2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Similar but Patched Code Considered Harmful

The Impact of Similar but Patched Code on Recurring Vulnerability Detection and How to Remove Them

Zixuan Tan[†], Jiayuan Zhou[‡], Xing Hu[†*], Shengyi Pan[†], Kui Liu[§], Xin Xia[§]
[†]Zhejiang University, Hangzhou, Zhejiang, China
[‡]Centre for Software Excellence, Huawei, Toronto, Canada
[§]Huawei, Hangzhou, Zhejiang, China
{tanzixuan, xinghu, shengyi.pan}@zju.edu.cn, jiayuan.zhou1@huawei.com, brucekuiliu@gmail.com, xin.xia@acm.org

***Abstract*—Identifying recurring vulnerabilities is crucial for ensuring software security. Clone-based techniques, while widely used, often generate many false alarms due to the existence of *similar but patched* (SBP) code, which is similar to vulnerable code but is not vulnerable due to having been patched. Although the SBP code poses a great challenge to the effectiveness of existing approaches, it has not yet been well explored.**

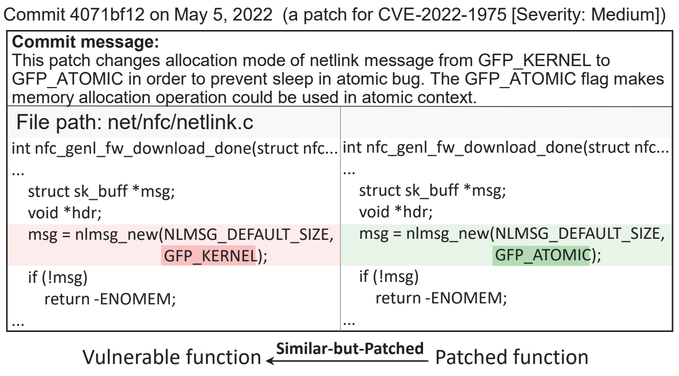
In this paper, we propose a programming language agnostic framework, Fixed Vulnerability Filter (FVF), to identify and filter such SBP instances in vulnerability detection. Different from existing studies that leverage function signatures, our approach analyzes code change histories to precisely pinpoint SBPs and consequently reduce false alarms. Evaluation under practical scenarios confirms the effectiveness and precision of our approach. Remarkably, FVF identifies and filters 65.1% of false alarms from four vulnerability detection tools (i.e., ReDeBug, VUDDY, MVP, and an elementary hash-based approach) without yielding false positives.

We further apply FVF to 1,081 real-world software projects and construct a real-world SBP dataset containing 6,827 SBP functions. Due to the SBP nature, the dataset can act as a strict benchmark to test the sensitivity of the vulnerability detection approach in distinguishing real vulnerabilities and SBPs. Using this dataset, we demonstrate the ineffectiveness of four state-of-the-art deep learning-based vulnerability detection approaches. Our dataset can help developers make a more realistic evaluation of vulnerability detection approaches and also paves the way for further exploration of real-world SBP scenarios.

***Index Terms*—Vulnerability Management, Software Maintenance, Software Security**

I. Introduction

Code reuse is one of the most frequent activities in software development [1]. By copying and pasting code snippets with or without modification, developers reuse existing code to improve the efficiency of programming. However, vulnerabilities in the original code may also spread to downstream software. For example, more than 60,000 open-source software projects are exposed to the vulnerability CVE-2017-12652 [2] for the reuse of unsafe code snippets in the popular graphic library *Libpng* [3], [4]. Due to poor software maintenance, these cloned *similar vulnerabilities* introduced from the reuse process are difficult to detect [4]–[8]. Therefore, it is crucial


Commit 4071bf12 on May 5, 2022 (a patch for CVE-2022-1975 [Severity: Medium])

Commit message:
This patch changes allocation mode of netlink message from GFP_KERNEL to GFP_ATOMIC in order to prevent sleep in atomic bug. The GFP_ATOMIC flag makes memory allocation operation could be used in atomic context.

File path: net/nfc/netlink.c

```
int nfc_genl_fw_download_done(struct nfc...   | int nfc_genl_fw_download_done(struct nfc...
...                                           | ...
  struct sk_buff *msg;                        |   struct sk_buff *msg;
  void *hdr;                                  |   void *hdr;
  msg = nlmsg_new(NLMSG_DEFAULT_SIZE,         |   msg = nlmsg_new(NLMSG_DEFAULT_SIZE,
                    GFP_KERNEL);              |                     GFP_ATOMIC);
  if (!msg)                                   |   if (!msg)
     return -ENOMEM;                          |      return -ENOMEM;
...                                           | ...
```

Fig. 1. An example showing the subtle difference between a vulnerable function (CVE-2022-1975) and the patched version [9].

for software maintainers to detect similar vulnerabilities in their codebases effectively.

The code-clone-detection-based (clone-based) approaches are commonly used to detect similar vulnerabilities [4], [10]–[13]. Generally, these techniques extract various signatures from vulnerable code and match similar code snippets as potential vulnerabilities. However, due to the subtle differences between vulnerable code and its corresponding patched versions [13], it is a challenge for clone-based approaches to differentiate them effectively (see Section IV-A for our experimental result). This often leads to the misidentification of such **Similar-but-Patched** (**SBP**) code as vulnerable, thus causing many false alarms. Figure 1 shows an example of SBP code in the Linux kernel [9] related to CVE-2022-1975 [14]. The vulnerability was simply fixed by altering an argument of the `nlmsg_new` function from `GFP_KERNEL` to `GFP_ATOMIC`). In this case, with only a single argument difference, the vulnerable function is closely similar to the patched version (SBP). Moreover, a piece of an SBP code could even exactly match a vulnerability. For example, if a vulnerability patch is reverted for various reasons (e.g., obsolescence or substitution with a better patch [15]), the reverted code would become the same as the vulnerable code, but without maintaining its vulnerability since the vulnerability condition will not be triggered (see Figure 6). This poses a significant challenge for clone-based approaches that aim to determine vulnerability by only examining current code.

*Corresponding author

In practice, the inability to distinguish vulnerable and SBP code can lead to a large number of false alarms, requiring substantial human effort to manually verify the results, which is not always feasible and hinders the application of these approaches [16]. MVP [13], proposed by Xiao et al., designed a function-level signature scheme to distinguish a vulnerability and an SBP. However, the proposed signature scheme is programming language-specific and lacks generalizability. Furthermore, MVP cannot handle the *reverted type SBP* because its signature is identical to that of the vulnerability.

In this study, we propose a programming language agnostic framework, **FVF** (**F**ixed **V**ulnerability **F**ilter), to reduce false alarms in clone-based vulnerability detection by identifying and filtering the SBP code. The core idea behind FVF is to leverage code change histories to determine whether the detected potentially vulnerable code snippet has already been patched. FVF works as a post-processing step for existing vulnerability detection approaches. When a potentially vulnerable code snippet similar to a known vulnerability is detected, FVF queries the vulnerability feature database for a *patch log*, which records the code change history of the vulnerability fix. It then retrieves the *function change log* of the target code snippets. Following existing recurring vulnerability detection approaches focusing on detecting function-level vulnerabilities, FVF generates the change and patch logs at the function level. If the *patch log* is detected in the *function change log*, it indicates that the potentially vulnerable code snippet has been patched previously (known as an SBP code snippet).

We evaluate the effectiveness of FVF in reducing false alarms (i.e., SBP) in real-world scenarios. We adopt nine major versions of two popular open-source projects, namely the Linux kernel [17] and Redis [18], to evaluate how FVF can improve existing clone-based vulnerability detection approaches. We employ four popular clone-based vulnerability detection approaches, ReDeBug [10], VUDDY [11], MVP [13], and implement a simple hash-based approach, as baseline vulnerability detectors. The experimental results show that the overall False Alarm Rate (FAR) for these detectors is 76.2%, which is far from satisfactory and impractical. After applying FVF, the overall FAR is reduced to 26.6%, with a significant improvement rate of 49.6%.

We further analyze where and why FVF makes false predictions, including false positives of FVF and false negatives, and we find no false positives and summarize 125 false negatives into two situations. Based on the findings, we conduct a qualitative study on the characteristics of filtered SBPs that confuse clone-based vulnerability detection approaches. We categorize 238 SBP code into three categories, which shed light on future research possibilities.

To evaluate the generalizability and scalability of FVF, we apply FVF on 1,081 historically vulnerable and popular open-source software (OSS) projects written in C, C++, and Java programming languages. In total, we collect 6,824 SBP functions and construct a dataset. Using the dataset, we study the prevalence of the SBP code in the real world and observe that 40% of OSS projects studied contain at least one instance of the SBP code. The results confirm the prevalence of the SBP phenomenon, emphasizing the need to address the challenge.

Besides clone-based vulnerability detection techniques, deep learning-based (DL-based) techniques have gained promising performance [19]–[21] in controlled lab environments. However, prior studies [21], [22] reveal that DL-based approaches sometimes leverage spurious features that are unrelated to the vulnerabilities, resulting in inferior performance in real-world scenarios. Given the subtle differences between vulnerability and SBP, DL-based approaches may also fail to distinguish them, leading to a large number of false alarms. Unfortunately, only a limited number of studies have considered SBP code and existing datasets such as Devign [23] and Big-Vul [24] overlook the inclusion of SBP code. As a result, DL-based approaches failed to learn SBP patterns during training, and evaluations become misaligned with real-world data distributions. The impact of the SBP code on DL-based approaches is not well explored.

Using the collected dataset, we evaluate the performance of state-of-the-art DL-based vulnerability detection approaches. We select two token-based approaches (LineVul [19] and VulBERTa [20]) and two graph-based approaches (Devign [23] and IVDetect [25]) for the study. We use these approaches to detect vulnerabilities on the SBP dataset to assess the impact of SBP on DL-based vulnerability detection approaches. The experimental results show that these approaches perform poorly on the dataset. All these approaches have a false alarm rate of more than 62%. The token-based approaches mistakenly predict almost all SBP code as vulnerable, and the two graph-based approaches have a false alarm rate of 64.9% and 62.9%, respectively. The results demonstrate the inability of these approaches to distinguish SBP from real vulnerabilities, thus emphasizing the discriminative effectiveness of our dataset. Our dataset can help developers make a more realistic evaluation of existing vulnerability detection tools and also paves the way for further exploration of real-world SBP scenarios.

Our contributions are summarized as follows:

- To the best of our knowledge, we are the first to systematically study the phenomenon of SBP and its impact on vulnerability detection. We find that while SBP code is prevalent in real-world scenarios, clone-based and DL-based vulnerability detection approaches are incapable of distinguishing SBP code, leading to a large number of false alarms in practice.
- We propose an effective framework, FVF, to identify SBP and help clone-based vulnerability detection approaches reduce false alarms. Experimental results show that FVF can significantly reduce false alarms of popular clone-based approaches such as VUDDY [11] and ReDeBug [10].
- We construct a real-world SBP dataset consisting of 6,824 SBP functions in three programming languages from 1,081 real-world projects using FVF, which can contribute to a more realistic evaluation of vulnerability detection tools. Our replication package can be accessed using the link [26].

The remainder of the paper is organized as follows: Section II describes the overview and design of our proposed framework. Section III and Section IV discuss the evaluation steps and results of FVF. In Section V, we discuss other features of FVF, such as programming language agnostic, and the impact of SBP code on promising deep learning-based approaches. In Section VI, we discuss the threats to the validity of our approach. Section VII summarizes related work in the field. We conclude the paper in Section VIII.

II. FVF: The Proposed Approach

The goal of FVF is to enhance existing clone-based vulnerability detectors by reducing false alarms caused by already patched vulnerabilities (i.e., SBP). The core idea is to take the code changes that fix historical vulnerabilities as a reference, to check whether the detected vulnerable code has been fixed in the past. In this section, we first present the overall framework and our design choices of FVF, followed by the details of each component.

A. Overall Framework

Following the existing recurring vulnerability detection approaches that focus on detecting function-level vulnerabilities, FVF also identifies SBP at the function level. The overall FVF framework, as shown in Figure 2, consists of four key components: ❶ **Function Change Log Generator**, ❷ **Vulnerability Patch Log Finder**, ❸ **Vulnerability Feature Database**, and ❹ **Fix Behavior Matcher**.

The vulnerability detector identifies vulnerabilities in the target repository and produces the detected potentially vulnerable functions ("Matched Function" in Figure 2) to the **Function Change Log Generator** to generate a *function change log*. Simultaneously, the detector produces the matched vulnerable function ("Vuln. Function" in Figure 2) to the **Vulnerability Patch Log Finder**, which queries the **Vulnerability Feature Database** to retrieve the *patch logs*. Finally, the **Fix Behavior Matcher** examines the *function change log* for the presence of fix behaviors indicated in the *patch logs*. If the same fix behavior is detected, the potentially vulnerable function is considered to have been previously fixed and is non-vulnerable (i.e., a false alarm). Otherwise, it remains potentially vulnerable and requires further review.

B. Vulnerability Detector

The Vulnerability Detector aims to detect recurring vulnerabilities in the target project. When analyzing a target code repository, the detector produces two types of information: the matched function ("Matched Function") and the known vulnerable function ("Vuln. Function"). The matched function represents the potentially vulnerable function detected in the target code repository. The detector outputs the location (including the name, parameter definitions, and return value) of the matched function. The vulnerable function refers to known vulnerable code that matches the detected potentially vulnerable function. By requiring only the vulnerable functions and the matched similar functions, FVF supports diverse types of recurring vulnerability detectors, varying from string matching methods like ReDeBug [10], signature-based methods like VUDDY [11], to more advanced slice-based techniques such as MVP [13] and TRACER [30]. This design ensures FVF's versatility and broad applicability. After detection, the matched and vulnerable functions are passed to the **Function Change Log Generator** and **Vulnerability Patch Log Finder**, respectively.

C. Function Change Log Generator

Given the matched function from the detector, the Function Change Log Generator retrieves the change history of the function in the version control system to build a *function change log*. A *function change log* is a sequence of code changes, denoted as $\langle \mathrm{fc}_1, \mathrm{fc}_2, ..., \mathrm{fc}_m \rangle$, where fc_i is a line-level code change on the function. The Function Change Log Generator first utilizes the *git log* command [31] with the file path and function name as parameters to obtain the change history of the function. Then all the change histories are concatenated in chronological order to generate the *function change log*.

As retrieving the history in a large codebase can be very costly and slow down the whole process, we set a retrieval window to limit how far back in the version history we should look for changes. We consider the original vulnerability fix date to be the earliest date we should retrieve, as a cloned vulnerability is unlikely to be fixed earlier than the original one recorded in CVE. Additionally, we empirically set a threshold of 50 to limit the number of retrieval operations. This threshold is a configurable option, and we discuss efficiency and performance under different thresholds in Section V-D.

In practice, we find the retrieved history may be truncated in specific cases when the matched function is renamed or moved. Hence, we retrieve file-level change histories as a supplement. Specifically, when the function-level retrieval stops before reaching the window size threshold, we further retrieve the file change history and extract all modifications to the function.

D. Vulnerability Patch Log Finder and Feature Database (DB)

The Vulnerability Patch Log Finder retrieves the appropriate *patch log* for the vulnerable function. It first gets the necessary information (e.g., the signatures and line numbers) about the vulnerable functions from the vulnerability detector. Then, it queries the Vulnerability Feature Database with CVE ID and vulnerable function signature for the *patch log*.

A *patch log* records all the fix actions, represented as a sequence of line-level code diffs concatenated from a series of patches on the function, chronologically. Figure 3 shows an example of generating a *patch log* for the function `tun_set_iff` and the vulnerability CVE-2018-7191 [27]. If multiple fixes have been applied to the same function, the last post-fix version is considered the fully fixed version. Thus, the *patch log* records each patch diff chronologically from the first patch to the last patch, making it naturally support multi-patch scenarios.

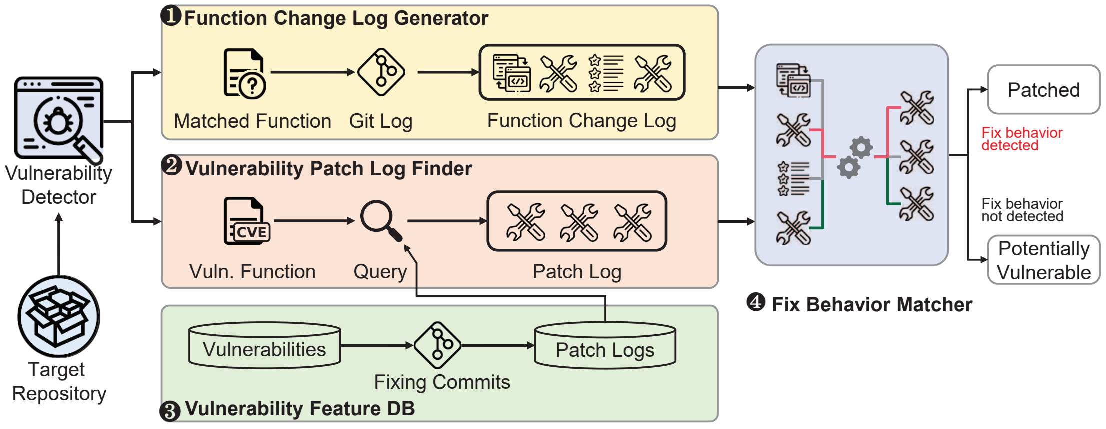
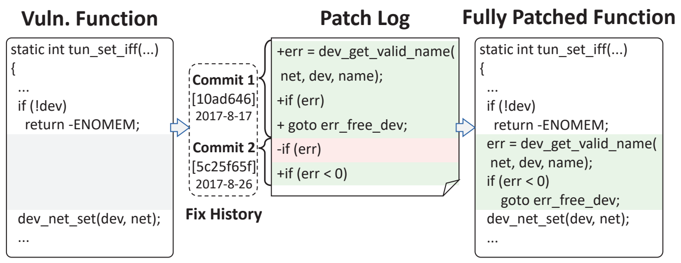


Fig. 2. Overview of FVF.



```
static int tun_set_iff(...)
{
  ...
  if (!dev)
   return -ENOMEM;



  dev_net_set(dev, net);
  ...
```

```
+err = dev_get_valid_name(
 net, dev, name);
+if (err)
+ goto err_free_dev;
-if (err)
+if (err < 0)
```

```
static int tun_set_iff(...)
{
  ...
  if (!dev)
   return -ENOMEM;
  err = dev_get_valid_name(
   net, dev, name);
  if (err < 0)
    goto err_free_dev;
  dev_net_set(dev, net);
  ...
```

Fig. 3. An example of generating the *patch log* for function `tun_set_iff` and vulnerability CVE-2018-7191 [27]. The *patch log* contains two fixes [28], [29].

The Vulnerability Feature Database stores the *patch logs* of all the disclosed vulnerabilities for the Vulnerability Patch Log Finder to query. Specifically, we collect the patch information of the disclosed vulnerabilities from vulnerability databases (e.g., NVD) and process to generate *patch logs* for each vulnerable function modified in the vulnerability patch. Besides, the database is continuously updated with the patch log of newly disclosed vulnerabilities, ensuring that the latest vulnerability information is always available for querying.

E. Fix Behavior Matcher

Given a *function change log* and a *patch log*, the Fix Behavior Matcher determines if the *patch log* is already contained in the *function change log*. If the condition is true, it means the function has fix behaviors in the past, suggesting the vulnerability has already been fixed. Hence, the detected result is an SBP case and is considered a false alarm.

Formally, for function change log $\text{FC\_Log} = \langle \text{fc}_1, ..., \text{fc}_m \rangle$ and patch log $\text{Pat\_Log} = \langle \text{pc}_1, ..., \text{pc}_n \rangle$, if there exists a set of indexes where $1 \leq i_1 < i_2 < ... < i_n \leq m$ such that $\mathit{Sim}(\text{pc}_j, \text{fc}_{i_j}) \geq \text{Threshold}$ for $1 \leq j \leq n$, we consider Pat_Log to be contained in FC_Log. If it is contained, it indicates that the potentially vulnerable function detected is an SBP one. Therefore, this detection result is a false alarm. Instead of direct string matching, we employ similarity to make FVF more robust when the cloned version has different literal representations, such as using different variable names or function names, etc. We denote *Sim* as a similarity calculator and calculate the BLEU-2 [32] score. The BLEU-2 score quantifies the similarity of 2-grams, i.e., consecutive pairs of words, making it especially applicable in scenarios where only the identifier name varies.

Since FVF is only interested in the subsequence of the *patch log* during the matching process, the code changes in the function history that are not related to the patch do not affect the result.

III. Experiments

In this paper, we aim to answer the following research questions:

RQ1: How effective is FVF in identifying SBP and reducing false alarms? Clone-based vulnerability detection approaches are often not practical as they struggle to distinguish vulnerabilities and SBP code, resulting in a large amount of false alarms [10], [13]. The goal of this RQ is to evaluate the false alarm rate of the existing clone-based vulnerability detection approaches and then the effectiveness of FVF in reducing false alarms.

RQ2: What are the false predictions of FVF in identifying false alarms? In this RQ, we look into the details of when FVF fails in identifying SBP code. Especially, we investigate the FPs (i.e., real recurring vulnerabilities that are incorrectly identified as SBP code snippets) and FNs (i.e., real false alarms that are not identified) of FVF.

RQ3: What are the characteristics of filtered SBP code snippets? In this RQ, we further conduct a qualitative study on the filtered SBP code snippets, to gain empirical insights on their characteristics, including the categories of SBP code snippets and the reason for each category.

A. Data Collection and Preprocessing

Our experiment data include two parts: 1) Vulnerability feature database and 2) Target project source code. The vulnerability feature database contains information on existing vulnerabilities, which is used by FVF. The target project is the project from which FVF identifies SBP code snippets. Figure 4 illustrates the workflow of our data collection process. We describe the process in detail below.

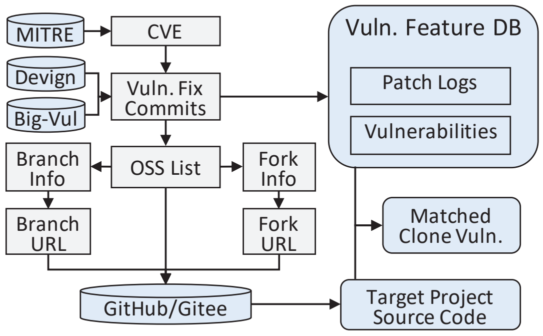


Fig. 4. An overview of the data collection approach.

1) Vulnerability feature database: The database includes two main parts: vulnerability information and *patch logs*. Initially, vulnerability information is collected from multiple reliable sources, followed by the generation of *patch logs*. The process of vulnerability feature database construction is outlined as follows:

Step 1: Collecting Vulnerability Fix Commits (VFCs). Vulnerability fix commits are commits in the version control system that fix vulnerabilities. We collect VFCs from two existing vulnerability datasets, namely Devign [23] and Big-Vul [24], resulting in 6,611 and 3,746 VFCs, respectively. Additionally, we extract more VFCs from the reference links of the CVE records in the authoritative vulnerability information source, the MITRE CVE database [33]. Note that we only keep VFCs modifying C/C++ source files. After deduplication, we collect a total of 15,636 VFCs distributed across 978 projects.

Step 2: Cleaning Noisy Data. It is important to note that not all changes within a VFC are related to fixing vulnerabilities. Some commits contain multiple intentions (e.g., code refactoring), which could introduce additional noise. We conduct a statistical analysis on the number of files modified in each commit within the dataset and find that the 99th percentile of the number of modified files per vulnerability fix is 10. Accordingly, VFCs that modify more than 10 files are excluded. We additionally exclude noisy commits that only revise comments or adjust white spaces.

Step 3: Extracting Vulnerability-Relevant Functions. We extract both pre- and post-commit versions of modified functions from each VFC using PyDriller [34]. Following previous studies [23], [24], the pre-commit version is labeled as vulnerable, while the post-commit version is considered patched. In the case of multiple VFCs for a single vulnerability, only the post-commit version of the latest VFC is considered patched; otherwise, it remains vulnerable.

Step 4: Generating Patch Log. Figure 3 shows an example of generating a *patch log*. A *patch log* is a sequence of modification lines beginning with '+' or '-', which records every code change made to the vulnerable function, from the initial VFC to the latest one chronologically. Specifically, the modification lines related to the vulnerable function in each VFC are concatenated into a unified sequence to build a *patch log*. In total, we collect 35,319 vulnerable functions, and 18,074 *patch logs* are generated.

TABLE I
DETAILS OF SELECTED BRANCHES OF TARGET OSS.

OSS	Branch	Abbr.	Status	Function	LoC
Linux Kernel	6.3-rc5	L.6.3	Mainline	657K	16M
	6.2.9	L.6.2	Stable	656K	16M
	5.15.105	L.5.15	LTS	616K	15M
Redis	7.0.10	R.7	Stable	7K	154K
	5.0.14	R.5	LTS	5K	105K

TABLE II
DETAILS OF SELECTED FORKS OF TARGET OSS.

OSS	Downstream Fork	Abbr.	Commits Ahead	Commits Behind	Last Update
Linux Kernel	Asahi Linux	L.A	2,230	32,401	2022-12
	Linux Kernel Lib.	L.L	975	80,518	2023-02
	O.H. Linux Kernel 5.10	L.O	11,076	273,473	2023-04
Redis	Birdisle	R.B	666	4,521	2023-04

2) Target project source code: We follow the existing clone-based vulnerability detection approaches [11], [13] to adopt the Linux kernel [17] and Redis [18], two widely used OSS projects with extensive branches and forks, as our target projects. Linux is a widely used operating system kernel, while Redis is a popular key-value database system extensively used by companies. Then we further collect the source code of the branches and forks of the target OSS projects.

Table I provides details of the branches we select for the Linux kernel and Redis. For Linux, we select the presently latest version (tag: ***6.3-rc5***) on the master branch, a stable release (tag: ***6.2.9***), and a long-term support version (tag: ***5.15.105***). For Redis, we select one stable version (tag: ***7.0.10***) and one long-term support branch (tag: ***5.0.14***).

In supplement of branches, we select OSS forks that make customization and still actively commit in the most recent six months. Details of the forks for the Linux kernel and Redis are outlined in Table II. The term *Commits Ahead* denotes the number of exclusive commits in the forked version, indicating the evolving history of the project. *Commits Behind* denotes the number of upstream commits not yet to be incorporated into the fork, indicating the level of outdatedness. For Linux, three downstream forks are chosen: ***Asahi Linux*** [35], which aims to adapt the Linux kernel to Apple silicon Mac computers. ***Linux Kernel Library*** [36], focusing on reusing Linux kernel code as a library for user-level applications. ***OpenHarmony Linux kernel 5.10*** [37], a customized Linux kernel maintained by OpenHarmony (O.H.), an open-source project for the Internet of Things (IoT) devices. Regarding Redis, one fork is selected: ***Birdisle*** [38], a modified Redis version that operates as a library within another process.

B. Baselines

We evaluate our approach with four baselines, including three well-known vulnerable code clone detection approaches (i.e., ReDeBug, VUDDY, and MVP) and a hash-based approach.

ReDeBug [10] extracts vulnerable signatures from vulnerability patches and leverages a pattern-matching approach to detect unpatched code clones. ReDeBug takes the deleted lines and the surrounding context lines in the patch file to generate the vulnerable signature.

VUDDY [11] employs abstraction and normalization techniques to generate coarse-grained and vulnerability-preserving function signatures for vulnerable code clone detection.

MVP [13] employs program slicing techniques to extract vulnerability and patch signatures, identifying recurring vulnerabilities that match the vulnerability signatures while not matching the patch signatures.

Hash-based is a simple function-matching approach that aims to evaluate the effectiveness of FVF even for such a trivial approach. This approach generates a hash ID for a given function as the signature of the function. For a given function, the approach compares the signature with all signatures of known vulnerable functions. If two signatures are matched, the function is considered to be a recurring vulnerability. We simply abstract the names and parameters of functions to obtain a slight generalization ability.

C. Test Scenarios

We evaluate the false alarm rate of baselines before and after applying our proposed approach in detecting vulnerabilities of target projects. Specifically, we run the baseline vulnerability detection tools on each target project, then apply FVF on the detected potential vulnerabilities, to identify false alarms caused by SBP.

Due to the inherent challenge of identifying vulnerabilities, there is no definitive ground truth on all possible vulnerabilities. To establish a fair and reliable ground truth, we engage a team of three security professionals, each with at least three years of experience in software security, to manually validate each potential vulnerability detected by the baselines.

The validation process starts with two security professionals independently labeling the potential vulnerabilities identified by the approach. Specifically, each professional is asked to first review the detailed disclosed information of the source vulnerability, including the CVE description and the vulnerability patch, to gain a comprehensive understanding of the vulnerability root cause. Then, based on such understanding of the source vulnerability, the professional evaluates whether the detected vulnerability is truly a recurring vulnerability. Subsequently, the labeling results of the two professionals are cross-verified. We employ Cohen's Kappa [39] to measure the inter-rater reliability, resulting in a value of 0.79, which indicates a substantial agreement. To increase reliability, the remaining discrepancies are resolved by the third professional using a similar review approach. The professionals have ample time for thorough inspections and are well compensated for their effort and expertise.

D. Evaluation Metrics

As our goal is to minimize the number of false alarms in clone-based vulnerability detection, we evaluate the baselines and the improvement made by FVF through three false-alarm-centric metrics: the number of false alarms, the false alarm rate (equivalent to false alarm precision), and the improvement rate. We do not calculate recall since we do not have the ground truth of detected negatives for the baseline detector.

False Alarm (FA) represents the number of false alarms produced by the baselines. A higher FA indicates a lower detection accuracy of an approach.

False Alarm Rate (FAR) is the ratio of false alarms to the total number of predicted positives. FAR reflects the accuracy of positive predictions made by the detector. A higher FAR indicates lower quality in detection performance.

Improvement Rate (IR) is the observed improvement rate of false alarms (i.e., the reduction in FAR) before and after applying our proposed approach. A higher IR indicates a more substantial enhancement brought by FVF.

E. Implementation Details

We implement FVF in 1.5k lines of Python code. Our experiments are performed on an Ubuntu 20.04 server, with 2 Intel Xeon Gold 6226R CPUs and 256G RAM. For ReDeBug [10], we follow the original configuration to set the length of n-gram to 4. For VUDDY [11], we use the online platform offered on their official website [40]. We first generate signatures locally and then upload them to the platform for detection. For MVP [13], we reimplement the algorithm strictly according to the methodology in the original paper and use the recommended parameters as the original paper as well. For the hash-based approach, we leverage the MD5 algorithm to generate hash IDs and compare the hash IDs.

IV. Experimental Results

A. RQ1: The Effectiveness of FVF in Identifying SBP and Reducing False Alarms

In total, the four baselines produce 470 potential vulnerabilities after deduplication. Three security professionals with at least three years of security experience manually verify all potential vulnerabilities produced. The manual verification process is described in detail in Section III-C. After verification, 112 potential vulnerabilities are confirmed as real (i.e., true positives, TP) in total. The results are presented in Table III. For the Linux kernel, all 75 TPs are detected in the L.L project, indicating a poor security maintenance status. Regarding Redis, the latest version (R.7) is well-maintained but missing fixes are found in R.5 and R.B.

Among all approaches, the hash-based model is the most trivial. For the Linux kernel, the detection results of the hash-based model are 100% false alarms except for the LKL (L.L) project. After applying FVF, the FAR decreases and ranges from 6.3% to 13.8%. The IR ranges from 50.0% to 88.9%. For Redis, the FAR ranges from 30.0% to 100.0%. After applying FVF, all the false alarms are identified and filtered.

VUDDY generates more coarse-grained function-level signatures than the hash-based model to achieve better performance. However, VUDDY performed the worst among the baselines. For the Linux kernel, all the results reported by VUDDY are false alarms, with the exception of the LKL project, which exhibits a notably high FAR of 84.2%. For Redis, the FAR ranges from 50.0% to 100.0%. One possible

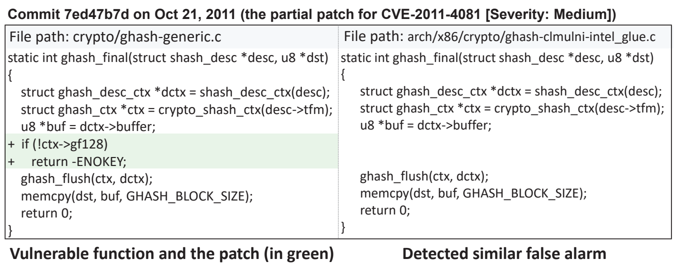
TABLE III
THE PERFORMANCE OF BASELINES IN DETECTING VULNERABILITIES IN LINUX AND REDIS BEFORE/AFTER ADOPTING FVF (TP: TRUE POSITIVE, FA: FALSE ALARM, FAR: FALSE ALARM RATE, IR: IMPROVEMENT RATE.)

Proj. Abbr.	Approach	TP	Original Perf.		After FVF		
			FA	FAR(%)	FA	FAR(%)	IR(%)
L.6.3	Hash-based	0	23	100.0	2	13.0	87.0
	VUDDY	0	13	100.0	2	15.4	84.6
	ReDeBug	0	90	100.0	38	42.2	57.8
	MVP	0	149	100.0	23	15.4	84.6
L.6.2	Hash-based	0	24	100.0	3	12.5	87.5
	VUDDY	0	13	100.0	2	15.4	84.6
	ReDeBug	0	91	100.0	38	41.8	58.2
	MVP	0	153	100.0	23	15.0	85.0
L5.15	Hash-based	0	29	100.0	4	13.8	86.2
	VUDDY	0	16	100.0	3	18.8	81.3
	ReDeBug	0	93	100.0	40	43.0	57.0
	MVP	37	165	81.7	28	13.9	67.8
L.A	Hash-based	0	27	100.0	3	11.1	88.9
	VUDDY	0	15	100.0	2	13.3	86.7
	ReDeBug	0	92	100.0	38	41.3	58.7
	MVP	0	149	100.0	23	15.4	84.6
L.L	Hash-based	21	27	56.3	3	6.3	50.0
	VUDDY	3	16	84.2	2	10.5	73.7
	ReDeBug	18	95	84.1	40	35.4	48.7
	MVP	41	149	78.4	21	11.1	67.3
L.O	Hash-based	0	29	100.0	4	13.8	86.2
	VUDDY	0	16	100.0	3	18.8	81.3
	ReDeBug	0	112	100.0	43	38.4	61.6
	MVP	47	157	77.0	27	13.2	63.8
R.7	Hash-based	0	5	100.0	0	0.0	100.0
	VUDDY	0	3	100.0	0	0.0	100.0
	ReDeBug	0	5	100.0	0	0.0	100.0
	MVP	0	13	100.0	0	0.0	100.0
R.5	Hash-based	7	3	30.0	0	0.0	30.0
	VUDDY	2	2	50.0	0	0.0	50.0
	ReDeBug	8	3	27.3	0	0.0	27.3
	MVP	4	10	71.4	0	0.0	71.4
R.B	Hash-based	7	3	30.0	0	0.0	30.0
	VUDDY	2	2	50.0	0	0.0	50.0
	ReDeBug	8	3	27.3	0	0.0	27.3
	MVP	49	4	7.5	0	0.0	7.5
Total	-	112	358	76.2	125	26.6	49.6

explanation for the high FA could be the quality of the vulnerability database. VUDDY relies on its private online database for detection, which may be outdated and not comprehensive. After enhancing VUDDY by applying FVF, the FAR in Linux is effectively reduced, dropping by 73.7%-86.7%. For Redis, all false alarms are identified and filtered.

ReDeBug constructs vulnerability signatures using partial information on vulnerabilities to identify cloned vulnerabilities, which makes ReDeBug more robust but also introduces more false alarms. For the Linux kernel, ReDeBug produces 90 to 112 false alarms, with a FAR of 100.0% across all versions except for the LKL, which is 84.1%. For Redis, the FAR ranges from 27.3% to 100.0%. After applying FVF, the FARs on the Linux kernel projects decreased significantly, dropping by 48.7% to 61.6%. Furthermore, FVF filtered out all false alarms on Redis.

MVP leverages code-slicing techniques on the vulnerable


Commit 7ed47b7d on Oct 21, 2011 (the partial patch for CVE-2011-4081 [Severity: Medium])

File path: crypto/ghash-generic.c

```
static int ghash_final(struct shash_desc *desc, u8 *dst)
{
   struct ghash_desc_ctx *dctx = shash_desc_ctx(desc);
   struct ghash_ctx *ctx = crypto_shash_ctx(desc->tfm);
   u8 *buf = dctx->buffer;
+  if (!ctx->gf128)
+     return -ENOKEY;
   ghash_flush(ctx, dctx);
   memcpy(dst, buf, GHASH_BLOCK_SIZE);
   return 0;
}
```

File path: arch/x86/crypto/ghash-clmulni-intel_glue.c

```
static int ghash_final(struct shash_desc *desc, u8 *dst)
{
    struct ghash_desc_ctx *dctx = shash_desc_ctx(desc);
    struct ghash_ctx *ctx = crypto_shash_ctx(desc->tfm);
    u8 *buf = dctx->buffer;


    ghash_flush(ctx, dctx);
    memcpy(dst, buf, GHASH_BLOCK_SIZE);
    return 0;
}
```

Vulnerable function and the patch (in green)
Detected similar false alarm

Fig. 5. A false alarm falls outside of SBP. The complete patch for CVE-2011-4081 [41] includes two identical code changes. Here we show the first.

and patched function to generate vulnerability signatures, which makes the signature more comprehensive in detecting potential similar vulnerabilities. However, it still predicts many false alarms due to not considering reverted-type SBPs. For the six versions of the Linux kernel, MVP predicts 149 to 214 similar vulnerabilities and the false alarm rate is between 77% and 100%. For Redis, MVP predicts 13 to 53 similar vulnerabilities and the false alarm rate is between 7.5% and 100%. After applying FVF, the FARs on the Linux kernel projects decreased significantly by 63.8% to 85.0%. For Redis, all the false alarms are identified and filtered by FVF.

In summary, the overall FAR of all four baselines is 76.2%, which is far from satisfactory and impractical. FVF identifies 233 fixed vulnerabilities in 358 potential vulnerabilities, reducing the overall FAR from 76.2% to 26.6%. Even with the trivial hash-based approach, FVF reduces FAR from 60.3% (32 out of 53) to 7.5% (4 out of 53). These results demonstrate the effectiveness of FVF in reducing false alarms in clone-based vulnerability detection.

RQ1 Result: The false alarm rate of the four existing clone-based vulnerability detection approaches are high and far from satisfactory. FVF is proven to be effective in reducing false alarms.

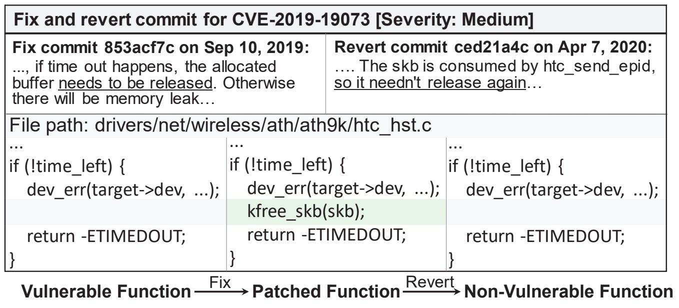
B. RQ2: False Predictions of FVF in Identifying False Alarms of Clone-based Vulnerability Detection Approaches

FVF aims to reduce the false alarms of clone-based vulnerability detection approaches by identifying SBP code snippets. In this RQ, we further look into the details of when FVF fails.

*1) **False Positives of FVF**:* False positives of FVF refer to real recurring vulnerabilities that are incorrectly filtered as SBP code by FVF. This may hinder the actual vulnerabilities and is often unacceptable. We manually verify all SBP code snippets identified in RQ1 and find no false positive case.

FVF employs an evidence-based process and adopts a conservative strategy to identify SBP code snippets. Specifically, FVF considers a vulnerable function detected by a clone-based vulnerability detection approach as SBP if and only if all code changes in the vulnerability patch log are rigorously contained in the function change log. This ensures that the SBP identified by FVF must have been historically patched.

*2) **False Negatives of FVF**:* False negatives are false alarms produced by the underlying clone-based vulnerability detectors but not filtered by FVF. While FVF can significantly


Fix and revert commit for CVE-2019-19073 [Severity: Medium]

Fix commit 853acf7c on Sep 10, 2019: ..., if time out happens, the allocated buffer needs to be released. Otherwise there will be memory leak…

Revert commit ced21a4c on Apr 7, 2020: …. The skb is consumed by htc_send_epid, so it needn't release again…

File path: drivers/net/wireless/ath/ath9k/htc_hst.c

```
...
if (!time_left) {
    dev_err(target->dev, ...);

    return -ETIMEDOUT;
}
```

```
...
if (!time_left) {
    dev_err(target->dev, ...);
    kfree_skb(skb);
    return -ETIMEDOUT;
}
```

```
...
if (!time_left) {
    dev_err(target->dev, ...);

    return -ETIMEDOUT;
}
```

Fig. 6. An example of patch reversion. Commit ced21a4c [43] reverts the change made by commit 853acf7c [44] – the fix for CVE-2019-19073 [45].

reduce false alarms (specifically, the SBP code) as shown in RQ1, there are cases where FVF misses certain false alarms. In RQ1, there are 125 false alarms that FVF does not identify. We manually conduct a qualitative study to investigate the reason why FVF fails and summarize two situations:

❶ **Lack of vulnerability trigger point.** We observe 90 cases that do not have fix behaviors in the past, however, are not vulnerable. What makes the difference is the calling context, i.e., the contextual conditions. The contextual conditions required for the vulnerabilities are not satisfied for the detected functions, therefore, they are false alarms.

Figure 5 shows a failed example for CVE-2011-4081 [42] and the detected false alarm. The root cause of the vulnerability is a null pointer dereference due to the pointer (`ctx->gf128`) could be null. The vulnerability was fixed on Oct 21, 2011, by adding a null pointer check and returning an error code in the null case. The detected false alarm, with the same name `ghash_final`, is also in the Linux project but exists in a different file. Since the pointer is not used within that file, there is no vulnerability or need for a null check.

In such cases, contextual information, such as function calls and value flow, becomes critical in determining if a similar function is truly a vulnerability. To identify this type of false alarm, *inter-procedural analysis* could be introduced. However, filtering such cases is out of the scope of FVF, while we focus on false alarms caused by SBP code.

❷ **Trivial clone false alarms.** There are 35 false alarms failed to be identified, which code is neither related to the vulnerabilities nor the patches. These cases are mostly short auxiliary functions or even non-function code and are hard to associate with any vulnerabilities. All these cases are produced by ReDeBug, which constructs vulnerability signatures using partial context lines in patch diffs. Although this improves the scalability, it also introduces more false alarms. Filtering out such false alarms falls is also out of the scope of FVF.

RQ2 Result: In our evaluation, FVF has no false positives and keeps a low false negative rate in reducing false alarms. It is important to note that the few missed false alarms are mainly out of the scope of FVF.

C. RQ3: The Characteristics of Identified SBPs

In total, FVF identified and filtered out 238 false alarms in RQ1. After manual verification, we confirm that all of them are already patched and should be filtered out. In this RQ, we further analyze the characteristics of these filtered false alarms to gain empirical insights. We categorize them into three categories: 1) Patch Reversion (126 cases), 2) Minor Difference (86 cases), and 3) Customized Patch (26 cases).

❶ **Patch Reversion** refers to the cases where a vulnerable function is first fixed, but then the modification is rolled back. There are 126 patch reversion cases in total. Reverting a commit involves undoing the change made before. It is found to be a common operation during software development due to reasons such as unexpected software regression [46] and the introduction of new bugs [47]. We observe that patch reversion also occurs in the context of vulnerability fixing.

Figure 6 shows an example of a patch of CVE-2019-19073 [44] and the corresponding reversion. The root cause is that, when timed out, the allocated socket buffer `skb` is forgotten to be released, resulting in a memory leak. However, the patch (made on Sep 10, 2019) was reverted on Apr 7, 2020, since `skb` is consumed by another function (`htc_send_epid`). Therefore, it is incorrect to release it again. After reverting the patch, the function is the same as the vulnerable one first reported in CVE-2019-19073 [45] but is no longer vulnerable. Most existing vulnerability detection approaches (e.g., the clone-based vulnerability detection [10], [11], [13], DL-based vulnerability detection [19], [20], [23] fail to distinguish reverted functions from vulnerable ones, since they only rely on the information of the function itself. This also verifies the unique advantages of our proposed FVF by considering the function change logs.

We further analyze the messages of the revert commits and identify two primary reasons for patch reversions: 1) *Contextual Change*. In 28 cases, the commit messages suggest that the vulnerable version is no longer vulnerable due to a change in context. 2) *Inadequate Patch*. In 18 cases, the original patch is found to be incorrect or inadequate to fix the vulnerability, and it is easier to develop a new patch from scratch rather than revise the original patch.

This phenomenon indicates the secrecy of the reverted patches. It is noteworthy that the cases of patch reversions are not rare, and the corresponding research is insufficient. So far, no baseline can handle the reverted patches.

❷ **Minor Difference** refers to false alarms where a patched function is wrongly detected as vulnerable, but not due to patch reversion or customized patch. There are 86 such cases in total. We analyze the difference between vulnerable and patched functions based on three metrics: lines of code (LOC), added lines of code (ALOC), and removed lines of code (RLOC). The median values are 4, 0, and 3, respectively. This confirms the challenges in distinguishing patched functions from vulnerable ones due to the subtle differences.

❸ **Customized Patch** refers to false alarms in which developers apply the original patches from upstream but then customize the patched code to fit the downstream. After customization, the signatures of the customized functions become similar to those of the vulnerable functions, resulting in false alarms of clone-based vulnerability detection approaches. We

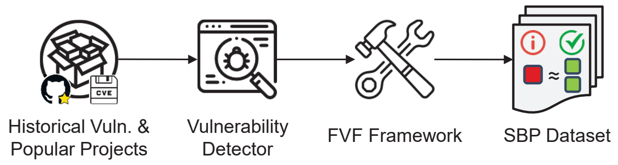


Fig. 7. SBP Dataset Construction.

observe eight false alarms of the customized patch. Due to the customization requirements, upstream patches may require additional development or refactoring, as discussed in previous studies [5], [48]. Some existing approaches, such as ReDeBug, also consider patch information. However, these approaches may still fail to distinguish customized patches from vulnerabilities. The reason is that most fix information is typically extracted from the mainline version, and customized patches are often not taken into consideration [48]. FVF can identify these false alarms by analyzing the code change history. By detecting the core fix behaviors, FVF can detect the function as a potential false alarm.

In conclusion, among the 238 filtered SBP-related false alarms, 52.9% (126 cases) are patch reversion false alarms, 36.1% (86 cases) are minor difference false alarms and 10.9% (26 cases) are customized patch false alarms. All three cloned vulnerability detection approaches fail to correctly distinguish these fixed vulnerabilities, generating a large number of false alarms. In contrast, our proposed approach, FVF, can effectively reduce these false alarms caused by SBP.

RQ3 Result: We categorize the 238 filtered false alarms into three categories and provide insights of each category: Patch Reversion, Minor Differences, and Customized Patch. Our analysis demonstrates the unique advantages of FVF in reducing the false alarms caused by SBP code by utilizing the function change logs.

V. Discussion

In this section, we further generalize FVF to large-scale real-world open-source software (OSS) projects and construct an SBP dataset. We aim to answer: 1) Can FVF be generalized to large-scale real-world projects? 2) What is the impact of the SBP code on deep learning-based vulnerability detection approaches? We also discuss the prevalence of SBP code in real-world projects, as well as the efficiency and the programming language agnostic nature of FVF.

A. Prevalence of SBP Code in Open-Source Projects

We leverage the same framework as in Section II-A to construct the SBP dataset. Figure 7 presents an overview of the construction of the dataset. We employ VUDDY-J, an extended version of VUDDY [11] that supports Java programming language, as the vulnerability detector, with vulnerability source collected from MITRE [33], to scan for vulnerable code clones on a large number of OSS projects. The detected potentially vulnerable code clones are further verified by FVF to identify the instances of SBP, which forms our dataset.

The scanned projects are selected from two sources: historically vulnerable projects from CVE and popular projects in the wild. Historically vulnerable projects provide cases where the fixed function (e.g., backported patches) is similar to the vulnerable function. Popular projects allow us to investigate the presence of the SBP phenomenon in the wild. The top 100 most starred projects on GitHub for the C, C++, and Java programming languages are selected, respectively. In total, we collected 1,081 projects for detection. Since a project may contain different software versions in different branches, the detection is only performed on active branches with code commits within the last five years.

After deduplication, we collect 6,827 SBP functions, which are similar to 3,945 vulnerable functions associated with 2,834 vulnerabilities. To ensure the functions included in our dataset are truly SBPs, we manually check 364 randomly sampled SBP functions (with a confidence level of 95% and a margin of error of 5%) and verify that all of them are patched. This is not ideal, but manually examining all samples requires enormous human labor and is not feasible. Moreover, we have shown in RQ2 that FVF, as an evidence-based approach, achieves a precision of 100% (i.e., no falsely identified SBPs). It is noteworthy that all the sampled functions are confirmed as real SBP code, demonstrating the accuracy of FVF and its low false positive rate.

We assess the prevalence of SBP code in three perspectives: project, vulnerability, and function. Out of the 1,081 OSS projects studied, 40% (430 projects) contain at least one instance of SBP. Among the 8,570 vulnerabilities, 31% (2,647 vulnerabilities) have at least one vulnerable function with SBP variants. Additionally, among the 22,471 vulnerable functions collected, 18% (3,945 vulnerable functions) have corresponding SBP variants. These results demonstrate the significant presence of SBP, emphasizing the importance of addressing the challenge of SBP.

B. Programming Language Agnostic

The SBP dataset is only collected from C, C++, and Java projects, however, the core design of FVF is programming language agnostic, and FVF can easily extend to other programming languages. In FVF, generation of the *function change log* and *patch log* only relies on the history tracking feature of version control systems (e.g., Git and SVN), which is a general feature regardless of programming languages. The matcher module applies log comparison, which is also not limited to specific languages. Hence, FVF is compatible with any programming language as long as the source code is managed with the standard version control system.

C. Impact of SBP code on Deep Learning-based Vulnerability Detection Approaches

DL-based vulnerability detection approaches have shown promising performance in controlled lab environments. However, the SBP cases are not well considered in the previous work. In the SBP dataset, the vulnerable code and the SBP

TABLE IV
PERFORMANCE OF STATE-OF-THE-ART DL-BASED VULNERABILITY DETECTION TECHNIQUES ON THE SBP DATASET.

Approach	Token-Based			Graph-Based	
	LineVul	**V-CNN**	**V-MLP**	**Devign**	**IVDetect**
False Alarms	6,822	6,827	6,644	2,743	2,388
FA Rate(%)	**63.4**	**64.4**	**63.5**	**64.9**	**62.9**

code share subtle differences, but the vulnerability just manifests itself in these differences.

In the experiment, we evaluate two token-based and two graph-based state-of-the-art DL-based vulnerability detection approaches with the SBP dataset. Similar to RQ1 (Section III-D), we employ FA and FAR as metrics. We select the following state-of-the-art DL-based approaches:

LineVul [19] utilizes a transformer-based language model to generate vector representations, enabling both the prediction of vulnerable functions and the localization of vulnerable lines.

VulBERTa [20] utilizes RoBERTa, a transformer-based deep learning model with a custom tokenization pipeline to detect vulnerabilities. It has two variants: a convolution neural network (V-CNN) and a multilayer perceptron (V-MLP).

Devign [23] utilizes a gated graph neural network (GGNN) to learn program dependencies features on code property graphs, enabling effectively detection of function vulnerabilities.

IVDetect [25] utilized a sliding window technique combined with an interpretable graph neural network to detect vulnerabilities and provide fine-grained explanations. It extracts five different scale features from the code and archives state-of-the-art performance on multiple datasets.

For all techniques, we use the pre-trained model weights provided by the authors and the default parameters mentioned in the original papers. As these deep learning models are designed mainly for C/C++, we only use the C and C++ data (about 97% of all).

The evaluation results are presented in Table IV. The average false alarm rate (FA Rate) of all models in the SBP dataset is as high as 63.8%. Specifically, token-based models predict almost all SBP cases as vulnerable. For example, the VulBERTa-CNN model predicts that all SBP samples are vulnerable. For the graph-based models, 64.9% (2,743) of the Devign predicted positives are SBP code, and 62.9% (2,388) of IVDetect predicted positives are SBP code.

The results demonstrate that the DL-based approaches also fail to distinguish the SBP code accurately and the SBP code poses large challenges to these approaches. Therefore, the SBP dataset can serve as a benchmark for evaluating the sensitivity of existing vulnerability detection approaches to accurately identify real vulnerabilities among the SBP data.

D. Time and Efficiency

In this section, we discuss the efficiency and overhead of FVF. FVF leverages the standard software version control system to retrieve *function change log*s, queries *patch logs* from the vulnerability feature database, and then uses a highly efficient code change log matching to match the two log sequences. The key overhead of FVF is the process of generation of function change logs. In a large repository with numerous commits, retrieving histories for specific functions or files is usually a heavy operation, which slows down the overall process. However, without sufficient change logs, FVF may fail to filter false alarms. Therefore, the size of the retrieval window is critical for the accuracy and efficiency of the algorithm. For this reason, we conduct an empirical analysis to determine the optimal retrieval window size.

TABLE V
THE TIME EFFICIENCY AND PERFORMANCE UNDER DIFFERENT THRESHOLDS OF RETRIEVAL WINDOW SIZE. (FA: FALSE ALARM, FAR: FALSE ALARM RATE, IR: IMPROVEMENT RATE, AVG.: AVERAGE TIME.)

Threshold	FA	FAR(%)	IR(%)	Time(s)	Avg.(s)
3	197	31.9	68.1	168	0.27
10	149	24.2	75.9	182	0.30
20	129	20.9	79.1	192	0.31
50	117	19.0	81.0	245	0.40
55	109	17.7	82.3	257	0.42

Figure 8 presents the empirical cumulative distribution function (ECDF) of required retrieval times of filtered false alarms in RQ1, which demonstrates that all filtered false alarms can be filtered within a maximum retrieval window size of 55. Additionally, the majority (93%) of successful cases are filtered by retrieving only a few (≤ 20) change histories. The "hard" cases, which require more retrievals and time, only constitute a minority ($\leq 7\%$) of the overall cases.

Then we experiment to determine the best retrieval window size for optimal performance and efficiency. We use the false alarm data in RQ1 (Section IV) and then reapply FVF under different maximum retrieval window size settings to test its performance. We use the same metrics FA, FAR, and IR. Additionally, we also measure the elapsed time and compute the average processing time for each case.

The results, as shown in Table V, indicate that with a larger retrieval window, the improvement rate increases, while the time and the average time are longer. Considering this trade-off between efficiency and performance, we empirically recommend a maximum retrieval window size of 50.

E. Extensibility of FVF

Similar to prior works, FVF is primarily designed for single-function similar vulnerability detection. However, FVF can also be extended to vulnerabilities across multiple functions (i.e., the inter-procedural vulnerabilities). Since the core design of FVF is to utilize the fix behavior, to detect similar inter-procedural vulnerabilities, FVF can further utilize all fix behaviors of relevant vulnerable functions to generate the multi-function patch log for the inter-procedural vulnerabilities, which we plan as future work.

VI. THREATS TO VALIDITY

Internal validity. Threats to internal validity are associated with bias and errors in the experiment. One potential threat is the absence of a ground truth in RQ1, where the predicted results are manually analyzed, potentially introducing bias. To

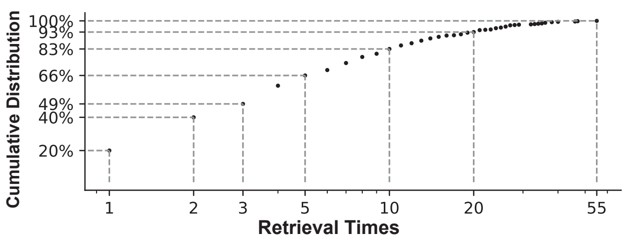


Fig. 8. ECDF of required retrieval times of filtered FAs.

mitigate the bias, we engage security professionals with at least three years of experience in software security to conduct the manual analysis. Another potential threat is the accuracy of the SBP dataset. To mitigate the threat, we conduct a manual verification process on a random sample of 364 SBP functions, with a confidence level of 95%. Another threat is the comprehensiveness of our vulnerability feature database. Vulnerabilities can be fixed through custom fixes that are different from the records in the CVE database. To mitigate the threat, we gather vulnerability fix commits from various public vulnerability databases and datasets to enhance our database.

External validity. Threats to external validity are related to the quality of vulnerability data. Previous research reveals that developers often group multiple changes into a single commit, resulting in a tangled code change [49], which produces large noise and bias. In FVF, we remove VFCs that modify more than 10 files or only modify comments, white spaces, or test/log files to mitigate the threat. This could still be improved using commit untangling techniques such as [50]. Another threat is associated with the selection of tested projects. In this study, we select projects with extensive branches and forks as our targets, but there may be better factors, we encourage future studies to investigate more factors in selection.

VII. Related Work

Clone-Based Vulnerability Detection. Various approaches have been proposed for detecting recurring or similar vulnerabilities [10]–[13], [51]–[56]. For example, Jiang et al. [57] and Kim et al. [11] consider code as token sequences to detect vulnerability clones. Bowman et al. [56] leverage code property graphs to enhance the robustness of the matching algorithm against code modifications. Xiao et al. [13] consider code snippets that match the vulnerable version and not match the fixed version as recurring vulnerabilities. Kang et al. [30] use taint analysis traces to detect recurring vulnerabilities but are limited to taint-style C/C++ vulnerabilities and do not incorporate code change history. Woo et al. [55] consider the oldest, disclosed, and patched versions of the vulnerable function to generate more robust signatures to find modified vulnerable code clones. However, these works focus only on the code snippets and overlook the extensive information contained in the change histories, resulting in many false alarms in clone-based vulnerability detection. In contrast, our work pays attention to the fix reversions of vulnerabilities in real-world scenarios, utilizes a broader history to make more comprehensive decisions, and significantly reduces the number of false alarms.

Deep Learning-Based Vulnerability Detection. Deep learning-based approaches have made periodic achievements in vulnerability detection [19], [20], [22], [23], [25], [58]–[64]. These works typically utilize deep neural networks to learn vulnerable patterns from various forms of code representation, such as lexical tokens [19], [20], program dependence graphs [58]–[61], [63], and a mixture of multiple representations [22], [23], [25], [64]. LineVul [19] employs a transformer-based architecture to generate vulnerability representations for line-level vulnerability detection. Devign [23] introduces GGNN to learn data and control dependencies features of vulnerable code. Shi et at. [62] train a graph convolutional network (GCN) on historically vulnerable functions and correlations to detect cloned vulnerabilities in downstream operating system distributions. Concoction [64] extracts both static and dynamic features of the code and uses a bidirectional Transformer network for vulnerability detection. However, these works often struggle to distinguish subtle differences between vulnerabilities and their corresponding fixed versions and therefore showed poor performance on SBP cases (see experimental results in Section V-C).

Other Kinds of Vulnerability Detection Approaches. Various works and techniques are also available in the field of vulnerability detection. Static analysis-based approaches [65]–[68] detect vulnerabilities through induction of possible variable values. Fuzzing-based approaches [69]–[71] aim to crash the program using random input to uncover vulnerabilities. Symbolic execution-based approaches [72]–[74] explore feasible execution paths by symbolizing variables to assist testing. Compared to these techniques, FVF is lightweight and does not require compiling or executing the code, which is a costly operation. Furthermore, FVF can also be complemented with these techniques. After eliminating SBP-style false alarms, other techniques can be applied for further validation.

VIII. Conclusion

In this paper, we focus on the SBP phenomenon in vulnerability detection. We propose a new programming language agnostic framework, FVF, to identify SBP cases and reduce false alarms in vulnerability detection. Our evaluation conducted with four cloned-based vulnerability detection tools and across nine versions of the Linux and the Redis project demonstrates that FVF can significantly reduce false alarm rates.

We further apply FVF to 1,081 real-world projects and construct a real-world SBP dataset containing 6,827 SBP functions. Using the dataset, we demonstrate the ineffectiveness of state-of-the-art DL-based vulnerability detection approaches in distinguishing SBP data. The dataset can help developers make a more realistic evaluation of existing vulnerability detection approaches and also paves the way for further exploration of real-world SBP scenarios.

Acknowledgments

This research is supported by the Ningbo Natural Science Foundation (No. 2023J292).

References

[1] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: a map of code duplicates on github," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[2] "CVE-2017-12652," https://nvd.nist.gov/vuln/detail/CVE-2017-12652, 2017.

[3] "Libpng: Portable network graphics support, official libpng repository," https://github.com/pnggroup/libpng, 2023.

[4] D. Reid, M. Jahanshahi, and A. Mockus, "The extent of orphan vulnerabilities from code reuse in open source software," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2104–2115.

[5] X. Tan, Y. Zhang, J. Cao, K. Sun, M. Zhang, and M. Yang, "Understanding the practice of security patch management across multiple branches in oss projects," in *Proceedings of the 31st ACM Web Conference (WWW)*, 2022, pp. 767–777.

[6] Z. Jiang, Y. Zhang, J. Xu, Q. Wen, Z. Wang, X. Zhang, X. Xing, M. Yang, and Z. Yang, "Pdiff: Semantic-based patch presence testing for downstream kernels," in *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020, pp. 1149–1163.

[7] S. Farhang, M. B. Kirdan, A. Laszka, and J. Grossklags, "Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities," *arXiv preprint arXiv:1905.09352*, 2019.

[8] V. H. Nguyen, S. Dashevskyi, and F. Massacci, "An automatic method for assessing the versions affected by a vulnerability," *Empirical Software Engineering (EMSE)*, vol. 21, pp. 2268–2297, 2016.

[9] "Nfc: netlink: fix sleep in atomic bug when firmware download timeout," https://github.com/torvalds/linux/commit/4071bf121d, 2024.

[10] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 48–62.

[11] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.

[12] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd annual conference on computer security applications*, 2016, pp. 201–213.

[13] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "{MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1165–1182.

[14] "CVE-2022-1975," https://nvd.nist.gov/vuln/detail/CVE-2022-1975, 2022.

[15] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated patch backporting in linux (experience paper)," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 633–645.

[16] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proceedings of 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 19–29.

[17] "Linux kernel source tree," https://github.com/torvalds/linux, 2024, accessed: 2024-03-02.

[18] "Redis is an in-memory database that persists on disk," https://github.com/redis/redis, 2024, accessed: 2024-03-02.

[19] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[20] H. Hanif and S. Maffeis, "Vulberta: Simplified source code pre-training for vulnerability detection," in *2022 International joint conference on neural networks (IJCNN)*. IEEE, 2022, pp. 1–8.

[21] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023.

[22] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[23] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[24] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.

[25] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 292–303.

[26] "Our replication package," https://github.com/Zixuan-Tan/FVF , 2024.

[27] "CVE-2018-7191," https://nvd.nist.gov/vuln/detail/CVE-2018-7191, 2018.

[28] "tun: call dev_get_valid_name() before register_netdevice()," https://github.com/torvalds/linux/commit/0ad646c81b, 2018.

[29] "tun: allow positive return values on dev_get_valid_name() call," https://github.com/torvalds/linux/commit/5c25f65fd1, 2018.

[30] W. Kang, B. Son, and K. Heo, "Tracer: Signature-based static analysis for detecting recurring vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1695–1708.

[31] "Git - git-log documentation," https://git-scm.com/docs/git-log, 2024, accessed: 2024-03-02.

[32] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[33] "Cve - mitre," https://cve.mitre.org/, 2024, accessed: 2024-03-02.

[34] D. Spadini, M. Aniche, and A. Bacchelli, *PyDriller: Python Framework for Mining Software Repositories*, 2018.

[35] "Asahi linux - linux on apple silicon," https://asahilinux.org/, 2024, accessed: 2024-03-02.

[36] "lkl/linux: Linux kernel source tree," https://github.com/lkl/linux, 2024, accessed: 2024-03-02.

[37] "Openharmony/kernel_linux_5.10," https://gitee.com/openharmony/ kernel_linux_5.10, 2024, accessed: 2024-03-02.

[38] "bmerry/birdisle: Birdisle is a modified version of redis that runs as a library inside another process." https://github.com/bmerry/birdisle, 2024, accessed: 2024-03-02.

[39] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[40] "Iotcube - cssa," https://iotcube.net/process/type/wf1, 2024, accessed: 2024-03-02.

[41] "crypto: ghash - avoid null pointer dereference if no key is set," https://github.com/torvalds/linux/commit/7ed47b7d14, 2011.

[42] "Cve-2011-4081," https://nvd.nist.gov/vuln/detail/CVE-2011-4081, 2011.

[43] "ath9k: Fix use-after-free read in htc_connect_service," https://github.com/torvalds/linux/commit/ced21a4c72, 2022.

[44] "ath9k_htc: release allocated buffer if timed out," https://github.com/torvalds/linux/commit/853acf7caf, 2019.

[45] "Cve-2019-19073," https://nvd.nist.gov/vuln/detail/CVE-2019-19073, 2019.

[46] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 471–482.

[47] J. Shimagaki, Y. Kamei, S. McIntosh, D. Pursehouse, and N. Ubayashi, "Why are commits being reverted?: A comparative study of industrial and open source projects," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 301–311.

[48] X. Wang, S. Wang, P. Feng, K. Sun, and S. Jajodia, "Patchdb: A large-scale security patch dataset," in *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 149–160.

[49] K. Herzig and A. Zeller, "The impact of tangled code changes," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 121–130.

[50] M. Wang, Z. Lin, Y. Zou, and B. Xie, "Cora: Decomposing and describing tangled code changes for reviewer," 11 2019, pp. 1050–1061.

[51] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[52] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 447–456.

[53] L. Cui, Z. Hao, Y. Jiao, H. Fei, and X. Yun, "Vuldetector: Detecting vulnerabilities using weighted feature graph comparison," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2004–2017, 2020.

[54] S. Salimi and M. Kharrazi, "Vulslicer: Vulnerability detection through code slicing," *J. Syst. Softw.*, vol. 193, p. 111450, 2022.

[55] S. Woo, H. Hong, E. Choi, and H. Lee, "{MOVERY}: A precise approach for modified vulnerable code clone discovery from modified {Open-Source} software components," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3037–3053.

[56] B. Bowman and H. H. Huang, "Vgraph: A robust vulnerable code clone detection system using code property triplets," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 53–69.

[57] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105, 2007.

[58] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[59] D. Zou, Y. Zhu, S. Xu, Z. Li, H. Jin, and H. Ye, "Interpreting deep learning-based vulnerability detector predictions based on heuristic searching," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–31, 2021.

[60] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

[61] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2821–2837, 2021.

[62] H. Shi, R. Wang, Y. Fu, Y. Jiang, J. Dong, K. Tang, and J. Sun, "Vulnerable code clone detection for operating system through correlation-induced learning," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 12, pp. 6551–6559, 2019.

[63] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2020.

[64] H. Wang, Z. Tang, S. H. Tan, J. Wang, Y. Liu, H. Fang, C. Xia, and Z. Wang, "Combining structured static code information and dynamic symbolic traces for software vulnerability prediction," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[65] "Clang static analyzer," https://clang-analyzer.llvm.org/, 2024, accessed: 2024-03-06.

[66] "Cppcheck - a tool for static c/c++ code analysis," https://cppcheck.sourceforge.io/, 2024, accessed: 2024-03-06.

[67] "Flawfinder," https://dwheeler.com/flawfinder/, 2024, accessed: 2024-03-06.

[68] K. Lu, A. Pakki, and Q. Wu, "Detecting {Missing-Check} bugs via semantic-and {Context-Aware} criticalness and constraints inferences," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1769–1786.

[69] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 725–741.

[70] "The afl++ fuzzing framework — aflplusplus," https://aflplus.plus/, 2024, accessed: 2024-03-02.

[71] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities," in *Network and Distributed Systems Security (NDSS) Symposium*, 2023.

[72] S. Anand, C. S. Păsăreanu, and W. Visser, "Jpf–se: A symbolic execution extension to java pathfinder," in *Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24-April 1, 2007. Proceedings 13*. Springer, 2007, pp. 134–138.

[73] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[74] D. A. Ramos and D. Engler, "{Under-Constrained} symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 49–64.