Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Sooel Son. 2022. HiddenCPG: large-scale vulnerable clone detection using subgraph isomorphism of code property graphs. In *Proceedings of the ACM Web Conference 2022.* 755–766.

[50] Wikepedia. 2024. *HITS algorithm.* Retrieved May 25, 2024 from https://en.wikipedia.org/wiki/HITS_algorithm

[51] Wikipedia. 2024. *PageRank.* Retrieved April 20, 2024 from https://en.wikipedia.org/wiki/PageRank

[52] wikipedia. 2024. *Precision and Recall.* Retrieved April 20, 2024 from https://en.wikipedia.org/wiki/Precision_and_recall

[53] Seunghoon Woo, Eunjin Choi, Heejo Lee, and Hakjoo Oh. 2023. {V1SCAN}: Discovering 1-day Vulnerabilities in Reused {C/C++} Open-source Software Components Using Code Classification Techniques. In *32nd USENIX Security Symposium.* 6541–6556.

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

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

[56] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A precise and scalable approach for identifying modified open-source software reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering.* 860–872.

[57] Jiahui Wu, Zhengzi Xu, Wei Tang, Lyuye Zhang, Yueming Wu, Chengyue Liu, Kairan Sun, Lida Zhao, and Yang Liu. 2023. Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions. In *2023 IEEE/ACM 45th International Conference on Software Engineering.* 270–282.

[58] Susheng Wu, Wenyan Song, Kaifeng Huang, Bihuan Chen, and Xin Peng. 2024. Identifying Affected Libraries and Their Ecosystems for Open Source Software Vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.* 1–12.

[59] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. 2020. {MVP}: Detecting Vulnerabilities using {Patch-Enhanced} Vulnerability Signatures. In *29th USENIX Security Symposium.* 1165–1182.

[60] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. 2022. Tracking patches for open source software vulnerabilities. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 860–871.

[61] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. ModX: binary level partially imported third-party library detection via program modularization and semantic matching. In *Proceedings of the 44th International Conference on Software Engineering.* 1393–1405.

[62] Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. 2004. Predicting source code changes by mining change history. *IEEE transactions on Software Engineering* 30, 9 (2004), 574–586.

[63] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2023. Vulnerability detection by learning from syntax-based execution paths of code. *IEEE Transactions on Software Engineering* (2023).

[64] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. 2021. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering.* 705–716.

[65] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

A Effectiveness Results on Target Projects of VMud

We present the project names to project identifiers, true positives, false positives, detected by VMud and the false negatives of VMud in Table 9.

Table 9: The Number of True Positives, False positives, Detected by VMud and the False Negatives VMud Missed

Id.	Project Name	TP (#)	FP (#)	FN (#)	Id.	Project Name	TP (#)	FP (#)	FN (#)	Id.	Project Name	TP (#)	FP (#)	FN (#)
1	hanwckf/rt-n56u	25	2	4	36	flatpak/flatpak	0	1	0	71	slact/nchan	1	0	0
2	ravynsoft/ravynos	4	0	0	37	AdAway/AdAway	21	0	4	72	ImageMagick/ImageMagick	1	1	0
3	Stichting-MINIX-Research-Foundation/minix	26	2	5	38	microsoft/WSL2-Linux-Kernel	0	7	0	73	momotech/MLN	1	0	0
4	seemoo-lab/nexmon	4	0	2	39	catboost/catboost	2	0	0	74	nodemcu/nodemcu-firmware	2	0	0
5	alibaba/tengine	1	1	0	40	mridgers/clink	1	0	0	75	saki4510t/UVCCamera	1	0	0
6	fastos/fastsocket	20	1	7	41	gpac/gpac	2	0	0	76	libretro/RetroArch	1	1	0
7	panda-re/panda	4	0	1	42	spotify/linux	0	8	0	77	wireshark/wireshark	1	0	0
8	tanersener/mobile-ffmpeg	6	0	0	43	emscripten-core/emscripten	1	0	0	78	ejoy/ejoy2d	1	0	0
9	premake/premake-core	3	1	1	44	Sunzxyong/Tiny	1	0	0	79	jart/cosmopolitan	1	0	0
10	civetweb/civetweb	3	0	0	45	lavabit/magma	1	0	0	80	arendst/Tasmota	1	0	0
11	madeye/proxydroid	1	0	0	46	ossec/ossec-hids	1	0	0	81	flipperdevices/flipperzero-firmware	1	0	0
12	y123456yz/reading-code-of-nginx-1.9.2	2	0	1	47	moby/hyperkit	1	0	0	82	nmap/nmap	0	1	0
13	freebsd/freebsd-src	4	1	0	48	Proxmark/proxmark3	1	0	0	83	RedisGraph/RedisGraph	1	0	0
14	CloverHackyColor/CloverBootloader	8	0	3	49	cesanta/mongoose	1	0	0	84	rofl0r/proxychains-ng	1	0	0
15	damonkohler/sl4a	10	1	6	50	alibaba/LVS	23	1	4	85	darktable-org/darktable	1	0	0
16	joncampbell123/dosbox-x	3	0	0	51	F-Stack/f-stack	8	1	0	86	antirez/sds	0	1	0
17	alibaba/AliOS-Things	5	1	5	52	OpenAtomFoundation/TencentOS-tiny	1	0	0	87	openssl/openssl	0	1	0
18	zlgopen/awtk	2	0	0	53	NetBSD/src	2	0	0	88	mabeijianxi/small-video-record	1	0	0
19	sumatrapdfreader/sumatrapdf	2	0	0	54	sdlpal/sdlpal	1	0	0	89	espressif/ESP8266_RTOS_SDK	1	0	0
20	nginx/nginx-releases	0	1	1	55	b4winckler/macvim	0	1	0	90	ctfs/write-ups-2016	1	0	0
21	peng-zhihui/ElectronBot	1	0	0	56	veracrypt/VeraCrypt	0	1	0	91	nginx/nginx	0	1	0
22	TelegramMessenger/Telegram-iOS	4	0	0	57	openzfs/zfs	1	0	0	92	jiangdongguo/AndroidUSBCamera	1	0	0
23	Tencent/xLua	1	1	0	58	antirez/disque	1	0	0	93	ntop/PF_RING	1	0	0
24	Cisco-Talos/pyrebox	3	0	1	59	rizinorg/rizin	6	0	1	94	h2o/h2o	1	0	0
25	aws/amazon-freertos	1	0	0	60	reactos/reactos	1	0	0	95	gozfree/gear-lib	0	0	1
26	yangchaojiang/yjPlay	2	0	0	61	nicolasff/webdis	1	0	0	96	omnirom/android_bootable_recovery	1	0	0
27	rogerclarkmelbourne/Arduino_STM32	1	0	0	62	FreeRDP/FreeRDP	0	4	0	97	TeamWin/Team-Win-Recovery-Project	1	0	0
28	rmtheis/tess-two	3	0	0	63	radareorg/radare2	2	0	0	98	coolwanglu/vim.js	0	1	0
29	eduard-permyakov/permafrost-engine	1	0	0	64	ApsaraDB/PolarDB-for-PostgreSQL	2	0	0	99	vifm/vifm	1	0	0
30	leixiaohua1020/simplest_ffmpeg_mobile	2	0	0	65	swaywm/sway	0	3	0	100	mattrajca/sudo-touchid	0	0	1
31	peng-zhihui/Dummy-Robot	1	0	0	66	mjolnirapp/mjolnir	1	1	0	101	micropython/micropython	1	0	0
32	armink/EasyLogger	1	0	0	67	greenplum-db/gpdb	0	1	0	102	InfiniTimeOrg/InfiniTime	1	0	0
33	Ralim/IronOS	1	0	0	68	systemd/systemd	0	1	0					
34	kbengine/kbengine	8	1	0	69	RediSearch/RediSearch	1	0	0					
35	endrazine/wcc	1	0	0	70	ultrajson/ultrajson	0	1	0					