MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures

Yang Xiao[1,2], *Bihuan Chen*[3*], *Chendong Yu*[1,2], *Zhengzi Xu*[4], *Zimu Yuan*[1,2], *Feng Li*[1,2], *Binghong Liu*[1,2], *Yang Liu*[4], *Wei Huo*[1,2*], *Wei Zou*[1,2], *Wenchang Shi*[5]

[1]*Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*

[2]*School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*

[3]*School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China*

[4]*School of Computer Science and Engineering, Nanyang Technological University, Singapore*

[5]*Renmin University of China, Beijing, China*

Abstract

Recurring vulnerabilities widely exist and remain undetected in real-world systems, which are often resulted from reused code base or shared code logic. However, the potentially small differences between vulnerable functions and their patched functions as well as the possibly large differences between vulnerable functions and target functions to be detected bring challenges to clone-based and function matching-based approaches to identify these recurring vulnerabilities, i.e., causing high false positives and false negatives.

In this paper, we propose a novel approach to detect recurring vulnerabilities with low false positives and low false negatives. We first use our novel program slicing to extract vulnerability and patch signatures from vulnerable function and its patched function at syntactic and semantic levels. Then a target function is identified as potentially vulnerable if it matches the vulnerability signature but does not match the patch signature. We implement our approach in a tool named MVP. Our evaluation on ten open-source systems has shown that, i) MVP significantly outperformed state-of-the-art clone-based and function matching-based recurring vulnerability detection approaches; ii) MVP detected recurring vulnerabilities that cannot be detected by general-purpose vulnerability detection approaches, i.e., two learning-based approaches and two commercial tools; and iii) MVP has detected 97 new vulnerabilities with 23 CVE identifiers assigned.

1 Introduction

Vulnerabilities can be exploited to attack software systems, threatening system security. Therefore, it is vital to detect and patch vulnerabilities in software systems as early as possible. Various techniques have been developed to detect vulnerabilities, e.g., static analysis (e.g., [17, 45, 61, 62, 75]), fuzzing (e.g., [9, 11, 39, 52, 63, 64, 67, 69, 70]), symbolic execution (e.g., [6, 10, 24, 60]) or manual auditing. Several advances have also been made to automatically patch vulnerabilities for the purpose of reducing patch deployment delays (e.g., [13, 15, 25, 47, 73]).

Due to reusing code base or sharing code logic (e.g., similar processing logic for similar objects in their different usages) in software systems, recurring vulnerabilities which share the similar characteristics with each other widely exist but remain undetected in real-world programs [46, 50, 75]. Therefore, recurring vulnerability detection has gained wide popularity, especially with the increased availability of vulnerabilities. The scope of this paper is to detect recurring vulnerabilities; i.e., given a vulnerability that behaves in a very specific way in a program, we detect whether other programs may have this specific behavior. Differently, general-purpose vulnerability detection techniques (e.g., [1, 2, 41, 44, 77]) leverage the general behaviors of a large fraction of vulnerabilities to find specific instances of these general behaviors in programs.

Existing Approaches. A general idea to detect recurring vulnerabilities is to match the source code of a target system with known vulnerabilities; and various approaches have been proposed (e.g., [28, 34, 38, 40, 41, 50, 56, 65, 77, 78]). Existing approaches can be classified into clone-based and function matching-based approaches.

Clone-based approaches (e.g., [28, 34, 38, 50, 78]) consider the recurring vulnerability detection problem as a code clone detection problem; i.e., they extract token- or syntax-level signature from a known vulnerability, and identify code clones to the signature as potentially vulnerable. Function matching-based approaches (e.g., [56, 65]) directly use vulnerable functions in a known vulnerability as the signature and detect code clones to those vulnerable functions. They do not consider any vulnerability characteristics as they are not designed particularly for recurring vulnerability detection.

However, due to the nature of clone detection and no consideration of how a vulnerability is fixed, clone-based and function matching-based approaches fail to differentiate the potentially small differences between vulnerable function and patched function, causing high false positives. Moreover,

* Bihuan Chen and Wei Huo are the corresponding authors.

these approaches fail to detect recurring vulnerabilities whose vulnerable functions have large code differences from those of the known vulnerability due to their imprecise vulnerability signature or pattern, leading to high false negatives.

Challenges. In summary, there are two main challenges in detecting recurring vulnerabilities with both low false positives and low false negatives. The first challenge is how to distinguish already patched vulnerabilities to reduce false positives. The second challenge is how to precisely generate the signature of a known vulnerability to reduce both false positives and false negatives.

Our Approach. To address the two challenges, we propose a novel recurring vulnerability detection approach, named MVP (**M**atching **V**ulnerabilities with **P**atches). Specifically, to address the first challenge, we not only generate a vulnerability signature but also a patch signature to capture how a vulnerability is caused and fixed. We leverage the vulnerability signature to search for potentially vulnerable functions, and use the patch signature to distinguish whether they are already patched or not. To address the second challenge, we propose a novel slicing method to extract only vulnerability-related and patch-related statements to generate vulnerability and patch signatures at both syntactic level and semantic level. Besides, we apply statement abstraction and entropy-based statement selection to further improve the accuracy of MVP.

Evaluation. We have implemented MVP, and evaluated it on ten open-source systems with 25,377 security patches. We compared MVP with two state-of-the-art, most closely related clone-based approaches (ReDeBug [28] and VUUDY [34]). The results indicate that MVP outperformed ReDeBug and VUUDY by improving precision by 74.5% and 75.6% and recall by 42.4% and 65.8%. MVP discovered 97 new vulnerabilities with 23 CVE identifiers assigned. We also compared MVP with function matching-based approaches (SourcererCC [56] and CCAligner [65]). The results show that MVP outperformed SourcererCC and CCAligner by improving precision by 83.1% and 83.3% and recall by 22.5% and 30.6%

Besides, we compared MVP with the-state-of-art learning-based approaches (VulDeePecker [41] and Devign [77]) and commercial tools (Coverity [2] and Checkmarx [1]) to demonstrate the incapability of such general-purpose vulnerability detection techniques in detecting recurring vulnerabilities.

Contribution. The main contributions of our work are:

- We proposed and implemented a novel recurring vulnerability detection approach by leveraging vulnerability and patch signatures through our novel slicing technique.
- We conducted intensive evaluation to compare MVP with four categories of state-of-the-art approaches. MVP significantly outperformed them in accuracy.
- We found 97 new vulnerabilities in ten open-source systems with 23 CVE identifiers assigned.

2 Motivation

2.1 Problems

We investigate the similarity among vulnerable function (V), patched function (P) and target function (T) to illustrate the problems of existing approaches. P is the result of applying a security patch on V; and T is a vulnerable function in a target system under detection. We use $Sim(f_1, f_2)$ to denote the similarity score between function f_1 and f_2.

We used 34,019 pairs of vulnerable functions and their corresponding patched functions in 25,377 security patches in ten projects (used in our evaluation), and used SourcererCC [56] to measure the similarity score of each pair. The results show that $Sim(V,P)$ is above 70% for 91.3% of pairs. Therefore, code differences between vulnerable and patched functions are small in most vulnerabilities. As a result, clone-based approaches may detect patched function as vulnerable because they take only vulnerable function as the signature (without taking patched function into consideration). Function matching-based approaches identify patched function as vulnerable if $Sim(V,P)$ is larger than its default similarity threshold (e.g., 70% for SourcererCC [56] and 60% for CCAligner [65]). In a word, when $Sim(V,P)$ is large, existing approaches can introduce high false positives.

On the other hand, if $Sim(V,T)$ is small, existing approaches cannot detect T. In fact, in all the truly vulnerable functions detected in our evaluation, 35.1% of them have a $Sim(V,T)$ of lower than 70% and existing approaches miss most of them (see Table 3). Specifically, clone-based approaches take a whole vulnerable function as the signature, thus it is more likely to miss T whose $Sim(V,T)$ is small. Function matching-based approaches cannot detect T as vulnerable if $Sim(V,T)$ is smaller than their default similarity threshold (e.g., 70% for SourcererCC [56] and 60% for CCAligner [65]). In summary, when $Sim(V,T)$ is small, existing approaches may introduce high false negatives.

2.2 A Motivating Example

To illustrate the limitation of existing approaches and to motivate the idea of MVP, we use a vulnerability in Qcacld-2.0 as a running example. Qcacld-2.0 is an open-source driver for Qualcomm WLAN, which is widely used in Android phones. Fig. 1 shows a patch of Qcacld-2.0, which fixes an out-of-bound access vulnerability in function `WDA_TxPacket`. The patch adds a sanitizing check for the local variable `vdev_id` at Line 18–22, which is used as an index to access the array `wma_handle->interfaces` in Line 28.

Clone-based Approach. For example, ReDeBug [28] uses pure syntax-level matching to find recurring vulnerabilities. Given a patch, ReDeBug takes lines prefixed by a "-" and context information (lines with no prefix) as vulnerable signature while removing blank lines, braces and comments. If

```
1 diff --git a/CORE/SERVICES/WMA/wma.c b/CORE/SERVICES/
      WMA/wma.c
2 index 0cb2ab8bd..cac414969 100644
3 --- a/CORE/SERVICES/WMA/wma.c
4 +++ b/CORE/SERVICES/WMA/wma.c
5 bool WDA_TxPacket(void *wma_context, void *tx_frame,
      eFrameType frmType, tpPESession psessionEntry) {
6     tp_wma_handle wma_handle = (tp_wma_handle)(
      wma_context);
7     int32_t is_high_latency;
8     u_int8_t downld_comp_required = 0;
9     tpAniSirGlobal pMac;
10    ol_txrx_vdev_handle txrx_vdev;
11    u_int8_t vdev_id = psessionEntry->smeSessionId;
12
13    if (NULL == wma_handle) {
14        printf("wma_handle is NULL\n");
15        return false;
16    }
17
18+   if (vdev_id >= wma_handle->max_bssid) {
19+       printf("Invalid vdev_id %u\n", vdev_id);
20+       return false;
21+   }
22+
23    pMac = (tpAniSirGlobal)vos_get_context(
      VOS_MOD_ID_PE, wma_context->vos_context);
24    if(!pMac) return false;
25    if (frmType >= HAL_TXRX_FRM_MAX) return false;
26    if (!((frmType == HAL_TXRX_FRM_802_11_MGMT) || (
      frmType == HAL_TXRX_FRM_802_11_DATA)))
27        return false;
28    txrx_vdev = wma_handle->interfaces[vdev_id].handle
      ;
29    if(!txrx_vdev) return false;
30    if (frmType == HAL_TXRX_FRM_802_11_DATA) {
31        adf_nbuf_t skb = (adf_nbuf_t)tx_frame;
32        adf_nbuf_t ret = ol_tx_non_std(txrx_vdev,
      ol_tx_spec_no_free, skb);
33        if (ret) { // do something }
34        is_high_latency = wdi_out_cfg_is_high_latency(
      txrx_vdev->pdev->ctrl_pdev);
35        downld_comp_required = is_high_latency &&
      tx_frm_ota_comp_cb;
36    }
37    if(downld_comp_required) { // do something }
38    return true;
39 error:
40    return false;
41 }
```

Figure 1: Patch for an Out-of-Bound Access Vulnerability

the signature is found in the target function, the target function will be identified as vulnerable. In Fig. 1, ReDeBug uses nearby 3 (by default) lines before and after the patch (Line 15–17 and Line 23–25) to generate the vulnerability signature. However, since in the patched function, Line 20–22 and Line 23–25 happen to have the same syntax as the vulnerability signature, ReDeBug mistakenly detects the patched function as vulnerable.

MVP. In Fig. 1, the root cause of vulnerability is that there is a missing check for the local variable `vdev_id`, which is used as an index to access an array later at Line 28. MVP builds data flow of the local variable `vdev_id` as the semantic signature. With the help of it, it can detect semantic-equivalent vulnerabilities whose syntax is slightly changed. The detailed signature extraction process will be discussed in § 3.2.

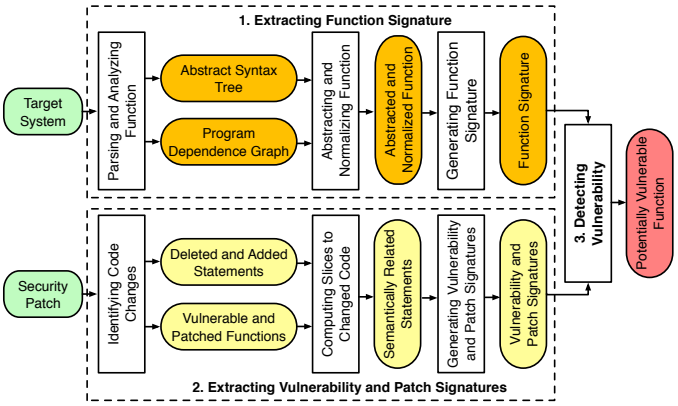


Figure 2: Overview of Our Approach (rounded rectangles represent inputs (green), intermediate results (yellow), and final outputs (red), and rectangles represent sub-steps in MVP)

3 Methodology

Fig. 2 shows the overview of MVP, which contains three steps. The *Extracting Function Signature* step (§ 3.2) takes a target system as an input, and generates a signature for each function in the target system. The *Extracting Vulnerability and Patch Signatures* step (§ 3.3) takes a security patch as an input, and generates a vulnerability signature and a patch signature to reflect a vulnerability from the perspective of how it is caused and how it is fixed. The final *Detecting Vulnerability* step (§ 3.4) determines whether each function in the target system is potentially vulnerable by matching its signature with the vulnerability and patch signatures.

3.1 Definition

This section introduces the key definitions used in our approach. We first define the function signature as follows.

Definition 3.1 (Function Signature)**.** Given a C/C++ function f, we define its signature as a tuple (f_{syn}, f_{sem}), where f_{syn} is a set of the hash values of all statements in the function; f_{sem} is a set of 3-tuple $(h_1, h_2, type)$ such that h_1 and h_2 denote hash values of two statements (i.e., $h_1, h_2 \in f_{syn}$), and $type \in \{data, control\}$ denotes the statement whose hash value is h_1 has a data or control dependency on the statement whose hash value is h_2.

f_{syn} captures the statements of a target function as the syntactic signature. f_{sem} captures data and control dependencies among statements in the function as semantic signature. They are providing complementary information of a function to help to improve the matching accuracy.

In the remaining of this paper, we assume that each vulnerability is within one function. We use (f_v, p_v) to denote the pair of a vulnerable function f_v and the patched function p_v after fixing the vulnerability in f_v.

Definition 3.2 (Function Patch)**.** Given a pair of functions (f_v, p_v), the function patch P_v consists of one or more hunks. A hunk is a basic unit in patch, which consists of context lines,

deleted lines and/or added lines. Deleted lines are lines in the f_v but missing from p_v, while added lines are lines missing in f_v but present in p_v. The first and last 3 lines in a hunk and lines between deleted and/or added lines are context lines.

Given a function pair (f_v, p_v) and the patch P_v, we further define S_{del} as the statements in the f_v but missing from p_v, S_{add} as the statements missing in f_v but present in p_v, S_{vul} as all statements in f_v, S_{pat} as all statements in p_v.

3.2 Extracting Function Signature

We extract f_{syn} and f_{sem} for each function f in three steps as explained in the next three subsections respectively.

3.2.1 Parsing and Analyzing Function

Given the source code of a target system as the input, we first apply a robust parser Joern [74] to parse the code and generate a code property graph which merges abstract syntax tree, control flow graph and program dependence graph into a joint data structure. From the code property graph, we first obtain all the functions in the target system, and then for each function, we generate its abstract syntax tree (AST) and program dependence graph (PDG).

3.2.2 Abstracting and Normalizing Function

As developers may reuse code snippets with renamed parameters/variables, we first perform abstraction to each function before extracting the signature to avoid false negatives. Specifically, we identify formal parameters, local variables and string literals from the AST of a function, and replace every occurrence of formal parameters, local variables and string literals respectively with a normalized symbol `PARAM`, `VARIABLE` and `STRING`, respectively.

There is an exception for format strings in our abstraction; i.e., instead of replacing a format string with `STRING`, we only reserve format specifiers, following the prototype `%[flags][width][.precision][length]specifier` [3], in a format string. The reason is that several types of vulnerabilities are related to format specifiers, such as format string vulnerability and stack-based buffer overflow vulnerability. For example, the patch of a stack-based buffer overflow vulnerability CVE-2018-7186 [5] just changed the format string "`protos=%s`" to "`protos=%490s`". We abstract these two string formats to "`%s`" and "`%490s`" respectively.

After abstraction, we apply normalization to each statement in the function body via removing all comments, braces, tabs, and white spaces. In this way, our approach becomes tolerant to changes to code formatting or comments.

3.2.3 Generating Function Signature

To generate the function signature, we first apply a hash function on each abstracted and normalized statement to compute a hash value. The syntactic signature of a function, f_{syn}, is thus represented as the set of computed hash values of statements.

Then, we extract data or control dependencies between two statements from the PDG of a function. Each dependency is denoted as a 3-tuple $(h_1, h_2, type)$, where h_1 and h_2 denote hash values of two statements (i.e., $h_1, h_2 \in f_{syn}$), and $type \in \{data, control\}$ denotes the statement whose hash value is h_1 has a data or control dependency on the statement whose hash value is h_2. The semantic signature of a function, denoted as f_{sem}, is thus represented as a set of extracted dependencies. Our abstraction and normalization could lose some information and lead to false positive in matching results. However, taking semantic information (i.e., control or data dependency between statements) into consideration can make up for the deficiency of abstraction and normalization.

Given the function code in Fig. 3(a), Fig. 3(b) is the result after our abstraction, where two formal parameters `str` and `target` are replaced with `PARAM`, two variables `i` and `num` are replaced with `VARIABLE`, and the string literal at Line 2

(a) Original Function Code

```
1  int count_character(char str[], char target) {
2    printf("The input string is:");
3    printf(str);
4    unsigned int i, num = 0;
5    for (i = 0; i < strlen(str); i++)
6      if (str[i] == target)
7        num += 1;
8    printf("\nTotal count of %c is %d\n", target, num);
9    return num;
10 }
```

(b) Abstracted Function Code

```
1  int count_character(char PARAM[], char PARAM) {
2    printf(STRING);
3    printf(PARAM);
4    unsigned int VARIABLE, VARIABLE = 0;
5    for (VARIABLE = 0; VARIABLE < strlen(PARAM);
       VARIABLE++)
6      if (PARAM[VARIABLE] == PARAM)
7        VARIABLE += 1;
8    printf("%c%d", PARAM, VARIABLE);
9    return VARIABLE;
10 }
```

(c) Normalized Function Code

```
1 printf(STRING);
2 printf(PARAM);
3 unsignedintVARIABLE,VARIABLE=0;
4 for(VARIABLE=0;VARIABLE<strlen(PARAM);VARIABLE++)
5 if(PARAM[VARIABLE]==PARAM)
6 VARIABLE+=1;
7 printf("%c%d",PARAM,VARIABLE);
8 return VARIABLE;
```

(d) Function Signature

```
1 [b603b5274b77a7e0343a2cee1a2bf153 (b603b5),
2  19663da837da5adf57815a71e8c43cc8 (19663d),
3  22d46299807c89d38e4b7c4a71aa4261 (22d462),
4  c8f314bf9eb06b41c2cffc558ab3488d (c8f314),
5  ce48ce953b21675299199dd00dc54ac1 (ce48ce),
6  c6b080f731106c91040b8ca37a772ec8 (c6b080),
7  4e4aab522d85d757afcbd2b05ce64041 (4e4aab),
8  cdaad6b9d8591ad71d3475ebe23a60d3 (cdaad6)]
9
10 [(22d462, c6b080, data), (22d462, 4e4aab, data),
11  (22d462, cdaad6, data), (c6b080, 4e4aab, data),
12  (c6b080, cdaad6, data), (c8f314, ce48ce, data),
13  (c8f314, ce48ce, control), (ce48ce, c6b080, control)]
```

Figure 3: An Example of Extracting Function Signature

is replaced with `STRING`, while the string literal at Line 8 is replaced with `%c%d` because it is a format string. Fig. 3(c) is the result of our normalization on Fig. 3(b). Finally, Fig. 3(d) gives the function signature, where Line 1–8 reports f_{syn} (i.e., Line 1 to 8 is the hash value of the statement at Line 1 to 8 in Fig. 3(c)) and Line 10–13 reports f_{sem}.

3.3 Extracting Vulnerability and Patch Signatures

Given a pair of functions (f_v, p_v), and its patch P_v, this section explains how to generate the signatures to capture the key statements related to vulnerability rather than include all the statements in f and p. In this way, we have small but accurate signatures for effective matching.

3.3.1 Identifying Code Changes

We first identify the changed files by parsing the header of a security patch (i.e., the diff file), and record the commits from which the vulnerable and patched versions of the changed files are obtained. Then, to identify the changed functions, we locate the deleted and added statements and their line numbers by parsing the diff file, and get all functions and their start and end line numbers in the vulnerable and patched versions of the changed files. As mentioned in Definition 3.2, there are context lines, deleted lines and/or added lines in a patch. If a statement includes one or more deleted (resp. added) lines, we regard the statement as deleted (resp. added) statement. Therefore, a partly modified statement is a deleted statement, an added statement, or a deleted statement and an added statement. By checking whether the line numbers of deleted (or added) statements are in the range of the start and end line number of a function in the vulnerable (or patched) version of the changed files, we identify all changed functions; and for each of them, we also extract S_{del} and S_{add}, S_{vul} and S_{pat}.

For example, as shown by the header in Fig. 1, the only changed file is `wma.c`, and its vulnerable and patched version can be obtained from commit `0cb2ab8bd` and `cac414969`. The line numbers of the three added statements are in the range of the start and end line number of the function `WDA_TxPacket`; and there is no deleted statement. Therefore, `WDA_TxPacket` is the only changed function in this patch.

3.3.2 Computing Slices to Changed Code

Neither the changed statements alone (i.e., S_{del} and S_{add}) nor all the statements in changed functions (i.e., S_{vul} and S_{pat}) can precisely capture how a vulnerability is caused and fixed. S_{del} and S_{add} may miss some statements that are relevant to a vulnerability through data or control dependencies; and S_{vul} and S_{pat} may include noisy statements that are not related to a vulnerability. Intuitively, slicing techniques [59] can be used to include relevant statements and exclude irrelevant statements; i.e., we can perform forward and backward slicing on the PDG

```
1  *sockaddr_len = sizeof(struct sockaddr_atmpvc);
2   addr = (struct sockaddr_atmpvc *)sockaddr;
3+  memset(addr, 0, sizeof(*addr));
4   addr->sap_family = AF_ATMPVC;
5   addr->sap_addr.itf = vcc->dev->number;
6   addr->sap_addr.vpi = vcc->vpi;
```

Listing 1: The Patch for CVE-2012-6546

of f_v (resp. p_v), using the deleted statements S_{del} (resp. the added statements S_{add}) as the slicing criterion.

For example, we set the added statement at Line 18 (i.e, S_{18}[1]) in Fig. 1 as the slicing criterion. The result of backward slicing includes the S_6, S_{11} and S_{13}, because S_{18} is data dependent on S_6 and S_{11} and control dependent on S_{13}. The result of forward slicing includes the statements at Line 19–40 as these statements are controlled by S_{18} directly/indirectly.

As shown in the above example, when a conditional statement is set as the slicing criterion, the result of forward slicing could contain too many statements where some of them are noisy as they are not related to the vulnerability. In fact, it is common to add a sanitizing check (i.e., conditional statement) in a security patch. If we include in the result of forward slicing only the statements which the conditional statement controls directly (e.g., S_{23} and S_{24} in Fig. 1, which are directly affected by S_{18}), vulnerability-related statements (e.g., S_{28}) are not included, failing to capture the vulnerability. In summary, if we choose all statements affected by a conditional statement, we may introduce much noise; and if we choose statements directly affected by a conditional statement, we may fail to capture the vulnerability.

Moreover, a patch can be just adding a function call without using its return value. For example, Listing 1 shows the patch for CVE-2012-6546, where a call to the function `memset` is added at Line 3 to avoid information leak. If we do not model the function `memset`, we cannot know that its first parameter is changed. However, it is infeasible to specifically model all function calls. As a result, if a function call statement is set as the slicing criterion, we only have the backward slicing result, but get no forward slicing result. Therefore, we fail to capture the statements which are related to a vulnerability with a traditional slicing method.

To address previous problems, we propose a novel slicing method to better capture a vulnerability with less noise than traditional slicing methods. In detail, we set each statement in S_{del} (resp. S_{add}) as the slicing criterion, and (i) perform normal backward slicing on the PDG of f_v (resp. p_v), obtaining all statements that have influence on the slicing criterion with respect to data and control dependencies, and (ii) perform customized forward slicing on the PDG of f_v (resp. p_v) according to different statement types of the slicing criterion.

- *Assignment statement.* We conduct normal forward slicing as there must be data flow from the assignment statement. For example, if we take S_{23} in Fig. 1 as slicing criterion and perform forward slicing, S_{24} is included.

[1]For the ease of presentation, we represent S_i as the statement at Line i.

- *Conditional statement.* We aim to include in the result of forward slicing only the statements that use the variables or parameters checked in the conditional statement. To this end, 1) we conduct backward slicing on data dependencies in the PDG to obtain the direct source for each variable or parameter in the conditional statement; e.g., in Fig. 1, the direct source for the used local variable `vdev_id` at Line 18 is S_{11}; 2) we set each statement in the previous backward slicing result as the slicing criterion, and perform normal forward slicing on data dependencies; e.g., the result of forward slicing on S_{11} includes S_{18}, S_{19}, S_{28}, S_{29}, S_{32}, S_{33}, S_{34}, S_{35} and S_{37}; and 3) only if the previous forward slicing result is empty, we perform normal forward slicing on control dependencies.
- *Return statement.* No dependency exists between the return value and the statements after the return statement. Therefore, there is no need for forward slicing. For instance, there is no need to perform forward slicing on S_{20} in Fig. 1.
- *Others.* Other types include function call statements with its return value not used. Similar to conditional statements, we conduct forward slicing following the same first and second steps for conditional statements.

We put the statements in S_{del} (resp. S_{add}) and the statements in their backward and forward slicing results together, denoted as S^{sem}_{del} (resp. S^{sem}_{add}). S^{sem}_{del} (resp. S^{sem}_{add}) has the semantically-related statements of all deleted (resp. added) statements in a changed function in a security patch.

3.3.3 Generating Vulnerability and Patch Signatures

A target function can be regarded as potentially vulnerable if it matches the vulnerability signature (i.e., how the vulnerability is caused) and does not match the patch signature (i.e., how the vulnerability is fixed). In other words, vulnerability signature can be used to find potentially vulnerable functions, and then patch signature can be used to distinguish whether they are already patched or not. In this way, we can reduce false positives. Guided by the above principle, we compute the vulnerability signature (i.e., V_{syn} and V_{sem}) and patch signature (i.e., P_{syn} and P_{sem}) at the syntactic and semantic level as follows.

$$V_{syn} = S^{sem}_{del} \cup (S_{vul} \cap S^{sem}_{add}) \quad (1)$$

$$V_{sem} = \{(s_1, s_2, type) \mid s_1, s_2 \in V_{syn}\} \quad (2)$$

$$T_{sem} = \{(s_1, s_2, type) \mid s_1, s_2 \in S^{sem}_{add}\} \quad (3)$$

$$P_{syn} = S^{sem}_{add} \setminus S_{vul} \quad (4)$$

$$P_{sem} = T_{sem} \setminus F^{sem}_{vul} \quad (5)$$

$$F^{sem}_{vul} = \{(s_1, s_2, type) \mid s_1, s_2 \in S_{vul}\} \quad (6)$$

We compute V_{syn} by Eq. 1. S^{sem}_{del} is the statements that are related to deleted statements, thus it is directly related to how a vulnerability is caused. However, S^{sem}_{del} may not include all the vulnerable statements, especially when S_{del} is empty (and hence S^{sem}_{del} is empty; i.e., there are only added statements in P_v). Therefore, we need to further consider $S_{vul} \cap S^{sem}_{add}$, i.e., the vulnerable statements in S_{vul} which are identified by slices to added statements. Using V_{syn}, we compute V_{sem} by Eq. 2, where $(s_1, s_2, type)$ denotes a $type \in \{data, control\}$ dependency between two statements in V_{syn}. We compute P_{syn} by Eq. 4, which denotes statements that only exist in patched function p_v. We compute P_{sem} by Eq. 3, 5 and 6, which represents data or control dependencies between the two statement that only exist in patched function p_v. T_{sem} is relations between statements in S^{sem}_{add}. With P_{syn} and P_{sem}, we can tell vulnerable function and patched function apart.

We observe that the number of statements in V_{syn} varies for different patches. If the number of statements is very large, V_{syn} may introduce noise and result in false negatives. For example, after we set S_{18} in Fig. 1 as the slicing criterion in our slicing method in § 3.3.2, V_{syn} includes S_6, S_{11}, S_{13}, S_{28}, S_{29}, S_{32}, S_{33}, S_{34}, S_{35} and S_{37}. However, there are some noisy statements (e.g., S_{35} and S_{37}) in V_{syn}, because they are hardly related to the cause of the vulnerability.

Therefore, we try to remove such noisy statements as many as possible. Based on our finding from vulnerabilities, the further the distance between the deleted/added statements and the statements in V_{syn}, the smaller the correlation between the statements in V_{syn} and the cause of a vulnerability. Hence, we propose an information entropy-based vulnerable statement selection method; i.e., we apply information theory [57] to use the information in each statement in V_{syn} to refine V_{syn}.

Specifically, let the total number of statements be N and the number of a statement $s \in V_{syn}$ be n in the target system. Then, the probability of s's occurrence in the target system, denoted as p, is computed as $p = \frac{n}{N}$. Based on information theory [57], the amount of information of s, denoted as $I(s)$, is computed as $I(s) = -\log(p) = -\log(\frac{n}{N}) = \log(\frac{N}{n})$. As $\log(\frac{N}{n})$ is in the range of $(0, +\infty)$ and varies greatly, it is not easy to compare the information. As $\log(\frac{N}{n}) \propto \frac{1}{n}$, we use $\hat{I}(s) = \frac{1}{n}$ to approximate $I(s)$, making it in the range of $(0, 1]$. Then, we compute the information in V_{syn}, denoted as $\hat{I}$, as $\hat{I} = \sum_{s \in V_{syn}} \hat{I}(s)$.

If $\hat{I}$ is larger than a maximum threshold t^I_{max}, meaning that V_{syn} includes too many statements that might be noisy, we iteratively remove from V_{syn} statements which are farthest from the slicing criterion (i.e., S_{del} and/or S_{add}) on the PDG until $\hat{I}$ is less than t^I_{max} or all statements in V_{syn} are directly connected with the slicing criterion on the PDG. Correspondingly, any dependency that is relevant to the removed statements are excluded from V_{sem}.

Finally, for statements in S_{del}, V_{syn}, V_{sem}, P_{syn} and P_{sem}, we apply the same abstraction, normalization and hashing procedures as introduced in § 3.2 on them, and replace the statements with their corresponding hash values in S_{del}, V_{syn}, V_{sem}, P_{syn} and P_{sem} for the ease of matching in § 3.4.

Table 1: Statistics about Target Systems and Security Patches

Target System	Version	Line (#)	Function (#)	Domain	NVD (#)	Commit (#)	Total (#)	Changed Function (#)
Linux kernel	v4.18	18,298,218	435,734	Operating System Kernel	1,628	17,618	18,495	19,904
FreeBSD	12.0	7,460,955	140,163	Operation System Kernel	160	3,656	3,716	7,703
ImageMagick	7.0.8-27	461,843	4,229	Image Processing	79	628	704	915
OpenJPEG	2.3.0	245,113	4,390	Image Processing	17	137	142	309
LibTIFF	v4-0-9	82,985	1,413	Image Processing	46	175	193	343
Libarchive	v3.3.3	194,050	3,283	Compression	15	141	152	353
Libming	0.4.8	73,888	2,375	Flash Processing	17	39	53	147
Libav	12.3	607,326	11,277	Video Processing	80	763	813	1,467
Asterisk	16.6.0	995,874	19,202	Communication Toolkit	7	556	533	2,080
Qcacld-2.0	le.4.0.4	490,638	7,541	WLAN Driver	44	561	576	1,157
Total	–	28,910,890	629,607	–	2,093	24,274	25,377	34,378

3.4 Detecting Vulnerability through Matching

Given the function signature (f_{syn}, f_{sem}) of every function in a target system as well as the deleted statements S_{del}, the vulnerability signature V_{syn} and V_{sem}, and the patch signature P_{syn} and P_{sem} in each changed function in a patch, we determine whether a function in the target system is potentially vulnerable based on *the principle that its signature matches the vulnerability signature but does not match the patch signature*.

Specifically, a target function is potentially vulnerable if it satisfies the following five conditions, i.e., **C1** to **C5**.

- **C1.** The target function must contain all deleted statements, if any; i.e., $\forall h \in S_{del}, h \in f_{syn}$.
- **C2.** The signature of the target function matches the vulnerability signature at the syntactic level; i.e., $\frac{|V_{syn} \cap f_{syn}|}{|V_{syn}|} > t_1$.
- **C3.** The signature of the target function does not match the patch signature at the syntactic level; i.e., $\frac{|P_{syn} \cap f_{syn}|}{|P_{syn}|} \leq t_2$.
- **C4.** The signature of the target function matches the vulnerability signature at the semantic level; i.e., $\frac{|V_{sem} \cap f_{sem}|}{|V_{sem}|} > t_3$.
- **C5.** The signature of the target function does not match the patch signature at the semantic level; i.e., $\frac{|P_{sem} \cap f_{sem}|}{|P_{sem}|} \leq t_4$.

C1 is to ensure that the deleted statements, which are directly related to how a vulnerability is caused, are retained in the target function. $\frac{|V_{syn} \cap f_{syn}|}{|V_{syn}|}$ in **C2** and $\frac{|V_{sem} \cap f_{sem}|}{|V_{sem}|}$ in **C4** respectively measure the degree of signature matching between target function and vulnerable function at the syntax and semantic level. $\frac{|P_{syn} \cap f_{syn}|}{|P_{syn}|}$ in **C3** and $\frac{|P_{sem} \cap f_{sem}|}{|P_{sem}|}$ in **C5** respectively measure the degree of signature matching between target function and patched function at the syntax and semantic level. According to our principle, if we set the threshold in t_1 and t_3 to 1, and the threshold in t_2 and t_4 to 0, the matching constraint is very strict, which would cause many false negatives. According to our sensitivity analysis in § 4.4, we believe that 0.8 (resp. 0.2) are empirically good values for t_1 and t_3 (resp. t_2 and t_4). These thresholds can be configured by users.

4 Evaluation

4.1 Evaluation Setup

Research Questions. Our evaluation aims to answer the following research questions.

- **RQ1.** How is the accuracy of MVP in detecting recurring vulnerabilities compared to state-of-the-art approaches?
- **RQ2.** How is the scalability of MVP in detecting recurring vulnerabilities compared to state-of-the-art approaches?
- **RQ3.** How is the sensitivity of the thresholds configurable in the matching component of MVP?
- **RQ4.** How does the adoption of statement abstraction and statement information contribute to the accuracy of MVP?
- **RQ5.** How is the performance of general-purpose vulnerability detection on detecting recurring vulnerabilities?

Dataset. We chose target systems that satisfied the following criteria. First, they are C/C++ open-source projects since MVP is designed to search vulnerabilities in C/C++ source code. Second, they contain sufficient security patches so that we can detect whether the vulnerabilities fixed by these security patches recur in the corresponding target systems. Third, they cover diverse application domains so that the generality of our approach can be evaluated.

Using the three criteria, we chose ten open-source projects. Table 1 reports their statistics. The lines of code range from 73,888 to 18,298,218, while the number of functions ranges from 1,413 to 435,734, which are large enough to show the scalability of MVP. The application domain includes operating system kernel, image processing, compression, flash processing, video processing, communication toolkit and WLAN driver, which is diverse enough to show MVP's generality.

For each project, we collected its security patches from National Vulnerability Database (NVD) [4]. The number of collected patches is reported in the column *NVD* of Table 1. Moreover, as software companies may tend to patch the vulnerabilities secretly instead of applying for CVE [72], a large number of security patches hide in commit history. To enrich the dataset, we obtained the commits which contain secretly patched vulnerabilities from our industrial collaborator. The number of security commits is reported in the column *Commit* of Table 1. The column *Total* presents the total number of security patches after removing duplicated cases. The column *Changed Function* reports the number of functions that are changed in security patches. In total, we collected 25,377 security patches, which result in 34,378 changed functions.

State-of-the-Art Approaches. To evaluate the accuracy of MVP, we selected state-of-the-art approaches from two categories. First, we selected clone-based recurring vulnera-

bility detection approaches, ReDeBug [28] and VUDDY [34], because ReDeBug is a common baseline and VUDDY is the most effective work in this direction. Second, we picked function matching-based approaches, SourcererCC [56] and CCAligner [65]. While not designed for recurring vulnerability detection, they were compared with MVP to demonstrate the importance of considering vulnerability characteristics.

Besides, to evaluate the worthwhileness of MVP, we selected state-of-the-art general-purpose vulnerability detection approaches from two categories. First, we selected learning-based vulnerability detection approaches, VulDeePecker [41] and Devign [77], because they are the most effective work to detect potentially vulnerable functions by learning from vulnerable functions. Second, we selected widely-used commercial static analysis-based vulnerability detection tools, Coverity [2] and Checkmarx [1]. While these general-purpose vulnerability detection approaches target a different problem than recurring vulnerability detection, we included them to demonstrate their incapability in detecting recurring vulnerabilities and the worthwhileness of MVP.

Evaluation Configuration. We have implemented MVP in 6,500 lines of Python code. Our experiments were run on a machine with 2.40 GHz Intel Xeon processor and 32G RAM, running Ubuntu 14.04. All the state-of-the-art approaches compared in the experiments were configured with the same setting as reported in their original papers.

4.2 Accuracy Evaluation (RQ1)

We compare MVP with with state-of-the-art approaches on the selected ten projects. The following section discuss each of the comparison in detail. We have conducted evaluation on six more projects, and the results are listed in Appendix A. Here, we adopt two widely used metrics, positive predictive value (a.k.a. precision) and true positive rate (a.k.a. recall), to evaluate the accuracy of different approaches. Eq. 7 and 8 show the equations to compute precision and recall, where TP, FP and FN denote true positive, false positive and false negative, respectively.

$$Precision = \frac{TP}{TP+FP} \tag{7}$$

$$Recall = \frac{TP}{TP+FN} \tag{8}$$

4.2.1 Ground Truth

We evaluated the accuracy by comparing false positives and false negatives. However, it is impossible to enumerate all vulnerabilities in a project. To have a fair ground truth, we used all the vulnerabilities that were detected by MVP and the state-of-the-art approaches in our evaluation; i.e., we manually analyzed potentially vulnerable functions detected by each approach and confirmed whether they were true positives.

```
1  {
2      int64_t l;
3      int digit;
4  -
5  +
6  +   if (char_cnt == 0)
7  +       return (0);
8  +
9      l = 0;
10     while (char_cnt-- > 0) {
11         if (*p >= '0' && *p <= '7')
```

Listing 2: The Patch for CVE-2017-14166

4.2.2 Comparison with ReDeBug and VUDDY

We ran MVP, ReDeBug and VUDDY by using each security patch in a project as an input to search recurring vulnerabilities in the project itself. Table 2 shows the accuracy of the three approaches. The first and second columns present the name of each project and the number of vulnerabilities in the ground truth respectively. The rest columns show the accuracy measurement for each of the three approaches.

Overall Results. MVP detected 116 potentially vulnerable functions with a precision of 83.6%. It missed 14 vulnerabilities, having a recall of 87.4%. Significantly, MVP had no false positive in two projects and no false negative in three projects. On the other hand, ReDeBug and VUDDY respectively reported 549 and 301 potentially vulnerable functions with a precision of 9.1% and 8.0%. Moreover, ReDeBug and VUDDY achieved a low recall of 45.0% and 21.7%. In summary, MVP significantly outperformed ReDeBug and VUDDY with respect to precision and recall in detecting recurring vulnerabilities; i.e., it improved precision of ReDeBug and VUDDY by 74.5% and 75.6%, while improving recall by 42.4% and 65.8%.

False Positive Analysis for MVP. We analyzed all false positives in MVP, and summarized three reasons. First, calling context is missing as we do not use inter-procedure analysis when we extract signatures at the semantic level. It caused 9 false positives. This is also one of the reasons for false positives in ReDeBug and VUDDY. For example, Listing 2 shows the patch for CVE-2017-14166, which is a heap-based buffer over-read vulnerability since the parameter `char_cnt` of function `atol8` in file `archive_read_support_format_xar.c` can be zero. Thus, the patch adds a check for `char_cnt`. MVP discovered a potentially vulnerable function `atol8` in file `archive_write_add_filter_uuencode.c`; but its parameter `char_cnt` cannot be zero because there exists a check before `atol8` is called. Inter-procedure analysis can be helpful but may hinder the scalability of MVP.

Second, semantic equivalence is not modeled; i.e., there can be different semantic-equivalent patches to fix a vulnerability, and thus we may falsely identify a target function as vulnerable when the target function contains a semantic-equivalent patch. It caused 4 false positives. For example, the patch in Listing 3 fixed kernel information leakage (i.e., CVE-2018-17155) in function `freebsd32_swapcontext`; i.e., it added a call to `bzero` to initialize variable `uc` with zeros before the

Table 2: Accuracy (i.e., True Positive, False Positive and False Negative) of ReDeBug, VUDDY and MVP

Target System	GT (#)	ReDeBug					VUDDY					MVP				
		TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
Linux kernel	32	12	286	20	4.0%	37.5%	9	49	23	15.5%	28.1%	25	6	7	80.6%	78.1%
FreeBSD	11	7	86	4	7.5%	63.6%	2	29	9	6.5%	18.2%	11	2	0	84.6%	100.0%
ImageMagick	16	7	14	9	33.3%	43.7%	0	5	16	0.0%	0.0%	14	2	2	87.5%	87.5%
OpenJPEG	16	10	7	6	58.8%	62.5%	2	1	14	66.7%	12.5%	16	1	0	94.1%	100.0%
LibTIFF	8	6	11	2	35.3%	75.0%	4	4	4	50.0%	50.0%	6	0	2	100.0%	75.0%
Libarchive	5	1	6	4	14.3%	20.0%	1	3	4	25.0%	20.0%	5	3	0	62.5%	100.0%
Libming	3	0	5	3	0.0%	0.0%	1	3	2	25.0%	33.3%	2	0	1	100.0%	66.7%
Libav	6	2	10	4	16.7%	33.3%	2	12	4	14.3%	33.3%	6	1	0	86.7%	100.0%
Asterisk	7	4	30	3	11.8%	57.1%	3	20	4	13.0%	42.9%	5	1	2	83.3%	71.4%
Qcacld-2.0	7	1	44	6	2.2%	14.3%	0	151	7	0%	0.0%	7	3	0	70.0%	100.0%
Total	111	50	499	61	9.1%	45.0%	24	277	87	8.0%	21.6%	97	19	14	83.6%	87.4%

```
1     if (uap->ucp == NULL)
2         ret = EINVAL;
3     else {
4+        bzero(&uc, sizeof(uc));
5         ia32_get_mcontext(td, &uc.uc_mcontext,
      GET_MC_CLEAR_RET);
6         PROC_LOCK(td->td_proc);
7         uc.uc_sigmask = td->td_sigmask;
```

Listing 3: The Patch for CVE-2018-17155

```
1  int freebsd32_getcontext(struct thread *td, struct
       freebsd32_getcontext_args *uap) {
2      ...
3      else {
4          memset(&uc, 0, sizeof(uc));
5          get_mcontext32(td, &uc.uc_mcontext,
       GET_MC_CLEAR_RET);
6          PROC_LOCK(td->td_proc);
7          ...
8      }
9      return (ret);
10 }
```

Listing 4: A Falsely Identified Vulnerable Function

data field of `uc` is assigned. The function in Listing 4 is detected as potentially vulnerable as it does not call `bzero`, but it is a false positive as it calls `memset` to initialize `uc` with zeros.

Third, extracted vulnerability or patch signature is not able to capture the characteristics of a vulnerability due to the various root causes of vulnerabilities. This caused 6 false positives. For example, `BUG_ON(!vreg)` at Line 7 in Listing 5, only put after Line 2 and 3, causes a vulnerability. MVP cannot include it into the vulnerable signature as it does not have any data/control dependency on the deleted statements at Line 2 and 3. Hence, the function in Listing 6 is falsely detected as it does not put `BUG_ON(!vreg)` after Line 4 and 5.

False Positive Analysis for ReDeBug and VUDDY. We analyzed all the false positives in ReDeBug and VUDDY. For ReDeBug, apart from missing calling context (leading to 2

```
1      int ret = 0;
2-     struct regulator *reg = vreg->reg;
3-     const char *name = vreg->name;
4+     struct regulator *reg;
5+     const char *name;
6      int min_uV, uA_load;
7      BUG_ON(!vreg);
8+     reg = vreg->reg;
9+     name = vreg->name;
10     if (regulator_count_voltages(reg) > 0) {
11         min_uV = on ? vreg->min_uV : 0;
12         ret = regulator_set_voltage(reg, min_uV, vreg->
       max_uV);
```

Listing 5: A Patch for FreeBSD

```
1  static int ufs_qcom_phy_cfg_vreg(struct device *dev,
       struct ufs_qcom_phy_vreg *vreg, bool on)
2  {
3      int ret = 0;
4      struct regulator *reg = vreg->reg;
5      const char *name = vreg->name;
6      ...
7      if (regulator_count_voltages(reg) > 0) {
8          ...
9      }
10 }
```

Listing 6: A Falsely Identified Vulnerable Function

```
1  @@ -2416,8 +2416,6 @@ static void nfsrvd_mkdirsub(
2      if (!nd->nd_repstat)
3          nd->nd_repstat = nfsrv_lockctrl(vp, &stp, &lop, &
       cf, clientid, &stateid, exp, nd, p);
4-     if (stp)
5-         FREE((caddr_t)stp, M_NFSDSTATE);
6      if (nd->nd_repstat) {
7          if (nd->nd_repstat == NFSERR_DENIED) {
8              NFSM_BUILD(tl, u_int32_t *, 7 * NFSX_UNSIGNED);
9  @@ -2439,6 +2437,8 @@ static void nfsrvd_mkdirsub(
10         }
11     }
12     vput(vp);
13+    if (stp)
14+        FREE((caddr_t)stp, M_NFSDSTATE);
15     NFSEXITCODE2(0, nd);
16     return (0);
17 nfsmout:
```

Listing 7: A Patch for Use After Free in FreeBSD

false positives), there are three major reasons. First, ReDeBug leverages each of the hunks in a changed function separately to match potentially vulnerable functions, and thus it suffers local matching, especially when the hunk has changes only to blank line, comment, header information, macro, or struct. It caused 421 false positives. For example, Listing 7 shows a patch for a use after free vulnerability in FreeBSD; i.e., variable `stp` is accessed after function `FREE` is called at Line 5, and the patch was to move the call to `FREE` after `stp` is accessed. The patch involves two hunks, i.e., one deletes the call to `FREE` and the other adds the call to `FREE`. ReDeBug may detect a target function matching the second hunk, causing false positives as `stp` is not accessed.

Second, ReDeBug uses a sliding window (of 4 lines of code by default) to match potentially vulnerable functions. As a result, when the last few added statements are the same to the statements before the added ones, the patched function might still be detected as potentially vulnerable. It caused 52 false positives. The example discussed in § 2.2 is such a case.

Third, ReDeBug does not use the semantics information,

```
1 @@ -2244,8 +2246,8 @@ set_regs(struct thread *td, struct
      reg *regs)
2       tp->tf_fs = regs->r_fs;
3       tp->tf_gs = regs->r_gs;
4       tp->tf_flags = TF_HASSEGS;
5-      set_pcb_flags(td->td_pcb, PCB_FULL_IRET);
6     }
7+    set_pcb_flags(td->td_pcb, PCB_FULL_IRET);
8     return (0);
9 }
```

Listing 8: The Patch for CVE-2014-4699

```
1 int cifs_close(struct inode *inode, struct file *file) {
2-    cifsFileInfo_put(file->private_data);
3-    file->private_data = NULL;
4+    if (file->private_data != NULL) {
5+       cifsFileInfo_put(file->private_data);
6+       file->private_data = NULL;
7+    }
8     return 0;
9 }
```

Listing 9: The Patch for CVE-2011-1771

which caused 24 false positives. For example, the statement at Line 5 was moved out of a conditional statement in the patch in Listing 8, changing the control dependency. However, ReDeBug cannot distinguish the vulnerable and patched functions. As we work at the function level and consider semantics information, we effectively prevent such false positives.

For VUDDY, apart from missing calling context (causing 8 false positives), another major reason is abstraction, which replaces formal parameters, local variables, data types and function calls with specific symbols. When some vulnerabilities are patched by only these abstracted items, VUDDY fails to distinguish the vulnerable and patched functions as they have the same hash value after abstraction. Besides, the hash values of unrelated functions might collide due to over-abstraction in VUDDY. These caused 269 false positives. For example, the patch in Listing 9 patched a null pointer dereference, caused by a missing null check for `file->private_data` before it was passed to `cifsFileInfo_put`. A target function in Listing 10 has the same hash value to this vulnerable function due to over-abstraction, but `kfree` can receive a null argument. However, VUDDY detects the target function as vulnerable. We do not apply abstraction on function calls and data types while using semantics to reduce such false positives.

False Negative Analysis. We analyzed all the false negatives in each approach. For MVP, it missed 3, 3 and 8 vulnerabilities, which were respectively detected by ReDeBug, VUDDY and our threshold sensitivity analysis in § 4.4. The reason is that MVP does not work at the hunk level but at the function level, which can bring noise into extracted signatures. Moreover, it does not apply abstraction on data types and function calls so that the signature is not generalized enough to capture some vulnerable cases. However, there is a trade-off between precision and recall. Our approach tries to maximize the precision, while maintaining a reasonably high recall.

For ReDeBug, it applies exact matching and does not apply abstraction. As a result, some renamed variables or parameters

```
1 static int ubifs_dir_release(struct inode *dir, struct
     file *file)
2 {
3    kfree(file->private_data);
4    file->private_data = NULL;
5    return 0;
6 }
```

Listing 10: A Falsely Identified Vulnerable Function

Table 3: Distribution of Vulnerabilities Detected by Different Approaches w.r.t. Similarity to Vulnerable Functions

Approach	10%*	20%	30%	40%	50%	60%	70%	80%	90%	100%
MVP	2	4	7	8	5	10	14	13	26	8
ReDeBug	0	1	5	2	1	3	3	11	16	8
VUDDY	0	0	0	0	0	0	2	3	13	6
SourcererCC	0	0	0	0	0	0	16	18	30	8
CCAligner	0	1	2	1	1	3	6	14	29	6
VulDeePecker	0	0	1	0	0	1	1	0	5	0
Devign	0	0	0	2	4	4	4	6	16	4
Coverity	0	0	0	1	0	2	0	0	0	1
Checkmarx	0	0	0	0	0	0	0	0	0	0
Ground Truth	2	4	7	8	8	10	16	18	30	8

* x% denotes the similarity score between vulnerable function and its corresponding matched target function.

may make exact matching fail, which caused 11 false negatives in ReDeBug. Besides, the context information around the deleted/added statements may be not related to the cause of a vulnerability, which caused 42 false negatives. Moreover, if the number of lines in a hunk is less than the sliding window size after blank lines, comments and braces are removed, ReDeBug will directly skip the hunk, leading to 8 false negatives. For VUDDY, it also uses exact matching. Thus, it may miss target functions with vulnerability-irrelevant differences from a vulnerable function. This caused all the 87 false negatives in VUDDY. Instead, we adopt partial matching and program slicing to reduce such false negatives.

4.2.3 Comparison with SourcererCC and CCAligner

We compared MVP against SourcererCC and CCAligner to indicate that code clone detection alone without considering any vulnerability characteristics is not suitable for recurring vulnerability detection. We used the vulnerable functions (derived from security patches in the column *Total* in Table 1) in each project as the input for SourcererCC and CCAligner, and considered the function clones to these vulnerable functions in each project as potentially vulnerable functions.

In total, SourcererCC and CCAligner respectively detected 15,555 and 23,889 function clones. Since all security patches we collected have been applied to the target systems used in our experiment, all function clones which have the same fully qualified name to vulnerable functions are actually patched functions and thus are false positives. After removing such false positives, we still had 2,982 and 10,033 function clones. Due to the large manual effort to analyze all these functions clones, we randomly sampled 15% of them to perform manual analysis. Our results show that the precision for SourcererCC and CCAligner is respectively 0.5% and 0.3%. Besides, we computed recall by checking whether those 111 vulnerable functions are in their function clones. It turns out that Sour-

Table 4: Performance Overhead of ReDeBug, VUDDY and MVP

Target System	ReDeBug			VUDDY			MVP		
	System Ana.	Patch Ana.	Matching	System Ana.	Patch Ana.	Matching	System Ana.	Patch Ana.	Matching
Linux kernel	1,883 s	0.68 ms	0.01 ms	6,974 s	3,846.17 ms	83.10 ms	37,545 s	7,178.31 ms	89.43 ms
FreeBSD	1,008 s	0.94 ms	0.03 ms	6,868 s	4,966.36 ms	63.24 ms	14,868 s	25,266.15 ms	63.24 ms
ImageMagick	35 s	1.27 ms	0.01 ms	221 s	7,228.69 ms	8.52 ms	595 s	20,859.38 ms	1.42 ms
OpenJPEG	11 s	1.40 ms	0.01 ms	251 s	5,697.18 ms	84.51 ms	574 s	15,640.85 ms	7.04 ms
LibTIFF	7 s	3.62 ms	0.01 ms	53 s	6,036.26 ms	108.81 ms	136 s	14,036.27 ms	0.51 ms
Libarchive	20 s	1.31 ms	0.01 ms	121 s	5,263.15 ms	39.47 ms	335 s	17,381.58 ms	1.97 ms
Libming	9 s	3.77 ms	0.01 ms	47 s	3,981.13 ms	113.21 ms	191 s	18,396.23 ms	1.89 ms
Libav	41.4 s	1.11 ms	0.01 ms	206 s	3,569.50 ms	29.52 ms	361 s	11,149.51 ms	2.21 ms
Asterisk	45.5 s	3.94 ms	0.01 ms	156 s	7,335.83 ms	125.70 ms	514 s	26,109.18 ms	6.00 ms
Qcacld-2.0	26 s	1.04 ms	0.01 ms	57 s	5,499.53 ms	517.36 ms	253 s	21,019.81 ms	3.04 ms

ererCC and CCAligner had a recall of 64.9% and 56.8%. Thus, MVP outperformed SourcererCC and CCAligner by improving precision by 83.1% and 83.3%, and recall by 22.5% and 30.6%.

4.2.4 Similarity of Vulnerable and Target Function

We measured the similarity score between the target functions detected as truly vulnerable by all the approaches in § 4.2 and their matched vulnerable functions (i.e., $Sim(V,T)$ as introduced in § 2.1). The results in Table 3 show that MVP can detect recurring vulnerabilities no matter T is similar or not similar to V, while other approaches tend to find recurring vulnerabilities only when T is similar to V.

4.3 Scalability Evaluation (RQ2)

To evaluate the scalability of MVP, we compared it against the two clone-based vulnerability detection approaches (i.e., ReDeBug and VUDDY) because they are the most closely related approaches to MVP which share similar processes. For the other clone detectors, as they work differently from MVP, we believe it is not fair to compare with them.

MVP, ReDeBug and VUDDY are all composed of three basic components: *system analysis* to extract the information of each target function in a target system, *patch analysis* to generate the signature of a vulnerability, and *matching* to search for vulnerability in target functions. As VUDDY only released the source code of the system analysis component, we had to implement the other two components based on their paper.

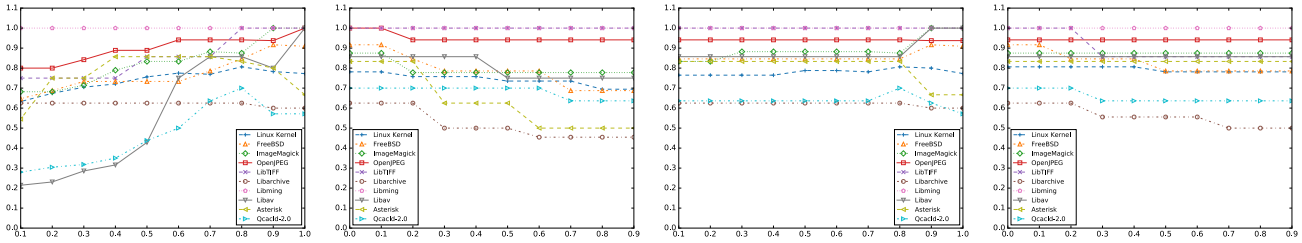
Table 4 reports the performance overhead of each component in each approach. Overall, the time consumption of the system analysis is approximately proportional to the size of the project. As VUDDY is syntax-based (i.e., it needs to use a parser to analysis the source code), it spent 8.4 times as much as ReDeBug (which is token-based) did in system analysis. Because MVP is semantics-based (i.e., it performs program analysis), it spent 3.2 times as much as VUDDY did in system analysis; e.g., MVP spent 10.4 hours for Linux kernel. However, system analysis is a one-time job, and its results can be reused for different security patches. In addition, we might reduce the time by using a demand-driven semantic analysis as most target functions do not match a vulnerable function at the syntactical level and thus semantics matching is not needed. Similar to system analysis, ReDeBug spent the least time in patch analysis. MVP took 3.3 times as much as VUDDY did in patch analysis. On average, MVP took 17,272.82 milliseconds to extract the signature of a vulnerability from a security patch. For matching, all three approaches were fast.

In summary, MVP is slower than ReDeBug and VUDDY, but it still scales to large systems. However, MVP has much higher precision, which significantly reduces the time used to manually validate potentially vulnerable functions. We believe MVP can save more time as manual validation often consumes most of the time in vulnerabilities analysis.

4.4 Threshold Sensitivity Analysis (RQ3)

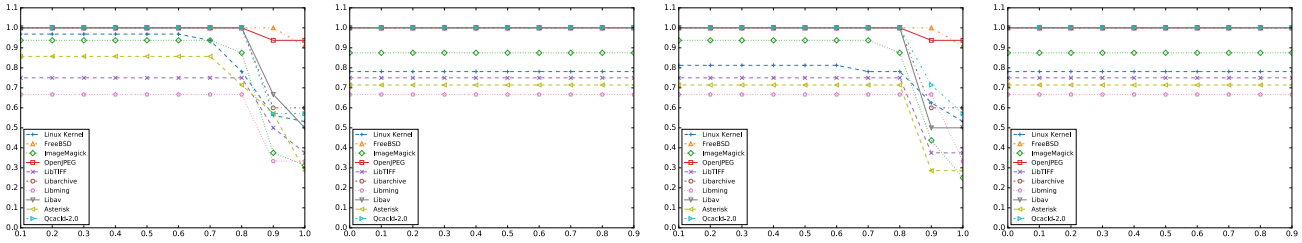
Four thresholds (i.e., t_1, t_2, t_3, t_4) are configurable in the matching step of MVP (§ 3.4). The default configuration is 0.8, 0.2, 0.8 and 0.2, which is used in the experiment in § 4.2. To evaluate the sensitivity of these thresholds to the accuracy of MVP, we reconfigured one threshold and fixed the other three, and ran MVP against the ten target systems. As t_1 and t_3 are used to determine whether a function signature matches the vulnerability signature, they were reconfigured from 0.1 to 1.0 by a step of 0.1. As t_2 and t_4 are adopted to determine whether a function signature does not match the patch signature, they were reconfigured from 0.0 to 0.9 by a step of 0.1. In total, 35 (i.e., 4×9 - 1) configurations of MVP were run. We analyzed their detection results, and found 8 vulnerabilities that were not detected by the default configuration in § 4.2.

Fig. 4 and 5 present the impact of four thresholds on precision and recall, respectively, where x-axis denotes the value of threshold, and y-axis denotes the precision or recall. Overall, before t_3 increased to 0.7, the precision and recall was almost stable in most target systems. As t_3 increased from 0.7 to 1.0, the precision increased and recall decreased. As t_1 increased from 0.1 to 0.8, the precision increased. Specifically, when t_1 and t_3 were configured to 0.9, the recall greatly decreased since many true positives were missed due to the strict matching condition. Thus, we believe 0.8 is a good value for t_1 and t_3. On the other hand, as t_2 increased from 0.0 to 0.2, the precision were changed slightly in most target systems. As t_2 and t_4 increased from 0.2 to 0.9, the precision decreased. Hence, we believe 0.2 is a good value for t_2 and t_4.



(a) t_1 (b) t_2 (c) t_3 (d) t_4

Figure 4: Precision vs. Four Threshold (x-axis denotes the value of threshold, and y-axis denotes precision)



(a) t_1 (b) t_2 (c) t_3 (d) t_4

Figure 5: Recall vs. Four Thresholds (x-axis denotes the value of threshold, and y-axis denotes Recall)

Table 5: Comparison of Threshold t^I_{max}

t^I_{max}	TP	FP	FN	Precision	Recall
None*	75	39	36	65.8%	67.6%
4	91	18	20	83.5%	82.0%
5	97	19	14	83.6%	87.4%
6	93	20	18	82.3%	83.8%
7	87	25	24	77.7%	78.4%
8	84	30	27	73.7%	75.7%

* It denotes disabling the adoption of statement information.

4.5 Contribution of Statement Abstraction and Statement Information (RQ4)

Contribution of Statement Abstraction. MVP uses statement abstraction in generating function signature (§ 3.2). We ran MVP by removing statement abstraction and analyzed the detected potentially vulnerable functions. The only difference was 11 more false negatives (i.e., 3 in Linux kernel, 5 in ImageMagick and 3 in Qcacld-2.0). Thus, statement abstraction improves MVP by detecting 12.8% more vulnerabilities.

Contribution of Statement Information. MVP adopts the information of statements in extracting the vulnerability signature (§ 3.3.3). To evaluate how the adoption of statement information contributes to the accuracy of MVP, and how the threshold t^I_{max} impacts the accuracy of MVP, we configured MVP by disabling the adoption of statement information, and changing t^I_{max} from 4 to 8, respectively. Then, we ran these six configurations of MVP against the ten target systems. We analyzed all the potentially vulnerable functions detected by them. Table 5 reports the true positives, false positives and false negatives of these configurations. The results indicate that i) the adoption of statement information improves the accuracy, and ii) 5 is empirically established as a good value for t^I_{max}.

4.6 Performance of General-Purpose Vulnerability Detection (RQ5)

4.6.1 Performance of VulDeePecker and Devign

We directly used VulDeePecker's and Devign's model which had been trained on their individual training dataset, and used the ground truth as the testing dataset to determine whether they can find any of 111 recurring vulnerabilities in the ground truth. It is worth mentioning that we did not use all the functions in the ten projects as the testing dataset. The reason is that the detected potentially vulnerable functions are not explainable; i.e., we do not know which vulnerability they are similar to, making it difficult and time-consuming to determine whether they are truly vulnerable or not.

The results show that VulDeePecker only detected 8 of the 111 vulnerabilities, having a recall of 7.2%; and Devign found 40 of them, achieving a recall of 36.0%. One main reason for high false negatives in VulDeePecker is that it can only handle functions that call specific library functions or APIs such as `strcpy`. As for Devign, it can only process a function whose nodes are less than 500 in AST, CFG or PDG. Besides, their training data may not include all security patches used in our experiment. Both approaches take vulnerable and non-vulnerable functions in Linux kernel as the training data. Nevertheless, VulDeePecker cannot detect any of the 32 recurring vulnerabilities in Linux kernel (i.e., a recall of 0%), while Devign can detect 19 of them (i.e., a recall of 59.3%). Thus, learning-based approaches may not be effective in discovering recurring vulnerabilities, and MVP is worthwhile.

4.6.2 Comparison with Coverity and Checkmarx

We ran Coverity and Checkmarx against the ten projects, and analyzed whether they could detect any of the 111 recurring

vulnerabilities in the ground truth. Not surprisingly, Coverity only detected four of them, i.e., one in FreeBSD, one in ImageMagick, one in OpenJPEG, and one in Libarchive, while Checkmarx cannot detect any of them. These results show that static scanners might not be effective in discovering recurring vulnerabilities, and MVP is worthwhile.

4.7 Limitations

MVP has a few underlying assumptions, which may limit its application. First, we focus on detecting recurring vulnerabilities which are Type-1, Type-2 and Type-3 clones [53]. In MVP, a target function is regarded as potentially vulnerable only if its function signature matches the vulnerability signature and does not match the patch signature at both syntactic and semantic level. As Type-4 is syntactically different, MVP cannot handle Type-4, which requires more semantic-based techniques like symbolic execution or dynamic analysis. However, this may significantly affect the scalability. ReDeBug handles Type-1, Type 2 and Type 3, while VUDDY handles Type-1 and Type-2.

Second, MVP uses Joern [74] to generate code property graph. We assume it is correct. Traditional approaches for generating data/control dependency graph need a working build environment to compile the project. For each pair of vulnerable and patched functions, we need to compile the whole project, which is time-consuming. To make MVP sufficiently general in practice, we use Joern, which can generate a combination of DDG, CDG and AST without compilation and support partial code. Experimental results demonstrate Joern can give good performance and acceptable precision.

Third, we cannot detect vulnerabilities whose patches are out of functions. Some vulnerabilities are fixed by only changing struct or macro, which are out of functions. MVP takes a function as an basic unit, therefore, any changes out of functions cannot be handled by MVP.

Besides, our accuracy evaluation has revealed some root causes that are not well handled. On the one hand, as discussed in § 4.2.2, there are three reasons for false positives in MVP: missing calling context, semantic equivalence, and improper signature extraction. For missing calling context, it can be addressed by inter-procedure analysis. However, as the analysis introduces additional time cost, the trade-off needs to be carefully explored. For semantic equivalence, we can adopt the methods proposed in [43], i.e., renaming variables, rewriting expressions, and rearranging control structures. However, they can only solve the problem partially because they cannot cover all types of the semantic equivalent code snippets. For improper signature extraction, one major reason is that we do not expand macros in functions. We can take the advantage of mature compilers to compile source files to expand macros. However, to expand macros for one function, compilers need to compile the whole project, which can be time-consuming. Therefore, the cost and benefit should be investigated.

On the other hand, as we only apply abstraction on formal parameters, local variables and strings, we cannot discover potentially vulnerable functions that have similar function calls or data types to the known vulnerable functions. Actually, we could take a two-step way to detect such potentially vulnerable functions without introducing many false positives. First, we tokenize each token in the source code of all the target systems, and learn a vector representation of each token through word embedding. Then for each statement, we apply abstraction, i.e., replacing formal parameters, local variables, strings, function calls and data types with symbols, followed by normalization and hashing. Thus, during matching, we can first match a target function's signature with the vulnerability signature using hashing values; if they match, we then compute the similarity of the two statements that have the same hashing value after replacing each token in the two statements with its vector representation. If the similarity is higher than a threshold, we treat the two statements as similar, and continue with other matching procedures in MVP to determine whether the target function is potentially vulnerable. In this way, we can improve MVP to detect the vulnerable functions that were detected by VUDDY in the experiment.

5 Related Work

We review the most closely related work in four aspects, i.e., code clone detection, clone-based vulnerability detection, learning-based vulnerability detection, and binary-level vulnerability detection.

Code Clone Detection. To detect code clones in four different types[8, 51, 54], a variety of techniques have been proposed. Some techniques focus on Type-I and Type-II clones (e.g., [26, 27, 29, 32, 37, 55, 68]), some techniques are designed to further detect Type-III clones (e.g., [7, 12, 14, 23, 36, 56, 65]), and some techniques are designed to detect Type-IV clones (e.g., [21, 30, 33, 35, 58, 66, 76]). These techniques are specifically designed to detect general code clones with high accuracy and scalability, but they do not target for accurately finding vulnerable code clones because vulnerabilities are often very subtle and context-sensitive.

Clone-Based Vulnerability Detection. Many approaches have been proposed to use clone detection to find vulnerable or buggy code clones. Zhou et al. [42] proposed CP-Miner, to detect bugs that caused by inconsistent identifier renaming in code clones. Jiang et al. [31] and Gabel et al. [22] proposed enhancement to CP-Miner. These approaches rely on inconsistencies in code clones to detect bugs. Differently, MVP aims to detect vulnerabilities with similar characteristics.

Nam et al. [50] proposed SecureSync to detect recurring vulnerabilities due to code and library reusing. It adopts extended abstract syntax trees and graph-based usage models to represent code fragments. Li and Ernst [38] developed a semantics-based buggy code clone detection approach CBCD. It generates the program dependence graph for buggy codes.

Then, it uses sub-graph isomorphism matching to discover potentially buggy code. Zou et al. [78] proposed SCVD to detect vulnerable code clones. It generates a program feature tree. Then, it utilizes tree-based matching to detect vulnerable code clones. Jang et al. [28] proposed a token-based approach ReDeBug to find unpatched code clones. It extracts vulnerable code files, tokenizes each of them by a sliding window of n lines of code. It applies k hash functions on all tokens and detects code clones via comparing the hashed tokens with the targets. Kim et al. [34] proposed VUDDY to detect vulnerable code clones. It applies abstraction and then hashes the normalized code body to generate fingerprints for each function. Finally, it matches for the fingerprint in the target system to detect vulnerable code clones. VUDDY is designed to discover Type-I and Type-II clones.

These approaches [28, 34, 38, 50, 78] share the same goal with ours. However, as they build vulnerability signature either broadly (e.g., from a whole function) or locally (e.g., from only several lines of code), they fail to capture the precise context of the vulnerability, which results in low accuracy. Instead, we leverage slicing to extract precise signatures from both vulnerable and patched functions to improve accuracy. Besides, signature-based clone detection approaches [28, 34] do not take patch information into consideration. They fail to differentiate the potentially small differences between vulnerable function and patched function. Instead, we not only generate a vulnerability signature but also a patch signature to capture how a vulnerability is caused and fixed.

Learning-Based Vulnerability Detection. Li et al. [40] designed VulPecker to detect vulnerabilities using code similarity analysis. It utilizes a set of features to characterize patches, and trains a model to select one of the existing code similarity analysis algorithms (e.g., [28, 38, 50]) for a specific patch. Li et al. [41] also proposed VulDeePecker to detect vulnerabilities using deep learning. For each program, it computes slices of library/API function calls, assembles them into code gadgets, and transforms code gadgets into vectors. Then, it trains a neural network model to decide whether a target program contains vulnerable code gadgets or not. Similarly, Lin et al. [44] developed a deep learning-based framework for detecting potentially vulnerable functions. Different from VulDeePecker, it encodes code fragments at the level of abstract syntax trees but not slices about functional calls. These learning-based approaches [40, 41, 44] target for general-purpose vulnerability detection, which is a different problem than recurring vulnerability detection.

Binary Level Vulnerability Detection. Several methods have proposed to detect vulnerabilities via binary-level matching techniques (e.g., [16, 18, 19, 20, 48, 49, 71]). These methods are at the binary level, which have different features and face different challenges. Therefore, we did not compare MVP with them.

6 Conclusions

In this paper, we have proposed and implemented a novel approach, named MVP, to discover recurring vulnerabilities with both low false positives and low false negatives. Our evaluation results have demonstrated that, MVP can significantly outperform the state-of-the-art recurring vulnerability detection approaches, and has detected 97 new vulnerabilities with 23 CVE identifiers assigned.

Acknowledgments

We thank our shepherd Giancarlo Pellegrino and anonymous reviewers for their comprehensive feedback. This research was supported (in part) by the National Natural Science Foundation of China (Grant No. 61602470, U1836209, 61802394, 61802067), National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No.NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Award No.NRF2018NCR-NSOE003-0001), and BINSEC: Binary Analysis For Security (Award No. NRF2016NCR-NCR002-026).

References

[1] Checkmarx. https://www.checkmarx.com.

[2] Coverity. https://scan.coverity.com.

[3] Format string. http://www.cplusplus.com/reference/cstdio/printf/.

[4] National vulnerability database. https://nvd.nist.gov/.

[5] Patch for cve-2018-7186. https://github.com/DanBloomberg/leptonica/commit/ee301cb2029db8a6289c5295daa42bba7715e99a.

[6] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 12–22, 2011.

[7] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998.

[8] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[10] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 725–741, 2015.

[11] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108, 2018.

[12] Yan Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186, 2014.

[13] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium*, pages 1253–1270, 2017.

[14] James R. Cordy and Chanchal Kumar Roy. The nicad clone detector. In *Proceedings of the IEEE 19th International Conference on Program Comprehension*, pages 219–220, 2011.

[15] Weidong Cui, Marcus Peinado, Helen J Wang, and Michael E Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 252–266, 2007.

[16] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[17] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering*, page 60–71, 2019.

[18] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 303–317, 2014.

[19] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, 2016.

[20] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

[21] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, pages 321–330, 2008.

[22] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 175–190, 2010.

[23] Nils Göde and Rainer Koschke. Incremental clone detection. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pages 219–228, 2009.

[24] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, 2008.

[25] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 618–635, 2016.

[26] Benjamin Hummel, Elmar Jürgens, Lars Heinemann, and Michael Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 1–9, 2010.

[27] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects. In *Proceedings of the 19th Working Conference on Reverse Engineering*, pages 387–391, 2012.

[28] Jiyong Jang, Maverick Woo, and David Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 48–62, 2012.

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

[30] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 81–92, 2009.

[31] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 55–64, 2007.

[32] 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.

[33] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwangkeun Yi. Mecc: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 301–310, 2011.

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

[35] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the International Static Analysis Symposium*, pages 40–56, 2001.

[36] Rainer Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process*, 26(8):747–769, 2014.

[37] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, 2006.

[38] Jingyue Li and Michael D Ernst. Cbcd: Cloned buggy code detector. In *Proceedings of the 34th International Conference on Software Engineering*, pages 310–320, 2012.

[39] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 533–544, 2019.

[40] 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*, pages 201–213, 2016.

[41] 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*, 2018.

[42] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[43] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. Antminer: mining more bugs by reducing noise interference. In *Proceedings of the 38th International Conference on Software Engineering*, pages 333–344, 2016.

[44] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics*, 14(7):3289–3297, 2018.

[45] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, pages 271–286, 2005.

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

[47] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102, 2009.

[48] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 709–724, 2015.

[49] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 406–415, 2014.

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

[51] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[52] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, 2017.

[53] 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.

[54] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[55] Hitesh Sajnani, Vaibhav Saini, and Cristina Lopes. A parallel and efficient approach to large scale clone detection. *Journal of Software: Evolution and Process*, 27(6):402–429, 2015.

[56] Hitesh Sajnani, Vaibhav Pratap Singh Saini, Jeffrey Svajlenko, Chanchal Kumar Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[57] Claude Elwood Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

[58] Abdullah Sheneamer and Jugal Kalita. Semantic clone detection using machine learning. In *Proceedings of the 15th IEEE International Conference on Machine Learning and Applications*, pages 1024–1028, 2016.

[59] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3):12, 2012.

[60] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, number 2016.

[61] Julien Vanegue and Shuvendu K Lahiri. Towards practical reactive security audit using extended static checkers. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 33–47, 2013.

[62] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, pages 257–267, 2000.

[63] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[64] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 579–594, 2017.

[65] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal Kumar Roy. Ccaligner: A token based large-gap clone detector. *Proceedings of IEEE/ACM 40th International Conference on Software Engineering*, pages 1066–1077, 2018.

[66] Huihui Wei and Ming Li. Positive and unlabeled learning for detecting software functional clones with adversarial training. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 2840–2846, 2018.

[67] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[68] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98, 2016.

[69] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522, 2013.

[70] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 146–157, 2019.

[71] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017.

[72] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: Security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering*, pages 462–472, 2017.

[73] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[74] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[75] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: exposing missing checks in source code for vulnerability discovery. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, pages 499–510, 2013.

[76] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *Proceedings of the 27th International Conference on Program Comprehension*, pages 70–80, 2019.

[77] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Proceedings of the Thirty-third International Conference on Neural Information Processing Systems*, pages 10197–10207, 2019.

[78] Deqing Zou, Hanchao Qi, Zhen Li, Song Wu, Hai Jin, Guozhong Sun, Sujuan Wang, and Yuyi Zhong. Scvd: A new semantics-based approach for cloned vulnerable code detection. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 325–344, 2017.

Table 6: Accuracy of MVP on Different Projects

Target System	Versions	Patches (#)	TP (#)	FP (#)
Curl	7_39_0 7_50_0 7_66_0	556	4	1
FFmpeg	n2.8.12 n3.3.6 n4.2	2319	2	0
Freetype2	VER-2-5-1 VER-2-6-2 VER-2-9-1	368	3	2
Radare2	2.0.1 2.7.0 3.2.1	618	5	1
VLC	1.3.0 3.0.0 4.0.0	1163	2	0
Wireshark	2.0.6 2.2.0 3.0.1	1181	3	1
Total	–	6205	19	5

Appendix

A Accuracy Evaluation on More Projects

To evaluate the accuracy of the default configuration of MVP on the other open-source projects, we selected six additional projects and used three versions for each of them. With a total of 6,205 security patches, MVP detected 24 potentially vulnerable functions, where 19 were true positives and 5 were false positives. The detailed results are reported in Table 6. These results indicate that MVP performs well on other target systems, and thus has good generality.

B Vulnerability Types

MVP found 97 recurring vulnerabilities in **RQ1**, detected 8 more recurring vulnerabilities in **RQ3** when configured to set different thresholds. The types of these vulnerabilities are listed in Table 7. These vulnerabilities covered a variety of different types. Although MVP does not take credit for the 8 vulnerabilities in **RQ3**, we believe MVP is capable of discovering recurring vulnerabilities of different types.

Table 7: Types of Detected Vulnerabilities

Type of Vulnerability	RQ1	RQ3	RQ4
Divide-by-Zero	3	0	0
Infinite Loop	7	0	0
Integer Overflow	8	0	0
Use-of-Uninitialized Value	9	1	0
Memory Leak	21	1	0
Null Pointer Dereference	14	3	0
Out-of-Bound Access	35	3	0
Total	97	8	0