PS^3: Precise Patch Presence Test based on Semantic Symbolic Signature

Qi Zhan
Zhejiang University
Hangzhou, China
qizhan@zju.edu.cn

Xing Hu*
Zhejiang University
Ningbo, China
xinghu@zju.edu.cn

Zhiyang Li
Zhejiang University
Hangzhou, China
misakalzy@zju.edu.cn

Xin Xia
Software Engineering Application Technology Lab, Huawei
China
xin.xia@acm.org

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Shanping Li
Zhejiang University
Hangzhou, China
shan@zju.edu.cn

ABSTRACT

During software development, vulnerabilities have posed a significant threat to users. Patches are the most effective way to combat vulnerabilities. In a large-scale software system, testing the presence of a security patch in every affected binary is crucial to ensure system security. Identifying whether a binary has been patched for a known vulnerability is challenging, as there may only be small differences between patched and vulnerable versions. Existing approaches mainly focus on detecting patches that are compiled in the same compiler options. However, it is common for developers to compile programs with very different compiler options in different situations, which causes inaccuracy for existing methods. In this paper, we propose a new approach named ***PS***^{3}, referring to *precise patch presence* test based on *semantic-level symbolic signature*. ***PS***^{3} exploits symbolic emulation to extract signatures that are stable under different compiler options. Then ***PS***^{3} can precisely test the presence of the patch by comparing the signatures between the reference and the target at semantic level.

To evaluate the effectiveness of our approach, we constructed a dataset consisting of 3,631 (CVE, binary) pairs of 62 recent CVEs in four C/C++ projects. The experimental results show that ***PS***^{3} achieves scores of 0.82, 0.97, and 0.89 in terms of precision, recall, and F1 score, respectively. ***PS***^{3} outperforms the state-of-the-art baselines by improving 33% in terms of F1 score and remains stable in different compiler options.

KEYWORDS

Patch presence test, Binary analysis, Software security

*Corresponding Author

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Qi Zhan, Xing Hu, Zhiyang Li, Xin Xia, David Lo, and Shanping Li. 2023. PS^3: Precise Patch Presence Test based on Semantic Symbolic Signature. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software vulnerabilities are security issues that can cause software systems to be attacked or abused by malicious users. Vulnerabilities may allow hackers or malware to invade the system and steal sensitive information, or allow attackers to remotely control the system and perform malicious operations [26]. The number of newly discovered vulnerabilities has increased rapidly in recent years, according to CVE data [14].

Applying patches is one of the most effective ways to combat vulnerabilities and improve system security [1]. Software developers continuously release patches to fix known vulnerabilities. In a large-scale software system, there may be thousands of binary files, which may be different versions and developed by various vendors or developers. Therefore, it is essential to ensure that these binary files have been patched for the corresponding vulnerabilities. The process to ensure this is referred to "patch presence test".

The concept of the patch presence test was first introduced by Zhang et al. [37] in 2018 to distinguish it from conventional vulnerability searching for binaries or binary function matching. We can formalize the concept of patch presence test by describing its input and output. The input consists of three parts: (1) information about a particular patch, usually a patch file that describes the code modification. (2) The source code or the *reference* binaries of the project. The *reference* binaries typically include a binary compiled from a vulnerable version of the project and another binary compiled from the patched version. (3) The *target* binary to be tested. The output of the patch presence test is a binary decision on whether the specific patch is present in the target binary or not.

Determining whether a given binary file has been patched for a corresponding vulnerability is challenging. The semantic gap between the source code and the binary makes it difficult to directly correlate modifications in the source code with changes in the assembly code [37]. Additionally, patches are usually small and subtle. Li et al. [25] show that up to 50% of security patches only modify

less than 7 lines of code. The traditional methods of matching binary functions are no longer effective in detecting the existence of patches due to the small differences present between the vulnerable and patched functions.

Many works have been proposed [23, 33, 35–37] to automate patch presence test. These approaches can be divided into two groups: similarity-based and signature-based. Similarity-based approaches [23, 35, 36] extract information from *reference* binaries, then measure similarity to the *target* binary and decide the presence of patches based on similarity between two *reference* to *target*. Signature-based approaches [33, 37] extract signatures from modification and attempt to verify their existence in the target. Both the two types of approach rely heavily on syntactic information. However, binaries compiled with different options can be quite different at syntax level [7]. **As a result, existing methods are only effective when reference binaries are compiled with the same compiler options as target binaries, making it difficult to test the binaries when the options are different.**

In addition, compiler flags or optimization levels cannot be easily extracted from a compiled binary [30]. Existing approaches usually resort to tools such as BinDiff [38] to identify reference binaries most similar to the target binary [37] or compiling all binaries with the same compiler options in experiments. Generating the reference binary most similar to the target takes a long time. Jiang et al. [23] noted that the use of BinDiff [38] took six minutes in an end-to-end test. **Efficiency is limited in large-scale software systems with thousands of binaries to test.**

To address these problems, we propose ***PS***^{3}, with the aim of testing the target binary precisely when the compiler options differ. The differences that represent the semantic information of the patch are stable in all downstream binaries, making them useful in the patch presence test. To capture semantic information in assembly code, we need to understand its context. To compare signatures precisely, we are required to check the equivalence at a semantic level rather than a syntax level. Following the intuition above, ***PS***^{3} extracts the semantic signatures based on symbolic emulation and matches them at a semantic level. The main idea of our approach is two-fold:

❶ In signature extraction, we use symbolic emulation to emulate the entire affected function and collect side effects (e.g., store data in memory) in a symbolic environment. From function entry, we execute the code in a symbolic environment and collect effects. Then we obtain four types of signatures from the effects of affected functions.

❷ In signature matching, we compare the signature extracted from the target binary with references. We utilize a theorem prover to prove the equivalence of two signatures that differ in syntax but are equal at semantic level. ***PS***^{3} is capable of precisely testing the target binary with respect to the compiler options by matching the semantic level signatures.

The binaries in the existing dataset are always compiled with the same compiler options [36], which cannot reflect the performance of the approaches when the compiler option differs. To fill this gap, we construct a dataset consisting of 3,631 (CVE, binary) pairs on 62 CVEs of four popular C/C++ projects. A test in our dataset is to check if a specific patch exists in a specific binary. We compile the binaries with combinations of different optimization levels and different compilers in the experimental phase.

Experimental results show that our approach outperforms state-of-the-art approaches by 0.22 in terms of the F1 score. We also evaluate the results of our approach compared to BinXray [36] for every combination of compiler and optimization level. The findings of our study establish the reliability and accuracy of our approach, indicating that ***PS***^{3} is capable of accurately capturing semantic signatures and successfully matching them with target binaries.

Contributions: In summary, the main contributions of this paper can be summarized as follows:

- We propose ***PS***^{3}, a signature-matching-based approach, which can precisely and efficiently test the presence of patch in target binaries by symbolic emulation.
- We construct a dataset comprising 62 CVEs and corresponding binaries compiled with various compiler options from four popular C/C++ projects, resulting in a total of 3,631 (CVE, binary) pairs.
- We systematically evaluate our approach on the dataset, and the results demonstrate its effectiveness in patch presence test with high accuracy.

The remainder of the paper is organized as follows. Section 2 presents the motivating example of our study. Sections 3 describe the design and implementation of the framework. Section 4 and Section 5 discuss the evaluation steps and results of ***PS***^{3}. In Section 6, we discuss the threats to validity of our approach and provide a case study of some representative patches. Section 7 summarizes the research related to the field. We conclude the paper and discuss the future work in Section 8.

2 MOTIVATING EXAMPLE

In this section, we walk through a motivating example to illustrate our intuition. Figure 1 shows the testing process of the security patch for CVE-2020-22019 [11], a vulnerability in FFmpeg. The modification in the source code introduces a condition to check if the value in `w` or `v` is less than 3, where `w` is the second parameter of the function `ff_vmafmotion_init` and `h` is the third parameter.

To test the presence of the patch, one straightforward idea is to find out what the binary code is compiled from the added condition and directly compare the extracted assembly code or similar pattern against the target binary. The above line of source code might be compiled to `cmpl $0x2, 0x14(%rsp); jle 0x1ab3b3`. However, matching the assembly code alone would not be helpful, as the concrete addresses and registers can vary across different binaries. For example, the target binary may use `ebx` or `ebp` as registers other than saving the value to stack. Additionally, the compare and jump pattern is very common in binaries, which can lead to many false positives if directly matched.

The above straightforward idea does not work well, as direct mapping of the source code to binary and matching cannot capture the semantic change of the patch. This limitation arises from register allocation and the absence of contextual information. From the assembly code `cmpl $0x2, 0x14(%rsp); jle 0x1ab3b3`, we know nothing about what the value pointed stands for in the current context. The requirement about context reminds us that we can track the whole dataflow in a symbolic environment from the

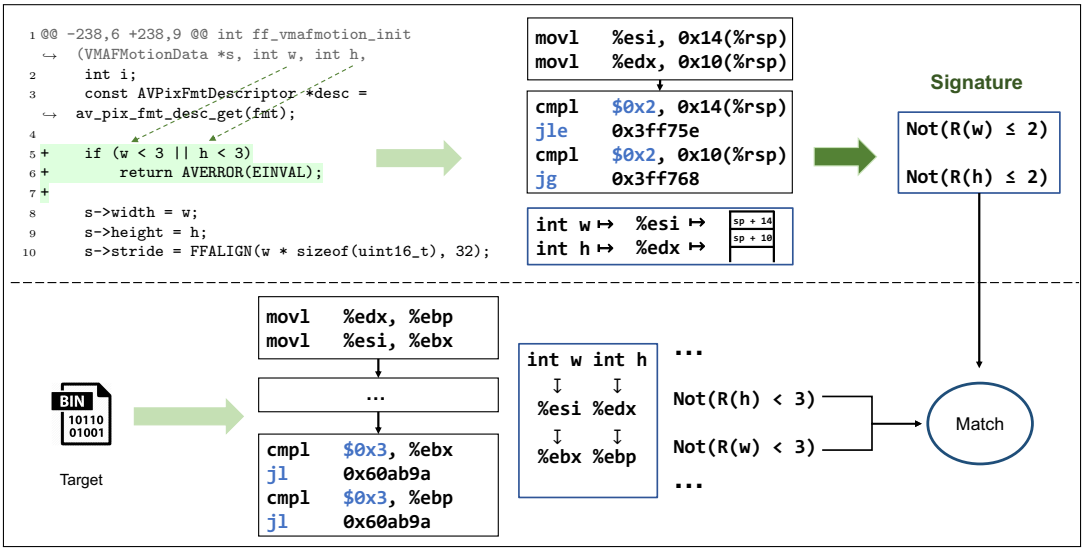

Figure 1: Motivating example

function entry point. To detect the status of the target binary after extracting the symbolic signatures, it suffices to test whether the signatures equivalent to `w < 3` and `h < 3` exist in the function.

Based on the idea mentioned above, we proceed with the following steps.

Extraction. We leverage symbolic emulation instead of concrete executing to track the changes in registers and memory relative to the function parameter and initial memory environment. After that, we extract signatures from the effects of the corresponding assembly code. In the prologue of the function shown in Figure 1, the register represents the second parameter of the function saved to stack with offset `0x14`. The code `cmpl $0x2, 0x14(%rsp)`, which corresponds to condition `w < 3` in the source code, can be traced back to the parameter `w`. The code `cmpl $0x2, 0x10(%rsp)` can be traced back to `h` in the same way. As a result, we obtain the signatures `Not(R(w) <= 2)` and `Not(R(h) <= 2)`, where `Not` denotes not jumping the branch. The signatures remain constant in different binaries because the order of parameter passing is consistent with typical compiler behavior. Therefore, we can utilize them to precisely test the target binary.

Matching. Once the signature is generated, we perform the same symbolic emulation on the target binary function directly to collect effectes. Since we cannot map the assembly code back to the source code in the target binary, all effects collecting in emulatoin are considered as potentially matched signatures. The remaining problem is how to match the effects collected from the target binary with the signatures. The signature `Not(R(h) < 3)` does not appear to be exactly equivalent to `Not(R(h) <= 2)` extracted from the target binary, but is semantically equal when `R(h)` is an integer. With the use of a theorem prover, we are able to ensure that the two expressions are equal at the semantic level. By matching the two expressions, we decide that the target binary is patched.

3 PROPOSED APPROACH

3.1 Overall Framework

The overall framework of ***PS***^{3} is illustrated in Figure 2. For simplicity, we assume that the patch modifies only one function. In practice, we classify the binary as vulnerable if one affected function is identified as vulnerable. The main components of our approach include a symbolic emulator that emulates the function and generates side effects, a signature extractor responsible for extracting defined signatures from the effects, and a matching engine that determines the presence or absence of the patch by comparing the signatures.

3.1.1 Symbolic Emulator. The emulator serves as the core of our approach. We first compile the source code into two reference binaries (that is, one vulnerable version and one patched version) with debug information and parse the difference file to extract the code modifications. Then, we utilize the debug information to identify which lines of reference binary code are deleted or added due to modifications in the source code. During the extraction phase, we perform symbolic emulation on the control flow graph of the binary function. For reference binaries, it is necessary to trace only the effects corresponding to the modified code, i.e., red curves for deleted code and green curves for added code in Figure 2. For the target binary, we need to collect the *complete* set of signatures in every possible affected function, i.e. gray curves since we cannot map the binary code back to the source code by debug information.

3.1.2 Signature Extractor. The extractor utilizes the emulator to perform symbolic emulation rather than concrete execution on the specific functions of the reference and target binaries, thus obtaining the effects. We extract these side effects as four types of signature: writes data to register, stores data to memory, conditions, and calls or returns value. The formal definition of signatures is depicted below. Once the signatures have been extracted from the

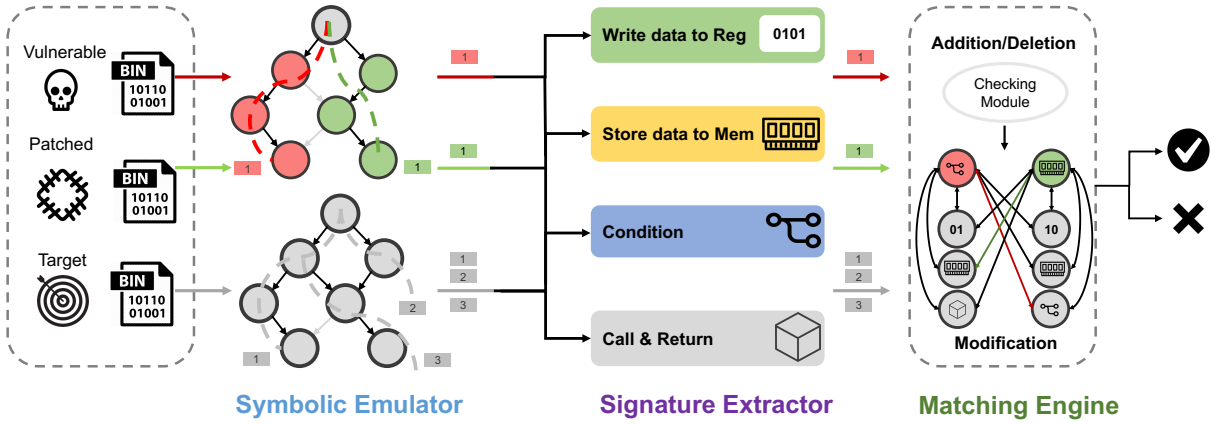

Figure 2: Overall framework of *PS*^{3}

two reference binaries successfully, we can always use the same signatures for every target rather than searching for the most similar reference each time, saving a lot of time in practice.

3.1.3 Matching Engine. In the matching phase, we compare the signatures of the target function with the two reference signatures one by one and determine whether the target binary has been patched or not.

3.2 Symbolic Emulation

The Algorithm 1 shows the high-level algorithm of symbolic emulation. During the emulation phase, we start from the entry of the function and perform a deep-first emulation (from line 4 to line 17). If a basic block is visited, it will be excluded during the next iteration (line 6). After emulating a basic block, we push the successor states to the queue (line 13) and collect side effects (line 11).

When emulating a basic block (line 19), we emulate all the instructions in the basic block. All calls to other functions are skipped and a symbolic value is assigned to the register representing the return value, which records the function call name and parameters encountered during the emulation. We simply ignore all other possible side effects during this process. As mentioned before, the register and memory accessed can be represented by a symbolic value for the first time, e.g., function parameters accessing, and the actual operations on registers or memories are replaced by symbolic operations.

The choice of memory model has always been a problem in symbolic execution, since we cannot make a precise presumption for a symbolic pointer [3]. The problem still exists in symbolic emulation. We need to maintain a mapping between the symbolized addresses and their corresponding symbolic values, which ensures that we can correctly reuse the symbolic values when reading memory. In ***PS***^{3} we use a simple memory model: Every symbolized address that is structurally different corresponds to a different symbolic value. Considering the widely used stack to save local variables, the calculation involved with stack pointer is maintained separately with offset.

The main difference between symbolic emulation in our approach and traditional symbolic execution is that we do not aim to solve the constraints to obtain the concrete values that can reach specific basic blocks of the function. Our approach focuses on the semantic effects that a patch to the source code represents, rather than on constraint solving and concrete values, since the source code changes are always reachable.

Algorithm 1: Symbolic emulation procedure

Input: Initial state, s_i
Output: Collected effects, T

```
1  Q ← a queue of states initialized of s_i ;
2  visited ← an array of booleans initialized to false;
3  T ← ∅ ;
4  while Q is not empty do
5      pop the last element s from Q;
6      if !visited[s] then
7          visited[s] ← true;
8          next_states = emulate_basic_block(s.block);
9          for state ∈ next_states do
10             if state is dead then
11                 T ← T ⋃ state.trace;
12             else
13                 push state to Q;
14             end
15         end
16     end
17 end
18 return T;
19 Function emulate_basic_block(block):
20     while state.successors less than 2 do
21         for instruction in block do
22             state = emulate(instruction);
23         end
24     end
25     return state;
```

3.3 Signature Definition

After generating the traces from the emulation, we need to extract useful information for matching. We define a binary signature as a group of side effects. Inspired by Egele et al. [18], we choose the call to function, write to register or memory, condition statement, and return value as signatures. We formalize the grammar of the signature in the Backus-Naur Form (BNF) as follows:

$$
\begin{array}{lcl}
\langle Signature \rangle & ::= & \langle FunctionCall \rangle \\
& | & \langle RegisterWrite \rangle \\
& | & \langle MemoryStore \rangle \\
& | & \langle Condition \rangle \\
& | & \langle Return \rangle \\
\langle FunctionCall \rangle & ::= & \text{name } \langle exp \rangle^* \\
\langle RegisterWrite \rangle & ::= & \text{offset } \langle exp \rangle \\
\langle MemoryStore \rangle & ::= & \langle exp \rangle \ \langle exp \rangle \\
\langle Condition \rangle & ::= & \langle exp \rangle \\
\langle Return \rangle & ::= & \text{name} \\
& | & \text{index} \\
\langle exp \rangle & ::= & \langle exp \rangle \text{ binop } \langle exp \rangle \\
& | & \text{unop } \langle exp \rangle \\
& | & \text{constant} \\
& | & \text{symbol}
\end{array}
$$

$\langle FunctionCall \rangle$ is represented as the name of the function followed by its parameters, $\langle RegisterWrite \rangle$ is represented by the index followed by the value to be written, and $\langle MemoryStore \rangle$ is represented by the address and the value to be written. A single Boolean expression represents a $\langle condition \rangle$. If the name of the function call can be found in the symbol table, the returned value can be labeled accordingly. Otherwise, we assign an incremental index to it. Finally, an expression $\langle exp \rangle$ can be a combination of other expressions and operators, or it can be a constant value or a symbol mentioned in symbolic emulation.

The collection of write to register and store to memory is straightforward. We collect the condition information in every branch. To collect information from the function call, we first obtain the name and number of parameters of a function call from the source code and use it to extract parameters from the actual function call. The return value is assigned after a function call.

There are many signatures to be extracted in the extraction phase as we record every possible behavior. However, some traces impact the decision, for example, common register write like "`RegWrite R0, 1`". Therefore, we implement certain rules to sanitize the signatures:

- For the signatures of writing to memory and register, we remove the signature if the value has been used later to ensure our signatures are small and precise, which reduced the duplicate matches.
- In our approach, we prefer modified hunks over hunks that simply add or remove content, since there are unique signatures for the two references, respectively, and we can make a more precise comparison between the vulnerable and patched functions.

3.4 Signature Matching

In the signature matching phase, we first check the extracted signatures before comparing them. If the signatures in the patched binary contain signatures in the vulnerable binary, we call it *a pure addition*, since the change is essentially an addition of semantics. In the same way, we call it *a pure deletion* if the situation is the opposite. If the modification is a pure addition or a pure deletion, we require all $\langle condition \rangle$ and $\langle call \rangle$ to match in the target binary. The target is classified as vulnerable or patched if the requirement is not satisfied, respectively. We do not decide the pure addition or deletion by looking at the difference file directly, since the result is easily affected by the syntax change in the source code. On the other hand, the semantic level signature is stable and they are not affected by syntax differences.

After checking procedure, we compare the signatures extracted from the reference and target binary one by one to decide whether the target has been patched or not. The *match* function shows the detailed comparison process. For the same type of signature, we compare every component of them and think that they are matched only when all components are equal.

$$
match(s_1, s_2) = \begin{cases}
s_1.name = s_2.name \wedge \bigwedge_i s_1.para_i = s_2.para_i, & t_1 = t_2 = \text{Call} \\
s_1.index = s_2.index \wedge s_1.value = s_2.value, & t_1 = t_2 = \text{RW} \\
s_1.addr = s_2.addr \wedge s_1.value = s_2.value, & t_1 = t_2 = \text{MW} \\
s_1.exp = s_2.exp, & t_1 = t_2 = \text{Condition} \\
s_1.name = s2.name \vee s_1.index = s_2.index, & t_1 = t_2 = \text{Return} \\
\text{false}, & \text{otherwise}
\end{cases}
$$

where t_1 is the type of s_1, t_2 is the type of s_2.

The score of reference to target is defined as the weighed sum of all matched signatures. The matching of call expressions and condition expressions is much more important than writing to memory or registers, since they are more likely to be unique. Therefore, we assign a higher weight to the matching of call and condition expressions than to the writes. By comparing the weighted sum of matched signatures between the vulnerable and patched reference binaries, we decided whether the binary is patched. If the score of the vulnerable reference to the target is not less than patched, we classify the hunk as vulnerable. Considering the purpose of the patch presence test, we determine that the function is patched only if every hunk in the function has been patched and that binary is patched only if every affected function has been patched.

Furthermore, we also incorporate source code information into the signature matching. When the parameter of a function call is a string, we will ignore it in the signature match. Technically, we assign a wildcard symbol to the parameter, and all other values will be considered equal to it. For example, when matching the signature `call(1, "string")`, the second parameter is ignored. Additionally, when the patch code is contained in a conditional statement, which is a common pattern in vulnerability fixes, we deduce that the target binary must contain a matching signature for $\langle Condition \rangle$. If there is no matching signature of the condition type, we directly classify the target as vulnerable. The patch code in the motivating example is contained in if (w<3 || h<3), so the target can be classified as vulnerable at once if there is no signature equal to `w<3` or `h<3`.

3.5 Implementation

PS^3 is built on top of the VEX intermediate representation (IR) [28] supported by angr [32]. VEX IR is an intermediate representation of the instruction sets used in the Valgrind dynamic binary instrumentation framework [29]. ***PS***^{3} can work with various architectures, since we can access the source code and compile it into the corresponding binaries. Angr is an open source binary program analysis framework in Python and has been used extensively in binary analysis [8, 17, 22], as well as patch presence test [23, 33, 37]. We use Z3 [15] to simplify the expression and calculation of the stack address offset to ensure that the memory mapping is more precise. In signature matching, we also use Z3 to prove equality of signatures nontrivial equation as in the motivating example.

4 EXPERIMENTAL SETUP

We evaluate the effectiveness of our approach following four research questions.

- **RQ1:** How effective is our approach at patch presence test compared to the state-of-the-art baselines?
- **RQ2:** How practical is our approach when testing binaries compiled with different options?
- **RQ3:** What is the result for each CVE and each project?
- **RQ4:** How efficient is ***PS***^{3}?

4.1 Dataset

To evaluate our approach, we need to test binaries with various compiler options. The binaries in the existing dataset [36, 37] are always compiled with the same options. They cannot reflect the ability to test binaries with various compiler options. We collect CVE information to build our dataset, as depicted in Figure 3. We chose OpenSSL[1], FFmpeg[2], Tcpdump[3], and Libxml2[4] as our evaluation projects. The four projects involved cryptographic protocols, video processing, packet and xml analyzer. They are widely used in studies on vulnerability matching and patch presence testing [33, 36]. We only consider these four popular C/C++ projects in our experiments, although the method we proposed is general for compiled language.

For these projects, we extract vulnerability information from its official website instead of the CVE website, as it provides well-documented security updates. In OpenSSL, the affected version and the fix version are well documented. This allows us to directly determine the time of the first occurrence of a vulnerability and the corresponding patched version. For other projects, we manually search the code base and decide on the start scope affected by the vulnerability. We selected CVEs from the past five years. For a vulnerability that exists in multiple branches, we choose only one branch for the test.

After collecting the CVE information and source code, we first generate difference files based on the patch commit. Then we compile the source code into two sets of binaries: reference binaries and target binaries.

[1]https://www.openssl.org/news/vulnerabilities.html
[2]https://ffmpeg.org/security.html
[3]https://www.tcpdump.org/public-cve-list.txt
[4]https://gitlab.gnome.org/GNOME/libxml2/-/releases

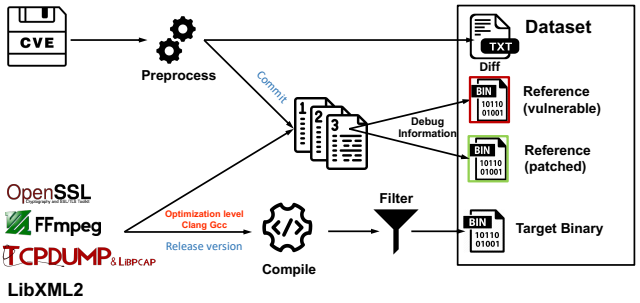

Figure 3: Dataset collection process

To generate reference binaries, we compile the corresponding source code using gcc compiler with `O0` optimization level and `-g` for debug information. We compile two reference binaries for each CVE. One binary represents the patched version, which is compiled by the patch commit, and the other represents the vulnerable version, which is compiled by the commit just before the patch commit.

To generate binaries for testing, we compile the corresponding source code using gcc and clang compilers with the optimization level ranging from `O0` to `O3` respectively, resulting in a total of **eight** different binaries. In our experiments, the gcc version is 9.4.0 while the clang version is 13.0.0. Kim et al. [24] have pointed out that the compiler version has a small impact compared to the optimization levels, so we only consider the optimization levels and ignore the different versions of the same compiler when generating target binaries. We choose to compile the source code to every release version, instead of compiling from the patch commit, as we did for generating the reference binaries. In addition, we manually filter out binaries when affected functions are missing in symbol tables due to function inline.

We determine the ground truth using the affected version information extracted from CVE. We consider the version in the *same branch* before the patched version as vulnerable and the patched and subsequent versions as patched. An item in our dataset is a (CVE, binary) pair that determine whether a specific binary contains a specific vulnerability, see Table 2 for example. As a result, we collected a total of 62 CVEs and created 3,631 pairs, consisting of 1,582 vulnerable pairs and 1,893 patched pairs. Detailed statistical data on the created dataset for each combination of configurations for each project can be found in Table 1.

4.2 Baselines

To evaluate the effectiveness of our approach, we compare it with the following two baseline models:

4.2.1 BinXray[36]. A state-of-the-art patch presence test tool without the presence of source code. BinXray uses basic block mapping to extract the signature of a patch by comparing vulnerable and patched binary programs. Then, it uses trace similarity to identify whether a target program is patched or not.

4.2.2 Asm2Vec[16]. A deep learning-based binary clone search method that is designed to combat code obfuscation and compiler

Table 1: Statistics of our dataset

Projects	#CVE	#Version	Clang & O0		Gcc & O0		Clang & O1		Gcc & O1		Clang & O2		Gcc & O2		Clang & O3		Gcc & O3		#Test
			#V	#P	#V	#P	#V	#P	#V	#P	#V	#P	#V	#P	#V	#P	#V	#P	
OpenSSL	23	32	143	161	147	163	137	135	144	152	137	135	132	114	137	150	132	114	2,233
FFmpeg	26	37	75	115	75	115	51	76	56	78	51	76	43	71	51	76	43	71	1,123
Tcpdump	11	4	33	11	33	11	21	7	24	8	21	7	24	8	21	7	21	6	263
LibXml2	2	2	2	0	4	0	2	0	1	0	1	0	0	0	1	0	1	0	12
Total	62	75	253	287	259	289	211	218	225	238	210	218	199	193	210	233	197	191	3,631

#P represents the number of patched pairs while #V represents vulnerable pairs.

Table 2: (CVE, binary) pair example

CVE	Target	Version	Option	Ground Truth
2018-14468	Tcpdump	4.9.3	O0 & Clang	patch

optimization. We chose it for comparison to demonstrate the effectiveness of directly utilizing binary code similarity for patch presence testing.

We choose not to select [23, 33] as our baselines, since there are no open source implementations available, making it difficult to compare them with our approach. FIBER [37] is primarily designed for patch presence tests on Android kernel images in AArch64 architecture. Sun et al. [33] pointed out that the signature generated by FIBER is kernel related and not suitable for other binaries, which cannot be directly compared with our approach using our dataset.

4.3 Metrics

Since the patch presence test is essentially a binary classification task, we select precision (P), recall (R) and F1 score (F1) as metrics to evaluate our approach with other baselines.

Precision refers to the case where the test correctly classifies a vulnerable binary. We define R_R as the number of truly vulnerable binary detected, R_I as the number of patched binary recognized as vulnerable, then precision can be expressed as:

$$P = \frac{R_R}{R_R + R_I}$$

Recall is the ratio of the number of actual vulnerable binary detected to the number of all vulnerable binary. R_R is defined as the number of truly vulnerable binary retrieved, and R_N is the number of all vulnerable binary in the dataset. Recall can be defined as follows:

$$R = \frac{R_R}{R_N}$$

F1 Score stands for the overall performance of the test, represented by the harmonic mean of precision and recall. Given P as the precision and R as the recall, the F1 score is calculated as follows:

$$F1 = \frac{2PR}{P + R}$$

4.4 Experimental Setting

PS^3 does not require additional configurations. We use IDA Pro binary, a disassembler tool [31] to dump the binary code of the functions for BinXray, which is the same as the BinXray used in their original experiments. To the best of our knowledge, no official open source implementation of Asm2Vec is available, so we utilized an unofficial implementation[5] followed [34] with default parameter settings. The experimental setup consists of an Intel CPU operating at 2.90GHz and running on Linux.

Table 3: Effectiveness of our approach vs. baselines

Approach	Precision	Recall	F1
Asm2Vec	0.60	0.50	0.65
BinXray*	0.51	0.96	0.67
PS^3	**0.82**	**0.97**	**0.89**

* In our experiments, BinXray returns the result of "unknown" for most targets when the compiler option combination is not gcc & O0. We consider all "unknown" binaries vulnerable for practical purposes and present the corresponding results.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Our approach vs. Baselines

To evaluate the effectiveness of ***PS***^{3}, we compare our approach with two state-of-the-art approaches, including BinXray [36] and Asm2Vec [16] in terms of precision, recall, and F1 score using our dataset. As shown in Table 3, our approach outperforms all state-of-the-art baselines in terms of precision, recall, and F1 score for practical purposes, achieving 0.82, 0.97, and 0.89 respectively.

Asm2Vec [16] is a state-of-the-art approach that learns assembly representations. We directly use the function-level representation to match the target function. Asm2Vec cannot distinguish vulnerable and patched functions, and precision and recall are both about 50%. The reason may be the lack of capturing small changes between vulnerable and patched functions, as we have previously proposed.

BinXray [36] is a state-of-the-art patch-based vulnerability matching tool and performs better than Asm2Vec, which achieves an F1 score of 0.67 in our dataset. For the tested binaries BinXray returns "Unknown", we make the most conservative assumption for it, i.e., considering all binaries as vulnerable.

Answer to RQ1: ***PS***^{3} can effectively identify the patch in the target functions with an F1 score 0.89. It outperforms the state-of-the-art approaches, Asm2Vec and BinXray, by 37% and 33%, respectively.

[5] https://github.com/oalieno/asm2vec-pytorch

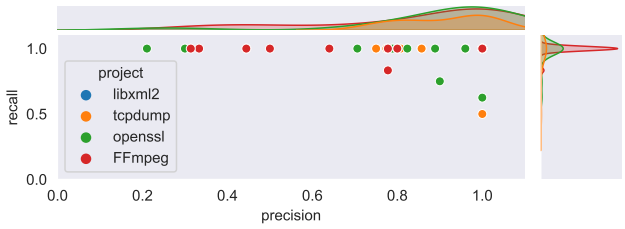

Figure 4: Results of every CVE in different projects

5.2 RQ2: Specific Compiler Optimizations and Compilers

To evaluate the effectiveness of our approach in different compiler options, we analyze the precision, recall and F1 score for every combination, as Table 4 shows. We have the following findings:

Finding 1. The precision and F1 score of our approach are always higher than the results of BinXray in each combination of compiler and optimization levels, showing that ***PS***^{3} can test the presence of patch. Both ***PS***^{3} and BinXray perform best on the combination of O0 & gcc, it is understandable since the reference binary is compiled with O0 & gcc.

Finding 2. The precision of our approach is affected by optimization levels, the F1 score falling within the range of 0.93 to 0.90 in the gcc compiler and 0.89 to 0.86 in the clang compiler, respectively. Although our result cannot ignore the differences between all optimization levels and always make the right decision, the situation migrated. The compiler also affected the accuracy of ***PS***^{3}, as different compilers generate and optimize assembly and can be quite different [24].

Finding 3. When analyzing binaries compiled with different compiler options, BinXray drops to only 50% precision, while it achieves 0.74 for O0 & gcc. BinXray cannot determine whether the binary is vulnerable or patched. The results indicate that BinXray can accurately determine vulnerability or patch status only when the target and reference binary are compiled using the same options, while in other situations it is impractical.

Compared to BinXray, which can even decide the status of binaries with different compiler options most of the time, ***PS***^{3} can actually capture the key semantic differences of patches rather than syntax differences in binaries, leading to a more precise decision with different compiler options.

> **Answer to RQ2:** ***PS***^{3} maintains high performance with different compiler options. Our approach yielded the greatest improvement in F1 score when compiled with gcc and O1 optimization level, resulting in a 44% increase.

5.3 RQ3: Results on Different Projects and CVEs

To better understand the results of the experiments, we also list the precision and recall of each CVE used in our dataset in Figure 4. For 35 out of 62 CVEs, ***PS***^{3} accurately distinguishes between vulnerable and patched versions, which achieve a perfect F1 score. The recall of 58 CVEs is 100% and the precison is lower respectively, since our approach is strict to decide a binary patched for practical use. In addition, our approach keep stable across various projects.

Our approach fails to identify four CVE (F1 score lower than 0.6). We check out each CVE manually and find that the five patches consist of one arithmetic optimization, one mentioned in the limitation below, and others are unclassified correctly since there is too much diff to confuse ***PS***^{3}.

> **Answer to RQ3:** ***PS***^{3} achieves a 100% recall rate for 94% of the CVEs and a 100% F1 score for 56% of the CVEs.

5.4 RQ4: Efficiency of our approach

We evaluate the efficiency of ***PS***^{3} by measuring the time cost. Table 5 shows the cost time of our approach and other baselines.

Preprocessing Time. Preprocessing is not required in our approach since ***PS***^{3} takes binaries and patch file as input directly. On the other hand, the preprocessing time of IDA Pro required by BinXray for each file is 110 seconds in our experiments. It takes an average of 257 seconds to extract affected functions in a file in the FFmpeg project.

Test Time. It takes around 7 seconds to complete a test in our approach. The test with maximum time cost is about 45 seconds. We manually check the test. It is a vulnerability of function `ff_mpeg4_decode_picture_header` in FFmpeg. There are a total of 570 basic blocks in the function, so emulation of the entire function can cost. In addition, signature matching is slower due to the increased number of signatures to be matched. Compared to our approach, BinXray is much faster in the test procedure, on average 60 milliseconds.

It seems that our approach is fairly slow compared to BinXray in terms of test time, but our approach is faster in practice considering the long preparation time in BinXray. In addition, a 7.4 seconds cost for determining a patch presence of a specific binary is acceptable for practical use. Furthermore, the signatures generated by our approach can be applied to multiple binaries directly, resulting in significant time savings.

> **Answer to RQ4:** ***PS***^{3} can test the patch presence of binaries with an average of 7.4 seconds, without the time-consuming preprocessing.

6 DISCUSSION

6.1 Qualitative Analysis

In this section, we show the strength and limitation of some representative security patches in our dataset.

Constant change. It is common to change constant operands in a patch, see Figure 5a for example. It may be hard to identify this, as neither the control flow graph nor the data flow is altered. To address this situation, ***PS***^{3} can extract semantic changes to detect such differences. For CVE-2018-0735, ***PS***^{3} can discover that the parameter of the function call `bn_wexpand` and `BN_consttime_swap` change from `group + 1` to `group + 2`, then matches it with the signatures extracted from target binaries.

Table 4: Results on different compiler options

Compiler option combination	BinXray*			***PS***^{3}		
	Precision	Recall	F1	Precision	Recall	F1
Gcc & O0	0.74	0.99	0.85	0.91	0.94	0.93
Gcc & O1	0.47	0.97	0.63	0.83	1	0.91
Gcc & O2	0.51	0.95	0.66	0.82	1	0.90
Gcc & O3	0.51	0.95	0.66	0.82	1	0.90
Clang & O0	0.46	0.97	0.63	0.83	0.95	0.89
Clang & O1	0.49	0.92	0.64	0.80	0.96	0.87
Clang & O2	0.48	0.96	0.64	0.77	0.96	0.86
Clang & O3	0.47	0.96	0.63	0.77	0.96	0.86

* We considering the "unknown" response as vulnerability in BinXray as we mentioned in RQ1.

Table 5: Time cost of our approach vs. BinXray (seconds)

Approach	Preprocess	Max	Min	Average
PS^3	-	45.1	2.1	7.4
BinXray	110	0.1	-	0.06

Add bounded code. Adding code to ensure a certain bound is also a common pattern in vulnerability patches, see Figure 5b for example, which only requires one single line to fix the vulnerability. Similarly to our motivating example, ***PS***^{3} is capable of tracing symbolic transitions and extracting the semantic signature `Mem(24 + SR(72)) == 0`, where `SR(72)` corresponds to the first function parameter and 24 corresponds to the corresponding structure offset in C. The target binary is tested by matching the single signature precisely.

Limitation. ***PS***^{3} only considers forward instructions based on the control flow graph and terminates the emulation after traversing all relevant basic blocks in reference binaries. In cases where the root of the vulnerability lies backward, ***PS***^{3} is unable to identify the differences and test the binary correctly. For example, CVE-2022-1343 [12], a vulnerability in OpenSSL, is shown in Figure 5c. The vulnerability is a wrong assignment to `ret`, which is later used as a return value. The difficulty in testing this patch is that one can only decide the existence of the patch after analyzing the backward instructions. Since both the vulnerable and patched functions call the function `X509_STORE_CTX_get_error` with the same argument, i.e. `ctx`, ***PS***^{3} cannot distinguish between the two different functions. As a result, our approach extracts the same signature for both functions, resulting in labeling the binary as vulnerable even if it is patched.

```
@@ -206,8 +206,8 @@ int ec_scalar_mul_ladder(const EC_GROUP *group, EC_POINT
↪  *r,
      */
     cardinality_bits = BN_num_bits(cardinality);
     group_top = bn_get_top(cardinality);
-    if ((bn_wexpand(k, group_top + 1) == NULL)
-        || (bn_wexpand(lambda, group_top + 1) == NULL)) {
+    if ((bn_wexpand(k, group_top + 2) == NULL)
+        || (bn_wexpand(lambda, group_top + 2) == NULL)) {
         ECerr(EC_F_EC_SCALAR_MUL_LADDER, ERR_R_BN_LIB);
         goto err;
     }
@@ -244,7 +244,7 @@ int ec_scalar_mul_ladder(const EC_GROUP *group, EC_POINT
↪  *r,
      * k := scalar + 2*cardinality
      */
     kbit = BN_is_bit_set(lambda, cardinality_bits);
-    BN_consttime_swap(kbit, k, lambda, group_top + 1);
+    BN_consttime_swap(kbit, k, lambda, group_top + 2);

     group_top = bn_get_top(group->field);
     if ((bn_wexpand(s->X, group_top) == NULL)
```

(a) Patch of CVE-2018-0735 [10]

```
@@ -100,6 +100,8 @@ void OPENSSL_LH_flush(OPENSSL_LHASH *lh)
         }
         lh->b[i] = NULL;
     }
+
+    lh->num_items = 0;
 }
```

(b) Patch of CVE-2022-1473 [13]

```
@@ -59,9 +59,10 @@ static int ocsp_verify_signer(X509 *signer, int response,

     ret = X509_verify_cert(ctx);
     if (ret <= 0) {
-        ret = X509_STORE_CTX_get_error(ctx);
+        int err = X509_STORE_CTX_get_error(ctx);
+
         ERR_raise_data(ERR_LIB_OCSP, OCSP_R_CERTIFICATE_VERIFY_ERROR,
-                       "Verify error: %s", X509_verify_cert_error_string(ret));
+                       "Verify error: %s", X509_verify_cert_error_string(err));
         goto end;
     }
     if (chain != NULL)
```

(c) Patch of CVE-2022-1343 [12]

Figure 5: Case study examples

6.2 Threats to Validity

6.2.1 Internal Validity. We used the `CFGFast` method from angr [32] in our implementation to build the control flow graph of a given function, which is fast but imprecise and may affect the results. Building a control flow graph for binaries is not easy and is sometimes difficult to analyze, in general [27]. Additionally, due to the limitation of VEX IR, our approach may encounter additional problems in practice. Angr is not able to handle all vector instructions effectively, and OpenSSL heavily utilizes vector instructions to accelerate execution in the target binary. As a result, symbolic value transitions in emulation may not work correctly because symbolic

Table 6: Summary of related work

Work	Approach		Language	Source code required
	signature	similarity		
FIBER [37]	✓		C/C++	✓
Osprey [33]	✓		C/C++	✓
PDiff [23]		✓	C/C++	✓
BinXray [36]		✓	C/C++	
BSCOUT [9]		✓	Java	✓
PHunter [35]		✓	Java	✓

register values cannot maintain valid information, leading to imprecise predictions.

In our approach, we assume that the entry address of the function to be analyzed is already known, which is also assumed in the related work [36, 37]. However, in practice, there are many stripped binaries. It is difficult to identify the entry point of a specific function in those binaries. To handle such cases, we can utilize binary function matching techniques as a preprocessing step in our approach to find the corresponding function, as previous work suggests [23, 37].

6.2.2 External Validity. We only consider vulnerabilities in four C/C++ projects, which may cause certain types of unusual vulnerability to be out of consideration. As mentioned in [5], the information about the affected version of the vulnerable software affected in the National Vulnerability Database (NVD) [14] is not always accurate, which can significantly affect the precision of our approach, although the information on the official project website is much more precise. In addition, ***PS***^{3} currently does not support an architecture other than Amd64, although it can be supported with some engineering effort.

7 RELATED WORK

7.1 Patch Presence Test

In this section, we review the main work closely related to patch presence test. The summary is shown in Table 6.

In 2018, Zhang et al. [37] first introduced the concept of "patch presence test" and distinguished it from conventional vulnerability search. Inspired by the heuristic that human analysts only examine the behavior of small and local code areas, Zhang et al. propose FIBER, which parses open source security vulnerability information to generate fine-grained binary signatures that reflect the changes caused by patch modifications. These signatures are used to determine whether the target binary has been patched. During the experimental phase, Zhang et al. evaluated FIBER using 107 real security patches from three different main vendors and eight images of the Android kernel, and the results showed that FIBER achieved an average accuracy of 94% without false positives.

Considering the overhead caused by the use of symbolic execution technology in FIBER [4], Sun et al. [33] proposed Osprey, which exploits light weight copy propagation and data flow slices without symbolic execution so that Osprey speeds up the testing process by more than 10 times without a much decrease in accuracy, which can still surpass 90%.

Jiang et al. [23] proposed PDiff, a system that uses downstream kernel images to perform highly reliable patch presence testing. Unlike previous research on patch presence testing, PDiff is based on the semantic similarity of patch checking and thus has a high fault tolerance for code changes. Tests on 398 kernel images corresponding to 51 patches showed that PDiff can achieve high-precision testing with an extremely low miss rate.

We choose signature match instead of similarity-based, since our objective is to provide a definite answer. Compared to the above approaches for C/C++ projects, we use lightweight symbolic emulation instead of symbolic execution to avoid time-consuming constraint solving. Additionally, we are the first to utilize a theorem prover to check the equivalence of signatures at a semantic level, while other approaches do not.

The works mentioned above are based on the source code, while Xu et al. [36] proposed BinXray, which does not presume the existence of the source code and patch commit. BinXray extracts patch signatures by comparing the given vulnerable code snippet and patched code snippet using a basic block mapping algorithm, where signatures are represented as a set of basic block traces. During the testing phase, the patch presence is determined by matching the basic block traces.

In addition to the detection of executable files compiled in the mainstream C/C++ languages mentioned above, Dai et al. [9] proposed BSCOUT, which is the first tool that can check the existence of patches in Java files. BSCOUT directly checks fine-grained patch semantics in the entire target executable without generating signatures. The results show that it exhibits significant accuracy regardless of the presence or absence of line number information (i.e., debug information) in the target executable.

Xie et al. [35] proposed a tool named PHunter to address the challenge introduced by code obfuscation in Android applications. PHunter uses coarse-grained characteristics to identify functions related to patches and then evaluates semantic similarity to decide whether the code has been patched.

7.2 Patch Presence Test vs. Other Works

Although patch presence test, vulnerability detection [2, 20, 21], and binary function matching [6, 18, 19] share some similarities, they refer to different tasks in essential. We have summarized the differences between patch presence testing and vulnerability detection, binary function matching as follows:

(1) Compare with *vulnerability detection.* The essential difference between the patch presence test and conventional vulnerability detection is whether information about a particular patch is available or not. In vulnerability detection, one should decide whether the binary contains *any* or a certain type of vulnerability. Patch presence test focuses on determining whether a *specific* patch exists for a particular vulnerability.
(2) Compared with *binary function matching.* Patch presence test can be seen as a particular type of binary function matching method, that is, comparing the target function with the patched and vulnerable reference function and determining

which one the target function is closer to. However, the general function matching methods do not work well due to the small differences between patched and vulnerable functions.

8 CONCLUSION AND FUTURE WORK

Patch presence test is a critical task. Existing methods are heavily based on syntax information. We present ***PS***^{3} that utilizes symbolic emulation to generate semantic patch signatures for fast and precise matching. To evaluate the performance of ***PS***^{3}, we created a dataset consisting of 3,631 tests. The experimental results show that our approach outperforms the baselines by improving 33% in terms of F1 score. In addition, ***PS***^{3} performs well with different compiler options.

When a patch is released to address a vulnerability in many Java projects, a corresponding test is often provided. It reminds us to take advantage of dynamic execution-based testing methods to perform patch presence tests. Therefore, in the future, we plan to start with symbolic emulation and combine dynamic execution methods to construct concrete input to functions. Running the function on the constructed input, we are able to precisely test the target binary.

The code and dataset we constructed are publicly available[6].

ACKNOWLEDGMENTS

This research was supported by the National Natural Science Foundation of China (No. 62141222) and the National Research Foundation, Singapore under its Industry Alignment Fund – Prepositioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. 2005. OPUS: Online Patches and Updates for Security. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Baltimore, MD) *(SSYM'05)*. USENIX Association, USA, 19.

[2] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. 2014. Automated Testing for SQL Injection Vulnerabilities: An Input Mutation Approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2610384.2610403

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018).

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (may 2018), 39 pages. https://doi.org/10.1145/3182657

[5] Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2352–2364. https://doi.org/10.1145/3510003.3510113

[6] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-Architecture Cross-OS Binary Search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 678–689. https://doi.org/10.1145/2950290.2950350

[7] Hui Chen. 2013. The influences of compiler optimization on binary files similarity detection. In *2013 the International Conference on Education Technology and Information System (ICETIS 2013)*. Atlantis Press, 971–975.

[8] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 1093–1110.

[9] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct whole patch presence test for java executables. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 1147–1164.

[10] National Vulnerability Database. 2018. CVE-2018-0735. https://nvd.nist.gov/vuln/detail/cve-2020-22019

[11] National Vulnerability Database. 2020. CVE-2020-22019. https://nvd.nist.gov/vuln/detail/CVE-2018-0735

[12] National Vulnerability Database. 2022. CVE-2022-1343. https://nvd.nist.gov/vuln/detail/CVE-2022-1343

[13] National Vulnerability Database. 2022. CVE-2022-1473. https://nvd.nist.gov/vuln/detail/CVE-2022-1473

[14] National Vulnerability Database. 2023. CVE: Vulnerabilities By Year. Retrieved July 1, 2023 from https://www.cvedetails.com/browse-by-date.php

[15] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[16] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. 472–489. https://doi.org/10.1109/SP.2019.00003

[17] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2018. Heaphopper: Bringing bounded model checking to heap implementation security. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 99–116.

[18] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 303–317.

[19] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-Based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 480–491. https://doi.org/10.1145/2976749.2978370

[20] Jian Gao, Yu Jiang, Zhe Liu, Xin Yang, Cong Wang, Xun Jiao, Zijiang Yang, and Jiaguang Sun. 2021. Semantic Learning and Emulation Based Cross-Platform Binary Vulnerability Seeker. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2575–2589. https://doi.org/10.1109/TSE.2019.2956932

[21] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 896–899. https://doi.org/10.1145/3238147.3240480

[22] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. 2020. SYMBION: Interleaving Symbolic with Concrete Execution. In *2020 IEEE Conference on Communications and Network Security (CNS)*. 1–10. https://doi.org/10.1109/CNS48642.2020.9162164

[23] 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. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1149–1163.

[24] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2023. Revisiting Binary Code Similarity Analysis Using Interpretable Feature Engineering and Lessons Learned. *IEEE Transactions on Software Engineering* 49, 4 (apr 2023), 1661–1682. https://doi.org/10.1109/tse.2022.3187689

[25] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2201–2215. https://doi.org/10.1145/3133956.3134072

[26] Liu Liu, Olivier De Vel, Qing-Long Han, Jun Zhang, and Yang Xiang. 2018. Detecting and Preventing Cyber Insider Threats: A Survey. *IEEE Communications Surveys & Tutorials* 20, 2 (2018), 1397–1417. https://doi.org/10.1109/COMST.2018.2800740

[27] Xiaozhu Meng and Barton P. Miller. 2016. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/2931037.2931047

[28] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (jun 2007), 89–100. https://doi.org/10.1145/1273442.1250746

[29] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[6] https://github.com/Qi-Zhan/ps3

[30] Davide Pizzolotto and Katsuro Inoue. 2021. Identifying Compiler and Optimization Level in Binary Code From Multiple Architectures. *IEEE Access* 9 (2021), 163461–163475. https://doi.org/10.1109/ACCESS.2021.3132950
[31] Hex Rays. 2023. IDA Pro. https://www.hex-rays.com/products/ida/
[32] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 138–157.
[33] Peiyuan Sun, Qiben Yan, Haoyi Zhou, and Jianxin Li. 2021. Osprey: A fast and accurate patch presence test framework for binaries. *Computer Communications* 173 (2021), 95–106.
[34] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: Jump-Aware Transformer for Binary Code Similarity. arXiv:2205.12713 [cs.CR]
[35] Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. 2023. Precise and Efficient Patch Presence Test for Android Applications against Code Obfuscation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 347–359.
[36] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–387.
[37] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries.. In *USENIX Security Symposium*. 887–902.
[38] Zynamics. 2021. bindiff. https://www.zynamics.com/bindiff.html