SymBisect: Accurate Bisection for Fuzzer-Exposed Vulnerabilities

Zheng Zhang *UC Riverside*
Yu Hao *UC Riverside*
Weiteng Chen *Microsoft Research*
Xiaochen Zou *UC Riverside*
Xingyu Li *UC Riverside*
Haonan Li *UC Riverside*
Yizhuo Zhai *UC Riverside*
Zhiyun Qian *UC Riverside*
Billy Lau *Google*

Abstract

The popularity of fuzzing has led to its tight integration into the software development process as a routine part of the build and test, i.e., continuous fuzzing. This has resulted in a substantial increase in the reporting of bugs in open-source software, including the Linux kernel. To keep up with the volume of bugs, it is crucial to automatically analyze the bugs to assist developers and maintainers. Bug bisection, i.e., locating the commit that introduced a vulnerability, is one such analysis that can reveal the range of affected software versions and help bug prioritization and patching. However, existing automated solutions fall short in a number of ways: most of them either (1) directly run the same PoC on older software versions without adapting to changes in bug-triggering conditions and are prone to broken dynamic environments or (2) require patches that may not be available when the bug is discovered. In this work, we take a different approach to looking for evidence of fuzzer-exposed vulnerabilities by looking for the underlying bug logic. In this way, we can perform bug bisection much more precisely and accurately. Specifically, we apply under-constrained symbolic execution with several principled guiding techniques to search for the presence of the bug logic efficiently. We show that our approach achieves significantly better accuracy than the state-of-the-art solution by 16% (from 74.7% to 90.7%).

1 Introduction

In recent years, large-scale programs such as the Linux kernel are being continuously fuzzed for the purpose of improving code quality and security [23, 11, 42, 19, 31, 24]. Such continuous fuzzing systems have been shown highly effective in identifying new bugs, e.g., syzbot [22] reports thousands of bugs in the Linux kernel.

While fuzzing is highly effective, this poses large workload to software developers and maintainers, as the continuous stream of bugs requires various analysis, e.g., bug triage and patching, that is often done largely manually today [27]. This is a hard problem as we already see over 8,000 bugs found by syzbot are auto-closed due to the lack of human investigations [22]. Thus, automating the analysis of fuzzer-exposed bugs is a worthwhile goal.

One important analysis that needs automation is bug bisection, i.e., the process of identifying commit that introduced a bug (also called vulnerability-contributing commits, or bug-inducing commits). It proves instrumental in various aspects. For example, it can help developers and maintainers understand the bug and facilitate patch development [7]; it can also pinpoint the vulnerable software versions to inform users about whether they need to worry about updating their software [12, 4].

To achieve this goal, researchers have proposed several automated approaches, but unfortunately they all have significant shortcomings.

The first type of approach directly executes the original PoC on older software versions to see which version would still crash after running the PoC. However, it is reported that such a dynamic solution suffers from several issues [3]: 1) Broken dynamic environment (e.g., build or runtime errors) leading to versions being skipped. 2) Accidental triggering of unrelated bugs. 3) Changes in the underlying bug-triggering condition.

The second type of approach requires patches, which may not be available at the time of the bug discovery. Even if the patches are available, such static solutions rely on heuristics which are inherently imprecise [9, 34]. This category includes the SZZ algorithm and its variants [12, 40], as well as most vulnerable code clone detection [26, 45]. For example, when given the code diff in a patch, their solutions consider the bug-inducing commit to be the one that introduces one or more lines in the code diff [12, 45]. However, such a solution does not take into account the bug-triggering conditions and can miss important details that are outside of the scope of the code diff in the patch.

There are also other methods, such as information retrieval-based bisection [10, 47, 43], which are usually based on bug reports. However, they either have low accuracy or require manual analysis.

Motivated by the above deficiencies, we take a different approach from the traditional methods. In particular, we aim to reason about the *presence of vulnerability logic* through static code analysis. Fundamentally, our approach investigates many more possible inputs beyond what's included in the original PoC. Furthermore, static methods can effectively circumvent a series of problems caused by the broken dynamic environments such as build errors. Finally, it does not require the development of patches in advance. To this end, we leverage **symbolic reasoning** which is the most precise way of confirming the presence of a vulnerability statically. A crucial characteristic of this approach is that it can automatically distinguish significant changes from vulnerability-irrelevant changes and effectively eliminate the influence of vulnerability-irrelevant changes on the results.

More specifically, we apply under-constrained symbolic execution [32] in different software versions to precisely identify the presence/absence of the same *vulnerablility logic* that is inherited from the released PoC. Then with a simple binary search algorithm, we can pinpoint the commit that introduced the vulnerability. To address the scalability challenges of symbolic execution, we leverage the trace associated with the PoC to guide the symbolic execution.

Following the methodology proposed in this paper, we apply it to the context of Linux kernel and the corresponding continuous fuzzing platform syzbot [22]. We show that it significantly outperforms state-of-the-art approaches in terms of accurately determining the vulnerable versions of bugs found with fuzzing. We summarize our contributions as follows:

- We developed a novel and drastically different solution of an automatic bisection tool called SYMBISECT, targeting fuzzer-exposed vulnerabilities. Our method is precise as it relies on looking for the presence of key vulnerability logic represented by symbolic formulas. We have implemented SYMBISECT for Linux kernel bugs reported on syzbot. We open-sourced the solution to facilitate the reproduction of results and further research [2].
- We proposed a new method to address the scalability problem in under-constraint symbolic execution in the Linux kernel. Our insight is that in the specific context of fuzzing results, we are able to use the knowledge of the vulnerability from the PoC to guide the symbolic execution in a principled fashion.
- We evaluated the performance of SYMBISECT against other state-of-the-art methods. We demonstrate that it not only achieves much higher accuracy than the PoC-based bisection but even outperforms the methods dependent on the presence of patches. Specifically, it can identify 83% of the vulnerable versions that elude detection in the PoC bisection implemented by Syzbot.

2 Background and Motivation

In recent years, fuzzing has played a significant role in discovering vulnerabilities in the Linux kernel [22, 23]. However, manual analysis of the extensive results generated by fuzzing has placed a tremendous burden on maintainers [1]. Automatic analysis of fuzzing results, such as identifying vulnerable versions and simultaneously identifying when vulnerabilities were introduced, is highly beneficial for understanding the logic of vulnerabilities, developing patches, notifying the respective maintainers, and backporting patches to vulnerable Long-Term-Support branches. For example, Rui Abreu *et al.* observed that automating bug bisection that pinpoints the bug-inducing commits can speed up fixing fuzzer-exposed bugs in Google's proprietary code on average by a factor of 2.23 [7]. In this paper, we define **bug-inducing commit** as a commit that introduces a software bug into a codebase [10]. It is possible that multiple commits (e.g., commit1 and commit2) contribute to the bug, and the last commit (commit2) makes the bug triggerable. In such cases, we consider the last commit the BIC because that is when a vulnerability is considered to exist.

Previous researchers have developed various types of methods for identifying bug-inducing commits, including PoC-based, Patch-based, and other approaches. However, each has its own limitations. Next, we will introduce them separately.

PoC-based bisection. The most straightforward approach is to dynamically re-execute the PoC that triggered the bug on older commits. This method is employed by the continuous fuzzing platform syzbot [22]. Specifically, syzbot starts bisection by running the same PoC with the commit on which the bug was discovered, ensures that it can reproduce the bug, and then goes back release-by-release (e.g., v5.4 to v5.3) to pinpoint the earliest release without the kernel crash (again using the same PoC). The predicate for bisection is binary (crash vs. no crash), not trying to differentiate between different crashes. This is intentional because bugs can manifest in a different ways (under different bug titles) [3]. However, this inevitably introduces false positives as unrelated bugs can sometimes be triggered. In fact, a small-scale analysis showed that unrelated bugs being triggered contributed to 66% of incorrect bisection [4]. In addition, such an approach also leads to false negatives, i.e., failing to report a kernel version being affected by the bug during bisection. They can be due to build/boot errors, bugs that are difficult to reproduce, and failing to account for changes in bug-triggering conditions (no adaptation in the original PoC). Overall, a previous small-scale study conducted by the syzbot team concludes that the bisection accuracy is only about 50% [4], highlighting the need for a better solution.

Patch-based bisection. This family of solutions is based on patches, including *SZZ algorithm and its variants*[40, 12] and most vulnerable code clone detection solutions [26, 45, 56, 37]. The basic idea of SZZ is to identify the bug-inducing commit by tracing the modified lines in the patch back to the most recent commit that introduced the lines. This method is static and effectively assumes the source lines removed or changed by the patch are responsible for introducing the bug. The SZZ algorithm has many variations, among which VSZZ [12] is the latest improvement aimed specifically at vulnerabilities (instead of general bugs). VSZZ modifies SZZ slightly by tracing back the commit history to the earliest commit (as opposed to the most recent) that introduces the deleted lines of a patch. However, such methods require patches input, which are not available at the time of bug discovery. Furthermore, the SZZ algorithm and its variants are fundamentally heuristics and their accuracies are limited [9]. Finally, they are unable to handle patches with only added lines [12], which are quite common in security patches (e.g., adding a bounds check).

Vulnerable code clone detection statically identifies sections of code that are similar or identical to known vulnerable code fragments [25, 36, 26, 28, 45, 49]. These methods rely on similarity comparisons of vulnerability-related code to determine the presence of a given vulnerability in the target program. The basis of similarity could be in terms of text, tokens, Abstract Syntax Trees (AST), Control Flow Graphs (CFG), or Program Data Graphs (PDG) [55]. Even though not originally intended for bisection, they can be directly applied to identify the earliest version similar to the known vulnerable version. V0Finder [45] represents the state-of-the-art in this category. In general, these methods are not designed to distinguish small changes made to the code base (e.g., via a bug-inducing commit). Besides, such methods still require an initial input of vulnerable code. The "vulnerable code" is generally defined as the whole patch function or a subset of lines within it. For example, VUDDY [26] uses the entire patch function, while others, like MVP [49], extract "relevant lines" through methods such as slicing, using deleted/added lines in the patch as slicing criteria. The "vulnerable code" can also be manually extracted, such as in HiddenCPG [44].

Other bisection. There are some other methods such as Information Retrieval (IR) approaches, that take bug information (usually from the bug report) as input and statically rank prior commits according to their "relevance to the bug" [10, 43, 13]. The advantage of IR-based methods is that they do not require patches and can still locate the buggy code based on bug reports or code coverage. Nevertheless, such methods still do not fundamentally attempt to verify the presence of vulnerability logic and instead only approximate them. As a result, the accuracy is also limited. They are usually good at identifying the top-N suspected bug-inducing commits. However, their accuracy declines significantly when N=1. Specifically, for Fonte [10], the state-of-the-art method in this category, the accuracy drops to only 36% when N=1, which is lower than the accuracy of PoC/Patch-based methods. Therefore, we do not consider such methods as the baseline for comparison.

Our insight. We observe that existing approaches have significant shortcomings. First and foremost, none of the above methods attempts to reason about the vulnerability logic when determining whether a particular version is affected by a bug. This motivates us to develop a solution that looks for the presence of the vulnerability logic in target software versions.

Specifically, we propose to leverage under-constrained symbolic execution to effectively address the shortcomings of existing solutions. Compared to PoC-based bisection, our solution (1) is static, thus sidestepping the challenges stemming from broken dynamic environments; (2) focuses on the specific vulnerability, allowing it to overlook other unrelated bugs; (3) considers more possible inputs and execution paths, alleviating the concern of changes in the underlying bug-triggering conditions.

Compared to the existing patch-based methods, our solution (1) does not require patches, which are not available when a fuzzer first finds the bugs; (2) looks for the presence of vulnerability logic as opposed to syntatic information such as the presence of certain source lines or tokens; (3) inspects the vulnerability logic beyond the scope of patch functions, allowing a much more complete and informed validation compared to heuristics that concentrate on only the code diff or the functions involved in patches.

3 Overview

In this section, we begin with a motivating example to provide a concise overview of why existing methods fall short and the intuition behind SymBisect. We will also discuss the main challenges of implementing our solution. Following that, we introduce the overall architecture of SymBisect.

```
                 The Bug-inducing Commit:
static struct bpf_map *htab_map_alloc(...)
1  - cost = S1*C1 + S2*S3;
     ......
2  - cost += S2*C2
3  - err = bpf_map_charge_init(..., cost);
4  - if (err)
   -     goto free_htab;
     ......
5    err = prealloc_init(...);

int bpf_map_charge_init(...,u64 size)
     ......
6    if (size >= U32_MAX - PAGE_SIZE)
        return -E2BIG;
                     The Patch:
static int prealloc_init(...)
     S3 = S3 + C2;
     ......
7  - htab->elems =bpf_map_area_alloc(S2*S3,
8  + htab->elems =bpf_map_area_alloc((u64)S2*S3,

S1: (u64)htab->n_buckets      C1: sizeof(struct bucket)
S2: (u64)htab->elem_size      C2: num_possible_cpus()
S3: htab->map.max_entries
```

Figure 1: The Bug-inducing commit and Patch of a vulnerability from syzbot

3.1 Motivating Example

Figure 1 illustrates an integer overflow vulnerability that leads to an out-of-bounds memory access. Specifically, the **bug-inducing commit** modifies the function `htab_map_alloc()`, which in turn calls function `bpf_map_charge_init()` and function `prealloc_init()`. Prior to this commit, the function `bpf_map_charge_init()` had a check at line 6, which checked the variable `size` to prevent any potential integer overflow in `prealloc_init()`. However, the removal of this safeguard paved the way for the occurrence of an integer overflow. To mitigate this vulnerability, the subsequent **patch** introduced a type-casting operation at line 8 within `prealloc_init()`, effectively preventing the risk of integer overflow.

Prior PoC-based tool executed the released PoC in versions preceding the patch. However, in this case, it triggered an unrelated bug, leading the kernel to crash before it could access the function `htab_map_alloc()`. Consequently, this resulted in an imprecise bisection result — syzbot thinks the kernel version is vulnerable and keep checking even earlier versions.

Prior Patch-based tools derive various forms of signatures, primarily syntactic, from the patch function `prealloc_init()`. In this case, the bug-inducing commit does not alter the patch function. Consequently, these solutions are unable to capture the commit and fail to differentiate versions preceding and following the bug-inducing commit. This leads to incorrect identification

```
            Symbolic execution trace (partly):
...... -> htab_map_alloc() -> bpf_map_charge_init()
                          -> prealloc_init() -> ......
                 Before inducing commit:
Line1  Assignment: cost = S1*C1 + S2*S3
Line2  Assignment: cost +=  S2*C2
Line6  Constraint  S1*C1 + S2(S3+C2) < U32_Max - 4096
Line7  Overflow condition: S2(S3+C2) > U32_Max
Not solvable => Not vulnerable
          After inducing commit (before patch):
Line8  Overflow condition: S2(S3+C2) > U32_Max
Solvable => Vulnerable
```

Figure 2: Vulnerability detection via symbolic execution

of the bug-inducing commit.

Our solution symbolically executes the relevant functions until it reaches the target source line and evaluates the symbolic constraints to check whether an out-of-bound memory access can occur — we know it is an out-of-bound bug from the bug report. Specifically, the symbolic execution starts from the syscall entry that triggered the bug (available from the call stack in the bug report). By enlarging the analysis scope, our solution effectively explores more of the state space and is not confined to the patch function. It effectively addresses both the limitation of patch-based bisection and potential changes in the bug-triggering conditions. Additionally, by disregarding unrelated bugs, it resolves the issues associated with PoC-based bisection.

Figure 2 illustrates a portion of the symbolic execution process. In the non-vulnerable version (prior to the bug-inducing commit), the variable `cost` is assigned in lines 1 and 2, with a subsequent check at line 6. While there are two branches in line 6, only one of them leads to the vulnerability point. Within this path, symbolic execution identifies a crucial constraint: S2(S3+C2) + S1*C1 <U32_Max - 4096. This constraint ensures that the overflow condition S2(S3+C2) >U32_Max is never satisfied, preventing any subsequent out-of-bounds occurrences (the OOB section isn't depicted in the figure). Consequently, this version is deemed non-vulnerable, which is correct. In contrast, in the vulnerable versions, the critical check against the cost is removed. As a result, the overflow condition becomes solvable by the symbolic execution engine, leading to an out-of-bounds (OOB) situation later on. Accordingly, our solution correctly classified this version as vulnerable.

3.2 Challenges and Insights

Despite the advantages of using symbolic execution to confirm the presence of vulnerability logic precisely, symbolic execution also faces its own challenges.

Scalability concerns. In the motivating example, we showed only a segment of the symbolic execution in Figure 2. In reality, our solution will encounter many more functions (i.e., starting from the syscall entry) and accumulate many more symbolic constraints. This can lead to the classic scalability challenge for symbolic execution, as the number of forked states may grow exponentially as the execution progresses. This makes the solution seemingly ill-suited for a large scope of analysis, especially against large-scale software such as the Linux kernel. Previous methods deal with this problem by confining the scope of symbolic execution to one specific function [54] or utilizing concolic execution[15]. Nevertheless, these methods are unsuitable for our purpose: the existence of a vulnerability is not determined by a single function, and we do not want to over-constrain the possible inputs through concolic or concrete execution.

Key observation. We observe that, to overcome the above challenge, it is possible to leverage fine-grained trace-level information about how the vulnerability is manifested (e.g., where the vulnerability is triggered, and which functions are involved) in the reported version to guide the exploration in the target version. This information allows us to distinguish the key statements from the unrelated ones for a specific vulnerability. As an illustration, coverage data can help de-prioritize less relevant execution paths. By utilizing this approach, SYMBISECT effectively narrows the scope of exploration, thus enhancing efficiency and mitigating the scalability challenge of symbolic execution.

3.3 System Architecture

As illustrated in Figure 3, our tool, denoted as SYMBISECT, requires three essential inputs for its operation:

- The source code of the program on which the bug was reported – we refer to it as the reference version. This version should be compilable and bootable as the fuzzer has successfully found the bug on this version.
- Proof of Concept (PoC): This is the executable or script that can reliably trigger the vulnerability in the reference version of the program.
- The source code of the program in potentially vulnerable target versions: These are the other versions of the program that we want to assess for the same vulnerability.

SYMBISECT is designed for vulnerabilities found through fuzzing. So both the compilable and bootable source code of the reference program and the PoC are naturally available when a vulnerability is found via fuzzing. With such inputs, SYMBISECT bisects the bug in a fashion similar to syzbot (except that SYMBISECT is completely static). It first evaluates historical versions backwards – one major release version at a time (e.g., v5.5 and then v5.4). Through this iterative pro-

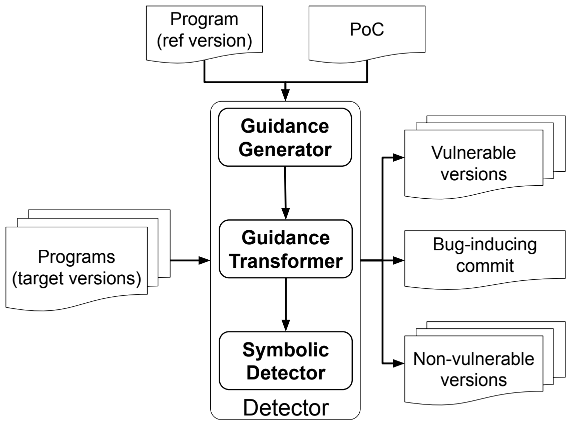

Figure 3: Overview of SYMBISECT

cess, SYMBISECT can identify the boundary or range for which the bug-inducing commit falls under (e.g., between v5.4 and v5.5). Subsequently, SYMBISECT follow a simple binary search procedure to pinpoint the specific commit that introduced the bug.

SYMBISECT consists of three primary components, designed to accurately identify vulnerabilities while also addressing scalability issues:

Guidance Generator. SYMBISECT first runs a PoC to trigger the specific vulnerability in the reference version of the program, thereby collecting essential execution traces. Utilizing these traces, SYMBISECT systematically produces three primary categories of guidance for subsequent symbolic detection. Firstly, SYMBISECT attempts to align the call stack trace (also called call trace in the syzbot bug report) of the execution on the target version to the one on the reference version (referred to as **Call Stack Guidance**) This effectively steers the exploration of symbolic execution towards the function where the vulnerability is observed. Secondly, besides the call stack trace, SYMBISECT also secondarily attempts to align the execution path down to the basic block level (referred to as **Path Guidance**). This is useful when there are a large number of possible execution paths that follow the same call stack. Lastly, SYMBISECT reuses the callees involved in indirect calls (referred as **Indirect Call Guidance**), thereby informing the symbolic detector to focus on a limited number of indirect call targets (as opposed to all possible ones computed using static analysis). More details are in §4.1.

Guidance Transformer. Upon identifying the above three kinds of guidance, SYMBISECT transforms them from the reference version into the target versions of the program. This enables a more efficient symbolic execution and a more accurate vulnerability detection process on these target versions. It's important to highlight that guidance translation between versions is done at the source code level, which remains stable and unaffected by compiler optimizations. To enhance the precision and robustness of those guidance when applied to differing target versions of the program, SYMBISECT employ multiple optimizations during the guidance transformer phase. More details about the guidance transformer will be provided in §4.2.

Symbolic Detector. The symbolic detector is a form of detector that can capture (or re-capture) the bugs that were reported by a fuzzer. The detector is applied to a target version, where it tracks all variables, especially symbolic variables, including the symbolic sizes of allocated objects. However, instead of attempting to find all possible bugs during the exploration (which is clearly not scalable), we narrowly focus on the specific bug at hand, with the help of the aforementioned guidance. Throughout the execution, the symbolic detector leverages guidance from preceding phases. Specifically, it dynamically adjusts the execution state schedule, aiming to alleviate path explosion. Additionally, the detector refines the callees of indirect function calls based on prior indirect call guidance. For vulnerability detection, the symbolic detector relies on call trace guidance to ensure accurate detection of the same vulnerability previously identified. More details about the under-constrained symbolic detector can be found in §4.3.

4 SYMBISECT Design

In this section, we delve into the intricacies of SYMBISECT's design by dissecting each component, discussing the challenges encountered, and illustrating our corresponding solutions.

4.1 Guidance Generator

Overall, this components attempts to guide SYMBISECT when SYMBISECT executes the PoC in the reference version of the program to trigger a specific vulnerability and collects the execution trace. Then, SYMBISECT produces three categories of guidance from the execution traces, which we explain below.

Call Stack Guidance. The call stack guidance represents the state of the call stack at the moment a vulnerability is triggered. This information can be readily collected when the corresponding bug is triggered in the reference version of the software under investigation. Utilizing the call stack guidance serves multiple purposes. First, it assists in identifying an appropriate **entry function** as the starting point for our symbolic detector. Second, it assists in pinpointing **the target line** where the vulnerability is triggered, allowing the symbolic detector to focus on the same vulnerability rather than any arbitrary vulnerability. We use call stack guidance to constrain the exploration of a target version so that it only explores the basic blocks that can potentially lead to the same stack trace. Correspondingly, we translate call stack guidance into basic-block-level priorities to guide the exploration.

- Highest Priority: basic blocks that dominate the basic blocks in the call stack will receive the highest priority. This indicates that their execution is essential for reaching the bug while maintaining the same call stack. The set of such basic blocks can be identified through the dominator analysis on the control flow graph of functions in the call stack.
- Lowest Priority: basic blocks, upon the execution of which can cause deviations from the call stack, will receive the lowest priority. Consequently, a symbolic detector should avoid executing any of these blocks. They can be identified through reachability analysis.

Path Guidance. In addition to the call stack guidance, we will need more fine-grained guidance if there are still too many possible execution paths that follow the same call stack. Specifically we propose to prioritize the execution path directly at the basic block level. The idea is that a basic block in the target version is likely to be non-critical if (1) the basic block is not executed in the reference version and (2) it remains unchanged in both the reference and target versions of the program. Therefore, the symbolic execution should prioritize the exploration of branches whose basic blocks have higher priority. Note that when there are conflicts, path guidance must yield to call stack guidance because the most critical goal is to ensure the vulnerable function being reached. We translate path guidance into the basic-block-level priorities as follows:

- High Priority: basic blocks covered by the execution trace in the reference version of the program will receive high priority (lower than the highest priority).
- Low Priority: basic block not covered by the execution trace in the reference will receive low priority.

Indirect Call Guidance. The indirect call guidance records the callee functions associated with each indirect function call encountered in the execution trace. Its primary role is to facilitate the accurate resolution of indirect function calls during the symbolic execution process, particularly in the target versions of the software under analysis.

4.2 Guidance Transformer

To enhance both the efficiency of symbolic execution and the precision of vulnerability detection in target program versions, it is essential to translate the three categories of guidance collected from a reference version. Specifically, one fundamental task is to map basic blocks from the binary form in the reference version, where execution traces are collected, to the LLVM IR in the target version, where symbolic execution is executed. One potential solution is first to map the binary-level basic blocks from reference to target. However, due to compiler optimizations, even if the source code lines are identical, their binary basic blocks may differ, making this solution undesirable.

Our solution employs source code as an intermediate representation to improve the mapping accuracy between the reference and target versions. The transformation sequence for basic blocks begins with the binary form in the reference version, moves to its source code, transitions to the source code in the target version, and ends in the LLVM IR of the target version. To facilitate these mappings, we use the debug information to transition between binary and source code and between source code and LLVM IR. Additionally, Git is employed for source code mapping between the reference and target versions. During the transformation sequence, we take care of multiple corner cases to enhance the precision (more details in §5.1).

After transforming the call stack to the target version, we verify the presence of the target line triggering the vulnerability. If absent, the target version is deemed non-vulnerable. If present, we examine whether there is a potential path from the entry to the target function in the call graph. A missing path directly results in a negative outcome.

Otherwise, the exploration of the target version will follow the aforementioned guidance. Finally, if there are basic blocks unique to the target version (never seen in the reference), we will assign a neutral priority level to them as it is unknown whether these basic blocks will be useful in triggering the vulnerability:

- Medium Priority: basic blocks unique to the target version, which do not map to any basic blocks in the reference version, will receive medium priority. Compared to the low priority basic blocks – the ones seen in reference version yet not exercised, we are less certain about the utility of such basic blocks; therefore we prefer to explore them with a higher priority compared to the basic blocks that were seen in both reference and target versions but not exercised in the reference.

Then, all the guidance (call stack, five lists of varying priorities, and indirect call mapping) are forwarded to the subsequent component.

4.3 Symbolic Detector

After generating guidance for the target version of the program under analysis, the symbolic detector conducts under-constrained symbolic execution on these targeted versions. Specifically, the detector monitors all variables within the program to identify potential vulnerability patterns, such as use-after-free or out-of-bound access errors. We propose multiple improvements to enhance the ability of under-constrained symbolic execution, including but not limited to handling symbolic addresses, and symbolic sizes of allocated memory. The details are described in §5.2. Throughout this execution, the symbolic detector utilizes the guidance generated in prior stages to enhance its effectiveness.

Call Stack Guidance. Symbolic execution is initiated at a selected entry function, determined by examining the call stack. Specifically, execution starts at the first meaningful function in the call stack — we choose to start at the syscall handler [24] (which is typically several layers behind the generic syscall entry). The symbolic execution process ends upon detecting a vulnerability (resulting in a positive output) or upon hitting a time constraint (yielding a negative output). Importantly, the detector only checks for vulnerabilities upon reaching the specified target line in the guidance, avoiding hitting any unrelated bugs accidentally. Also, the basic blocks with the lowest priority are prohibited from execution.

Path guidance. When symbolic execution encounters a symbolic condition, it forks to explore both true and false branches. This forking behavior primarily contributes to the path explosion in symbolic execution. The path guidance is employed to address this. This approach prioritizes exploration by first traversing branches with higher priority. When two branches have the same priority, one is randomly selected to be explored first.

Indirect Call Guidance. During symbolic execution, if we observe the indirect call target being assigned explicitly to a function pointer, we can unambiguously determine the indirect call target. Otherwise, we initially refer to the indirect call guidance to identify the indirect call target. If we find a match for the specific indirect call, we use the specific target from the guidance directly. Otherwise, we utilize the state-of-the-art type-based analysis [29] to resolve indirect calls (which may produce multiple targets).

5 Implementation

In total, the implementation of SymBisect has 4,726 LoC Python code for the Guidance Generator and Guidance Transformer and 4,347 LoC of C++ for the Symbolic Detector atop KLEE [16]. In the following sections, we will delve into further details regarding the Guidance Transformer (§4.2), and Symbolic Detector (§4.3). In summary, SymBisect collects the original trace at the binary level, the generated guidances are at the source code level, and the symbolic execution engine is based on LLVM IR.

5.1 Guidance Transformer

Code formatting. Because we employ source code as an intermediate representation during the guidance transformer, we require each source code line to be associated with only a single basic block. To achieve this, we develop a simple source code formatter that divides composite lines into simpler ones. For instance, splitting "} else if(cond){" into two distinct lines. This is done for both the reference version and the target version at the beginning.

Accurate coverage collector. SymBisect leverage KCOV mechanism to discern which sections of the code have been covered. Syzkaller offers a tool to save the coverage data from KCOV. However, this operation is not always reliable. When the kernel crashes, some coverage can be lost. To improve this, SymBisect modifies the kernel to record the KCOV buffer in a log upon a kernel crash.

Refine guidance. The source code level guidance is generated using DWARF debug information, which may not always be accurate. Specifically, compiler optimizations (compiling the Linux kernel with -O0 is generally not supported), e.g., function inlining, and reordering, can lead to inaccuracies when mapping basic blocks in binary instructions into their corresponding source code lines with debug information – we find that the coverage of many basic blocks can be lost. To mitigate such impact, we implement an analysis of the basic blacks with the control flow graph and the dominator tree. We recover potentially lost basic block coverage under the following two conditions: 1) Should a line within a BB be marked as covered by a test case, it is necessary to mark all lines within that same BB as covered as well. 2) In instances where a covered BB is dominated by another BB (indicating that it is invariably executed after the dominating BB), it's essential that the dominator BB is also marked as covered.

5.2 Symbolic Detector

The types of vulnerabilities supported by SymBisect are determined by symbolic engine and detectors. SymBisect currently support bugs that exhibit out-of-bound (OOB) memory access and use-after-free (UAF) errors. This is because OOB/UAF vulnerabilities are the ones that are generally considered more security-critical and commonly exploited [57, 15, 48]. Note that the root causes of these vulnerabilities can vary, e.g., the underlying causes might be integer overflow and type confusion but they exhibit OOB memory access as an error. Given that syzbot categorizes vulnerabilities based on their security impact, we follow the same categorization. Currently, SymBisect has a few limitations: (1) it does not support race condition bugs (that are not supported by KLEE), and (2) it does not support bugs that require reasoning across multiple syscalls (instead the detector focuses on the last syscall that triggered the bug). Below, we discuss some modifications made to KLEE to achieve more accurate results.

Under constraint symbolic variables. We choose to symbolize all variables without concrete values in static environments, including global variables and arguments of system calls. This approach allows us to explore a broader range of potential execution paths during our analysis.

Symbolic address. In its original form, KLEE does not adequately support under-constrained symbolic addresses. When it encounters read/write operations to a symbolic address, KLEE typically generates a specific concrete address based on the current constraints.

The logic KLEE employs for dealing with under-constrained symbolic addresses is not reliable, particularly when faced with a multitude of such addresses. There might be instances where a symbolic address does not map onto any existing object. In such cases, arbitrarily concretizing this address to an existing object and proceeding with read/write operations can lead to incorrect outcomes.

Instead, we apply an improved mechanism in UCKLEE [18] to deal with symbolic addresses that have not been encountered before. When attempting to write/read to such a novel symbolic address, our system allocates a new object. Besides that, we maintain mappings between symbolic and concrete addresses. Therefore, subsequent attempts to access the same symbolic address will, in reality, be directed toward the corresponding concrete object as per the mapping. This procedure ensures that each symbolic address is consistently linked to a unique concrete object, thereby improving the precision of read-/write operations and overall analysis.

Symbolic size. The original way KLEE allocates a new object with symbolic size is also not suitable for our situation. Specifically, if the size is symbolic, it generates a specific concrete size, and then KLEE tries to half its size until the size is no larger than a small constant (i.e., 128 in KLEE v2.2).

In our under-constraint cases, it will result in many objects with small sizes, such automatic concretization may result in the inaccuracy of the results. For example, if there is a path that can only be explored with a size larger than the constant, it will always be skipped.

Instead, we implement a solution similar to the previous work[41] to handle this issue. We choose to track the symbolic sizes. We allocate the object with a large constant size in memory to make sure that the intended access to the object won't be missed and log the symbolic size. When there is a check against the size of an object, we always use the symbolic size.

Function modeling. To improve the scalability of symbolic execution, we manually model more general library functions belonging, such as strcpy(), malloc().

Vulnerability checker. The under-constraint nature of our symbolic execution will introduce some false positives when asserting the presence of vulnerability logic. To mitigate the problem, we concentrate on detecting the vulnerability on the corresponding line (called target line) in the target version where the vulnerability is triggered – we require the same line to be present in the reference and target version.

Once reaching the target line, for each read/write operation, we extract the address (usually symbolic) and find the corresponding object. If no corresponding object is found (usually happens in UAF cases), instead of allocating new under-constrained memory, we report the vulnerability directly. Otherwise, SYMBISECT compares the offset with the size of the object under current constraints. If the offset can be larger than the size (usually happens in OOB cases), SYMBISECT reports the vulnerability and terminates the execution. Finally, if none of these is detected, SYMBISECT keeps exploring various execution paths until a time limit is reached or runs out of paths to explore, leading to a negative result.

6 Evaluation

In this section, we evaluate SYMBISECT based on the following three research questions.

- RQ1: How precisely does SYMBISECT identify the vulnerable versions for a specific vulnerability? How precisely does it determine the exact bug-inducing commit? What factors influence the accuracy?
- RQ3: How effective is SYMBISECT, when compared with state-of-the-art (PoC-based/patch-based) bug-inducing commit identification methods?
- RQ4: How efficient is SYMBISECT in conducting its analysis? Specifically, how does the provided guidance/exploration strategy improve efficiency?

Evaluation Target and Vulnerability Dataset. We assess SYMBISECT on Linux kernel bugs reported on syzbot [22]. This choice is made due to several factors. First, syzbot is among the earliest and most mature continuous fuzzing platforms and the Linux kernel is among the most popular open source software. Second, the Linux kernel is the largest software that is being continuously fuzzed today. Third, there are a variety and a large number of bugs reported on syzbot continuously, which require bisection. Specifically, in our evaluation, we utilize SYMBISECT to conduct bisection on the Linux mainline branch. We believe our solution generalizes beyond the Linux kernel as it is likely more complex than most other software.

We consider adding support for other types an important but orthogonal exercise (see discussion in §7). As mentioned, SYMBISECT currently supports bugs that exhibit OOB and UAF impact (and no race conditions involved). Therefore, we randomly sampled 50 bugs from syzbot reports that meet the following requirements: 1) reported in the last 4 years. 2) labeled to have OOB or UAF impact. 3) not race conditions (which our symbolic detector currently does not support). 4) with PoCs and the bugs can be reproduced. 5) the corresponding patch has a "Fixes:" tag (to be explained below).

A "Fixes:" tag is included in a patch that points to one or more previous commits that are considered to introduce the corresponding bug. We treat it as the ground truth because we verified that they are consistent with our definition of bug-inducing commits (see later for "ground truth verification"). Note that SYMBISECT does not require the presence of a "Fixes:" tag to operate; we choose such bugs to merely simplify the evaluation process.

For each vulnerability, our tool begins with the released vulnerable version and inspects every major release version (e.g., v5.10) until the oldest version, v4.20, in our dataset. Versions prior to v4.20 present compatibility issues with the Clang/LLVM toolchain. While more engineering work might address this, it diverts from our primary focus. If the released vulnerable version is not on the Linux mainline branch, we find the corresponding commit (with the same Linux kernel version) as the starting point on the mainline. In total, our dataset consists of 645 bug-version pairs. We will determine whether each version is affected by a bug (vulnerable vs. non-vulnerable). We evaluate the accuracy of SYMBISECT against these bug-version pairs. Subsequently, to evaluate bug-inducing commit identification, we retained the bugs introduced after v4.20 (32 in total): SYMBISECT employs a binary search between the latest non-vulnerable version and version on which the bug was reported by syzbot to pinpoint the exact bug-inducing commit.

All experiments are conducted in Ubuntu-20.04 with 1TB memory and Intel(R) Xeon(R) Gold 6248 20 Core CPU @ 2.50GHz * 2. For each bug-version pair, we allocate a single CPU core for a maximum of 10,000 seconds of symbolic execution.

Comparison Targets. We compare SYMBISECT with the three following lines of work:

Tools	TP	FP	TN	FN	Accuracy	Precision	Recall	F-1 Score
SymBisect	237	29	348	31	90.7%	89.1%	88.4%	88.7%
Syzbot(PoC)	146	27	350	122	76.9%	84.4%	54.5%	66.2%
V0Finder	138	0	377	130	79.8%	100.0%	51.5%	68.0%
VSZZ	250	145	232	18	74.7%	63.4%	93.3%	75.4%

Table 1: **The results of vulnerable versions detection**

Tools	correct	incorrect	Accuracy
SymBisect	24	8	75%
Syzbot	16	16	50%
V0Finder	11	21	34.375%
VSZZ	18	14	56.25%

Table 2: **The results of bug-inducing commit identification**

- PoC-based bisection. Syzbot bisects bugs with PoCs to find the commit that introduced the bug [3]. We employ a crawler to directly retrieve results from the website. In instances where bisection results are unavailable, we execute the PoC on the target kernels to get the results.
- Patch-based bisection with SZZ algorithm. As described in §2.2, this line of research assesses vulnerability-(un)affected versions by locating the vulnerability-introducing commit with SZZ and its variants. In this line of work, VSZZ [12] is the state-of-the-art tool and it's open source. We set up VSZZ with their default options according to the tutorials[6].
- Patch-based bisection with vulnerable code clone detection. These methods are based on code similarity comparison. V0Finder [45] is a recent vulnerable code clone detector that is used to discover the first version where a vulnerability is introduced. We set up V0Finder with their default options according to the tutorials[5].

Evaluation metrics. For the evaluation of determining the vulnerable versions for a specific vulnerability, for each bug-version pair, we will get a verdict as true positives (TP), true negatives (TN), false positives (FP), or false negatives (FN). Then we calculate the corresponding accuracy, precision, recall, and F1 score. For pinpointing the precise bug-inducing commit, we received a binary result (either identifying the correct bug-inducing commit or not) from which we calculated the accuracy.

Ground truth verification. To ensure that the "Fixes:" tag is consistent with the bug-inducing commit we defined, we carried out the following verification for all such tags in our dataset: 1) If the PoC triggers the reported bug in the target version, then that version is deemed vulnerable. 2) If the path from the entry function to the target line is absent in the target version (for example, if the target function or line does not exist), then it is considered not vulnerable. 3) For versions that cannot be verified through the previous two steps, we manually analyze the logic of the vulnerability to determine if it exists in the target version.

After the verification, we found that the vast majority of "fixes" tags are consistent with the bug-inducing commit we defined. The only exception was a vulnerability introduced by two adjacent commits. The first commit defined a function related to the vulnerability, and the second introduced the caller of this function. According to our definition, the second is the bug-inducing commit, but the "fixes" tag pointed to the first. Interestingly, we noted that these two adjacent commits were merged into the Linux mainline branch together in a single merge commit, thus not affecting our evaluation results.

6.1 Accuracy of SymBisect (RQ1)

Accuracy of vulnerable version detection. As shown in Table 1, SymBisect achieves an overall accuracy of 90.7% over 645 versions, higher than all existing tools. Note that this evaluation is performed on a per-bug-version-pair basis.

Accuracy of bug-inducing commit identification. Table 2 shows the results of bug-inducing commit identification, SymBisect outperformed all the other cases with an accuracy of 75%. The reason the accuracy is lower (than vulnerable version detection) is that it aggregates the results from all kernel versions for a single bug. For example, if the vulnerability was introduced in v5.3, we might correctly label v5.4 as vulnerable; however, if we mistakenly labeled v5.3 as non-vulnerable, then we still will end up with an incorrect bisection result for the specific bug (FN). Upon manual inspection, we discovered that among these eight cases of inaccuracy, five were due to FPs, and three resulted from FNs.

False positives in SymBisect. SymBisect has 29 false positives (misidentifying non-vulnerable versions as vulnerable). The FPs generated by SymBisect tool arise from the intrinsic characteristics of under-constrained symbolic executions. For example, global variables are symbolized in our approaches, allowing the constraints to represent them as potentially holding any

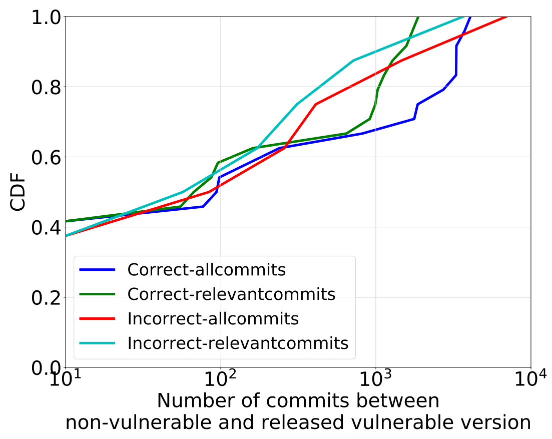

Figure 4: Comparison of commit number between correct and incorrect cases

value of the specified type. However, such a variable could be hard-coded somewhere that symbolic execution cannot access. Consequently, such under-constraining can lead to SYMBISECT concluding infeasible behaviors in practice.

As an example false positive, we find an OOB bug that arises from a lack of checks against socket types. In the kernel, different types of sockets possess different sizes. The mappings between the socket type and the corresponding structure sizes are stored as global variables in the kernel, which are symbolized in our detectors. When under-constrained, the symbolized mapping can produce any sizes from a given socket type, leading to false positives.

False negatives in SYMBISECT. Our evaluation records 31 false negatives (misidentifying vulnerable versions as non-vulnerable). The primary cause of FNs is the scalability issue. Certain vulnerabilities can be triggered only via a specific path, which might not be covered in the symbolic execution due to the time threshold, despite our effort to apply principled guidance during exploration. Moreover, the guidance may not be complete due to differences between the reference and target versions. If the symbolic execution lacks accurate guidance, it is likely to encounter scalability issues due to the complexity of kernels.

An example of this challenge occurs when a vulnerability site is influenced by a check against a pivotal variable. The vulnerability can be triggered only when this variable is set to a particular value in preceding functions. Yet, the distance between this value assignment and the condition check is substantial, with many functions with many state forks interspersed. Even with our guidance, satisfying such a nuanced condition in a limited time to activate the vulnerability proves challenging, resulting in false negatives.

Factors that influence accuracy. We hypothesize there are certain factors that affect the bisection accuracy. For instance, a plausible factor is the distance between the bug-inducing commit and the commit on which the bug is discovered. This is because the farther away the two commits are, the more changes may occur to the underlying bug-triggering condition.

To evaluate the distance factor, we counted the total number of commits between the released vulnerable version and the non-vulnerable version (the version before the bug-inducing commit) on the Linux mainline branch. Note that on the Linux mainline branch, a merge commit may combine updates from multiple commits on other branches, and we did not break down this merge commit to recount the number of commits. Another variation of the distance factor is to count only the "relevant commits" — commits that modify the files within the call stack responsible for triggering the bug — between the vulnerable and non-vulnerable versions.

We investigated the two aforementioned factors, and the results are presented in Figure 4. Contrary to our expectations, there appears to be little correlation between the two distance metrics and the accuracy of bisection in our dataset. Specifically, as illustrated in the figure, 37.5% (3 out of 8) of the incorrect cases (Incorrect-all commits) had very limited space (fewer than 10 commits in total) between bug-inducing commit and vulnerability-finding commit. This proportion is close to that of the correct cases (Correct-all commits), where 41.7% (10 out of 24) also had limited space.

We analyzed bugs that have more than 100 "relevant commits" (i.e., with large distances), which constitutes 14 bugs in total. The accuracy of SYMBISECT was 71.4% (10 out of 14). In general, we find that our method is effective in eliminating the influences of unrelated code changes. Even if those commits modified files included in the call stack (or even directly modified the corresponding functions), as long as they do not affect the existence of the vulnerability logic, our symbolic execution-based methods often can exclude their impact on the results.

6.2 Comparison (RQ3)

As shown in Table 1, SYMBISECT outperforms other tools effectively. It achieves higher accuracy (90.7% compared to the 77.1% average of preceding tools) and higher F1 scores (88.7% as opposed to 69.8%) than all previous tools. As expected, we observed that the main reasons for inaccuracies in existing PoC-based methods are the broken dynamical environment, inadvertent triggering of unrelated bugs, and evolving bug-triggering conditions as the code progresses. The fail-

Reason	FN	FP	Solved in SYMBISECT
Hard to reproduce	38	0	15
Detector not introduced	8	0	8
Build/boot errors	14	0	14
Config disabled	9	0	9
Trigger another bug	0	27	27
Over-constraint on inputs	53	0	53
Total	149		126

Table 3: **The reasons of PoC-based method failed**

ures of patched-based tools are due to their dependence on unreliable syntactic information and only consider a limited portion of bug-related code. In comparison, our solution based on static symbolic reasoning aims to capture the logic of the specific vulnerability and extend its scrutiny to a much broader context beyond the confines of the patched function.

Improvements over syzbot bisection. Table 3 outlines the reasons for the PoC-based method's failures in our dataset. The first five types are cited from the official syzbot documentation[3], while the final reason, "over-constraint" is a reason we observed. In fact, we find that it is the most common reason for inaccuracies. Notably, SYMBISECT has effectively addressed 83% of the inaccuracies associated with the PoC-based approach. We will now detail the causes of each failure and how SYMBISECT addressed them, as follows:

- Vulnerability with low probability of triggering. PoC-based approach often struggles to reproduce bugs that have a very low probability of triggering even in the released version that corresponds to the PoC. At present, for every target version, syzbot conducts testing only 10 times [3]. It is probable that vulnerabilities may not be triggered within these limited attempts. The under-constraint feature of SYMBISECT enhances its capability to fulfill the preconditions necessary for triggering the bug. As a result, SYMBISECT yields accurate results for 15 of the 38 cases within the given time threshold.
- Detector not introduced. The PoC-based approach is dependent on specific detectors, like the KASAN sanitizer. Until these detectors are integrated into the kernel, PoCs cannot detect vulnerabilities effectively. In contrast, SYMBISECT is equipped with its own symbolic execution detector, eliminating the need for reliance on sanitizers in the Linux kernel.
- Build/boot errors. As we discussed in §2, the static feature of SYMBISECT bypasses the problem resulting from kernel boot errors.
- Config disabled. As PoC-bisection goes back in time, certain kernel configs may be forcefully disabled when they conflict with the other config options. In contrast, since our solution does not require the compilation of the entire kernel, we can simply force other config conflicts to be disabled and make sure the vulnerable modules involved are compiled into LLVM bitcode for our analysis.
- Accidental triggering of unrelated bugs. The PoC has the potential to activate unrelated kernel bugs that break the program. Current syzbot does not look at the exact crash, nor does it attempt to distinguish between different types of crashes, leading to some FPs. In contrast, our tool focuses on the specific bug only upon reaching the target line (and analyze its associated operations). This allows us to effectively sidestep this issue.
- Over-constraint on inputs. This is essentially due to changes in the underlying bug-triggering conditions. Executing the original PoC does not always activate the bug in some vulnerable versions. Input mutations become necessary under these circumstances. The under-constrained symbolic execution approach treats all potential entry function arguments and global values comprehensively, effectively addressing these false negatives.

  Figure 5 presents an OOB vulnerability. Specifically, in function `mpol_parse_str()` if the str variable starts with "=", the flags variable will reference the first byte of str. If a certain condition at line 2 is met, the program skips to line 4. Here, a write operation occurs that exceeds the boundaries of str, leading to an out-of-bounds write. The PoC-based syzbot bisection incorrectly pinpoints a bug-inducing commit which modified the function `shmem_parse_one`—the caller of the `mpol_parse_str()` function. Prior to this misidentified commit, another check at line 8 was in place against the opt variable. The initial PoC fails this check, causing syzbot to label versions before this commit as non-vulnerable. However, by using a different input that bypasses this check, the bug remains exploitable. Instead, SYMBISECT symbolizes the inputs, making it easier to bypass such checks as long as a feasible solution exist.

Improvements over V0Finder. V0Finder failed to discover 107 vulnerable versions out of 230 cases, resulting in a low recall of 46.5%. The main reason is that V0Finder does a strict syntactic similarity comparison for the whole function. Specifically, after normalization and abstraction, it concludes that the target version is vulnerable only if the patch functions are strictly the same as those in the released version. Thus it cannot detect the vulnerable cases that are syntactically different, but convey the same vulnerable functionality.

```
                  The vulnerable function:
int mpol_parse_str(char *str,...)
1   char *flags = strchr(str, '=');
2   if(condition)
3       goto out
    ......
4   if (flags)
5       *flags++ = '\0';
    ......
out:
6   if (flags)
7       *--flags = '=';
```

```
          The incorrect Bug-inducing Commit
          (Identified by Syzbot Bisection):
static int shmem_parse_one(...)
    ......
8 - else if (!strcmp(opt, "mpol")) {
  -     ......
9 -     if (mpol_parse_str(value, &ctx->mpol))
        ......
10+ if (IS_ENABLED(CONFIG_NUMA)) {
  +     ......
11+     if (mpol_parse_str(param->string, &ctx->mpol))
```

Figure 5: Case study of syzbot FN

```
                       The Patch:
int qrtr_endpoint_post(...)
     struct qrtr_cb *cb;
1 - unsigned int size;
2 + size_t size;
    ......
3   if (len != ALIGN(size, 4) + hdrlen)
    goto err;
```

```
           The incorrect Bug-inducing Commit
               (Identified by V0Finder):
int qrtr_endpoint_post(...)
   ......
+   if (cb->type == QRTR_TYPE_NEW_SERVER) {
+       const struct qrtr_ctrl_pkt *pkt = data + hdrlen;
+       qrtr_node_assign(node, le32_to_cpu(pkt->server.node));
+   }
```

Figure 6: Case study of V0Finder FN

In Figure 6, we see an illustrative example. Here, a 4-byte size variable is prone to an overflow at line 3. To address this, the patch modifies the variable's size to 8 bytes. However, the bug-inducing commit pinpointed by V0Finder is actually a feature enhancement commit, unrelated to the vulnerability. This commit introduces multiple lines into the patched function. Due to this, V0Finder incorrectly designates all preceding versions as non-vulnerable, leading to a multitude of false negatives.

SymBisect, instead of syntactic comparison, extracts accurate semantic information. Thus it can distinguish vulnerability-irrelevant changes from significant changes effectively. Furthermore, it does not rely on patches. Whether the patch changes a function or not is irrelevant to SymBisect. As a result, SymBisect can eliminate a large number of FN cases of V0Finder. This significant advantage is largely due to the differing

```
                      The Patch:
int squashfs_read_data(...)
1 - TRACE("Block @ 0x%llx, %scompressed size %d\n", index,
2 + TRACE("Block @ 0x%llx, %scompressed size %d\n", index - 2,
        compressed ? "" : "un", length);
      }
3 + if (length < 0 || length > output->length ||
  +         (index + length) > msblk->bytes_used) {
4 +     res = -EIO;
5 +     goto out;
  +  }
```

```
          The incorrect Bug-inducing Commit :
    ...... (initialize the file)
6 +  TRACE("Block @ 0x%llx, %scompressed size %d\n", index,
```

```
            The correct Bug-inducing Commit :
int squashfs_read_data(...)
    ......
7   TRACE("Block @ 0x%llx, %scompressed size %d\n", index,
  -         compressed ? "" : "un", length);
8 - if (length < 0 || length > output->length ||
  -         (index + length) > msblk->bytes_used)
9 -     goto block_release;
```

Figure 7: Case study of VSZZ FP

foundational design principles of the two systems.

Improvements over VSZZ. VSZZ processes a patch as input and identifies the vulnerability-introducing commit by backtracing the patch's deleted lines through the code commit history to the earliest instance, facilitated by line matching. The earliest commit where these deleted lines were initialized is then marked as the commit that induced the bug. When multiple deleted lines originate from different commits, VSZZ selects the earliest of those commits as the bug-inducing commit. If the patch does not have any deleted lines, VSZZ identifies the commit that initialized the file mentioned in the patch as the bug-inducing commit.

Figure 7 illustrates a typical scenario where the underlying assumption fails, leading to a false positive. The deleted line 1 in the patch function is not created by the vulnerability-inducing commit, leading to backtracing to an earlier point. All commits situated between the commit identified by VSZZ and the actual inducing commit will be marked as FPs. In detail, the vulnerability was brought into the codebase by a commit that removed a certain validation check at line 8, then the vulnerability was patched by putting the check back in. However, the line they removed from the patch was just for logging that is not really related to the vulnerability. VSZZ traced this logging line back to when the whole function was first added, resulting in some FPs.

Basically, the commit that introduces the vulnerability may not alter the patch function at all, as demonstrated in our motivating example. Even if it does alter the patch function, it may not modify the deleted lines in the patch, just as in the above example. Furthermore, even

Strategy	Implementation
SymBisect	Exploration + Indirect call + Stack + Path
Pure Exploration	Exploration + Indirect call
Pure Re-tracing	Indirect call + Stack + Path
Stack	Exploration + Indirect call + Stack
Path	Exploration + Indirect call + Path

Table 4: **The relationship between strategy and guidance**

if the bug-introducing commit does change the deleted lines, it may only modify them rather than create them. In such cases, VSZZ may backtrace beyond the actual bug-introducing commit. These factors contribute to 112 false positives, a significantly higher figure than those seen with the other methods.

In contrast, our semantic method does not hinge on such a strong assumption. The symbolic execution engine accurately extracts semantic information, clarifying their relationship with the vulnerability.

6.3 Scalability of Different Exploration Strategies (RQ4)

To understand how the guidance helps with the overall results, we conduct a comparative study against alternative strategies. Fundamentally, SymBisect balances the exploration (i.e., allowing execution of the basic blocks in the medium-priority list) with re-tracing (i.e., aligning the execution trace with the one in the reference version). Therefore, we consider the following strategies that fall under various places in the spectrum: (1) pure exploration without any re-tracing or guidance (no consideration of basic block priorities), (2) pure re-tracing strictly following path guidance (i.e., when a branch leads to a high/highest priority exists, the other branches are prohibited from execution), (3) exploration with call stack guidance only. (4) exploration with path guidance only.

Table 4 shows the relationship between various strategies and specific guidance. "Indirect call", "Stack", and "Path" represent indirect call guidance, call stack guidance, and path guidance, respectively. For strategies with a combination of exploration and certain guidance, we assign different priorities to different paths based on guidance as defined in §4.1. In general, we prioritize the execution of higher priority branches, and do not prohibit the execution of branches unless the basic blocks are marked as the lowest priority (which can never reach the target line). In contrast, "Pure Re-tracing", when one branch leads to a high/highest priority basic block, the execution of other branches is prohibited. For all strategies, we employ the same entry function and target line and activate the indirect call guidance to ensure a fair comparison.

Results. The results are presented in Figure 8. The X-axis represents the symbolic execution time required to reach the target line, while the Y-axis shows the percentages of cases where the execution time falls within the range [0, X]. We have the following observations:

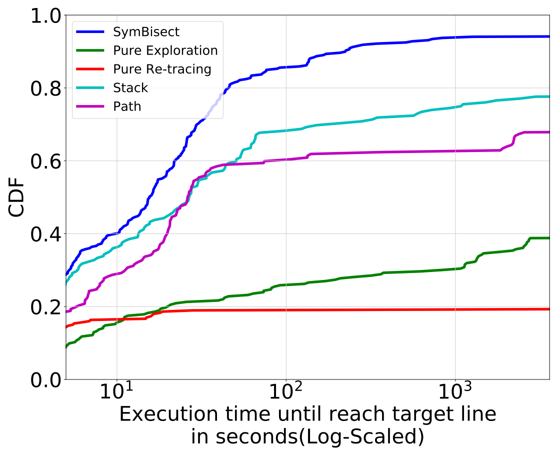

Figure 8: Scalability Evaluation

We can see that SymBisect performs the best. Pure re-tracing performs the worst, as it often fails to account for changes in the underlying bug-triggering condition and cannot reach the target line (e.g., the symbolic exploration is killed earlier than expected). Pure exploration performs second to last because it has too many execution paths to cover, resulting in path explosion. The remaining two strategies with limited guidance perform better than pure exploration but worse than SymBisect. When comparing call stack guidance against path guidance, we find that the former performs slightly better than the latter. This is consistent with our strategy in SymBisect where call stack guidance takes precedence over path guidance. In other words, guiding the execution toward the target function is more aligned with the end goal of reaching the target line of code.

7 Discussion

Exploration range. As discussed in §2, Relying solely on patch functions presents inherent disadvantages, prompting us to explore entire traces in order to gather comprehensive information relevant to vulnerabilities within the program. However, these traces may encompass thousands of functions, with the majority of them unrelated to the vulnerability at hand. Consequently, achieving a balance between scalability and accuracy primarily relies on determining the appropriate exploration range. While we employ specific heuristics to limit the range, there is still room for a more systematic approach to this decision-making process. For example, we envision one can apply static analysis (less precise but more scalable) to identify the vulnerability-related functions in advance, then skipping the unrelated functions when applying symbolic execution. Developing such a solution would significantly improve our capability to identify and address vulnerabilities without overwhelming our resources.

Support more bug types. The types of vulnerabilities supported by SymBisect depend on the symbolic engine it is based on (currently KLEE) and the detectors built on top of it (or provided by KLEE itself). SymBisect currently supports bugs that manifest as OOB and UAF, including type confusion and integer overflow bugs that manifest as OOB. There are a few types of bugs that are interesting to support for future improvements: (1) additional bug types such as use-before-initialization [52], (2) bugs that require precise reasoning across multiple syscalls, and (3) race conditions bugs. For (1), it requires additional symbolic detectors to recognize other bug types. For (2), symbolic execution across multiple syscalls is feasible but presents an additional scalability challenge. For example, in some OOB cases, the allocation and use of the vulnerable object occur in different system calls. Without analyzing the allocation, the analysis of the subsequent syscall on use will be under-constrained and therefore potentially lead to false positives. This means we will need to first collect the symbolic expression for the object size (in one syscall), and then reason about whether the use can go out-of-bounds (in another syscall). We envision an optimization to terminate the symbolic execution of the allocation syscall earlier, as soon as the object size info is collected and leave other unexplored variables under-constrained. For (3), there are specialized symbolic detectors that can detect specific race condition bugs, e.g., multi-reads and double-fetch [50]. In the context of bisection, we envision that a more general approach is to recognize the interleaving points [51] and record the desired interleaving during the execution of the PoC in the reference version and use it to guide the execution of the target version.

Support bugs without PoCs. When a fuzzer discovers bugs, it usually generates a corresponding PoC, but there are exceptions. In some cases, syzakaller only produces a bug report. We wish to point out that our tool does not necessarily have to rely on PoCs. Instead, as long as we can obtain traces that trigger the vulnerability, it would be sufficient to guide the symbolic execution. For example, with hardware support (e.g., Intel Processor Trace [21]), we envision bug reports can be accompanied with corresponding control flow information.

8 Related Work

Under-constrained symbolic execution in OS kernels. UCKLEE [33] represents the initial implementation of an under-constrained symbolic execution virtual machine based on KLEE. It is primarily utilized for patch verification as well as rule-based generalized checks, encompassing areas such as memory leaks, uninitialized data, and user input vulnerabilities. UBITect [52] and IncreLux [53] utilize under-constrained symbolic execution to identify feasible paths and mitigate false positives in static analysis when detecting Use-Before-Initialization (UBI) bugs. SID [46] aims to distinguish security-related patches from other bug fixes, which is different from our work. It attempts to set up a model for several types of vulnerabilities with the help of under-constraint symbolic execution, rather than simply extracting and comparing characteristics. Besides, previous studies that attempted to perform symbolic execution on operating system kernels addressed the scalability issues using the following methods: 1) Decrease the scope of symbolic execution when analyzing operating system kernels. For example, performing intra-procedural analysis on a specific function such as the patch function [54] [46]. However, the approach may not be suitable for our purposes. The existence of a vulnerability is not determined by a single function. 2) Concretizing symbolic inputs and global variables [15, 48]. In our cases, it will result in an over-constraint problem.

Dynamic vulnerable version identification. The information about the affected versions of a vulnerability is quite important [38]. Dai et al. [17] proposed a PoC migration approach that takes a PoC as input and migrates the PoC to verify other affected versions. However, it specifically targets user-space programs. Furthermore, as demonstrated in § 6, over-constraint on inputs is only one of the causes of failure.

Code clone detection. If two code fragments are highly similar, with only minor modifications, or identical due to copy-paste, then one fragment may be considered a code clone of the other [55]. Code clone detection is widely used in software engineering tasks such as program understanding, plagiarism detection, copyright infringement investigation, and code compaction [8, 35, 36, 39, 20]. These techniques are designed to detect general code clones with high accuracy and scalability. However, they do not aim to precisely reason about security properties of the code, which may be influenced by small changes while still preserving "similarity". In addition, vulnerable code clone detection [26, 25, 14, 49, 56, 14, 45] usually perform code clone detection on what they define as vulnerability-related code (a few lines within the patch function or the entire function, sometimes manually extracted [44]). However, the lack of vulnerability logic reasoning makes them imprecise, as demonstrated in our evaluation.

Information-retrieval-based bisection. Locus [43] was the initial method to pinpoint bugs at the software change level using token similarities from bug reports. ChangeLocator [47] determines Bug-Inducing Commit (BIC) using crash call stack information. Orca [13] ranks commits based on bug symptoms, like exception messages or customer feedback. Bug2Commit [30] aggregates features from bug reports and commit, averaging their vector representations. FONTE [10] identifies BIC via test coverage. It ranks commits by the suspiciousness of their modifications. Despite their scalability, these methods fall short in accuracy. As mentioned in the Background section, The state-of-the-art, Fonte, only reaches a 36% accuracy rate.

9 Conclusion

The identification of vulnerable versions of Open Source Software and pinpointing bug-inducing commits are crucial for vulnerabilities uncovered through fuzzing. In response to this, we introduce SYMBISECT, a precise methodology grounded in symbolic analysis. The central principle is that detailed symbolic information tends to be more stable compared to both the original PoC and syntactic similarity assessments during software evolution. Our experimental results confirm that SYMBISECT not only significantly surpasses the existing PoC-based approach in terms of accuracy, but also outperforms methods that rely on patches. With the insights gained from SYMBISECT about vulnerable versions, developers can precisely locate the bug-inducing commit. This empowers them to address the potential threats brought about by fuzzing vulnerabilities, thus promoting a more secure software ecosystem.

Acknowledgment

We thank anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Science Foundation under Grant #2155213, #2247881 and a Google Gift.

References

[1] Linux Kernel Faces Reduction in Long-Term Support Due to Maintenance Challenges. `https://www.linuxjournal.com/content/linux-kernel-reduction-longterm-support`.

[2] SymBisect Source Code. `https://github.com/zhangzhenghsy/SymBisect`.

[3] Syzbot Bisection. `https://android.googlesource.com/platform/external/syzkaller/+/HEAD/docs/syzbot.md#bisection`.

[4] Syzbot Bisection Motivation. `https://lore.kernel.org/all/CACT4Y+Y3nN=nLEkHXLFcX7vxp_vs1JrD=8auJ3cX9we6TQHO+w@mail.gmail.com/T/#u`.

[5] V0Finder Source Code. `https://github.com/WOOSEUNGHOON/V0Finderpublic`.

[6] VSZZ Source Code. `https://figshare.com/ndownloader/files/31748777`.

[7] R. Abreu, F. Ivančić, F. Nikšić, H. Ravanbakhsh, and R. Viswanathan. Reducing time-to-fix for fuzzer bugs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1126–1130. IEEE, 2021.

[8] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool. A systematic review on code clone detection. *IEEE access*, 7:86121–86144, 2019.

[9] N. Alexopoulos, M. Brack, J. P. Wagner, T. Grube, and M. Mühlhäuser. How long do vulnerabilities live in the code? a {Large-Scale} empirical measurement study on {FOSS} vulnerability lifetimes. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 359–376, 2022.

[10] G. An, J. Hong, N. Kim, and S. Yoo. Fonte: Finding bug inducing commits from failures. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 589–601. IEEE, 2023.

[11] C. Ascherm, S. Schumilo, T. Blazytko, R. Gawlik, , and T. Holz. Fuzzing with input-to-state correspondence. NDSS, 2019.

[12] L. Bao, X. Xia, A. E. Hassan, and X. Yang. V-szz: automatic identification of version ranges affected by cve vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2352–2364, 2022.

[13] R. Bhagwan, R. Kumar, C. S. Maddila, and A. A. Philip. Orca: Differential bug localization in {Large-Scale} services. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 493–509, 2018.

[14] B. Bowman and H. H. Huang. Vgraph: A robust vulnerable code clone detection system using code

property triplets. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 53–69. IEEE, 2020.

[15] W. Chen, X. Zou, G. Li, and Z. Qian. Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. USENIX Security, 2020.

[16] D. E. Cristian Cadar, Daniel Dunbar. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008) December 8-10, 2008, San Diego, CA, USA.

[17] J. Dai, Y. Zhang, H. Xu, H. Lyu, Z. Wu, X. Xing, and M. Yang. Facilitating vulnerability assessment through poc migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3300–3317, 2021.

[18] D. E. David A Ramos. Under-constrained symbolic execution: Correctness checking for real code. USENIX Security, 2015.

[19] eng Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[20] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 516–527, 2020.

[21] X. Ge, W. Cui, and T. Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[22] Google. Google syzbot. `https://syzkaller.appspot.com/upstream/`.

[23] Google. Google syzkaller. `https://github.com/google/syzkaller`.

[24] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3262–3278. IEEE Computer Society, 2023.

[25] J. Jang, A. Agrawal, and D. Brumley. Redebug: finding unpatched code clones in entire os distributions. Oakland'12.

[26] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. Oakland'17.

[27] X. Li, Z. Zhang, Z. Qian, T. Jaeger, and C. Song. An investigation of patch porting practices of the linux kernel ecosystem. *arXiv preprint arXiv:2402.05212*, 2024.

[28] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. ACSAC'16.

[29] K. Lu and H. Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[30] V. Murali, L. Gross, R. Qian, and S. Chandra. Industry-scale ir-based bug localization: A perspective from facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 188–197. IEEE, 2021.

[31] H. Peng, Y. Shoshitaishvili, and M. Payer. Tfuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*. IEEE, 2018.

[32] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. USENIX Security'15.

[33] D. A. Ramos and D. R. Engler. Under-constrained symbolic execution: Correctness checking for real code. In J. Jung and T. Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 49–64. USENIX Association, 2015.

[34] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 25:1294–1340, 2020.

[35] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[36] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*, pages 1157–1168, 2016.

[37] E. C. H. L. Seunghoon Woo, Hyunji Hong. Movery: A precise approach for modified vulnerable code clone discovery from modified open-source software components. USENIX Security, 2022.

[38] Y. Shi, Y. Zhang, T. Luo, X. Mao, and M. Yang. Precise (un) affected version analysis for web vulnerabilities. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[39] G. Shobha, A. Rana, V. Kansal, and S. Tanwar. Code clone detection—a systematic review. *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 2*, pages 645–655, 2021.

[40] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[41] D. Trabish, S. Itzhaky, and N. Rinetzky. A bounded symbolic-size model for symbolic execution. In D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, editors, *ESEC/FSE*, pages 1190–1201. ACM, 2021.

[42] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh. Syzvegas: Beating kernel fuzzing odds with reinforcement learning. USENIX Security, 2021.

[43] M. Wen, R. Wu, and S.-C. Cheung. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 262–273, 2016.

[44] S. Wi, S. Woo, J. J. Whang, and S. Son. Hiddencpg: large-scale vulnerable clone detection using subgraph isomorphism of code property graphs. In *Proceedings of the ACM Web Conference 2022*, pages 755–766, 2022.

[45] S. Woo, D. Lee, S. Park, H. Lee, and S. Dietrich. V0finder: Discovering the correct origin of publicly reported software vulnerabilities. In *USENIX Security Symposium*, pages 3041–3058, 2021.

[46] Q. Wu, Y. He, S. McCamant, and K. Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. NDSS, 2020.

[47] R. Wu, M. Wen, S.-C. Cheung, and H. Zhang. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering*, 23:2866–2900, 2018.

[48] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. {FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, 2018.

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

[50] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678. IEEE, 2018.

[51] T. Yavuz. Sift: A tool for property directed symbolic execution of multithreaded software. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 433–443, 2022.

[52] Y. Zhai, Y. Hao, H. Zhang, D. Wang, C. Song, Z. Qian, M. Lesani, S. V. Krishnamurthy, and P. L. Yu. Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel. In *ESEC/FSE*, pages 221–232. ACM, 2020.

[53] Y. Zhai, Y. Hao, Z. Zhang, W. Chen, G. Li, Z. Qian, C. Song, M. Sridharan, S. V. Krishnamurthy, T. Jaeger, and P. L. Yu. Progressive scrutiny: Incremental detection of UBI bugs in the linux kernel. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[54] H. Zhang and Z. Qian. Precise and accurate patch presence test for binaries. USENIX Security, 2018.

[55] H. Zhang and K. Sakurai. A survey of software clone detection from security perspective. *IEEE Access*, 9:48157–48173, 2021.

[56] D. Zou, H. Qi, Z. Li, S. Wu, H. Jin, G. Sun, S. Wang, and Y. Zhong. Scvd: A new semantics-based approach for cloned vulnerable code detection. In *DIMVA*, pages 325–344. Springer, 2017.

[57] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian. {SyzScope}: Revealing {High-Risk} security impacts of {Fuzzer-Exposed} bugs in linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3201–3217, 2022.