RESEARCH-ARTICLE

PLocator: Fine-Grained Patch Presence Test in Binaries via Patch Code Localization

CHAOPENG DONG, Institute of Information Engineering, Beijing, China

JINGDONG GUO, Institute of Information Engineering, Beijing, China

SHOUGUO YANG, Beijing Institute of Technology, Beijing, China

YANG XIAO, Institute of Information Engineering, Beijing, China

YI LI, Nanyang Technological University, Singapore City, Singapore

HONG LI, Institute of Information Engineering, Beijing, China

View all

Published: 01 October 2025
Accepted: 25 September 2025
Revised: 25 August 2025
Received: 29 January 2025

PLocator: Fine-Grained Patch Presence Test in Binaries via Patch Code Localization

CHAOPENG DONG* and JINGDONG GUO, Institute of Information Engineering, CAS and School of Cyber Security, University of Chinese Academy of Sciences, China
SHOUGUO YANG, Zhongguancun Laboratory, China
YANG XIAO†, Institute of Information Engineering, CAS and School of Cyber Security, University of Chinese Academy of Sciences, China
YI LI, Nanyang Technological University, Singapore
HONG LI, ZHI LI, and LIMIN SUN‡, Institute of Information Engineering, CAS and School of Cyber Security, University of Chinese Academy of Sciences, China

1-day vulnerabilities in binaries have become a major threat to software security. Patch presence test is one of the effective ways to detect the vulnerability. However, existing patch presence test works do not perform well in practical scenarios due to the interference from the various compilers and optimizations, patch-similar code blocks, and irrelevant functions in stripped binaries. In this paper, we propose a novel approach named *PLocator*, which leverages constants from both the patch code and its context, extracted from the control flow graph, to form the anchors and accurately locate the real patch code in the target function, offering a practical solution for real-world vulnerability detection scenarios.

To evaluate the effectiveness of *PLocator*, we collected 73 CVEs and constructed two datasets with and without the irrelevant functions, comprising 1,090 and 27,250 functions, respectively. Moreover, we set three different experiments, i.e., Same, XO (cross-optimizations), and XC (cross-compilers), to evaluate the performance of existing patch presence test approaches and *PLocator*. The results demonstrate that *PLocator* outperforms the second state-of-the-art approach on accuracy by 44.3% (without irrelevant functions) and 74.9% (with irrelevant functions), indicating that *PLocator* is more practical for the patch presence task.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Vulnerability Detection, Patch Presence Test, Binary Code Similarity Detection

*This work was done while the first author was at Nanyang Technological University.

†Corresponding Author
‡Corresponding Author

Authors' Contact Information: Chaopeng Dong, dongchaopeng@iie.ac.cn; Jingdong Guo, guojingdong@iie.ac.cn, Institute of Information Engineering, CAS and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China; Shouguo Yang, yangshouguo@outlook.com, Zhongguancun Laboratory, Beijing, China; Yang Xiao, xiaoyang@iie.ac.cn, Institute of Information Engineering, CAS and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China; Yi Li, yi_li@ntu.edu.sg, Nanyang Technological University, Singapore, Singapore; Hong Li, lihong@iie.ac.cn; Zhi Li, lizhi@iie.ac.cn; Limin Sun, sunlimin@iie.ac.cn, Institute of Information Engineering, CAS and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China.

ACM 1557-7392/2025/10-ART
https://doi.org/10.1145/3770079

1 INTRODUCTION

1-day (disclosed) vulnerabilities have emerged as a significant threat in software systems due to the widespread adoption of open-source software (OSS). According to the CVE data [8], the number of confirmed vulnerabilities has surged from 7,928 to 29,065 over the past decade. As highlighted in Synopsys's report [5], 89% of the codebases incorporate open-source code that has been stagnant for at least four years and components that have not received updates for two years, rendering them highly susceptible to 1-day vulnerabilities.

Given a known vulnerability, e.g., a vulnerable function, the task of *1-Day Vulnerability Detection* aims to locate all homologous vulnerable functions within the target binary (which can be viewed as a set of functions). The difficulty of identifying 1-day vulnerabilities lies in the potentially enormous amount of irrelevant functions within the target binary. To accomplish this task, an efficient and effective technique called *Binary Code Similarity Detection* (BCSD) [16, 17, 19, 21, 22, 25, 27, 28, 30, 32, 36] has been proposed, which compares two or more pieces of binary code to identify their similarities and differences [23]. Given a reference vulnerable function f_{vul}, BCSD retrieves potentially vulnerable functions from a large codebase efficiently with high recall. However, it struggles to distinguish vulnerable functions from fixed ones since the differences between them are often subtle [45], which leads to false positives and false negatives. Thus, *Patch Presence Test* [44] is proposed as a follow-up procedure to address the limitation of BCSD. A patch fixes the vulnerability; therefore, patch presence test is effective only when the detected patch code is related to the corresponding vulnerability. In other words, patch presence reflects the vulnerability status only when the binary is known to be a vulnerable version. Its input normally includes a patch file, two reference functions (i.e., reference vulnerable function f_{vul} and reference fixed function f_{fix}) of the project, and a set of functions to be tested. The features generated from the patch and two reference functions determine whether a known security patch has been applied to the target functions.

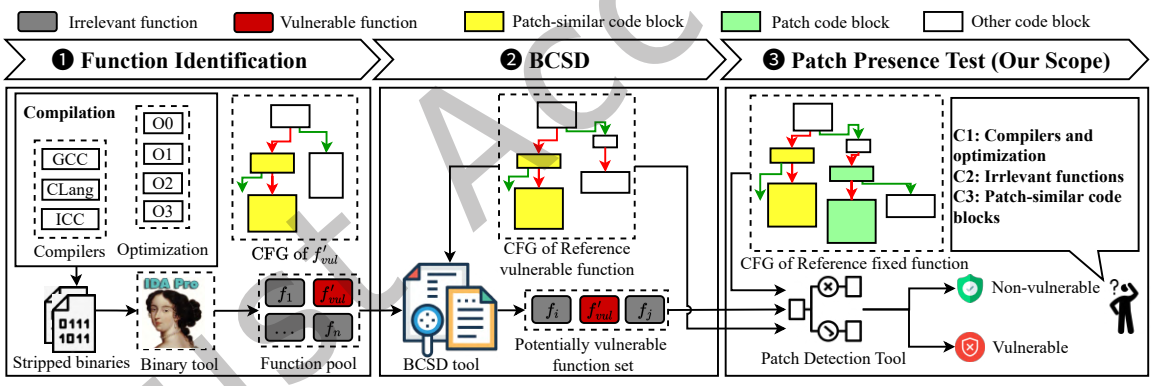


Fig. 1. Pipeline of the 1-day vulnerability detection task for stripped binaries, comprising three steps: ❶ Function Identification (identify function boundaries in binaries), ❷ Function Matching (match the reference vulnerable function with the candidate functions in the function pool), and ❸ Patch Presence Test (distinguishing between vulnerable and non-vulnerable functions via patch detection). The patch-similar (yellow) code blocks present code and structure similar to the patch (green) ones, which mislead the patch detection tool to identify the input vulnerable function f'_{vul} as non-vulnerable, as highlighted in the dashed red rectangle. **C1**, **C2**, and **C3** denote the three core challenges for the patch presence test.

Pipeline of 1-Day Vulnerability Detection. Figure 1 shows the pipeline of the 1-day vulnerability detection task. Given stripped binaries that are compiled from various compilers and optimization levels: **❶ Function Identification.** The boundaries of binary functions are identified by the binary analysis tool (e.g., IDA [4], Ghidra [13]),

thereby constituting a function pool comprising candidate functions; **❷ Binary Code Similarity Detection (BCSD)** A BCSD tool (e.g., jTrans [32]) matches the reference vulnerable function with candidates in the function pool, selecting the top-K most similar ones as the potentially vulnerable function set; **❸ Patch Presence Test.** Finally, the patch detection tool is employed to classify the remaining functions as either vulnerable or non-vulnerable based on the patch code and reference functions.

Our Scope. Owing to advancements in function identification and matching provided by current binary analysis and BCSD tools, our scope mainly focuses on the patch presence test phase (step ❸), which takes the set of potentially vulnerable functions provided by the BCSD tool and outputs their categories as vulnerable or non-vulnerable. Our objective is not to conduct general semantic vulnerability detection, but rather to determine whether a given binary contains a known vulnerability based on a provided patch and two reference functions.

Existing Approaches and Challenges. The current techniques for *patch presence test* still suffer from many limitations when integrated into the vulnerability detection workflow. We roughly divide the existing *patch presence test* techniques into *syntactic-based* and *semantic-based* approaches. Syntactic-based approaches [37, 38, 44] concentrate on the syntactic modifications made in patches. These approaches are efficient, owing to the lightweight comparison over the syntactic features. Semantic-based approaches [24, 41, 43] attempt to capture the semantic information by simulating the program's execution with the symbolic execution technique. During the execution, the side effects (e.g., memory status and constraints) are recorded as semantic information of the target, which is then compared with the references to determine the patch presence. However, the emulation process is time-consuming, making it unsuitable for large-scale testing. Despite the recent advances, three challenges remain in the patch presence test for binaries.

- **C1 Interference from various compilers and optimizations.** Developers frequently utilize diverse compilers and optimizations to satisfy specific requirements when producing binaries, resulting in numerous false negatives for syntactic-based methods. This occurs because the binary code generated with different compilers and optimizations diverges significantly [18]. Syntactic-based methods cannot distinguish whether the differences in binary code are due to compilation settings or patches.
- **C2 Interference from binary strip and irrelevant functions.** Most of the *patch presence test* approaches assume that the binary contains function symbols [24, 43, 44] or the BCSD tool can retrieve the target functions precisely [38] so that they only consider the difference between vulnerable and fixed functions and treat the *patch presence test* as a binary classification task (i.e., vulnerable or fixed). However, we argue that such a hypothesis is problematic, given the fact that for real-world binaries they often strip the symbols and the BCSD tool reports many false positives when the size of the function pool is large [32, 40], as discussed in a preliminary study on BCSD § 2.2. As such, when the input functions are irrelevant to the vulnerability, these *patch presence test* approaches still classify them as vulnerable or fixed (grey rectangles in the "Potentially vulnerable function set" section of Figure 1), which have little meaning and cause many false positives.
- **C3 Interference from patch-similar code blocks.** Code blocks that are similar or identical are common in binaries. For example, the same error handling code (e.g., throwing exceptions or printing errors) may be reused to fix similar vulnerabilities [29]. Therefore, focusing on the patch code without considering the context and the relationship between them may lead to misidentification (i.e., taking vulnerable functions with patch-similar code as fixed), even for semantic-based methods.

Our Approach. To address the challenges, we propose *PLocator* (Patch Locator), an innovative approach that can accurately and efficiently detect 1-day vulnerabilities in binaries.

To tackle **C1**, we use the constants extracted from the two types of key instructions (condition comparison and function call) as anchors, which are stable across compilers and optimizations. To distinguish the key instructions

with the same values, we also extract the constant values associated with the key instruction through data flow dependencies as auxiliary information. We generate the signatures based on the extracted anchors to reduce the interference from various compilers and optimizations.

To tackle **C2**, we first filter irrelevant functions at a coarse-grained level by comparing the matched unique anchor values between the reference functions and target functions. Subsequently, we classify the target functions as irrelevant by examining whether the patch code and its context are matched or not.

To tackle **C3**, we utilize the context of the patch code to eliminate patch-similar code blocks (i.e., yellow blocks mentioned in Figure 1) by verifying whether the matched patch code and context code satisfy the original control flow relationship in the patch.

Evaluation. To evaluate the effectiveness of our work from multiple perspectives, we first collected 73 CVEs and 112 distinct vulnerable functions from four popular projects. And then, we constructed two datasets: dataset without irrelevant functions $\mathcal{D}_{\text{-irr}}$ and dataset with irrelevant functions$\mathcal{D}_{\text{+irr}}$, based on the collected data and compiled binaries at four different optimization levels (i.e., O0 to O3) and two compilers (i.e., GCC and Clang). To evaluate the performance of baselines and *PLocator* in different scenarios, we set three different experiments, i.e., Same, XO (cross-optimization levels), and XC (cross-compilers), and evaluate them both on the two datasets. The experimental results demonstrate that *PLocator* outperforms the second-best state-of-the-art approach on accuracy by 44.3% (without irrelevant functions) and 74.9% (with irrelevant functions), respectively. Additionally, *PLocator* completes the detection process in a short amount of time, averaging 0.14 seconds per function, making it suitable for large-scale detection.

Contributions. Overall, we summarize our contributions as follows:

(1) We systematically outline the pipeline for the 1-day vulnerability detection and present the limitations of existing patch presence test approaches through a preliminary study of the BCSD tool and a motivating example.
(2) To address these challenges, we propose *PLocator*, a novel approach that leverages constants in both the patch code and its surrounding context, capable not only of accurately distinguishing between vulnerable and fixed functions, but also of substantially reducing the manual effort required to verify detected vulnerabilities by mitigating interference from irrelevant functions. We implement a prototype of *PLocator* and release the code and dataset at https://github.com/GentleCP/PLocator-public.
(3) We collect the original data from NVD, comprising 73 CVEs and 112 distinct vulnerable functions, and construct two datasets using four optimization levels and two compilers, generating a substantial number of test cases to evaluate the effectiveness of various patch presence test approaches across three experiments. The results demonstrate the effectiveness of *PLocator* in accurately and efficiently detecting 1-day vulnerabilities.

2 BACKGROUND AND MOTIVATION

In this section, we begin by presenting the key terms and definitions employed throughout the paper to ensure clarity and ease of reference. We then provide a summary of related patch presence approaches and highlight their limitations. Subsequently, we conduct a preliminary study to expose the limitations of BCSD and demonstrate the necessity of the patch presence test. Finally, we provide a motivating example to elucidate the rationale behind our approach.

2.1 Terms and Definitions

1-day vulnerability and 1-day vulnerability detection. 1-day vulnerabilities refer to publicly disclosed vulnerabilities in software for which patches are available but have not yet been applied [15]. 1-day vulnerability

detection aims to identify the presence of such vulnerabilities within software. Specifically, our study targets the detection of 1-day vulnerabilities in binaries.

Patch and patch code. A patch addresses a vulnerability and is typically released upon the disclosure of a 1-day vulnerability. The patch code refers to a concise segment of code within the patch, encompassing both the removed and newly added statements.

Reference functions. The reference functions are those used to generate features for the patch presence test, comprising two types: the vulnerable reference function and the fixed reference function. The former is a binary function extracted from the vulnerable binary (before the patch applied), while the latter is extracted from the fixed binary (with the patch applied).

Patch Presence Test. One of the techniques for 1-day vulnerability detection that determine the presence or absence of a patch in corresponding binary functions. The input includes a patch, two reference functions, and one or multiple binary functions under examination, while the output is the classification of the functions as either vulnerable or non-vulnerable. The assumption that patch presence indicates vulnerability status holds only when the binary is already known to contain the vulnerable function. This restriction is necessary because the presence or absence of a patch has meaning only if the vulnerable code exists in the first place; otherwise, patch absence does not imply vulnerability.

Context code. The code associated with the patch code that within the same control flow path. Based on its execution order relative to the patch code, it can be further classified into backward context code (executed before the patch code) and forward context code (executed after the patch code).

Anchor and auxiliary information. A stable data structure that remains consistent across varying compilers and optimization levels, composed of an anchor value v, an anchor type t, and auxiliary information aux, denoted as $\mathcal{A} = \langle v, t, aux \rangle$. Auxiliary information is a sequence of constants that associated with the anchor through data dependencies, denoted as $aux = \langle (v_1, t_1), \ldots, (v_n, t_n) \rangle$, where v_i and t_i represent the value and type of the i^{th} constant, respectively.

Anchor graph. A directed graph consisting of anchors that follow the control flow of the control flow graph (CFG), denoted as $AG = (V, E)$, where $V = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ represents the set of anchors, and $E = \{(\mathcal{A}_i, \mathcal{A}_j) \mid \mathcal{A}_i \in V, \mathcal{A}_j \in V\}$ denotes the set of edges connecting the anchors.

Anchor path. A sequence of anchors following the control flow within AG, employed to represent the patch code and context code for matching, denoted as $AP = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle, \mathcal{A}_i \in AG$. We refer to the anchor path extracted from the patch code as the patch (anchor) path, the one derived from the backward context code as the backward context (anchor) path, and the one obtained from the forward context code as the forward context (anchor) path, respectively.

Candidate functions and function pool. Candidate functions refer to those extracted from stripped binaries, forming the function pool for BCSD.

Target functions. Functions within the potentially vulnerable function set retrieved by the BCSD tool. As *PLocator* is dedicated to the patch presence test, these functions are designated as the targets to be tested.

Called Functions. The set of subfunctions invoked within the body of the function.

2.2 Preliminary Study on BCSD

To expose the limitations of BCSD on 1-day vulnerability detection and the interference from the irrelevant functions for patch presence test, we first conduct a preliminary study on jTrans, i.e., a state-of-the-art BCSD approach, by matching the reference vulnerable functions against a large function pool across different compilation optimizations. The details of the experimental setup are presented in § 5.1. We treat vulnerable functions as positive, fixed, and irrelevant functions as negative, and employ precision (P), recall (R), and F1-score (F1) as the

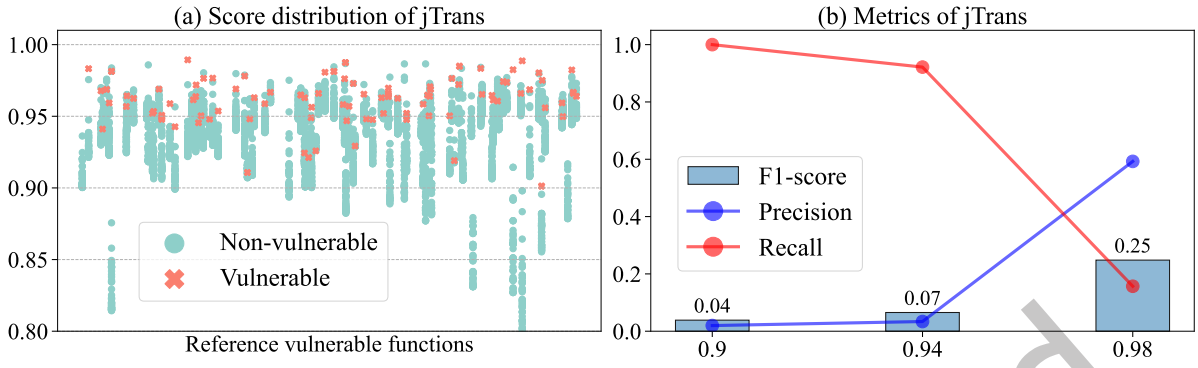


Fig. 2. The results of a preliminary study on BCSD, conducted using jTrans across various compilation optimization levels. Figure (a) illustrates the distribution of the BCSD similarity scores for different functions, where the x-axis represents different reference vulnerable functions, and the y-axis denotes the similarities of the top-50 most similar functions per reference function retrieved from a function pool containing 6,000 candidates. Figure (b) illustrates jTrans's performance metrics across varying similarity thresholds (x-axis); functions exceeding the threshold are classified as "vulnerable", whereas those below it are considered "non-vulnerable".

evaluation metrics, defined as follows:

$$P = \frac{TP}{TP+FP}, R = \frac{TP}{TP+FN}, F1 = \frac{2 \cdot P \cdot R}{P+R} \tag{1}$$

where TP, FP, and FN denote the number of vulnerable functions correctly identified as vulnerable, non-vulnerable functions mistakenly classified as vulnerable, and vulnerable functions incorrectly classified as non-vulnerable, respectively.

Figure 2 illustrates the similarity distribution obtained by jTrans and the corresponding evaluation metrics under varying BCSD thresholds. As shown, jTrans effectively recalls the vulnerable functions but erroneously includes a substantial number of non-vulnerable functions in the retrieved set, as mentioned in § 1. Increasing the threshold in Figure 2b mitigates false positives but significantly compromises recall, which is unacceptable. This underscores the necessity for the patch presence test approach to precisely differentiate between vulnerable and non-vulnerable (including fixed and irrelevant) functions as a critical downstream task of BCSD.

2.3 Motivating Example

We use CVE-2014-3470 [1], a vulnerability of the famous cryptographic protocols software OpenSSL, to motivate our approach, as depicted in Figure 3. The patch code in Figure 3a introduces a check on variable "`s->session->sess_cert`" and alerts error with two function calls. Figure 3b to Figure 3d presents the CFGs of fixed reference function, two vulnerable functions compiled with gcc-O0, gcc-O3 and Clang-O0, respectively.

Limitations of Existing Approaches. When detecting vulnerabilities in the two vulnerable functions, we found that both types of patch presence test approaches exhibit limitations.

(1) *Syntactic-based approaches struggle to discern whether changes are introduced by the patch or compiler options.* These methods typically normalize instructions by abstracting addresses (e.g., `jmp 0x8048EF` → `jmp address`), memory (e.g., `mov [ebp], edx` → `mov mem, edx`), and registers (e.g., `mov ebp, esp` → `mov reg, reg`) to mitigate compilation differences. However, binary code with different compilers and optimization levels may implement the same source code with distinct lines of code and totally different

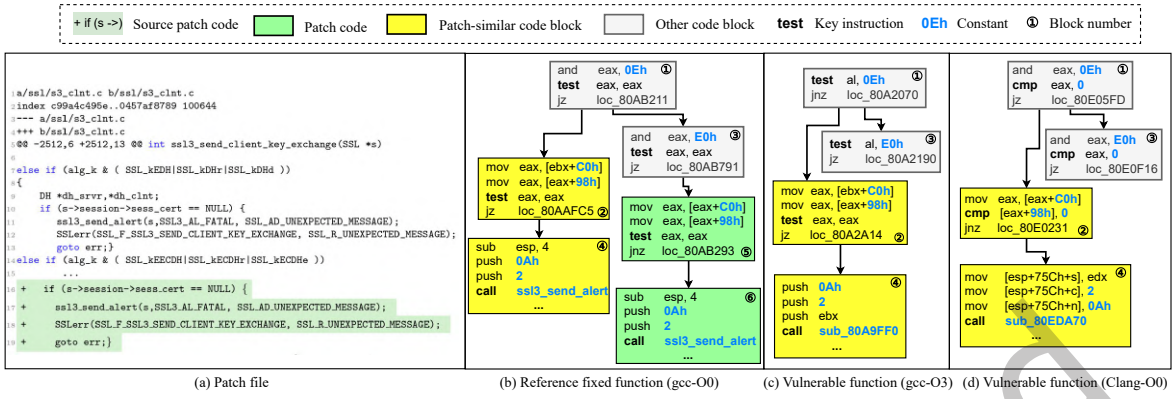


Fig. 3. Motivating example for CVE-2014-3470. Figure (a) shows the patch file, where the patch introduces a conditional check and an error-handling function call, highlighted with a green background. Figure (b) depicts the fixed version of the function compiled with GCC at O0: green blocks indicate the actual patch code, and orange blocks represent patch-similar code. Figures (c) and (d) show two vulnerable versions of the function, compiled with GCC-O3 and Clang-O0, respectively. In all binary figures, key instructions (conditional comparison and function call) are marked in bold, and their associated constants are highlighted in bold blue for clarity.

instructions. For example, for the same source code statement (line 14 in Figure 3a), reference fixed function (Figure 3b) implements it with three instructions in block ① and the comparison instruction (`test eax, eax` in ③). In contrast, the vulnerable function (Figure 3c) implements the same operation with only two instructions in ①, and the vulnerable function (Figure 3d) utilizes a different comparison instruction (`cmp eax, 0`). Such variability undermines stability when relying solely on normalized instructions.

(2) *Semantic-based approaches fail to distinguish patch-similar code from the real patch code due to their identical semantic features, resulting in misidentification.*. For example, the two yellow code blocks in Figure 3b present similar assembly code as the green code blocks (i.e., patch). Methods that extract semantic features from the green code blocks mistakenly identify the yellow blocks as the patch code, leading to the incorrect classification of the two vulnerable functions as fixed.

Observations and Illustration. We present two major observations to address the aforementioned limitations.

- **Observation 1.** By comparing the CFGs of three functions carefully, we found that two types of key instructions (condition comparison and function call) and the constants associated with them through data dependencies remain consistent across compilers and optimizations, highlighted in **bold text**. It is worth noting that not all constants within functions remain stable across different compilation settings. For instance, in the reference fixed function (Figure 3b, block ④), a stack operation (`sub esp, 4`) is explicitly executed, followed by parameter pushing before the call to `ssl3_send_alert`. In contrast, the vulnerable function (Figure 3d) moves parameters onto the stack directly using the `esp` register, resulting in the absence of the constant 4. Instead, we only extract constants that are associated with the key instruction through data dependencies to avoid such cases.
- **Observation 2.** By inspecting the green (patch) and yellow code blocks in Figure 3b, we found that their distinguishing factor lies in their prerequisites, wherein one involves an `and` operation with **0Eh**, while the other is **E0h**.

These two observations motivate us to leverage both the patch and its contextual information, specifically, the constants associated with key instructions, to locate the patch code and verify its presence. However, to ensure that the selected key instructions are suitable for the patch presence test, it is essential to assess their *stability* (i.e., consistency across compilers and optimization levels) and *universality* (i.e., prevalence across real-world patches). To this end, we conduct the following empirical study:

(1) **Stability.** To evaluate stability, we select 224 distinct vulnerable and fixed functions from our dataset. For each function, we randomly choose two vulnerable functions compiled with different compilers and optimization levels. We then match the key instructions across vulnerable functions using debug information by mapping their addresses to source code line numbers and comparing their presence. We compute the matching proportion as $\phi = \frac{M}{N}$, where M is the number of matched key instructions and N is the total number of key instructions to be matched. The results show a median matching proportion (ϕ) of 94.2%, demonstrating the high stability of key instructions.
(2) **Universality.** To evaluate universality, we manually analyze the patches of 73 CVEs in our dataset, counting the number of patches that introduce modifications involving condition comparisons or function calls. The results indicate that 98.6% of the patches contain at least one such modification, confirming the widespread occurrence of key instructions in patch code.

Compared to the stability and universality of constants in key instructions, other potential features exhibit low reliability and are therefore excluded from consideration. For instance, memory access patterns can vary significantly across different compilers and optimization levels. To illustrate, initializing a variable with $s = 0$ may generate the instruction ``mov DWORD PTR [ebp-4], 0'' under O0, whereas under O2/O3, it may be replaced with ``xor eax, eax'', leading to divergent memory access behaviors. Similarly, structural features such as loops demonstrate low universality in patches; only 6.8% of the 73 analyzed patches involve modifications to loop structures.

With the above observation and confirmation of the key instruction's stability across compilers and optimization levels and universality in the patches, we illustrate the process of *PLocator* for patch presence test as follows:

Signature generation. To determine whether the two vulnerable functions contain patch code or not, we first locate the patch code in the reference function (Figure 3b), extracted from the reference binaries with debug information preserved. Then we extract the anchors from the patch code (i.e., bold blue text in blocks ⑤ and ⑥). Subsequently, we further extract the anchors in blocks ① and ③ of the patch as the context.

Patch Detection. Given the two vulnerable functions extracted from the target stripped binaries, where debug symbols are absent, we match the anchors of the patch and its context in two vulnerable functions (Figure 3c and Figure 3d). For function calls without symbols (e.g., **sub_80A9FF0**), we match them with the BCSD tool (e.g., jTrans) by limiting the search space to a small range to reduce false positives. More details are discussed in § 4.3.2. Afterward, the patch-similar code (i.e., yellow blocks) will be filtered out since it does not satisfy the original control flow relationship between the patch and its context (i.e., patch block ② does not follow the execution after context block ③). Thus, we determine that the two vulnerable functions are vulnerable.

3 APPLICATION SCENARIO OF *PLOCATOR*

PLocator proves especially valuable in real-world scenarios where source code is unavailable and only binaries are provided.

Vulnerability Analysts: Analysts often examine firmware images from IoT devices (e.g., routers) where vendors do not disclose source code. Assessing whether a vulnerability has been patched thus necessitates binary-level analysis. In this case, *PLocator* can be utilized to detect vulnerable functions, significantly alleviating their workload.

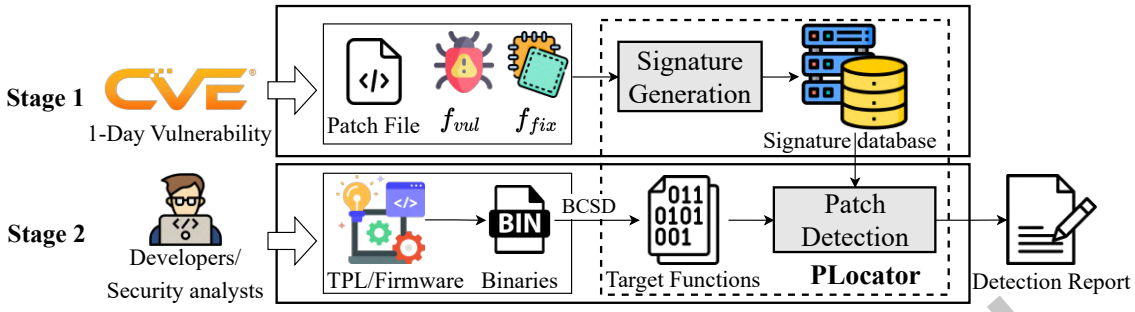


Fig. 4. The application scenario of *PLocator*, consisting of two stages. The dashed rectangle highlights the components executed by *PLocator*. In **Stage 1**, *PLocator* analyzes known 1-day vulnerabilities and builds the signature database. In **Stage 2**, given binaries extracted from TPLs or firmware, *PLocator* collaborates with a BCSD tool to retrieve target functions and classifies them as vulnerable or non-vulnerable by matching against the signature DB.

Developers and Third-Party Libraries: Developers typically have access only to their own project's source code. Yet modern software development relies heavily on third-party libraries (TPLs), as mentioned in [37, 42], many of which are distributed solely in binary form. In this case, developers can leverage *PLocator* to detect vulnerabilities within these TPLs before integrating them into their software ecosystems.

As illustrated in Figure 4, the application involves two key stages. In **Stage 1**, *PLocator* constructs a vulnerability signature database from vulnerability data obtained from NVD [6]. Using inputs such as patch files and reference functions (vulnerable and fixed), *PLocator* extracts anchor-based signatures and stores them in a centralized feature database. This stage can be performed once by trusted maintainers or vulnerability researchers. In **Stage 2**, *PLocator* is used to scan vulnerabilities in binaries extracted from the software. Given binaries, *PLocator* collaborates with an existing BCSD tool (e.g., jTrans) to retrieve target functions and then classifies each one as either vulnerable or non-vulnerable by matching them against the signatures in the database. This process produces a vulnerability report that alerts users to the presence of unpatched 1-day vulnerabilities in their software, without requiring access to source code.

4 METHODOLOGY

4.1 Overview

The goal of *PLocator* is to locate the patch code in a set of binary functions with the given patch and two reference functions, and then determine the categories of the target functions.

Figure 5 depicts the workflow of *PLocator*, comprising two main stages: *Signature Generation* (Offline) and *Patch Detection* (Online).

During the signature generation, *PLocator* aims to generate the signatures based on the two reference functions extracted from the compiled binaries and a given patch file. First, ❶ *PLocator* maps the patch code into the corresponding control flow graphs (CFGs) of two reference functions (f_{vul} and f_{fix}) based on the given patch file and debug information in compiled binaries. Second, ❷ *PLocator* constructs the anchor graphs for and ❸ extracts the anchor paths to form the signature for both reference functions.

During the patch detection stage, *PLocator* preprocesses the input target function by constructing the target anchor graph (AG) in the same manner as the previous stage. Subsequently, *PLocator* ❹ filters out irrelevant functions based on the set of anchors in two reference functions at a coarse-grained level. After that, ❺ *PLocator* matches the extracted anchor paths in the signatures within the target AG, and ❻ verifies the matched patch

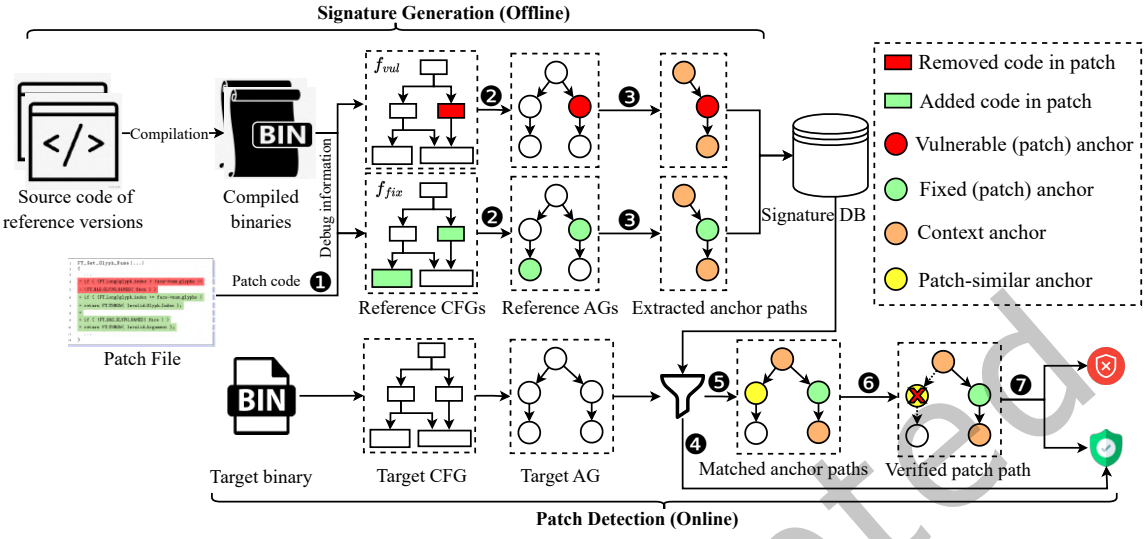


Fig. 5. The workflow of *PLocator* comprises two main stages (signature generation and patch detection). In signature generation (offline), there are three main steps: ❶ patch code mapping (map the patch in the patch file into the CFG blocks), ❷ anchor graph construction (construct anchor graph (AG) from the CFG), and ❸ anchor path extraction (extract patch-related anchor paths in the anchor graph). In patch detection (online), there are four main steps: ❹ irrelevant function filtering (filter out irrelevant functions directly), ❺ anchor path matching (match the extracted anchor paths within the target AG), ❻ patch path verification (verify the matched patch path to eliminate patch-similar code), and ❼ function classification (classify the target function into vulnerable or non-vulnerable).

path to eliminate the patch-similar code. Ultimately, ❼ *PLocator* classifies the functions as either vulnerable or non-vulnerable based on the verified anchor paths between the two reference functions and the target function.

4.2 Signature Generation

Signature generation generates representative features of the patch code and its context code based on the patch file and two reference functions. The process comprises three steps: patch code mapping, anchor graph construction, and anchor path extraction.

4.2.1 Patch Code Mapping. Patch code mapping aims to pinpoint the patch code in reference binary functions with the given patch file (i.e., deleted code in the reference vulnerable function f_{vul} and added code in the reference fixed function f_{fix}) and debug information from the reference binaries.

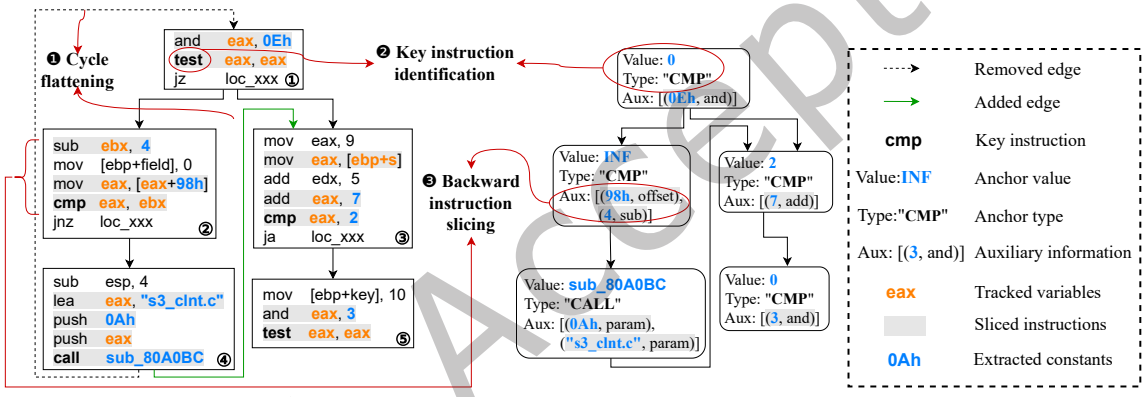
To achieve the object, we first standardize and compute the hash value [12] of the source code lines in the patch file and two reference functions to minimize the influence of non-semantic modifications and improve the comparison efficiency, which is accomplished by eliminating all comments, braces, tabs, and white spaces, similar to previous studies [33–35]. Subsequently, we split the patch code into two parts (deleted and added), and map them to the source code of the reference versions. Considering that identical code statements may recur within a function (e.g., `goto error`), we pinpoint the real patch code with its context source code in the patch file. Ultimately, for the mapped source code, we employ addr2line [7], a command-line tool that converts binary addresses into source file names and line numbers using debug information, to identify the patch blocks in two reference binary functions.

Table 1. The types of key instructions and the corresponding anchor values.

Key instruction type	Key example	Anchor value and type	Sliced instruction type	Sliced example	Aux
Condition comparison	`cmp eax, 2`	Value: 2 Type: "CMP"	Assignment	`mov eax, [eax + 0Eh]`	(0Eh, "offset")
			Arithmetic	`add eax, 2`	(2, "add")
Function call	`call foobar`	Value: foobar[1] Type: "CALL"	Parameter	`push 0Ah` `mov [esp], 3`	(0Ah, "param") (3, "param")

[1] We do not utilize function names for matching when unavailable.

4.2.2 Anchor Graph Construction. To ensure a stable signature across different compilers and optimization levels, we begin by constructing the anchor graph ($AG_{ref} = (V, E)$) to facilitate subsequent anchor path extraction, where V denotes the set of anchors and E represents the set of edges between anchors, derived from the control flow within the CFG. As mentioned in § 2.3, we extract anchors from the two types of key instructions, *condition comparison* and *function call*, which heavily influence the control flow of the program and persist across compilers and optimizations. The generation consists of three main steps as follows:



(a) Control flow graph (CFG) (b) Anchor graph (AG)

Fig. 6. Example of anchor graph construction. Figures (a) and (b) present the control flow graph (CFG) and the corresponding constructed anchor graph (AG), respectively. In ❶ cycle flattening, the back (dashed) edge is removed, and a new (green) edge is added to flatten the cycle. In ❷ key instruction identification, the key instruction, highlighted in bold black text, is identified, and its anchor value and type are extracted to initialize the anchor node. In ❸ backward instruction slicing, the variables within the key instruction are traced for slicing, and the relevant constants in the sliced instructions are extracted to constitute the auxiliary information ("Aux") for the anchors.

❶ Cycle flattening. In the presence of a cycle within a program, it becomes imperative to break the cycle to avert the generation of infinite paths. We remove cycles in the control flow graph (CFG) by flattening them (i.e., treating the cycle as a single iteration), following the standard strategy adopted by prior works such as BinXRay [38] and PDiff [24]. Specifically, we identify and remove back edges (e.g., C → A in cycle A → B → C → A) that introduce cycles, thereby transforming the CFG into a directed acyclic graph. To preserve critical control-flow connectivity, we additionally introduce edges from the source node of each back edge to the successors of the cycle entry, excluding those within the cycle itself, thereby effectively modeling a single iteration of the cycle. This flattening allows us to enumerate all simple paths without risking infinite recursion.

While this process abstracts away cycle semantics to some degree, it has minimal impact on our analysis. Our method focuses on identifying stable anchors and their control-flow context, which are typically preserved even under single-iteration cycle flattening. For example, blocks ①, ②, and ④ form a cycle for the CFG in Figure 6a. To flatten the cycle, we remove the back edge (④, ①) and introduce a new edge (④, ③).

❷ **Key instruction identification.** As shown in Figure 6a, we first identify two types of key instructions in CFG based on the known instructions for specific operations (i.e., `test`, `cmp` for condition comparison, and `call`, `jmp` for function call). After that, we extract the anchor value from the key instruction and determine the anchor type based on the key instruction type. The first three columns in Table 1 give two examples of the extracted anchor values and types. For other key instruction formats, such as `test eax, eax` for condition comparison (i.e., compare whether the variable is 0 or *None* through and operation) and `jmp loc_80AAF63` for indirect function call (i.e., where the jump target is the start address of a function), we convert them into equivalent key instruction formats as shown in column 2 of Table 1 to extract the anchor values and types based on the predefined rules. *Note that although we extract the anchor value of the function name for call instruction, we do not utilize it for matching directly when the function name is unavailable (e.g., sub_80A0BC).*

❸ **Backward instruction slicing.** Merely using constants in two types of key instructions is insufficient, as multiple anchors with identical values and types may exist in the same function. For example, two comparison instructions (`test eax, eax`) in block ① and ⑤ in Figure 6a generate the same anchor value 0. The main difference between these two key instructions lies in the previous instructions associated through variable **eax**, where one involves *and* operation with **0Eh**, while the other is **3**. *Intuitively, slicing techniques [31] can be employed to incorporate relevant instructions and extract the associated constants.* Specifically, we can perform backward slicing on the blocks, track variables in the key instruction, and use them as the slicing criterion. For instance, we set the variables (**eax** and **ebx**) in `cmp eax, ebx` of block ② in Figure 6a as the slicing criterion and slice the instructions that are associated through data dependencies. The slicing results are highlighted with grey text background. From the sliced instructions, we update the tracking variables and extract the relevant constants as well as their types as auxiliary information of the anchor. For function call instruction, we track the variables and extract constants from its parameters according to the calling conventions. In *x86* architecture, the parameters are moved into the stack so that we slice the `push` instruction (e.g., `push eax`) and `mov` operation taking register `esp` as the target (e.g., `mov [esp], 3`). As shown in Table 1, we roughly divide the sliced instruction types into three categories: assignment, arithmetic, and parameter. The last three columns show the sliced example and the corresponding extracted auxiliary information.

Ultimately, we extract the anchors to construct the anchor graph (AG) by following the original control flow of the CFG, as illustrated in Figure 6b.

4.2.3 Anchor Path Extraction. After constructing the anchor graph as the foundation, we need to extract useful information related to the patch and context code for patch presence test. The whole process consists of three main steps: ❶ **Candidate patch path extraction**, ❷ **Patch path selection**, and ❸ **Context path extraction.**

❶ **Candidate patch paths extraction.** Given the potential for multiple code modifications within a patch to span diverse control flows, we strive to identify and select the most representative patch path to enhance matching efficiency. To accomplish this, we first need to generate the candidate patch paths for two reference anchor graphs as follows:

(1) We extract the patch anchor graph, comprising solely the patch code, from the reference anchor graph ($AG = (V, E)$), guided by the mapped patch code regions. This graph is denoted as $AG_{patch} = (V', E')$, where $V' \subseteq V$ denotes the patch anchor nodes, and $E' \subseteq E$ denotes the patch anchor edges.
(2) We divide the patch anchors into subgraphs by computing the weakly connected components of the patch anchor graph AG_{patch}. Two anchors are considered to belong to the same subgraph if they are connected through any undirected path, regardless of edge direction. Specifically, we implement this

Algorithm 1: Subgraph dividing

Input: An anchor graph AG
Output: A list of weakly connected subgraphs $L_{AG_{sub}}$

```
S_visited, L_AG_sub ← ∅, ⟨⟩;
for n_i ∈ AG do
    if n_i ∉ S_visited then
        S_sub ← BFS(AG, n_i);
        S_visited ← S_visited ⋃ S_sub;
        L_AG_sub.add(GetSubGraph (AG, S_sub));
end
return L_AG_sub;
Function BFS(AG, n):
    S_sub, S_start ← ∅, {n};
    while S_start ≠ ∅ do
        S_temp ← S_start;
        S_start ← ∅;
        for n_j ∈ S_temp do
            if n_j ∉ S_sub then
                S_sub.add(n_j);
                S_start ← S_start ⋃ Predecessors (AG, n_j);
                S_start ← S_start ⋃ Successors (AG, n_j);
                S_start ← S_start \ S_sub;
        end
    end
    return S_sub;
```

using a breadth-first search (BFS) [14] traversal as shown in Algorithm 1. Starting from each unvisited node in AG_{patch}, we perform BFS over both predecessors and successors to collect all reachable nodes, forming one weakly connected component, with nodes stored in S_{sub}. This process is repeated until all nodes are assigned to a subgraph. Each such component defines a subgraph containing the corresponding anchors and their connecting edges in AG_{patch}.

(3) We enumerate all simple paths from the entry nodes to the exit nodes within each subgraph, regarding them as candidate patch paths.

❷ **Patch path selection.** Among all the candidate patch paths, we aim to select the most distinguishable and representative one for each reference function. The selection principle is to ensure that the patch path effectively distinguishes the vulnerable function from the fixed one, while also enhancing the identifiability of the patch path in the target function. Therefore, we set the following two priorities for patch path selection.

- *Pry-1: Exclusive to a single reference function.* The patch path includes anchors that are present in one reference function and are absent in the other. This priority ensures that the selected path can effectively differentiate the two reference functions.
- *Pry-2: Highest path weight.* The patch path possesses the highest path weight. Anchors that appear less frequently in the anchor graph (AG) are more distinctive and easier to identify in the target function. Therefore, we assign higher weight to rarer anchors using inverse term frequency: $w_{\mathcal{A}} = \frac{1}{TF_{\mathcal{A}}}$, where $TF_{\mathcal{A}}$

is the term frequency of anchor $\mathcal{A}$. For a patch path composed of n anchors $AP = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, the path weight is defined by two critical elements: the maximum anchor weight on the path and the length of AP.

$$w(AP) = (\alpha, \beta) = (\max\{w_{\mathcal{A}_1}, \ldots, w_{\mathcal{A}_n}\}, n), w_{\mathcal{A}_i} \in AP \tag{2}$$

The weight α captures the presence of a highly distinctive anchor, allowing the path to stand out during matching. The weight β ensures that the patch path encapsulates richer semantic content by incorporating as many anchors as possible. The integration of weights α and β thus enhances the path's identifiability. To determine which patch path has a higher weight, we compare the path weight of two patch paths, $w(AP_1) = (\alpha_1, \beta_1), w(AP_2) = (\alpha_2, \beta_2)$ with the following equation:

$$w(AP_1) \prec w(AP_2) \iff \begin{cases} \alpha_1 < \alpha_2, \text{ or} \\ \alpha_1 = \alpha_2 \text{ and } \beta_1 < \beta_2 \end{cases} \tag{3}$$

This lexicographic ordering ensures that paths containing more distinctive anchors (higher α) are preferred, and among those with equal distinctiveness, longer paths (higher β) are favored.

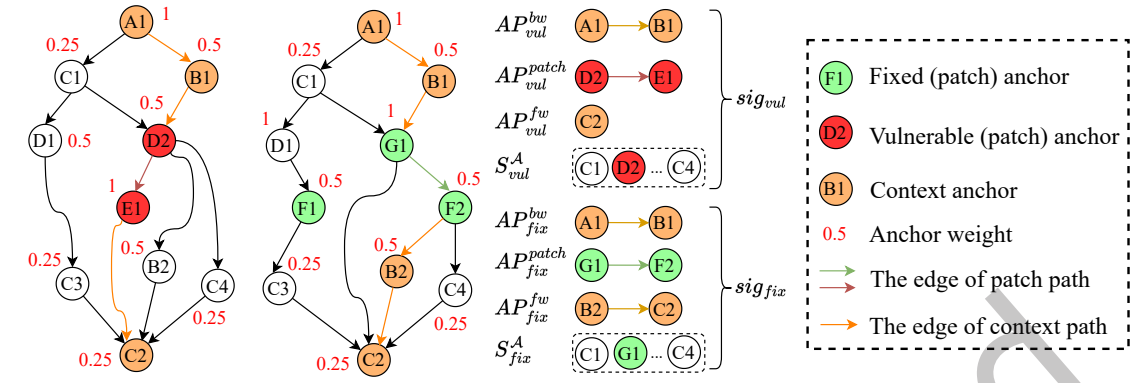
Pry-1 is first applied to select the patch path; if multiple candidate patch paths satisfy *Pry-1*, or none do, then *Pry-2* is subsequently used to select the patch path with the highest weight. We denote the selected reference patch paths extracted from the two reference functions as $AP_{ref}^{patch},\ ref \in \{vul, fix\}$. We extract patch paths from both reference functions, rather than relying on a single one, to accurately capture minimal patch modifications, such as a parameter change within a function, by comparing the features matched in the target function with the two reference functions. If only one of them is provided, we can not distinguish which one is closer to the target function, which may lead to false identification.

❸ **Context path extraction.** The patch path alone is often insufficient to fully capture the semantics of a patch. As discussed in § 2.3, incorporating surrounding context code helps distinguish true patch code from the patch-similar code. To extract the context paths, we perform bidirectional exploration from the patch path AP_{ref}^{patch}:

- For the backward context path AP_{ref}^{bw}, we start from the first anchor in AP_{ref}^{patch} and traverse backward toward the entry node of the anchor graph (AG).
- For the forward context path AP_{ref}^{fw}, we start from the last anchor in AP_{ref}^{patch} and traverse forward toward the exit node of AG.

In both directions, we use a greedy strategy [10], recursively selecting the predecessor or successor with the highest anchor weight at each step to form the context path. The context paths comprise both the backward and forward context paths, denoted as AP_{ref}^{bw} and AP_{ref}^{fw}.

Example of anchor path extraction. Figure 7 presents an example of anchor path extraction. ❶ Initially, we extract the candidate patch path in the two reference functions, which are ⟨D2, E1⟩, ⟨F1⟩, and ⟨G1, F2⟩, respectively. ❷ Then, we select the patch path in each reference function with the two priorities, which are ⟨D2, E1⟩ (satisfying *Pry-1* owing to the presence of E1 that only exists in Figure 7a), and ⟨G1, F2⟩ (satisfying *Pry-1* and *Pry-2* owing to the presence of G1 with the highest weight in Figure 7b). ❸ Ultimately, we extract the context paths corresponding to the two patch paths using the strategy described above. Taking the backward context path extraction in Figure 7a as an example, we begin from the patch anchor D2 and examine its predecessors. Among them, B1 has a higher weight than C1; therefore, B1 is selected as an anchor in the backward context path. Similarly, we extract the anchors for the context paths and present the backward context path $AP_{ref}^{bw},\ ref \in \{vul, fix\}$ and the forward context path $AP_{ref}^{fw},\ ref \in \{vul, fix\}$ in Figure 7c.



(a) AG of the vulnerable reference function (b) AG of the fixed reference function (c) Sigatures extracted from the two reference functions

Fig. 7. Example of anchor path extraction. Figures (a) and (b) show the AGs of the two reference functions, and Figure (c) shows the generated signature composed of the extracted anchor paths (backward context path AP^{bw}_{ref}, patch path AP^{patch}_{ref}, and forward context path AP^{fw}_{ref}) and the set of anchors $S^{\mathcal{A}}_{ref}$ in AG, where $ref \in \{vul, fix\}$. The fixed function newly introduced three anchor nodes F1, F2, G1, and removed the anchor nodes D2, E1 in the vulnerable function. Each anchor (node) is labeled with a letter and a number (e.g., C3); anchors sharing the same letter (e.g., C) possess identical anchors. Both the vulnerable anchor and the fixed anchor are patch anchors.

4.2.4 Signature Definition. We define our signature as a structured representation that captures the whole function and the code that is only related to the patch of both the vulnerable and fixed reference functions. It is constructed using the anchor graph AG_{ref}, $ref \in \{vul, fix\}$, as well as the extracted backward context, patch path, and forward context paths from each reference function. Formally, the signature is defined as:

$$sig_{ref} = (\mathcal{S}_{ref}, AP^{bw}_{ref}, AP^{patch}_{ref}, AP^{fw}_{ref})\ ref \in \{vul, fix\} \tag{4}$$

Here, $\mathcal{S}_{ref} = \{\mathcal{A}_1.v, \ldots, \mathcal{A}_n.v\}$, $\mathcal{A}_i \in AG_{ref}$, $ref \in \{vul, fix\}$, where $\mathcal{A}_i.v$ is the value of the anchor $\mathcal{A}_i$. AP^{bw}_{ref}, AP^{patch}_{ref}, AP^{fw}_{ref} are the reference backward context path, patch path, and forward path, respectively. $\mathcal{S}_{ref}$ is used to assess the target function at a coarse level, which enhances robustness when anchor paths are sparse or overly generic, mitigating false positives during classification and improving detection efficiency. The reference anchor paths in two reference functions are used to assess the target function at a fine-grained level, which precisely locates the patch code in the target function and distinguishes the vulnerable and fixed functions.

4.3 Patch Detection

During the patch detection, we filter out irrelevant functions to enhance both matching accuracy and efficiency, and locate the patch code within the target function using the provided signatures for function classification. To complete the following steps, we first generate the anchor graph of the target function (AG_{tgt}) in the same way as described in § 4.2.2 for detection.

4.3.1 Irrelevant Function Filtering. As discussed in our preliminary study on BCSD § 2.2, patch presence testing must account for the interference of irrelevant functions, owing to the limitations of current BCSD tools when dealing with large function pools. Therefore, before locating the patch code within the target function, we aim to

filter out irrelevant functions by performing a coarse-grained evaluation of all anchors in AG_{tgt}, thereby also enhancing detection efficiency.

To achieve this, we utilize the set of anchor values in the signature $\mathcal{S}_{ref}$ and the set of anchor values in the target anchor graph AG_{tgt} to compute the matching proportion of unique anchor values between the reference AG AG_{ref} and AG_{tgt} using the equation as follows:

$$\rho(\mathcal{S}_{ref}, \mathcal{S}_{tgt}) = \frac{|\mathcal{S}_{ref} \bigcap \mathcal{S}_{tgt}|}{\mathcal{S}_{ref}}, ref \in \{vul, fix\} \tag{5}$$

It will likely be irrelevant if a function matches only a small number of anchor values (i.e., a low ρ). Thus, the target function is immediately classified as non-vulnerable (irrelevant) if neither the proportion between the reference vulnerable function and the target function ($\rho(\mathcal{S}_{vul}, \mathcal{S}_{tgt})$), nor that between the reference fixed function and the target function ($\rho(\mathcal{S}_{fix}, \mathcal{S}_{tgt})$), surpasses the irrelevant function filtering threshold T_{iff}.

4.3.2 Anchor Path Matching. To verify the presence of the signature in the target function, we attempt to match its three anchor paths against the target function. For a given reference anchor path, the anchor path matching process aims to identify the most similar anchor path within the target function to serve as the match. To this end, two key challenges must be addressed during this process as follows:

(1) *The number of false positives generated by the BCSD tool increases as the number of candidates grows.* We employ the BCSD tool to match anchors of the `CALL` type in the binary, as function names are typically absent in stripped binaries. However, given that a binary typically comprises thousands of functions, directly matching anchors of the `CALL` type within the functions in the binary is impractical and would lead to many false positives. The false positives of BCSD tools primarily consist of irrelevant functions that exhibit structural and instructional similarities to the query function. These can be categorized into two types: 1) irrelevant functions outside the calling scope of the target function, and 2) irrelevant functions called by the target function but failing to satisfy the structural context observed in the reference function.
(2) *The target anchor graph may encompass a large number of anchor nodes, resulting in a substantial increase in the number of anchor paths to be matched.* The number of anchor nodes in the target function may range from dozens to hundreds. Consequently, exhaustively enumerating all simple paths within the target function and matching them against the reference anchor path leads to a path explosion problem, rendering the process computationally intensive and impractical.

To solve the challenges, we reduce the number of candidate functions and anchor paths for matching with two strategies as follows:

Strategy 1: *Narrow down the search space based on values and caller constraints.* The rationale is that most unrelated either differ in their values and auxiliary information, or lie outside the calling scope of the target function. By restricting candidates to only those consistent in value or called by the target function, unrelated anchors are filtered out early. For `CMP`-type anchors, we directly match them based on their value and auxiliary information. For `CALL`-type anchors, we first attempt to match them by function name, if available. Otherwise, we employ the BCSD tool but restrict its search space to only those functions called by the target function, rather than all functions in the binary. This focused search eliminates irrelevant functions early and substantially reduces false positives.

Strategy 2: *Filter out candidate anchors based on the distance between anchors in adjacent candidate sets.* The rationale is that anchors from different paths typically break the control-flow relationships defined by the reference anchor path. By requiring both reachability and proximity to the preceding anchor, this strategy ensures that only candidates consistent with the intended control-flow logic are preserved. Anchor path matching is treated as the reverse of anchor path extraction within the target function. Once a predecessor anchor is matched, subsequent candidate anchors are filtered based on their distance from the previously matched anchor within the

target anchor graph. Only the closest and structurally valid candidate is preserved, while others are discarded. To this end, we define the distance between the two anchors ($\mathcal{A}_1$ and $\mathcal{A}_2$) in AG as follows:

$$d(\mathcal{A}_1, \mathcal{A}_2) = \begin{cases} \infty & \text{There is no path from } \mathcal{A}_1 \text{ to } \mathcal{A}_2, \text{i.e., unreachable} \\ |p(AG, \mathcal{A}_1, \mathcal{A}_2)| & \text{The length of the shortest path } p \text{ from } \mathcal{A}_1 \text{ to } \mathcal{A}_2 \text{ in AG} \end{cases} \tag{6}$$

Strategy 1 serves as a frontline filter for Strategy 2. It prunes the candidate pool through value checks and caller constraints, ensuring that only a small and relevant subset of anchors is retained. Strategy 2 then functions as a structural validator, enforcing that the matches also preserve the control-flow logic observed in the reference. By combining these two strategies, we can substantially alleviate the aforementioned two challenges.

Anchor Path Matching Algorithm. Motivated by these two strategies, we propose the anchor path matching as shown in Algorithm 2. For convenient reference, we list all the abbreviations mentioned in the algorithm as follows:

- AP_{ref}: Reference anchor path, one of the inputs of the algorithm.
- AG_{tgt}: Anchor graph of the target function, one of the inputs of the algorithm.
- L_{AP}: The list of matched anchor paths, output of the algorithm.
- l_{max}: Current maximum length of the matched anchor path.
- AP: Current matched anchor path, it will be updated during the recursive process.
- i: The index of the anchor in AP_{ref} currently being matched.

First (line 1), we initialize L_{AP} as an empty list, and set l_{max} to 0. The core matching process is implemented via depth-first search (DFS) [9], which includes three main steps:

(1) Check matching progress (lines 5–10): The matching of the reference anchor path proceeds anchor by anchor. Thus, when the index i reaches the length of AP_{ref}, it signifies that the matching has reached the end (i.e., no further reference anchors remain to be matched). At this point, the matched anchor path AP is added to L_{AP}, and l_{max} is updated accordingly.
(2) Retrieve candidate anchor set (lines 11–14): The candidate anchor set S_{cand} is retrieved using the function `CandAnchorSet`, which is implemented based on **Strategy 1**. The branching logic based on $AP.length$ determines whether the AP has a previously matched anchor to serve as a constraint. When $AP.length = 0$ (empty path), no anchors have been matched yet, meaning any candidate anchors generated by Strategy 1 can serve as valid starting points. When $AP.length > 0$ (non-empty path), at least one anchor has already been matched, so subsequent candidates must conform to the constraints outlined in **Strategy 2**. Only those anchors that are closest to the last anchor in AP (i.e., $AP[-1]$) will be selected to form the candidate anchor set S_{cand}.
(3) Update matched anchor path (lines 15-21): For each anchor in S_{cand}, a new matched path is formed by appending it to AP, and the function `DFS` is called recursively to continue the matching process.

Example. Figure 8 presents an example of anchor path matching. Given the reference anchor path ⟨A1, B1, A2, A3, C1⟩ and the target anchor graph AG_{tgt}, we first derive the list of candidate anchor sets $L_{S_{cand}}$ according to **Strategy 1**, as illustrated in Figure 8b. Next, we apply **Strategy 2** to select anchors from each candidate set. Starting from A1', only B1' is selected due to its minimal distance to A1'. Similarly, A2', A3', and C1' are subsequently selected, forming the matched anchor path. Although a path starting from A2' to A4' is also explored, its length is shorter than that of the current matched path. As a result, the final matched anchor path is ⟨A1', B1', A2', A3', C1'⟩.

Given version updates and code variations, we require the context path to achieve a matching ratio exceeding the context path matching threshold T_{cpm}, which filters out incorrectly identified paths while tolerating minor discrepancies. In contrast, we require the anchors to be fully matched for the patch path, ensuring the patch's presence.

Algorithm 2: Anchor path matching

Input: A reference anchor path AP_{ref} and the target anchor graph AG_{tgt}
Output: The list of matched anchor paths L_{AP}

$L_{AP}, l_{max} \leftarrow 0$;
DFS $(0, \langle\rangle)$;
return L_{AP};
Function DFS(i, AP)**:**
  if $i = AP_{ref}.length$ **then** // Matching reaches the end
    if $AP.length > l_{max}$ **then** // A longer matched anchor path is found
      $L_{AP}, l_{max} \leftarrow \langle AP\rangle, AP.length$
    else if $AP.length = l_{max}$ **then**
      L_{AP}.add(AP)
    return;
  if $AP.length = 0$ **then** // No previously matched anchor S_{cand}
    $S_{cand} \leftarrow$ CandAnchorSet $(AG_{tgt}, AP_{ref}[i])$;
  else
    $S_{cand} \leftarrow$ Anchors of CandAnchorSet $(AG_{tgt}, AP_{ref}[i])$ that closest to $AP[-1]$;
  if $S_{cand} = \emptyset$ **then** // No candidate anchors for the current reference anchor
    DFS $(i + 1, AP)$;
  else
    for $\mathcal{A}_i \in S_{cand}$ **do**
      AP.add$(\mathcal{A}_i)$;
      DFS $(i + 1, AP)$;
    end

4.3.3 Patch Path Verification. As outlined in § 2.3, relying solely on the patch code for matching may result in the misidentification of patch-similar code blocks within the vulnerable function as real patch code, causing the vulnerable function to be incorrectly classified as non-vulnerable. Therefore, it is essential to incorporate the context path to verify the authenticity of the matched patch path.

Our central insight is that *patch-similar code often fails to preserve the control-flow relationships between the patch and its surrounding context observed in the reference function.* To make this explicit, we first need to define the distance between two anchor paths. Let $AP_1 = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ and $AP_2 = \langle \mathcal{A}'_1, \ldots, \mathcal{A}'_m \rangle$ be two anchor paths. Since our computing targets (i.e., context path and patch path) are extracted within the same control flow in the AG, we define the control flow distance between them as:

$$d_{AP}(AP_1, AP_2) = d(\mathcal{A}_n, \mathcal{A}'_1) \tag{7}$$

In other words, it is computed as the distance between the nearest anchors between the two anchor paths, i.e., the last anchor of AP_1 and the first anchor of AP_2. This definition reflects the expected execution order: the patch code is typically executed after the backward context and before the forward context within the same control flow path.

To ensure structural consistency, the distances between the matched anchor paths must not exceed those observed in the reference function. Specifically, a matched patch path is considered valid only if it satisfies:

$$d_{AP}(AP^{bw}_{match}, AP^{patch}_{match}) \leq d_{AP}(AP^{bw}_{ref}, AP^{patch}_{ref}) \wedge d_{AP}(AP^{patch}_{match}, AP^{fw}_{match}) \leq d_{AP}(AP^{patch}_{ref}, AP^{fw}_{ref}) \tag{8}$$

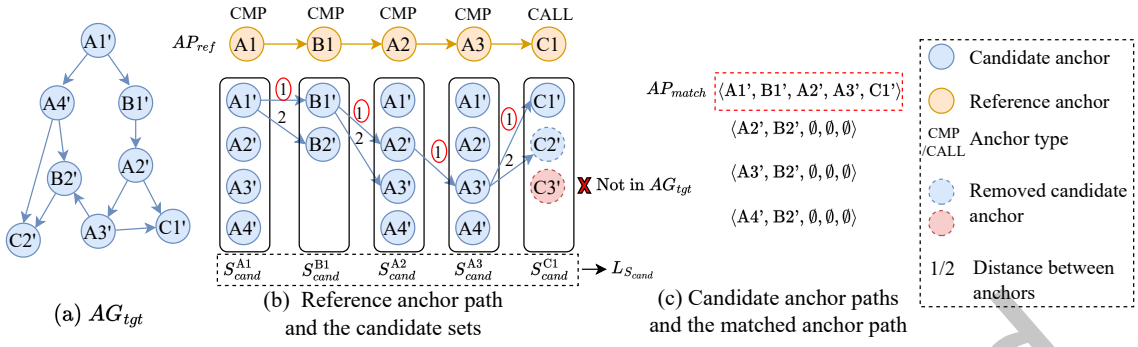


Fig. 8. Example of anchor path matching. Figure (a) shows the target anchor graph AG_{tgt}. Figure (b) displays the reference anchor path AP_{ref} alongside the corresponding candidate anchor sets for each anchor in AP_{ref}, denoted as $L_{S_{cand}}$. Figure (c) depicts the candidate and matched anchor path AP_{match} (highlighted by a red dashed rectangle). The symbol $\emptyset$ indicates that the reference anchor node does not match any candidate nodes. C3' is removed from the candidate set due to its absence in AG_{tgt} (i.e., not called by the target function).

The first inequality in Equation (8) ensures that the structural relationship between the backward context path and the patch path in the target function is at least as close as that in the reference function. The second inequality in Equation (8) enforces the same condition between the patch path and the forward context path. These constraints ensure that both the patch and its context maintain coherence, mirroring the structure observed in the reference function, and thus eliminate patch-similar code that fails to satisfy the condition. If multiple matched patch paths satisfy the condition, we select the one with the largest number of context matching anchors.

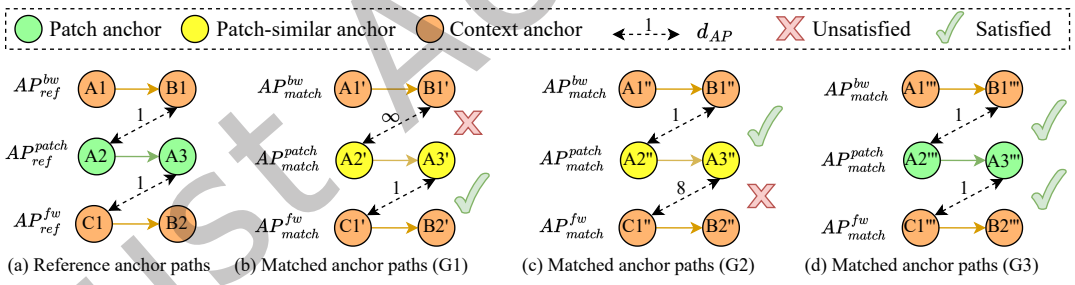


Fig. 9. Example of patch path verification. Figure (a) shows the reference anchor paths, including the reference backward context path (AP^{bw}_{ref}), patch path (AP^{patch}_{ref}, and AP^{fw}_{ref}), along with the distance between the context paths and the patch path. Figures (b) to (d) depict three groups (G1, G2, G3) of matched anchor paths. The matched patch paths in G1 and G2, derived from patch-similar code, are deemed invalid due to unsatisfied conditions, whereas the one in G3, extracted from the real patch code, is considered valid as it satisfies the distance constraints.

Example. Figure 9 present an example of patch path verification. Given the reference anchor paths extracted from the reference function in Figure 9a, we evaluate the validity of the matched patch paths across three distinct groups (G1, G2, G3), as illustrated in Figure 9b to Figure 9d. In Figure 9b (G1), the distance between the matched backward context path and the patch path is ∞, indicating that the patch code is unreachable

from its backward context, thereby rendering the matched patch path invalid. In Figure 9c (G2), the distance between the patch path and the forward context path is 8, which exceeds the reference distance of 1, resulting in another invalid match. Only the patch path in Figure 9d (G3) satisfies both conditions ($d_{AP}(B1''', A2''') \leq d_{AP}(B1, A2) \wedge d_{AP}(A3''', C1''') \leq d_{AP}(A3, C1)$), and is thus deemed valid.

4.3.4 Function Classification. Using the matched patch, context paths, and the signatures of two reference functions, we aim to categorize the function as vulnerable or non-vulnerable through a decision tree. In light of the diverse nature of code modifications within patches, we categorize them into three distinct types as follows:

(1) *Type 1: Both deletion and addition.* The patch comprises both deleted and newly added code statements, leading to both sig_{vul} and sig_{fix} being generated. In this case, a vulnerable function should contain sig_{vul} and not contain sig_{fix}.
(2) *Type 2: Deletion only.* The patch consists solely of deleted code statements, with no new code introduced, resulting only sig_{vul} being generated. In this case, a vulnerable function should be identified by the presence of sig_{vul}.
(3) *Type 3: Addition only.* The patch consists solely of added code statements, without code deleted, resulting in only sig_{fix} being generated. In this case, A vulnerable function should be identified by the absence of sig_{fix}. However, irrelevant functions may also lack sig_{fix}. To differentiate the vulnerable function from irrelevant ones, we classify functions whose context paths match but lack a corresponding patch path in sig_{fix} as vulnerable, as they omit the added patch code while preserving the contextual structure found in the fixed function.

For patches of differing modification types, corresponding tailored strategies should be employed. Furthermore, in our design, to guarantee the recall of anchor paths, we do not insist on the auxiliary information within anchors being perfectly matched, as such details may vary slightly across compilations. Consequently, when both sig_{vul} and sig_{fix} are present and both the patch and context paths match, we proceed to sequentially compare the patch path score and context path score to ascertain the function's classification. The scores for the matched anchor path are calculated based on the matched auxiliary information (i.e., the greater the extent of auxiliary information matched, the higher the likelihood of corresponding to the respective category) as shown in Equation (9).

$$s = \frac{\sum_{i=0..|AP_{ref}|} LCS(AP_{ref}[i].\text{aux}, AP_{match}[i].\text{aux})}{|AP_{ref}|} \tag{9}$$

AP_{ref} and AP_{match} denote the reference anchor path and the matched anchor path, respectively. By enumerating the anchors in AP_{ref} and AP_{match}, we compare the corresponding auxiliary information within each anchor using the longest common subsequence (LCS) [11]. The average matching score of each anchor in AP_{ref} is then calculated as the definitive matching score of a reference anchor path. The context score is derived as the sum of the backward context path score and the forward context path score.

Figure 10 presents the decision tree for function classification, designed based on the above strategies. Let us examine the red decision route as a representative case. Following this route: ❶, both signatures are present, necessitating verification of whether their corresponding patch paths match. ❷ Upon confirming that both patch paths exhibit matches, we proceed to compute the patch scores between each matched path and its respective reference patch paths within the signatures. ❸ The result shows that the patch path in sig_{vul} achieves a higher score compared to the one in sig_{fix}, consequently leading to the classification of the function as vulnerable.

5 EVALUATION

We aim to address the following **research questions (RQs)**.

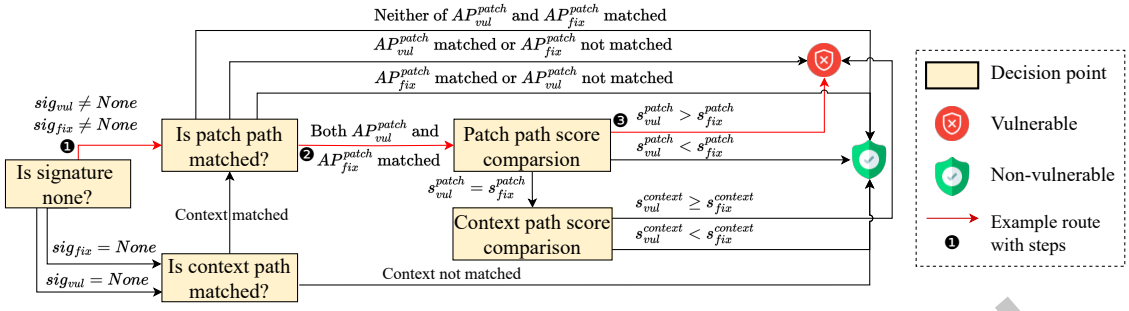


Fig. 10. The decision tree for function classification. The classification process begins by verifying whether the signatures are *None*, a condition that occurs when no patch path is extracted from the reference function. Based on each outcome, the tree proceeds to the next relevant condition. Here, $AP_{ref}^{patch}, ref \in$ vul, fix denote the patch anchor paths in the vulnerable and fixed reference functions, respectively. The symbols s_{ref}^{patch} and $s_{ref}^{context}$ represent the matching scores for the patch path and the context path. The red path highlights an example decision route, with each step emphasized for illustration.

- **RQ1.** How accurate is *PLocator* in identifying vulnerable and fixed functions with the same compilation settings, across different optimization levels, and different compilers, compared to the state-of-the-art approaches?
- **RQ2.** How does *PLocator*'s efficiency in the detection process compare to state-of-the-art approaches?
- **RQ3.** How effective is *PLocator* when irrelevant functions are involved, in comparison to state-of-the-art approaches? To what extent do the two components of *PLocator*, namely irrelevant function filtering and patch path verification, contribute to enhancing accuracy (i.e., through an ablation study)?

RQ1 and **RQ2** are employed to compare *PLocator* with existing patch presence test approaches, using the same experimental setup as in their studies [38, 41]. **RQ3** is used to compare *PLocator* with baselines in a more challenging and practical scenario where irrelevant functions are present.

5.1 Experimental Setup

5.1.1 Dataset. Figure 11 depicts the pipeline employed for constructing our dataset and evaluating the patch presence test approaches, which includes three main steps:

- ❶ **Data Collection.** We select four real-world projects spanning diverse application domains, protocol encryption, packet processing, XML parsing, and font rendering, all of which have been extensively evaluated in prior studies [41, 43], and download their source code repositories. We then collect vulnerabilities associated with these projects, including vulnerable and fixed versions, along with corresponding patch files from NVD [6]. Subsequently, we extract the names of vulnerable functions from the patch files to compile a list of vulnerable and fixed functions.
- ❷ **Binary preprocessing.** All vulnerable and fixed version of the project (identified through release tags in Git commit) is compiled using GCC 9.4.0 and Clang 6.0 with O0–O3 optimization levels as the original binaries. Subsequently, we strip the binaries to simulate real-world scenarios where debug information is absent and utilize the binary analysis tool (IDA Pro) to identify the functions in stripped binaries to form the function pool. To label the functions in the pool, we construct the mapping of the function names and the function addresses for each original binary. A function is labeled as vulnerable only if it comes from a vulnerable binary and its address matches a vulnerable function in the original binary. Otherwise, the function is labeled as non-vulnerable.

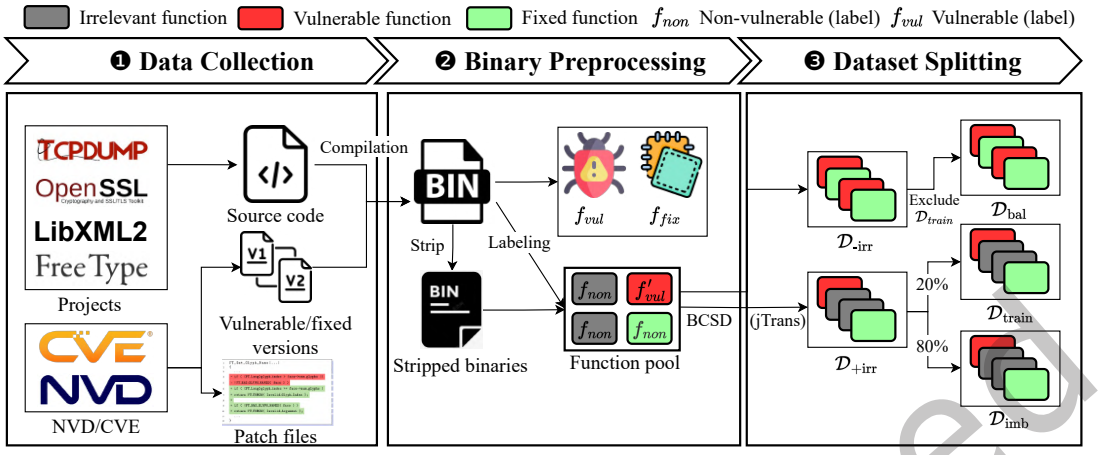


Fig. 11. Pipeline for dataset construction. The process consists of three main stages: ❶ **Data Collection.** Collect source code and corresponding vulnerability information, including vulnerable/fixed versions and patch files for each project. ❷ **Binary Preprocessing.** Compile source code using different compilers and optimization levels to generate binaries, following binary stripping. Extract reference functions (f_{vul} and f_{fix}) and label functions in the stripped binaries using function name and address mappings from the original binaries. ❸ **Dataset Splitting.** Construct the dataset for evaluation based on the labeled functions.

- ❸ **Dataset Splitting.** We construct two datasets to evaluate the performance of patch presence test approaches. Specifically, we collect all the vulnerable and fixed functions in the pool to construct the dataset without irrelevant functions $\mathcal{D}_{\text{-irr}}$. We use the BCSD tool jTrans to retrieve the top-50 most similar functions (ensuring inclusion of the vulnerable ones) for each reference vulnerable function to construct the dataset with irrelevant functions $\mathcal{D}_{+\text{irr}}$. We construct this dataset using the BCSD tool to simulate realistic usage scenarios, wherein BCSD is typically employed to retrieve candidate functions from stripped binaries, as it can efficiently filter out large numbers of irrelevant functions, allowing the patch presence test approaches to focus their more precise but costlier analysis on a manageable subset, as demonstrated in BinXRay [38] and Robin [41]. BCSD is a prerequisite for *PLocator* as it provides an essential pre-filtering step that efficiently narrows the vast search space in stripped binaries, rapidly identifying a smaller set of candidate functions more likely to contain the vulnerable code. To determine the thresholds introduced in earlier sections, we partition $\mathcal{D}_{+\text{irr}}$ into two subsets using a 2:8 ratio: a training dataset $\mathcal{D}_{train}$ (20%) and a imbalanced testing dataset $\mathcal{D}_{imb}$ (80%). Additionally, we exclude the vulnerable and fixed functions in $\mathcal{D}_{train}$ from $\mathcal{D}_{\text{-irr}}$ to derive the balanced testing set $\mathcal{D}_{bal}$.

Table 2 presents the statistical summary of the original data and the two constructed datasets. In total, we collect 73 CVEs encompassing 224 distinct vulnerable and fixed functions. The first six columns detail the original vulnerability data and the corresponding compiled binaries, while the final five columns report the statistical composition of the dataset without irrelevant functions $\mathcal{D}_{\text{-irr}}$ and the dataset with irrelevant functions $\mathcal{D}_{+\text{irr}}$, respectively.

PLocator operates under the assumption that function boundaries in stripped binaries can be accurately identified, relying on functions extracted by a preceding BCSD tool or those recognized by binary analysis frameworks such as IDA Pro. According to our dataset statistics, approximately 2.7% of the vulnerable and fixed

Table 2. Statistical results of our dataset. f_{vul}, f_{fix}, and f_{irr} refer to the vulnerable, fixed, and irrelevant functions, respectively. Version and Binary indicate the release tag and the compiled binary, respectively.

Original data						$\mathcal{D}_{\text{-irr}}$		$\mathcal{D}_{\text{+irr}}$		
Projects	# CVE	# f_{vul}	# f_{fix}	# Version	# Binary	# f_{vul}	# f_{fix}	# f_{vul}	# f_{fix}	# f_{irr}
OpenSSL	34	50	50	21	105	248	248	311	311	10,091
Freetype	7	9	9	5	25	45	45	45	39	1,822
Tcpdump	25	39	39	2	10	182	182	172	164	11,088
Libxml2	7	14	14	4	20	70	70	76	41	3,090
Total	73	112	112	32	160	545	545	604	555	26,091

functions could not be identified by IDA Pro. The majority of these functions originate from binaries compiled with the Clang compiler, likely due to IDA Pro's limited support for Clang-generated binaries. To ensure a fair comparison across all compilers and optimization levels, we exclude the corresponding vulnerable and fixed functions under all compilers and optimization levels from the evaluation.

5.1.2 Experiments and Metrics. To compare the effectiveness of the approaches on different compilers and optimizations, we set three experiments: *1) Same.* The reference and target functions are from binaries with the same compiler (gcc) and optimization level (O0). *2) XO (Cross-optimizations).* The reference and target functions are from binaries with different optimization levels (O1 to O3). *3) XC (Cross-compilers).* The reference and target functions are from binaries with different compilers (GCC and Clang). Same as in § 2.2, we adopt precision, recall, and F1-score as the primary metrics to evaluate the effectiveness of the approaches. During our experiments, we observed that the baseline methods supported only a subset of the test cases in our dataset, primarily due to failures in signature generation, timeouts, and unexpected exceptions. Moreover, failed test cases are not reported as positives, which may result in an inflated precision. To account for this, we introduce two additional metrics, support rate (SR) and accuracy (ACC), defined as follows:

$$\text{SR} = \frac{\text{TC}_{\text{sup}}}{\text{TC}_{\text{all}}}, \text{Acc} = \frac{\text{TC}_{\text{cor}}}{\text{TC}_{\text{all}}} \tag{10}$$

where TC_{sup} denotes the number of successfully supported test cases, TC_{cor} and TC_{all} represents the total number of test cases, respectively.

5.2 Baseline Methods

We compare *PLocator* with three state-of-the-art patch presence test approaches, one is syntactic-based, while the other two are semantic-based methods. We also include the BCSD tool jTrans as a baseline comparison to highlight the improvements achieved by *PLocator* over it.

- **BinXray [38]**. A syntactic-based patch presence test approach, which uses basic block mapping to extract the execution traces of the patch from two reference binary functions and compares them to the trace extracted from the target function.
- **Robin [41]**. A semantic-based patch presence test approach. It employs symbolic execution to extract the malicious function input (MFI) that triggers the vulnerable code. The MFI is then fed into the target, and similarities are calculated with the captured semantic features to determine the presence of a patch.

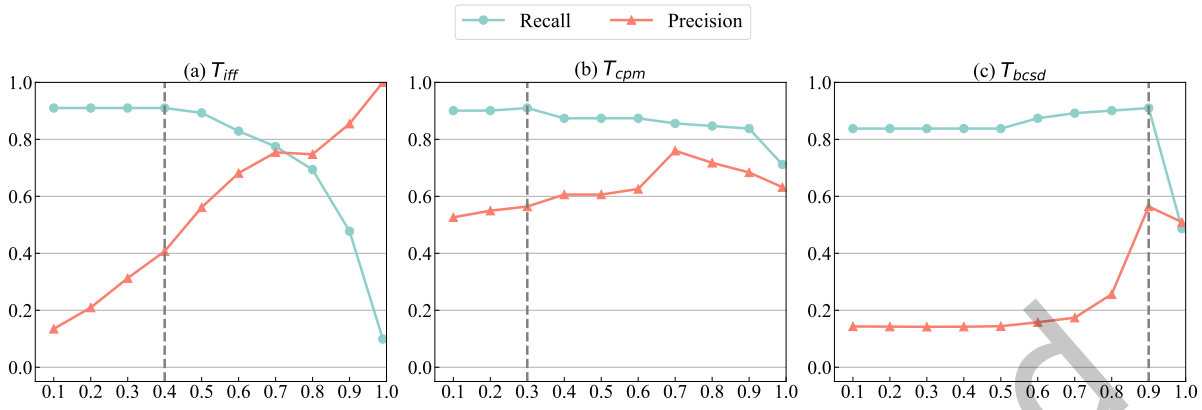


Fig. 12. Threshold selection process. The x-axis and y-axis represent the threshold values and the corresponding metric scores (precision and recall), respectively. The optimal threshold is chosen as the one achieving the highest recall while maintaining relatively high precision, highlighted by grey dotted lines.

- PS^3 **[43]**. A semantic-based approach extracts the features from the vulnerable and fixed functions through symbolic execution and matches them in the target binary function.
- **jTrans [32]**. The state-of-the-art BCSD approach segments assembly code based on jump instructions, embeds the resulting sequences using the BERT architecture [20], and employs two pre-training tasks, Masked Language Modeling (MLM) and Jump Target Prediction (JTP), to capture function semantics.

For the above baselines, we use the original implementations as described in their papers. We regard the target functions as "unknown" when the baselines return an uncertain result (i.e., with a score of 0) or fail to identify the target (i.e., "unknown" or time out). We exclude other related approaches from the experiments for two main reasons: 1) They did not release the source code (e.g., PDiff [24], SPAIN [39]). 2) They perform worse than our select baselines in the previous studies (e.g., Fiber [44], PMatch [26]) To evaluate the effects of two components (i.e., irrelevant function filtering and patch path verification) in *PLocator*, we further set up three configurations:

- $PLocator_{-IFF}$: *PLocator* without Irrelevant Function Filtering.
- $PLocator_{-PPV}$: *PLocator* without Patch Path Verification.
- $PLocator_{-Both}$: *PLocator* without the both components.

5.3 Implementation

5.3.1 Environment and Tools. *PLocator* is implemented in Python with 3,365 lines of code. We utilize IDA Pro 7.5 [4] and IDAPython to disassemble the binary functions and construct the CFGs. All experiments, except for BCSD, were conducted on a laptop with an Intel 16-Core i9-9880 2.30GHz processor, 64 GB of memory, running Ubuntu 22.04 OS. We implemented jTrans as our BCSD tool on a server equipped with an Intel Xeon Gold 5218 CPU @ 2.30GHz, 1 TB of memory, and 2 Nvidia Tesla V100 GPUs (32GB each).

5.3.2 Threshold Selection. We vary the three thresholds T_{iff} (irrelevant function filtering), T_{cpm} (patch path verification), and T_{bcsd} (called functions BCSD) from 0.1 to 1 to identify the thresholds yielding the best results (i.e., highest recall and higher precision). The threshold selection results are shown in Figure 12, in which T_{iff}, T_{cpm} and T_{bcsd} yield the best results at 0.4, 0.3 and 0.9, respectively.

Table 3. The results on the balanced test dataset $\mathcal{D}_{\text{bal}}$ (%).

Experiment	**Same**					**XO**					**XC**					**Average**				
Metric	P	R	F1	SR	Acc	P	R	F1	SR	Acc	P	R	F1	SR	Acc	P	R	F1	SR	Acc
BinXray	92.3	49.3	64.3	53.4	49.3	77.8	12.8	22.0	18.0	13.9	60.0	4.1	7.7	8.2	4.8	76.7	22.1	31.3	26.6	22.7
Robin	72.4	86.3	78.8	92.1	72.6	56.2	74.4	64.1	92.1	54.1	57.7	41.1	48.0	89.2	48.0	62.1	67.3	63.6	89.7	58.2
PS3	79.7	69.9	74.5	78.1	65.1	57.0	52.1	54.4	78.1	45.4	56.1	50.7	53.2	75.3	43.2	64.3	57.5	60.7	77.2	51.2
PLocator	**87.0**	**91.8**	**89.3**	**100.0**	**89.0**	**77.5**	**84.9**	**81.1**	**100.0**	**80.1**	**83.3**	**82.3**	**82.8**	**100.0**	**82.9**	**82.6**	**86.3**	**84.4**	**100.0**	**84.0**

5.4 RQ1: Result of Patch Presence Test excluding Irrelevant Functions

In this research question, we focus solely on identifying vulnerable functions among fixed functions, consistent with previous works. For each test case in the balanced test dataset $\mathcal{D}_{bal}$, we categorize it based on each method's output and compute the metrics presented in Table 3.

Syntactic-based method (BinXRay) result analysis. BinXRay supports only 53.4%, 18.0%, and 8.2% of test cases in $\mathcal{D}_{bal}$ for the three experiments, resulting in extremely low accuracy. The unexpectedly low support rate of BinXRay primarily stems from a fundamental design flaw. BinXRay employs a block mapping algorithm to match reference blocks with blocks in the target function. Although it normalizes instructions within blocks (i.e., replacing specific addresses and registers with unified symbols), the matching process remains vulnerable to variations introduced by differing compilation settings, which may yield entirely divergent instruction sequences for identical source code. As a result, it often fails to extract its core matching unit, named trace (i.e., a sequence of consecutive code blocks), leading to a lack of support. Furthermore, when patch modifications are extensive, the extraction of traces will also fail. In such cases, BinXRay simply returns "Too much difference," signaling its inability to support the functions.

Semantic-based methods (Robin and PS^3) results analysis. Robin and PS^3 exhibit relatively stable performance and support a greater number of test cases compared to BinXRay, particularly in the XO and XC experiments. On average, Robin achieves a recall of 67.3% and a precision of 62.1% across the three experiments, while PS^3 attains a recall of 57.5% and a precision of 64.3%, respectively. The suboptimal results of these two semantic-based methods are primarily attributed to their neglect of patch-similar code and inherent design limitations. To clearly illustrate their predictions for different test cases, we plot the score of each test case for three approaches as shown in Figure 13. All three approaches classify the target functions based on the matched features of two reference functions (i.e., vulnerable and fixed). For each experiment under a specific approach (e.g., Robin-Same), two figures are provided:

- *Test Case Score Scatter Plot (left figure).* Each test case is plotted as a node based on the reference vulnerability and its score predicted by the approach. The *x-axis* represents different vulnerabilities, while the *y-axis* indicates the score of test cases under the corresponding vulnerabilities.
- *Test Case Score Curve Plot (right figure).* The *x-axis* represents the probability density of test cases at a specific score, sharing the *y-axis* with the scatter plot. A higher *x-axis* value indicates a greater number of test cases with the corresponding score.

Under ideal circumstances, all vulnerable test cases should yield negative scores, whereas fixed ones should yield positive scores. As shown in Figure 13, Robin and PS^3 accurately predict the test cases in the Same experiments, but generate more false positives and false negatives in the other two experiments. Moreover, the absolute value of scores decreased from the Same experiment to the other two experiments. *In that case, although they claim to select semantic features resilient to varying compilation settings, many of the features they depend on are lost*

under different compiler and optimization conditions, causing the boundary between vulnerable and fixed functions to become indistinct. The results of PS^3 are similar to Robin's since they both depend on emulation of the target function and match features from the reference against the target. The key distinction between them lies in their choice of semantic features and the collected traces from the target function. For feature selection, PS^3 assumes that function names remain unchanged in the target function and uses function call names as features. In contrast, Robin selects function call arguments and ignores function call names, achieving better performance on datasets containing stripped binaries. For trace collection, PS^3 enumerates all blocks of the target function. In contrast, Robin focuses only on traces related to the vulnerable and fixed code, based on their Malicious Function Input (MFI), which mitigates interference from irrelevant code to some extent.

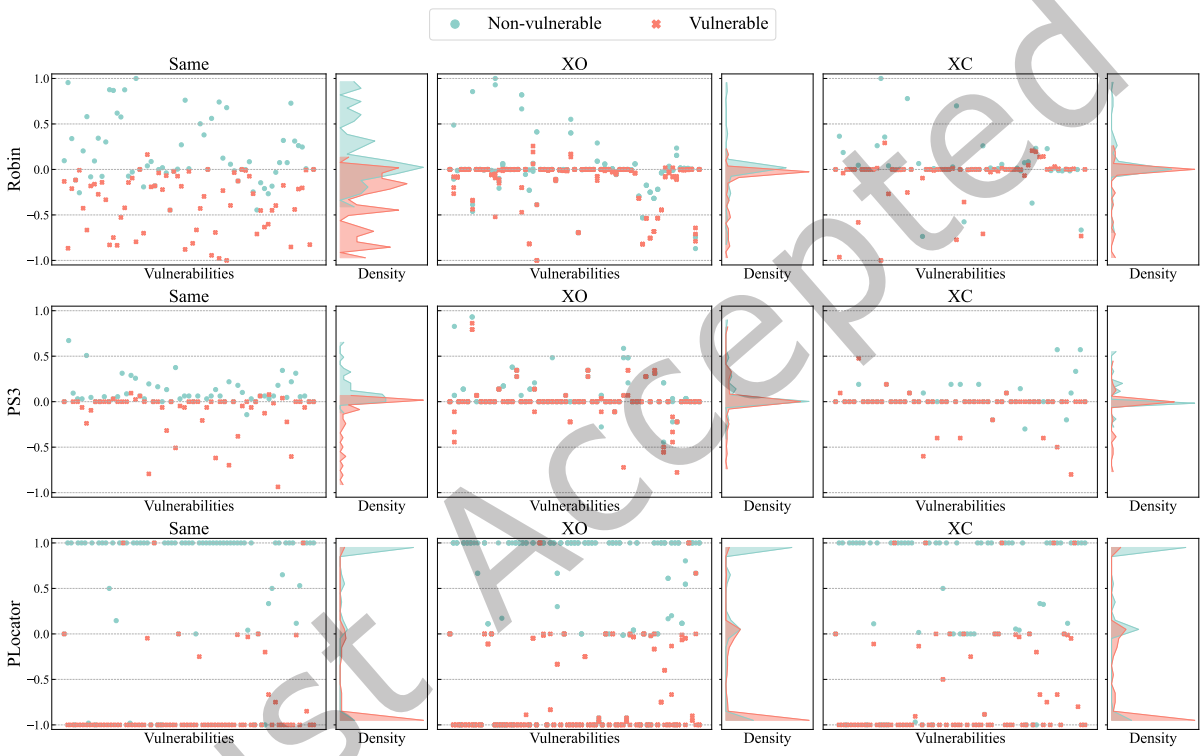


Fig. 13. Scatter and density plots of test case scores for Robin, PS^3, and *PLocator* on the balanced dataset $\mathcal{D}_{bal}$. Each method's output is visualized using a normalized score s, computed as $s = \frac{N_{vul} - N_{fix}}{\max(N_{vul}, N_{fix})}$, where N_{vul} and N_{fix} represent the number of matched vulnerable and fixed features, respectively. The score s ranges from −1 to 1. In the scatter plot, the x-axis represents different vulnerabilities (test cases), while the y-axis shows the corresponding normalized score. In the probability density plot, the x-axis shows the score distribution, and the y-axis indicates the relative density. Ideally, vulnerable functions should appear below the zero-score line ($s < 0$), while non-vulnerable functions should lie at or above it ($s \geq 0$).

PLocator result analysis.. Across all the experiments, *PLocator* significantly outperformed the baselines, achieving a precision of 82.6% and a recall of 86.3%, outperforming the second-best approach (Robin) by 33.0% and 28.2%, respectively. As depicted in Figure 13, *PLocator* effectively separates the test cases into their true categories, even under different compilers and optimizations. The similarity in the absolute score values across the three

experiments highlights the robustness of the features selected by *PLocator*. We manually analyze *PLocator*'s false positives and false negatives to conclude three primary causes:

1) undetectable patch modifications. Some patches will not affect the anchors extracted from the two references, resulting in the same signatures generated for vulnerable and fixed functions. For example, the patch of `CVE-2014-0224` only added an assignment statement as shown in Figure 14b, resulting in the signature of vulnerable and fixed to be the same.

2) Anchor path variation caused by compilers and optimizations. In some special cases, the anchor paths across different compilers and optimizations may show disparity. For example, as illustrated in Figure 14c, the order of three conditions in the function `ssl_get_algorithm2` at O0 changes when the function is compiled with O2, leading to a corresponding change in the order of anchors in the patch path. The two conditions in the function `icmp_print` at O0 are optimized as a single condition by casting the expression `*v5 - 11` as an unsigned integer, which includes the same semantics. These code variations lead to degraded performance metrics in the XO and XC experiments compared to the Same experiment.

3) Insufficient patch or context information. PLocator fails to detect the patch and context path precisely due to insufficient information, such as an anchor path with only a few anchors or does not contain rich auxiliary information. These anchor paths can be mistakenly identified in other similar code blocks and result in false identification.

PLocator for tiny patch modification identification.. Owing to the patch code localization and decision tree for function classification, *PLocator* can even detect patches with tiny modifications. For instance, the patch for `CVE-2017-13031` modifies only a single parameter, as illustrated in Figure 14a. *PLocator* successfully located the patch code in the target and accurately classified the true label by determining which parameter (i.e., represented as auxiliary information) is closer to the target.

Patch similar code interference analysis. Based on the statistics from our analysis of 73 CVEs across four projects, 17.7% of the vulnerable functions contain patch-similar code, affecting 140 test cases in $\mathcal{D}_{bal}$. These patch-similar code segments span no more than four lines of code, primarily involving function call modifications or the addition of simple checks. Robin and PS^3 correctly identify only 62% and 19% of these test cases, respectively, whereas *PLocator* accurately identifies 82%, owing to its patch path verification mechanism.

> **Answer to RQ1.** *PLocator* effectively distinguishes vulnerable functions from fixed ones, achieving an average of 82.6% precision, 86.3% recall, and 84.4% F1-score across the three experiments. Compared to the second-best approach, *PLocator* consistently outperforms it under varying compilation settings, with precision and recall improvements of 33.0% and 28.2%, respectively. The results indicate that *PLocator* effectively differentiates between vulnerable and fixed functions.

5.5 RQ2: Result of Patch Detection Efficiency

Figure 15 illustrates the time cost for a single test case across different approaches. The time includes the binary preprocessing and the patch presence test. The left figure displays box plots representing the time cost for each test case, while the right figure shows the mean and median time costs for each approach. As observed, BinXRay exhibits exceptional speed, averaging only 0.03 seconds per test case by focusing exclusively on syntactic-level feature extraction and comparison. In contrast, Robin and PS^3 require an average of 3.23 seconds and 17.7 seconds per test case, respectively, with the majority of their time spent on symbolic execution. For larger target functions exceeding 500 blocks, these methods become significantly more time-consuming due to the emulation of the target and may even fail due to unforeseen exceptions.

```
a/print-frag6.c b/print-frag6.c
index 03836adbb..fbcabc5b0 100644
--- a/print-frag6.c
+++ b/print-frag6.c
@@ -41,7 +41,7 @@ frag6_print(netdissect_options *ndo,
register const u_char *bp, register const u)
    dp = (const struct ip6_frag *)bp;
    ip6 = (const struct ip6_hdr *)bp2;

-   ND_TCHECK(dp->ip6f_offflg);
+   ND_TCHECK(*dp);
    if (ndo->ndo_vflag) {
      ND_PRINT((ndo, "frag (0x%08x:%d|%ld)",
```

(a) Patch of CVE-2017-13031 [3].

```
a/ssl/s3_clnt.c b/ssl/s3_clnt.c
index a6b3c01afa..c99a4c495e 100644
--- a/ssl/s3_clnt.c
+++ b/ssl/s3_clnt.c
@@ -915,6 +916,7 @@ int ssl3_get_server_hello(SSL *s)
        SSLerr(SSL_F_SSL3_GET_SERVER_HELLO,
        SSL_R_ATTEMPT_TO_REUSE_SESSION_IN_DIFFERENT_CONTEXT);
        goto f_err;
    }
+   s->s3->flags |= SSL3_FLAGS_CCS_OK;
    s->hit=1;
}
else  /* a miss or crap from the other end */
```

(b) Patch of CVE-2014-0224 [2]

```
// CVE-2013-6449 ssl_get_algorithm2 00
if (s->version >> 8 == 3 &&
    s->version > 770 &&
    s->s3->tmp.new_cipher->algorithm2 == 49200)
// CVE-2013-6449 ssl_get_algorithm2 02
if ( s->s3->tmp.new_cipher->algorithm2 != 49200 ||
     s->version >> 8 != 3 ||
     (result = 131200, s->version <= 770))

// CVE-2017-12895 icmp_print 00
if (*bp_0 == 11 || *bp_0 == 12)
// CVE-2017-12895 icmp_print 02
if ((unsigned __int8)(*v5 - 11) > 1u)
```

(c) Changes caused by optimization

Fig. 14. Examples of *PLocator*'s test cases. Figure (a) illustrates a test case successfully identified by *PLocator*, featuring a subtle patch modification involving a single parameter change in a function call. Figure (b) depicts a test case beyond *PLocator*'s current capability, where the patch introduces only an assignment statement. Figure (c) shows a test case in which *PLocator* produces an erroneous result due to variations in compilation settings.

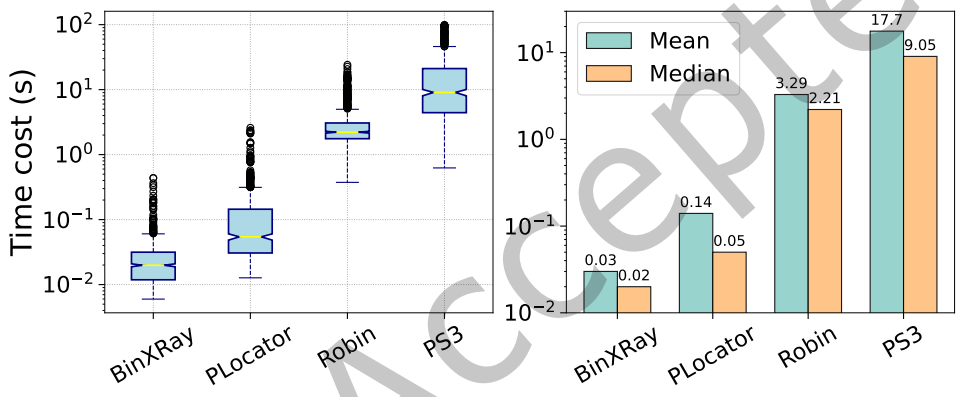


Fig. 15. The time costs of various approaches for the patch detection are presented, with values for Robin and PS^3 displayed on a separate scale for clarity. The left figure shows box plots representing the distribution of time costs across test cases, while the right figure depicts the mean and median time costs for each approach. All reported time costs include binary preprocessing (e.g., CFG extraction) and patch presence test for each function.

PLocator achieves an average detection time of 0.14 seconds per test case, maintaining consistent time efficiency regardless of function size. This remarkable efficiency is a result of the meticulously designed strategy aimed at reducing the number of candidate functions and anchor paths for matching, as outlined in § 4.3.2. By minimizing the number of potential anchors based on their values, auxiliary information, and the distance between adjacent anchors, we effectively reduce the search space, thereby significantly lowering the number of anchor paths to match. Additionally, all features are extracted directly from the CFG, eliminating the need for extra feature generation costs and enabling a rapid matching process.

> **Answer to RQ2.** *PLocator* efficiently performs patch detection in a minimal amount of time with 0.14s per target function on average, providing enhanced scalability for practical application in large-scale scenarios, all while maintaining superior accuracy compared to the baselines.

Table 4. The results of the patch presence test on the imbalanced dataset $\mathcal{D}_{imb}$ (%).

Experiment	**Same**					**XO**					**XC**					**Average**				
Metric	P	R	F1	SR	Acc	P	R	F1	SR	Acc	P	R	F1	SR	Acc	P	R	F1	SR	Acc
jTrans	3.5	98.7	6.7	100.0	35.6	5.2	72.8	9.7	100.0	71.1	2.1	66.2	4.1	100.0	27.2	3.6	79.2	6.8	100.0	23.3
BinXRay	88.9	43.2	58.2	5.3	5.1	60.0	5.9	10.8	3.5	3.3	23.1	4.1	6.9	4.2	3.8	57.3	17.7	25.3	4.3	4.1
Robin	5.1	83.4	9.6	88.7	52.0	3.8	69.3	7.2	88.7	50.4	3.5	44.6	6.6	85.8	56.3	4.1	65.9	7.8	87.7	52.9
PS3	3.6	70.3	6.8	75.9	30.6	2.9	52.0	5.5	61.6	23.5	3.0	54.1	5.8	71.8	30.4	3.2	58.8	6.0	69.8	28.2
PLocator$_{-Both}$	3.7	79.7	7.1	100.0	51.1	3.1	75.7	6.0	100.0	49.3	4.0	78.4	7.6	100.0	55.4	3.6	78.0	6.9	100.0	51.9
PLocator$_{-IFF}$	10.4	91.9	18.7	100.0	81.2	8.5	86.6	15.5	100.0	79.9	10.6	85.1	18.8	100.0	82.8	9.8	87.9	17.7	100.0	81.3
PLocator$_{-PPV}$	12.0	79.7	20.8	100.0	85.8	10.7	74.3	18.7	100.0	86.2	14.5	78.4	24.5	100.0	88.7	12.4	77.5	21.3	100.0	86.7
PLocator	**21.4**	**91.9**	**34.7**	**100.0**	**91.9**	**20.7**	**85.2**	**33.3**	**100.0**	**92.7**	**23.1**	**83.8**	**36.2**	**100.0**	**93.1**	**21.7**	**86.9**	**34.7**	**100.0**	**92.5**

* The high precision of BinXRay is because it returns "unknown" for most test cases, resulting in low false positives.

5.6 RQ3: Result of Patch Presence Test including Irrelevant Functions

In real-world scenarios (i.e., stripped binaries), irrelevant functions cannot be disregarded and must also be identified by patch presence test approaches, which is a more challenging task that has been overlooked by prior works [38, 43, 44]. To assess whether the approaches perform effectively when irrelevant functions are involved, we ran the baselines and *PLocator* on $\mathcal{D}_{+imb}$ to evaluate their performance in the patch presence test task and calculated the metrics based on their outputs on test cases. The results are presented in Table 4. The first five rows display the results of the baselines, while the last four rows show the results for *PLocator* and the ablation study on two of its components.

Comparison to the BCSD tool. As shown in Table 4, we include the BCSD tool jTrans as a baseline on the imbalanced dataset. jTrans achieves a precision of only 3.6%, requiring analysts to manually verify about $\frac{1}{3.6\%} = 27.8$ functions per vulnerability. By contrast, *PLocator* attains a precision of 21.7%, reducing the effort to only $\frac{1}{21.7\%} = 4.6$ functions per vulnerability. Although the precision is still modest, it represents an 83.5% reduction in manual verification effort, which is a meaningful improvement in practice. Thus, while *PLocator* does not entirely eliminate false positives, it significantly improves the practicality and efficiency of patch presence testing compared to BCSD alone.

Irrelevant functions interference analysis. As shown in Table 4, the support rate of BinXRay quickly drops from 58.2% to 5.3% in the Same experiment due to the large difference between the irrelevant functions and the two reference functions. Robin and PS^3 fail to identify a significant number of irrelevant functions, misclassifying them as vulnerable with an extremely low precision of 4.1% and 3.2%, respectively. This means that, for each vulnerability, they would incorrectly classify 25 and 31 test cases as vulnerable from the top 50 candidates, reflecting their ineffectiveness in scenarios where irrelevant functions are present. In contrast, *PLocator* recalls 86.9% of vulnerable functions with a precision of 21.7% across the three experiments on average, indicating that only 4 to 5 out of the top 50 candidates are incorrectly classified as vulnerable. Compared to the original top 50 candidates requiring verification, *PLocator* significantly reduces the analysts' workload, leaving only one-tenth of the test cases to be examined. This highlights its superior capability in distinguishing irrelevant functions from truly vulnerable or fixed ones.

Similar to **RQ1**, we present the score distribution of Robin, PS^3, and *PLocator* as depicted in Figure 16 for test cases with irrelevant functions. *Under ideal circumstances, all irrelevant test cases should yield a score of 0, indicating*

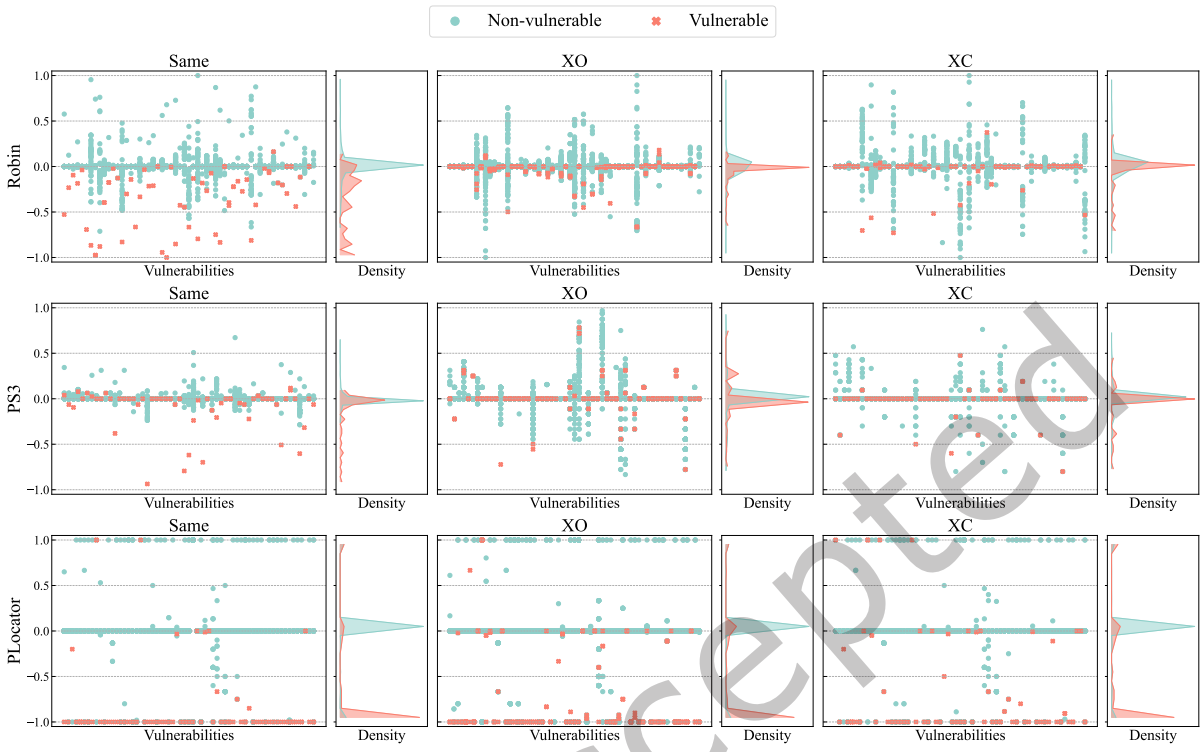


Fig. 16. Scatter and density plots of test case scores for Robin PS^3, and PLocator on the imblanced test dataset $\mathcal{D}_{imb}$. We normalize the score s produced by each method by subtracting the number of matched vulnerable features N_{vul} from the number of matched fixed features N_{fix}, then scaling the result to the range $(-1, 1)$ by dividing it by the maximum match number, i.e., $s = \frac{N_{vul} - N_{fix}}{\max(N_{vul}, N_{fix})}$. The y-axis represents the normalized score, while the x-axis denotes different vulnerabilities in the scatter plot and the probability density of test cases in the curve plot, respectively. Vulnerable functions (nodes) should appear below the zero-score line (i.e., $s < 0$), whereas non-vulnerable functions (nodes) should lie above or on the zero-score line (i.e., $s \geq 0$).

that they are neither vulnerable nor fixed. As observed, many irrelevant functions receive scores with high absolute values in both Robin and PS^3 across the three experiments, even surpassing those of vulnerable and fixed functions. This makes it infeasible to distinguish vulnerable and fixed functions from irrelevant ones by setting a score threshold, as it may inadvertently filter out true positives. These irrelevant functions contain features (either syntactic or semantic) extracted from the reference functions, which leads to their misidentification as vulnerable or fixed. This highlights that the identification of irrelevant functions is crucial and cannot be disregarded in the patch presence test task. In contrast, *PLocator* assigns accurate scores to the majority of test cases, effectively separating vulnerable and non-vulnerable test cases into two distinct groups: Scores greater than or equal to 0 (non-vulnerable functions) and scores less than 0 (vulnerable functions) , as demonstrated by the curve plots in Figure 16 across the three experiments.

Ablation study. As shown in the last four rows of Table 4, PPV ($PLocator_{-IFF}$) improves the recall by 12.7% and increase the precision from 3.6% to 9.8%. This component distinguishes the real patch code from patch-similar code by verifying the control logic between the patch and its context, enabling the successful recall of positives

previously misidentified. Additionally, many irrelevant functions that fail the patch path verification are correctly excluded, reducing false positives. IFF ($PLocator_{-PPV}$) primarily increase the precision to 12.4% with only a slight decrease in recall (1%). This component compares the entire function rather than focusing solely on features related to the patch code and its context, proving effective, particularly when the patch path and its context are short and provide limited information. When both components are employed, *PLocator* achieves optimal results, attaining a recall of 86.9% and a precision of 21.7%, as demonstrated in the last row of Table 4.

Answer to RQ3. *PLocator* effectively identifies vulnerable functions within a function pool containing numerous irrelevant functions, outperforming the second-best approach by 74.9% in accuracy, underscoring its practical value for the patch presence test task. By integrating its two core components, *PLocator* achieves optimal performance, boosting precision by a factor of five and improving recall and accuracy by 11.4% and 78.2%, respectively.

6 DISCUSSION

6.1 Threats to Validity

Internal validity. PLocator relies on a BCSD tool to match called functions between the reference and target functions when function names are unavailable. However, the effectiveness of this matching is contingent upon the performance of the BCSD tool. To address this, we require the tool to support cross-compiler and cross-optimization-level matching, and adopt jTrans, a state-of-the-art BCSD tool, to mitigate this threat. The selection of functions in an imbalanced dataset may influence the performance of patch presence test methods. To address this, we employ jTrans to recall the top-50 most similar functions from a large function set as target functions, rather than randomly sampling from the pool, thereby better simulating real-world scenarios. These functions are challenging to identify using the BCSD tool and can serve as a valuable benchmark to demonstrate the effectiveness of patch presence testing as a follow-up work to the function matching.

External validity. We focus on four widely studied projects previously evaluated in related work [38, 41], which may result in some specific vulnerabilities being overlooked. To mitigate this threat, these projects are deliberately chosen from diverse application domains, namely protocol encryption, packet processing, XML parsing, and font rendering. Furthermore, we collect vulnerabilities spanning seven distinct types and encompassing a range of function sizes to preserve the diversity and representativeness of our dataset.

6.2 Limitations of *PLocator*

PLocator has some limitations, which need to be overcome in future works.

(1) *PLocator* has to locate the patch code in the reference binary functions before generating the signatures. The core process patch code mapping § 4.2.1 requires the source patch file and the source code of both the vulnerable and fixed functions, which is unavailable in disclosed software. In future work, we aim to integrate *PLocator* with approaches such as SPAIN [39], which identify patch code directly from two reference binaries (vulnerable and fixed). With the assistance of such tools, *PLocator* would no longer require the source code of the reference functions and could instead generate signatures directly from the binaries.
(2) *PLocator* employs patch path verification to differentiate real patch code from patch-similar code by enforcing distance constraints between the matched context code and the patch code. However, when the patch-similar code maintains a consistent control flow relationship with the context and satisfies the distance constraints, *PLocator* may fail to distinguish such cases, resulting in misidentification.

(3) *PLocator* detects vulnerabilities based on specific given vulnerable and fixed functions, and cannot generalize to other vulnerabilities stemming from the same root cause but differing in implementation, which is also an inherent limitation shared by existing patch presence testing approaches. Considering the vast number of disclosed vulnerabilities in the NVD [6], *PLocator* plays a valuable role in detecting a large volume of such 1-day vulnerabilities in software systems. In the future, we would like to improve *PLocator* by capturing broader patterns of vulnerabilities, enabling detection beyond the specific vulnerable functions.

6.3 *PLocator* vs. Related Work

In this section, we review the advancements of existing methods in the patch presence test field, and compare them with *PLocator* on three challenges mentioned in § 1. The existing patch presence test methods can be roughly divided into two types based on the type of features they use, which are syntactic-based (T1) and semantic-based (T2). The patch presence test conducted by the methods includes two major stages, which are concluded as **Feature Generation** and **Feature Matching**. The methods are summarized in Table 5.

syntactic-based patch presence test. Fiber [44] was the pioneering work to introduce the concept of patch presence testing. It identifies representative patch modifications using predefined change selection rules, then attempts a coarse-grained match of the patch code within the target function based on block structure (i.e., CFG topology), followed by a precise comparison of the ASTs of the reference and target functions to ascertain the patch status. BinXRay [38] maps the changed patch blocks in binary by normalizing the instructions (i.e., replacing address, register, and memory with flags) and a block mapping algorithm. Then it extracts the traces at the block level to compare the similarity between the target function and the two reference functions to determine whether the patch is applied. PatchDiscovery [37] mitigates the over-fitting problem encountered in BinXRay by prioritizing the more representative key blocks (i.e., blocks with more instructions changed).

In summary, existing syntax-based methods demonstrate both effectiveness and efficiency when the reference and target binaries are compiled under identical settings. However, such consistency is seldom encountered in real-world scenarios. Their performance is highly sensitive to variations in compilers and optimization levels, which drastically reshape the syntactic and structural properties of the code, thereby constraining their practical applicability. Consequently, they fail to address challenge **C1**. Furthermore, these approaches rely on normalized instructions and block structures to locate corresponding code in the target function, an approach that often fails when the function is irrelevant to the reference. The extracted features or signatures do not differentiate between context and patch code, leading to false positives where patch-similar code is mistakenly identified as the actual patch. Therefore, they are also incapable of addressing challenges **C2** and **C3**.

Semantic-based patch presence test. PDiff [24] stands as one of the pioneering semantic-based approaches for patch presence testing. It identifies patch-related code blocks and derives semantic features—such as path constraints, memory states, and function call sequences—by simulating program execution through symbolic analysis. These features are encoded as path digests, and similarity between the reference and target functions is computed accordingly for classification. Robin [41] builds upon the observation that patches often alter execution paths, typically diverting conditions that would trigger vulnerabilities into error-handling routines. Leveraging this insight, Robin crafts inputs that can activate the vulnerable code, termed malicious function inputs (MFIs), and monitors the resulting side effects in the target function to inform classification. PS^3 [43] adopts a strategy akin to PDiff, generating semantic signatures from both the vulnerable and fixed reference versions. In the matching phase, it evaluates each semantic feature against those extracted from the target, aggregating the results into a final score using a weighted summation of all matched signatures.

In summary, existing semantic-based methods seek to capture the underlying semantics of patches by simulating program execution through symbolic execution. During the simulation of the two reference functions, they record

Table 5. Comparison between *PLocator* and related works. T1 and T2 refer to the two types of the patch presence test, which are syntactic-based (T1) and semantic-based (T2), respectively. The methodology is divided into two stages: **Feature generation** and **Feature matching**. The three columns labeled **C1**, **C2**, and **C3** correspond to the challenges outlined in section Introduction. ✔and ✘denote whether each challenge is effectively addressed or not, respectively.

Method	Feature Generation	Feature Matching	C1	C2	C3
Fiber (T1)	Root instructions and corresponding blocks	Rough matching with blocks, precise matching with ASTs	✘	✘	✘
BinXRay (T1)	Traces acorss patch code with normalized instructions	Mapping blocks and compare traces	✘	✘	✘
PatchDiscovery (T1)	Block paths with key basic blocks selected	Mapping blocks and compare paths	✘	✘	✘
PDiff (T2)	Patch digits alone the patch affected paths	Matching patch digits between the reference and target	✔	✘	✘
Robin (T2)	Malicious function input (MFI) and side effects	Matching side effects by feeding MFI into the target	✔	✘	✘
PS^3 (T2)	Side effects extracted through emulation	Matching side effects present in the target	✔	✘	✘
PLocator (T1)	Anchors and anchor paths	Matching and verify the anchor paths in the target	✔	✔	✔

side effects, such as memory states and path constraints, as semantic features. These features are then matched with those extracted from the target function to determine its classification. This enables them to effectively mitigate the impact of variations in compilers and optimization levels (**C1**). However, these methods often fail to account for the interference introduced by irrelevant yet semantically similar functions or patch-similar code within the vulnerable function. Such code fragments can exhibit comparable semantic characteristics, resulting in the misidentification of patch presence or absence. Consequently, these approaches fall short in addressing challenges **C2** and **C3**.

Comparison of PLocator with the existing methods. Compared to existing patch presence test methods, *PLocator* successfully addresses all three key challenges while maintaining high efficiency as a syntax-based approach. For **C1**, rather than tracking side effects through function-level execution simulation, *PLocator* adopts a novel representation of patch-related code that remains stable across compilation settings and enables efficient detection. For **C2**, prior methods often assume that function names are preserved in the target binary or that a BCSD tool can accurately retrieve the vulnerable or fixed functions, overlooking the influence of irrelevant functions in stripped binaries. *PLocator* mitigates this interference at both coarse-grained and fine-grained levels through several steps: filtering, matching, and verification, achieving superior accuracy and recall compared to baseline methods. For **C3**, earlier approaches typically treat the patch-related code as a monolithic unit during matching, which can lead to false positives when patch-similar code exists. In contrast, *PLocator* separates context from patch code and matches them independently to verify authenticity, thereby enhancing detection accuracy.

In summary, *PLocator* advances the patch presence test in practical settings, positioning itself as a more effective follow-up to BCSD methods in the detection of 1-day vulnerabilities.

7 CONCLUSION

Patch presence test is a critical and challenging task, serving as a follow-up work of binary code similarity detection (BCSD) in the 1-day vulnerability detection pipeline. Existing methods face difficulties in addressing interference from diverse compilers, optimizations, irrelevant functions, and patch-similar code blocks. We introduce *PLocator*, a novel approach leveraging patches and their context, represented through anchors, to accurately locate patch code and detect vulnerabilities in binaries with exceptional speed. Comprehensive experiments on two datasets (with and without irrelevant functions) demonstrate that *PLocator* significantly outperforms state-of-the-art approaches by a considerable margin.

REFERENCES

[1] 2014. CVE-2014-3470. https://nvd.nist.gov/vuln/detail/CVE-2014-3470.

[2] 2017. CVE-2014-0224. https://nvd.nist.gov/vuln/detail/CVE-2017-13031.
[3] 2017. CVE-2017-13031. https://nvd.nist.gov/vuln/detail/CVE-2017-13031.
[4] 2022. The best-of-breed binary code analysis tool, an indispensable item in the toolbox of world-class software analysts, reverse engineers, malware analyst and cybersecurity professionals. https://hex-rays.com/ida-pro/.
[5] 2023. Open source security and risk analysis report. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html.
[6] 2023. The U.S. government repository of standards based vulnerability management data represented using the Security Content Automation Protocol (SCAP). https://nvd.nist.gov/.
[7] 2024. Convert addresses into file names and line numbers.. https://linux.die.net/man/1/addr2line.
[8] 2024. CVEDetails. The ultimate security vulnerability data source. https://www.cvedetails.com/.
[9] 2024. Depth-first search. https://en.wikipedia.org/wiki/Depth-first_search.
[10] 2024. Greedy algorithm. https://en.wikipedia.org/wiki/Greedy_algorithm.
[11] 2024. Longest common subsequence. https://en.wikipedia.org/wiki/Longest_common_subsequence.
[12] 2024. MD5. https://en.wikipedia.org/wiki/MD5.
[13] 2024. A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission. https://www.ghidra-sre.org/.
[14] 2025. Breadth first search. https://en.wikipedia.org/wiki/Breadth-first_search.
[15] 2025. One-day, n-day, and zero-day vulnerabilities explained. https://fieldeffect.com/blog/1-day-0-day-vulnerabilities-explained.
[16] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2007. Code normalization for self-mutating malware. *IEEE Security & Privacy* 5, 2 (2007), 46–54.
[17] Silvio Cesare, Yang Xiang, and Wanlei Zhou. 2013. Control flow-based malware variantdetection. *IEEE Transactions on Dependable and Secure Computing* 11, 4 (2013), 307–317.
[18] Hui Chen. 2013. The Influences of Compiler Optimization on Binary Files Similarity Detection. https://api.semanticscholar.org/CorpusID:56574415
[19] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014).
[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *North American Chapter of the Association for Computational Linguistics*. https://api.semanticscholar.org/CorpusID:52967399
[21] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS*.
[22] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *ASE*. 896–899.
[23] Irfan Ul Haq and Juan Caballero. 2019. A Survey of Binary Code Similarity. *Comput. Surveys* (2019).
[24] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (2020). https://api.semanticscholar.org/CorpusID:226228290
[25] Christopher Krügel, Engin Kirda, Darren Mutz, William K. Robertson, and Giovanni Vigna. 2005. Polymorphic Worm Detection Using Structural Information of Executables. In *International Symposium on Recent Advances in Intrusion Detection*. https://api.semanticscholar.org/CorpusID:5066319
[26] Zhe Lang, Shouguo Yang, Yiran Cheng, Xiaoling Zhang, Zhiqiang Shi, and Limin Sun. 2021. PMatch: Semantic-based Patch Detection for Binary Programs. *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)* (2021), 1–10. https://api.semanticscholar.org/CorpusID:246080265
[27] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014). https://api.semanticscholar.org/CorpusID:7209096
[28] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Transactions on Software Engineering* 43 (2017), 1157–1177. https://api.semanticscholar.org/CorpusID:23190950
[29] Martin Reddy. 2011. Chapter 4 - Design. In *API Design for C++*, Martin Reddy (Ed.). Morgan Kaufmann, Boston, 105–150. https://doi.org/10.1016/B978-0-12-385003-4.00004-X
[30] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel J. Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *International Symposium on Software Testing and Analysis*. https://api.semanticscholar.org/CorpusID:6480274
[31] Josep Silva. 2012. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.* 44 (2012), 12:1–12:41. https://api.semanticscholar.org/CorpusID:16374826

[32] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Hang Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: jump-aware transformer for binary code similarity detection. *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022).

[33] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. 2022. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In *USENIX Security Symposium*. https://api.semanticscholar.org/CorpusID:251582778

[34] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In *ICSE*. 860–872.

[35] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *USENIX Security Symposium*. https://api.semanticscholar.org/CorpusID:219003482

[36] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *CCS*. 363–376.

[37] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, Zhaohui Zhou, Haijun Wang, and Ting Liu. 2023. PatchDiscovery: Patch Presence Test for Identifying Binary Vulnerabilities Based on Key Basic Blocks. *IEEE Transactions on Software Engineering* 49 (2023), 5279–5294. https://api.semanticscholar.org/CorpusID:265233966

[38] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020). https://api.semanticscholar.org/CorpusID:220497409

[39] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), 462–472.

[40] Shouguo Yang, Chaopeng Dong, Yang Xiao, Yiran Cheng, Zhiqiang Shi, Zhi Li, and Limin Sun. 2023. Asteria-Pro: Enhancing Deep Learning-based Binary Code Similarity Detection by Incorporating Domain Knowledge. *ACM Transactions on Software Engineering and Methodology* 33 (2023), 1 – 40. https://api.semanticscholar.org/CorpusID:255372539

[41] Shouguo Yang, Zhengzi Xu, Yang Xiao, Zhe Lang, Wei Tang, Yang Liu, Zhiqiang Shi, Hong Li, and Limin Sun. 2023. Towards Practical Binary Code Similarity Detection: Vulnerability Verification via Patch Semantic Analysis. *ACM Transactions on Software Engineering and Methodology* 32 (2023), 1 – 29. https://api.semanticscholar.org/CorpusID:259178169

[42] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, Aihua Piao, Jingling Xuey, and Wei Huo. 2019. B2SFinder: Detecting Open-Source Software Reuse in COTS Software. In *ASE*. 1038–1049.

[43] Qi Zhan, Xing Hu, Zhiyang Li, Xin Xia, David Lo, and Shanping Li. 2023. PS3: Precise Patch Presence Test Based on Semantic Symbolic Signature. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)* (2023), 2061–2072. https://api.semanticscholar.org/CorpusID:265695297

[44] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries. In *USENIX Security Symposium*. https://api.semanticscholar.org/CorpusID:51684590

[45] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. 2020. PatchScope: Memory Object Centric Patch Diffing. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (2020). https://api.semanticscholar.org/CorpusID:226228378