This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

SNIPTEST: Fuzzing Multi-Level Code Slices for Validating Vulnerabilities

Aniruddhan Murali, *Student Member, IEEE*, Noble Saji Mathews, *Student Member, IEEE*, Mahmoud Alfadel, *Member, IEEE*, Meng Xu *Member, IEEE* and Meiyappan Nagappan

Abstract—Modern software systems are increasingly complex, and static analysis tools are commonly used to identify potentially vulnerable code by issuing warnings. However, these warnings often require manual inspection to confirm whether the reported issues are real, making the process time-consuming and error-prone. Directed fuzzing has emerged as a powerful automated technique to validate the warnings. However, applying it to the entire project in response to each warning is computationally infeasible, often requiring days of execution to achieve only incremental improvements in code coverage.

We present SNIPTEST, an execution-based warning triage framework that generates and fuzzes compiled code slices centered around static-analysis warnings. Rather than proving exploitability in the full program, SNIPTEST provides evidence about how a warning behaves under progressively expanded sliced execution contexts. Unlike prior approaches that extract slices from the program entry point or limit slice size for filtering, SNIPTEST constructs slices of arbitrary size and compiles them into standalone testable units. It employs a layer-by-layer slicing strategy, incrementally expanding context around the target location to validate potential vulnerabilities with increasing precision. We evaluate SNIPTEST on a benchmark of 97 true vulnerabilities and 97 false alarms across three real-world projects. SNIPTEST produces Possible True Positive evidence for 53 of 97 confirmed vulnerabilities (54.6%) by triggering the corresponding bug oracle consistently across all three analyzed slice levels, while the remaining cases are unreachable. Particularly, in 40.2% of these cases, it exploits the vulnerability along the observed execution path, matching the top three stack frames. On the 97 confirmed false alarms, SNIPTEST produces Possible False Positive evidence for 54 cases (55.6%) by reaching the warning without triggering the bug oracle, but misclassifies 28 cases (28.8%),and the remaining cases are unreached. Compared with AFLGo under our benchmark configuration, unseeded SNIPTEST reaches more locations than unseeded AFLGo and achieves comparable reachability to seeded AFLGo. Furthermore, SNIPTEST enhances efficiency, achieving a 5.5–$10.6\times$ speedup in fuzzing time compared to seeded or unseeded directed fuzzers. We also compare SNIPTEST with LLM based approach for classifying static analysis warnings LLM4SA. We find that SNIPTEST outperforms LLM4SA with an F1 score of 0.791 compared to 0.360 for LLM4SA. Finally, we demonstrate the practical relevance of SNIPTEST by identifying three new vulnerabilities in two open source projects leading to disclosure of CVE-2025-11964.

Index Terms—Fuzzing, Vulnerability detection, Code slicing

1 INTRODUCTION

Static analysis tools are widely used to detect potentially harmful bugs (i.e., vulnerabilities) early in the software development lifecycle [18], [47], [54], [61]. However, they often generate a large volume of warnings, many of which are false positives [17], [40], [49], [50]. In parallel, new code are frequently committed into the codebase, and these changes must be carefully reviewed to prevent regressions in production systems [30]. As a result, developers face a costly and error-prone triage process to determine which warnings or commits require urgent attention and which can be deprioritized.

A promising idea is dynamic validation—a family of techniques that attempt to confirm whether a suspicious code location actually results in failures at runtime. Among these, fuzzing is particularly intriguing due to its effectiveness in uncovering vulnerabilities [29], [46]. However, applying fuzzing to large codebases or individual warnings is challenging: writing tailored harnesses for each warning is labor-intensive; while general fuzzing often reach a coverage plateau before reaching the intended code location [23].

These limitations have led to the emergence of directed fuzzing, which is further categorized into distance-based [24], [27], [41] and slicing-based. In slicing-based fuzzing, a program slice (i.e., code closely related to the warning or commit) is isolated and fuzzed independently [28], [34], [38], [48]. Compared with distance-based fuzzing, slicing-based fuzzing starts at code locations that are much closer to the suspicious code, which significantly increases the chances of hitting the target. While one cannot get an end-to-end input that proves the existence of a bug, the under-constrained nature of a slice is especially useful for ruling out false bug reports, i.e., if a bug cannot be triggered with even less constraints on the inputs, it is more likely to be a false alarm. This is the core idea behind FUZZSLICE [48].

However, current slicing-based techniques face a key limitation. Program slices are generated ahead and independent of fuzzing. More specifically, the scope of slices is often predetermined based on heuristics and does not adapt to fuzzing results. This implies that the fuzzer may either omit necessary context (if the slice is too small [48]) or introduce excessive complexity (if the slice is too large [24]).

Mahmoud Alfadel is with the University of Calgary, Canada. The remaining authors are with the David R. Cheriton School of Computer Science, University of Waterloo, Canada.
E-mail:{a25murali, noblesaji.mathews, mei.nagappan, meng.xu.cs}@uwaterloo.ca, mahmoud.alfadel@ucalgary.ca

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

Furthermore, predetermined slices imply a limited set of execution paths or entry points, which can lead to false negatives by ignoring alternative call chains that may reach the same vulnerability [32]. These constraints can limit their ability to expose exploitable vulnerabilities or correctly dismiss benign warnings.

To address these challenges, we introduce SNIPTEST, a compositional fuzzing framework that validates vulnerabilities through slices of increasing context. Instead of relying on a fixed set of slices, SNIPTEST organizes validation around *slice levels*. At Level 0, SNIPTEST generates a slice rooted at the function that contains the warning. At Level 1, it expands the analysis to immediate callers of that function and their dependencies. At Level 2, it further expands to higher-level callers, and so on. Critically, this leveled design is not merely about testing larger slices. The power lies in treating the trajectory of outcomes across levels as a diagnostic signal. A warning that triggers only in the most underconstrained slice (Level 0) is likely brittle and context-sensitive—it may disappear once caller guards or initialization operations are reintroduced. In contrast, a warning that persists as we expand the calling context points to a systemic issue that warrants prioritization. By leveraging the sequence of outcomes rather than any single slice in isolation, SNIPTEST elevates slice-based fuzzing from a filtering technique to reactive slicing for distinguishing benign anomalies from genuinely critical vulnerabilities. Figure 2 illustrates this process using a call graph in which the vulnerable function is progressively embedded in broader calling contexts.

Our work SNIPTEST improves on FUZZSLICE [48] to support multi-level slicing—the idea of progressively expanding slices backward from a vulnerability. Multi-level slicing introduces new challenges such as handling an explosion of paths at higher levels, and avoiding redundant fuzzing across similar slices. SNIPTEST tackles these challenges through dedicated optimizations, time-bounded slicing, and a staged evaluation design that makes multi-level analysis tractable and meaningful (Sections 5 & 6). SNIPTEST tests each bug across multiple slice levels to assess not just whether it can trigger a vulnerability, but how consistently it does so across all slices, providing a clearer indication of the bug likelihood to represent an exploitable vulnerability.

Throughout the paper, we use "validation" in a pragmatic sense, i.e., SNIPTEST produces execution evidence for warning triage in slice context, rather than proving full-program exploitability. A warning classified as a possible true positive indicates that SNIPTEST triggered the corresponding bug oracle within the analyzed sliced contexts, while a possible false positive indicates that the warning was reached without triggering the bug oracle in those contexts.

Our study contributes the following advances:

- **Multi-context warning triage.** We introduce a warning-triage protocol that organizes executable slices by call-graph distance from a warning. SNIPTEST changes the unit of reasoning from one local function slice to a family of progressively constrained execution contexts. It aggregates observations within each level and across levels to determine whether the warning persists, disappears, or becomes unreachable as caller context is restored. To the best of our knowledge, SNIPTEST is the first work to dynamically adjust code slices based on fuzzing feedback.
- **Engineering support for scalable multi-level analysis.** We introduce caching, pruning, staged stopping, parallel execution and wrapper-variable trimming to make the construction and fuzzing of multiple caller-rooted slices practical. These are engineering extensions around the low-level slice construction inherited from FUZZSLICE.
- **Comprehensive evaluation.** We evaluate SNIPTEST across four key dimensions: effectiveness, fidelity, efficiency in detecting true vulnerabilities and false alarms. Our work is comprehensive due to the indepth discussion of true positives, true negatives, false positives and false negatives - rigor not commonly found in prior work [39], [48], [51]. We also provide direct comparison with the directed fuzzer AFLGo [24] and the LLM-based approach LLM4SA [55]. SNIPTEST achieves a 5.5-10.6x speedup in fuzzing time over seeded or unseeded AFL-Go. Additionally, SNIPTEST outperforms LLM4SA, with an F1 score of 0.791 versus 0.360.
- **Practical vulnerability detection.** We demonstrate the practicality of SNIPTEST by detecting 3 real world vulnerabilities in 2 Open Source projects [3], [10], [11] leading to disclosure of CVE-2025-11964 [13].
- **Artifact & Reproducibility.** Our replication package and datasets are publicly available.

2 Background on Fuzzing Approaches

In this section, we provide background on coverage-guided fuzzers and directed fuzzers.

Coverage-guided fuzzers. Fuzzing is an automated software testing technique that generates program inputs to expose crashes, bugs, and security vulnerabilities. Traditional fuzzers operate in an undirected manner: they aim to maximize overall code coverage or execution diversity without prioritizing any specific program locations. Popular coverage-guided fuzzers, such as AFL and libFuzzer [1], [2], fall into this category and have been widely successful in discovering shallow and medium-depth bugs.

Directed fuzzers. Directed fuzzing extends this model by biasing input generation toward specific targets, such as known vulnerable code locations, patches, or warnings reported by static analysis tools. Instead of maximizing global coverage, directed fuzzers such as AFLGo and HawkEye [24], [35], guide execution toward target locations using metrics such as control-flow distance or reachability. This approach is particularly effective for deep or hard-to-reach bugs but typically requires accurate target specification and may incur additional overhead in analysis and guidance.

3 Why SnipTest? A Motivating Example

Our goal is to validate whether suspicious code locations, such as those emerged by static analysis tools or recently changed, correspond to actual vulnerabilities. For instance, if a tool reports a potential memory leak or buffer overflow, we aim to determine whether the issue is a true bug or a

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696


```
1  struct btrace {                                   Level 1
2      char* p;
3      char* q;
4  };
5                                                    Level 0
6  void btrace_alloc(struct btrace* X) {
7      X = (struct btrace*) malloc(sizeof(struct
       btrace));  // Allocate X
8      X->p = (char*) malloc(10);
                // Allocate p
9      X->q = (char*) malloc(10);
                // Allocate q
10 }
11
12 void btrace_clear(struct btrace* X) {
13     free(X->q);       // Free q
14     free(X);          // Free X
15     // Memory leak: 'p' is not freed
16 }
17
18 void use_btrace(char** s) {
19     struct btrace* X;
20     btrace_alloc(X);      // Allocate memory
21     doSomething1(X->p, s); // Use p
22     doSomething2(X->q, s); // Use q
23     btrace_clear(X);       // Clear q and X
24 }
```

Fig. 1: Example from `binutils` project [22]. The pointer `p` is allocated but never freed, leading to a memory leak.

false alarm. Manually inspecting each warning or fuzzing the entire program in response to each one is inefficient and often ineffective [23], [25].

To illustrate our approach, we present a motivating example from the `binutils` project [22]. Listing 1 shows simplified C code involving three functions: the function `btrace_alloc` performs heap allocation for a struct and its fields, `btrace_clear` partially deallocates the structure, and `use_btrace` orchestrates the full flow.

When a static analysis tool such as Infer [5] analyzes this code, it reports a memory leak associated with the field `p` in the `btrace_alloc` function. Validating this warning using whole-program fuzzing, such as fuzzing the entire `binutils` project, is slow, often requiring days of execution and project-specific fuzzing dictionaries, and may still fail to trigger the vulnerability.

SNIPTEST adopts a more targeted strategy. It begins by generating and fuzzing a minimal code slice containing the warning location, which we refer to as Level 0. In this example, Level 0 includes only the function `btrace_alloc`. When fuzzed in isolation, a bug oracle e.g. LeakSanitizer [8] reports memory leaks for all three heap-allocated components: `p`, `q`, and `X`.

To improve diagnostic precision, SNIPTEST incrementally expands the slice. At Level 1, SNIPTEST includes the caller `use_btrace` and its dependencies, such as `btrace_clear`. This broader slice enables both allocation and deallocation logic to be exercised. Fuzzing at this level suppresses the leak reports for `q` and `X`, as valid deallocation paths are now included. However, the warning for `p` persists, since `p` is never freed. At higher levels, we would still expect `p` to be leaked.

This example highlights the core insight behind SNIPTEST: by observing how warnings behave across incrementally expanded slice levels, we can distinguish between true bugs and false alarms. If a warning consistently triggers across multiple levels, it is likely signaling a true vulnerability. If the code is executed but the warning does not manifest, it is likely a false alarm. That is, iteratively growing the code slice around the warning and checking if the warning persists, SNIPTEST can offer a lightweight yet semantically grounded way to prioritize warnings without requiring full-program fuzzing.

4 Related Work

Program slicing is well-studied, with most work focusing on uncompiled backward slices from entry points to vulnerabilities [21], [42], [44]. However, these slices are often overly large for arbitrary code points [19], [20], [36], [53]. In contrast, SNIPTEST operates at the function level, terminating backward slices at different functions, and finally compiling and linking small slices to enable quick exploration.

A common technique for dynamically validating vulnerabilities is directed fuzzing. Prior work has proposed various strategies for directing fuzzers toward specific program regions [26], [27], [35], [37], [43], [57], primarily by mutating inputs to increase the likelihood of reaching a target location. Therefore directed fuzzers can be used to verify static analysis warnings. The key distinction between SNIPTEST and directed fuzzing lies in their entry points: while directed fuzzers typically begin at the MAIN function and explore the entire program, SNIPTEST generates small slices around warning locations and confines fuzzing within these slices. Input mutation strategies from directed fuzzing are orthogonal to SNIPTEST and could be integrated into its fuzzing engine.

FUZZSLICE is the closest implementation and methodological predecessor to SNIPTEST. Given a target function, FUZZSLICE resolves dependencies, extracts a compilable function-rooted slice, generates a fuzzing wrapper, and tests the resulting binary. Its unit of reasoning is a single underconstrained local slice, and its primary use case is false-positive filtering. SNIPTEST reuses this slice-materialization operation—it does not introduce a new low-level slicing algorithm. Rather, its contribution lies in the surrounding framework: selecting multiple caller roots via static call-graph analysis, grouping slices by caller distance, testing multiple contexts per level, aggregating observations within and across levels, and applying staged stopping and scalability optimizations. Level 0 is the closest comparable setting to FUZZSLICE, as both analyze a slice rooted at the warning-containing function. However, SNIPTEST is not identical even at Level 0, owing to wrapper-generation improvements. More importantly, Level 0 is only the initial step in a multi-level warning-triage process. While FUZZSLICE focuses on underconstrained exploration of fine-grained slices primarily for false-positive filtering, SNIPTEST supports adjustable slice sizes, enabling improved precision and more effective vulnerability detection. Beyond that, SNIPTEST incorporates static analysis to automatically discover ancestor functions, expands slices based on fuzzing outcomes, and introduces multiple optimizations to classify warning behavior across progressively larger code contexts as benign, vulnerable, or unreachable.

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

Runtime bug oracles such as AddressSanitizer, LeakSanitizer, UndefinedBehaviourSanitizer and MemorySanitizer [14], [15], [16], [52] are used by SNIPTEST to detect several issues (e.g., out-of-bounds access, null pointer dereference, timeouts, double free), but other oracles—such as developer-provided assertions or even LLM-based assertions—can be integrated at runtime without modifying the framework. The design of effective oracles is complementary to, and independent of, SNIPTEST itself. As more expressive sanitizers or assertions become available, SNIPTEST (like any fuzzing system) can naturally detect a broader class of bugs, including logic errors.

Other approaches for identifying actionable static-analysis warnings include machine-learning–based techniques and, more recently, LLM-based approaches. Machine-learning–based methods train classifiers to distinguish vulnerable from non-vulnerable code using features extracted from static-analysis outputs, such as warning types, code metrics, or data-flow characteristics, and they require large, high-quality labeled datasets [33], [58], [59]. LLM-based approaches, in contrast, leverage pretrained language models to reason about code semantics and warning context, reducing the need for handcrafted features but still relying heavily on the quality and availability of static-analysis information or prompt context [45], [56]. These approaches differ fundamentally from SNIPTEST; they operate at the prediction level without executing the program.

5 SNIPTEST APPROACH

SNIPTEST aims to validate potential vulnerabilities reported by static analysis tools or recently changed code by generating and testing executable code slices that include the reported vulnerability across various program contexts. This is done through four key steps: **(1)** Construct a static function-level call graph; **(2)** Identify the function enclosing the vulnerability; **(3)** Generate and compile slices of increasing context; **(4)** Fuzz the resulting binaries and classify the vulnerability. We describe each step below.

(1) Construct a static function-level call graph. To assess whether a vulnerability is likely to be triggered at runtime, we first need to understand the possible execution paths in the program. Therefore, we build a static call graph that maps caller-callee relationships between functions. We use Tree-sitter [6], to extract the Abstract Syntax Tree (AST), identifying function definitions and calls to build the function-level call graph.

(2) Identify the function enclosing the vulnerability. Static analysis tools often generate warnings at specific code locations, but without runtime context. Hence, identifying the function enclosing the vulnerability is necessary to isolate the affected code region. In this step, using the AST from Step 1, we search for the exact function enclosing the given warning and designate it as the vulnerable function. This function forms the starting point for all code slices.

(3) Generate and compile slices of increasing context. Analyzing a vulnerability warning in isolation may lead to incomplete or misleading results, as its execution may depend on a broader calling context, making it necessary to systematically expand the code slice to capture relevant execution paths. At the same time, expanding too much can introduce unnecessary complexity and computational overhead. We balance these tradeoffs by iteratively growing the code slice around the vulnerable function. This involves two key sub-steps:

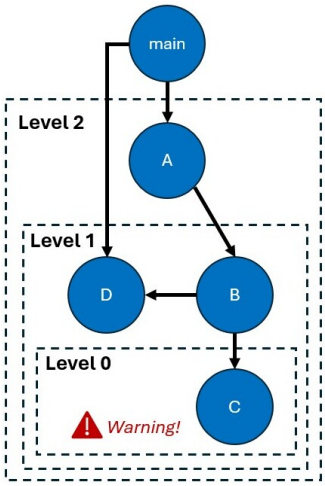


Fig. 2: An example illustrating slice levels where each node represents a function.

(3a) Define and grow the code slice. To systematically expand the execution context, we need a structured way to group slices based on their size and context. We introduce the concept of slice level, which groups slices based on their distance from the vulnerable function in the call graph. Our slicing process follows these principles:

- Level 0: Contains only the vulnerable function.
- Level 1: Adds all immediate callers of the vulnerable function.
- Level 2: Adds callers of Level 1 functions, etc.

Each level adds more calling context, allowing us to test whether the vulnerability warning depends on external inputs or conditions. If a vulnerability is triggered at Level N, further expansion is necessary as higher levels may reveal hidden preconditions that prevent the vulnerability from being exploited. If the suspected code location is reached sufficiently but no vulnerability is triggered, further expansion may be unnecessary [48].

Figure 2 shows a call graph where `main` calls both `A` and `D`; `A` calls `B`, which in turn calls `C` and `D`. The vulnerable function is `C`. The slice expands progressively, i.e., in Level 0, only function `C` is included (minimal slice). Level 1 includes functions `B`, `C`, and `D` in one slice, as `B` has a maximum calling distance of 1 from `C` and `B`'s dependencies are `C` and `D`. Level 2 further includes function `A` in a single slice, with a maximum calling distance of 2 from `C`. The slice includes `A` and its recursive dependencies `B`, `C`, `D`. Depending on the callgraph, there can be multiple slices at a given level.

(3b) Compile code slices using FUZZSLICE. After SNIPTEST selects an entry function for a particular slice level, it invokes FUZZSLICE to compile that function-rooted slice. FUZZSLICE resolves cross-file dependencies, removes code unnecessary for compilation, produces a standalone binary, and generates a baseline fuzzing wrapper. In this architecture, FUZZSLICE serves as a slice-construction component. FUZZSLICE does not construct SNIPTEST's static call graph,

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

select caller roots, define slice levels, schedule exploration across levels, aggregate outcomes, or produce final warning-level classifications. These operations belong to SNIPTEST's multi-level orchestration and evidence-aggregation methodology. SNIPTEST also extends FUZZSLICE's wrapper generation by trimming unused wrapper variables and adding support for fuzzing enums and bitfield data types. These changes improve input efficiency and broaden the set of functions that can be tested.

For each slice level, we identify all functions whose distance from the vulnerable function matches that level. In Figure 2, for instance, only function `C` (the vulnerable function) is sliced at Level 0. At Level 1, we slice function `B`, which is one level up from `C`, and its dependencies. At Level 2, function `A` is also included.

This step finally produces multiple binaries, each containing a code slice with a defined amount of context.

(4) Fuzz binaries and classify the vulnerability. In this step, we fuzz all generated binaries at each slice level to determine whether the suspected code location is potentially a true bug or a false alarm.

At this stage, multiple code slices have been compiled and linked, each corresponding to a different execution path that leads to the suspected function. We iteratively perform fuzzing, starting from binaries at Level 0, followed by those at Level 1, and so forth. SNIPTEST uses AFL++ [31] to fuzz the binaries and collects coverage information using llvm-coverage [2] in order to track execution frequencies of each line, alongside the results of runtime bug oracles (e.g., ASAN [52] and UBSAN [16]) to detect vulnerabilities.

The challenge at this stage is to consolidate the fuzzing results across different execution paths and slice levels to make a final classification of the vulnerability warning. We perform classification at three stages:

(1) Classification per binary. Each binary is fuzzed independently, and its behavior is classified into one of three states based on coverage and bug detection results:

(a) Triggered: The bug oracle detects a vulnerability at the warning location, confirming that the warning has been triggered in the code slice.
(b) Reached: The warning location is executed, but no bug is detected by the bug oracle, suggesting that while the path is feasible, it does not necessarily lead to an actual vulnerability.
(c) Not Reachable: The warning location is never executed during fuzzing, indicating that no execution path in the slice reaches it.

(2) Classification per slice level. Since multiple binaries may exist at a given slice level–each representing different execution paths–the fuzzing results must be aggregated at this level before making further decisions. We aggregate the results from all binaries at a given slice level as follows.

(a) Vulnerable (V): If at least one binary at the current slice level is triggered, then a vulnerable execution path has been identified. This suggests that the warning location is likely a true positive.
(b) Benign (B): If no slice at the current level is triggered, but at least one slice is reached, it indicates that execution paths to the vulnerability warning exist, but no vulnerabilities have been exploited. This suggests that the warning is likely a false positive, as an extensive search fails to identify an offending input that triggers the bug.
(c) Not Reachable (NR): If all binaries at the current level classify the warning as Not Reachable, no feasible execution path leads to the warning at this level. In such cases, the warning location remains undetermined, and we cannot make a conclusive classification at this slice level.

(3) Final classification across all levels. After analyzing individual binaries and consolidating the results at each slice level, the final classification of the warning is determined based on its behavior across multiple levels. If the analysis is limited to a predefined number of slice levels (e.g., Level 0 to Level 2), we make the final decision as follows:

(a) Possible True Positive (PTP): If the warning is consistently labeled as "Vulnerable" at all slice levels, it is classified as a PTP, indicating a high likelihood of an actual vulnerability.
(b) Possible False Positive (PFP): If the warning is reached through "Benign" paths at lower slice levels and remains benign or "Not Reachable" at higher levels, it is classified as a PFP. This reflects our validated assumption (Section 7) that warnings benign at shallow slices are unlikely to become vulnerable in deeper ones [48].
(c) Not Reachable (NR): If the warning remains "Not Reachable" at all slice levels, it means that conclusions cannot be drawn about the validity of the warning, as it was never reached during fuzzing.

The final classification rule is a heuristic evidence-aggregation strategy rather than a formal proof of vulnerability or benignness. Its purpose is to support warning triage by summarizing how a warning behaves across increasingly broader sliced execution contexts. Consistent triggering across slice levels increases confidence that the warning should be prioritized, while reaching the warning without triggering the bug oracle provides evidence that the warning may be benign under the explored contexts. Accordingly, we use cautious labels—Possible True Positive, Possible False Positive, and Not Reachable—to reflect the evidential nature of the classification.

6 SNIPTEST OPTIMIZATIONS

While SNIPTEST systematically tests vulnerability warnings through iterative slicing and fuzzing, practical challenges may arise. Specifically, as slice levels grow, the number of execution paths can increase exponentially, leading to excessive slice creation and redundant fuzzing. Since each execution path requires a separate binary for analysis, inefficient slicing and fuzzing can significantly inflate computational costs.

To address these challenges, SNIPTEST proposes a set of optimizations that fall into two categories: (1) Slice Creation Optimizations; (2) Slice Fuzzing Optimizations. Below, we describe each set of optimizations in detail.

6.1 Slice Creation Optimizations

These optimizations focus on reducing time to construct slices, especially when slices overlap across levels.

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

(O1) Cache minimized slices during recursive slicing. When slicing a function, SNIPTEST recursively minimizes its dependencies to construct a compilable unit. At higher slice levels, these dependencies often overlap across different slices. To eliminate redundant work, SNIPTEST caches minimized slices for individual dependencies. When a previously minimized dependency is needed again, it is retrieved directly from the cache, avoiding re-minimization and re-compilation. This also helps prevent redundant compilation when there are cyclic function calls in the project. This reduces the cost of constructing large slices at higher levels.

(O2) Parallelize slice creation. Slice creation can be efficiently parallelized across multiple cores since it involves copying files from the project and trimming code extraneous to each slice, followed by compiling the slice [48]. Workloads are distributed across multiple cores with centralized slice caching.

6.2 Slice Fuzzing Optimizations

These optimizations focus on improving the efficiency of fuzzing once the slices have been constructed.

(O3) Skip redundant fuzzing after triggering a vulnerable path. During fuzzing at a given slice level, SNIPTEST generates multiple binaries corresponding to different ancestor functions. Once fuzzing triggers the vulnerability in any binary, further fuzzing at that level becomes redundant for classification purposes. Thus, SNIPTEST halts fuzzing remaining slices at that level immediately after the first successful vulnerability exploit.

(O4) Skip fuzzing at higher levels for non-triggering warnings. If slices at a given level reach the warning location but do not trigger the bug oracle, SNIPTEST skips fuzzing higher-level slices for that warning. This optimization acts as an early stopping rule for this specific case, i.e., once a warning has been exercised without triggering a failure, additional caller context is unlikely to make the same warning vulnerable under our validation model. This optimization does not define the full stopping condition of SNIPTEST, but it implements one stopping decision used to avoid redundant higher-level fuzzing.

(O5) Parallelize fuzzing of independent slices. Each slice results in a single standalone binary, independent of other slices. SNIPTEST exploits this by enabling parallel fuzzing of slices across cores or machines without requiring synchronization. Unlike full-program fuzzing, which needs complex input sharing across threads, fuzzing independent slices achieves near-linear scalability, hence reducing total analysis time.

(O6) Remove code irrelevant to warning execution. To accelerate fuzzing, SNIPTEST prunes code that cannot influence the vulnerability's execution. Using static intra-procedural AST analysis, SNIPTEST removes code appearing after the vulnerability location that is not enclosed within control-flow structures (e.g., `if` or `while` blocks). This minimization can reduce fuzzing overhead and improve the probability of rapidly triggering the vulnerability. This pruning step is intra-procedural and hence unaffected by cycles in the call graph. In the case of higher slice levels, we prune the entry-point function by removing code appearing after the last vulnerable function call leading to the vulnerability.

7 EVALUATION

We organize the evaluation of SNIPTEST around the following research questions:

- **RQ1:** How effective is SNIPTEST at validating true vulnerabilities across slice levels?
- **RQ2:** How effective is SNIPTEST at filtering out false alarms across slice levels?
- **RQ3:** How accurately does SNIPTEST reproduce the actual execution paths of true vulnerabilities in stack traces?
- **RQ4:** How efficient is SNIPTEST with its slice-level optimizations?
- **RQ5:** How does SNIPTEST compare with directed fuzzers in efficiency and target reachability?
- **RQ6:** How does SNIPTEST compare with LLM based approaches for classifying vulnerabilities?
- **RQ7:** How does SNIPTEST compare with slice based approach FUZZSLICE?

To answer these questions, we first describe our evaluation setup in Section 7.1, including dataset construction and SNIPTEST configuration. We then report results for each RQ. in Sections 7.2–7.8.

7.1 Evaluation Setup

In this section, we describe the datasets, the environment, and the configuration used to evaluate SNIPTEST.

Dataset. SNIPTEST is designed to validate individual code locations, such as those emerged by static analyzers or introduced in recent commits, by slicing and fuzzing the surrounding program context. To evaluate SNIPTEST, we require datasets with ground truth: confirmed true vulnerabilities and false alarms.

We use the RevBugbench framework [60] to inject true vulnerabilities into three real-world projects: libxml2, zstd, and PROJ. RevBugbench accepts a bug fix pattern as input, specifying both the syntactic structure and semantic conditions of a bug. If a code location matches this pattern, RevBugbench "reverts" the fix, reintroducing a likely bug into the code. The resulting buggy version contains a true bug in the "reverted" function, while a corresponding warning of the same function in the fixed version (i.e., without the reverted fix) serves as a false alarm.

We validate the behavior of each injected and fixed variant by running existing test suites of the project, confirming that: (1) the injected version causes a failure, and (2) the fixed version passes the test suite without triggering the bug. This allows us to create two sets of suspicious code locations per project: one with confirmed vulnerabilities and one with confirmed false alarms.

Table 1 provides statistics on the projects in our benchmark. Of the three projects, zstd contains the largest number of vulnerabilities (49), followed by libxml2 (38), and PROJ (10). They include 20 heap overflow, 59 stack overflow, 3 floating-point exceptions, 5 global overflows, 8 memory leaks, and 2 stack use-after-scope vulnerabilities. The 97 true vulnerabilities forms **dataset-TP** and 97 false alarms

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

TABLE 1: Statistics on the repositories in our benchmark.

Repository	Dataset-TP (#True Vulns.)	Dataset-FP (#False Alarms)
libxml2	38	38
zstd	49	49
PROJ	10	10

forms **dataset-FP**. Both datasets form the foundation for all subsequent experiments.

We use RevBugbench for realistic evaluations, as it preserves full call hierarchies in real-world projects by injecting bugs, in contrast to synthetic benchmarks like Juliet [7]. On the other hand, CVE-based datasets allow evaluating multiple real vulnerabilities but require switching artifacts across project versions. RevBugbench avoids this, enabling consistent slicing and fair evaluation with directed fuzzing all on the same project version.

Configuration. We evaluate SNIPTEST on each examined code location using iterative slicing and fuzzing at *three* slice levels, with time budgets increasing by complexity: 5, 7.5, and 10 minutes for Level 0, 1, and 2, respectively. We limited experiments to Level 2 due to practical scalability constraints, not because detection plateaued. Higher slice levels cause a rapid growth in the number of slices, substantially increasing fuzzing time. Experiments are first run without optimizations (for RQ1–RQ3) to assess classification coverage, then repeated with optimizations (for RQ4–RQ5) to measure time savings, without affecting accuracy. A bare-metal Kubernetes cluster with two AMD EPYC 9224 (48-core, 2.5 GHz) worker nodes, each with 1 TB RAM, is used to execute the experiments. Fuzzing jobs are pinned to a single core and allocated 20 GB RAM, with a maximum of 20 jobs running concurrently.

7.2 RQ1: How effective is SNIPTEST at validating true vulnerabilities across slice levels?

Goal and Approach. RQ1 evaluates whether SNIPTEST can trigger true vulnerabilities across varying slice levels. We evaluate this by applying SNIPTEST to dataset-TP. For each vulnerability location, we generate slices at three levels (Level 0, 1, 2), incrementally expanding the surrounding code context. At each slice level, we classify the outcome as (1) Vulnerable (V): bug is triggered, (2) Benign (B): location reached but no vulnerability triggered, or (3) Not Reachable (NR): location not reached.

Results. Table 2 summarizes the classification results of the examined vulnerable code locations. From the table, we observe that **SNIPTEST triggers the bug oracle consistently across all analyzed levels for 53 of 97 confirmed vulnerabilities (54.6%), without any misclassifications.** That is, SNIPTEST correctly triggers the vulnerability at all three slice levels for 53 vulnerable code locations (54.6%), including 20/38 in libxml2, 29/49 in zstd, and 4/10 in PROJ. In all cases where a vulnerability location is reachable, the bug is also triggered, resulting in no misclassifications.

We also observe that vulnerability reachability declines at higher slice levels due to increased complexity. For example, at Level 2, 33 vulnerable code locations (34%) that were reachable and triggered at Level 0 become Not Reachable. This drop is due to added dependencies in deeper slices, such as function pointers, environment-specific conditions, and file system interactions, which complicate fuzzing and require longer exploration time. For projects such as PROJ, a larger fraction of injected locations becomes dependent on function pointers as the slice expands, causing more Level 2 slices to be classified as NR.

TABLE 2: SNIPTEST classification outcomes for true vulnerabilities across three slice levels. **V** = Bug oracle is triggered at the location in at least one slice at the level, **B** = Location is reached in at least one slice but the bug oracle is not alerted, **NR** = Not Reachable (location unreached across slice level), **PTP** = possible true positive, **PFP** = possible false positive. The locations consistently crashing across all slice levels are only considered as possible true positives.

Project	# Vulns.	Lvl	V	B	NR	PTP	PFP	NR
libxml2	38	0	30	0	8	20	0	18
		1	26	0	12			
		2	20	0	18			
zstd	49	0	47	0	2	29	0	20
		1	38	0	11			
		2	29	0	20			
PROJ	10	0	9	0	1	4	0	6
		1	7	0	3			
		2	4	0	6			
Total	**97**	–	–	–	–	**53 (54.6%)**	**0 (0%)**	**44 (45.3%)**

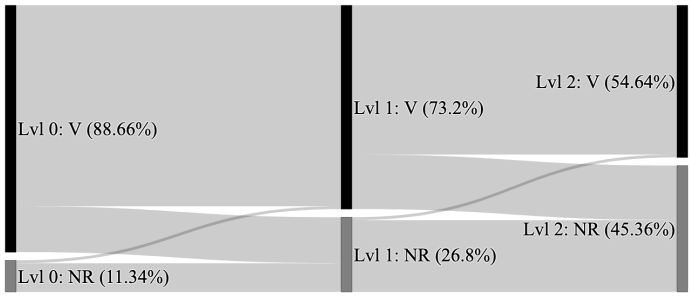


Fig. 3: Sankey diagram showing classification transitions of true vulnerabilities across slice levels on dataset-TP.

Expanding slicing scope might trigger previously unreachable vulnerabilities. Although larger slices can complicate fuzzing, they can also expose vulnerabilities that were unreachable in smaller contexts. In 3.1% of the examined cases, we find that vulnerable code locations that were Not Reachable at lower slice levels became reachable and triggered at higher levels. These cases highlight the importance of analyzing multiple slice levels to uncover vulnerabilities that depend on a broader initialization or environmental setup.

Across all slice levels, no vulnerability is observed to shift from vulnerable to benign behavior. The Sankey diagram in Figure 3 visualizes classification transitions across levels. Among all true vulnerabilities, we find that no bug that was initially classified as vulnerable ever transitions to being classified as benign at higher levels. This observed stability supports the empirical basis of SNIPTEST cross-level aggregation in our benchmark.

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

TABLE 3: SNIPTEST classification outcomes for false alarms across three slice levels. **V** = Bug oracle is triggered at the location in at least one slice at the level, **B** = Location is reached in at least one slice but the bug oracle is not alerted, **NR** = Not Reachable (location unreached across slice level), **PTP** = possible true positive, **PFP** = possible false positive. The locations that are benign at lower levels and unreachable or benign at higher levels are tallied together to obtain possible false positives.

Repository	# Vulns.	Lvl	V	B	NR	PTP	PFP	NR
libxml2	38	0	14	17	7	7	21	10
		1	11	14	13			
		2	7	12	19			
zstd	49	0	30	13	6	20	26	3
		1	24	18	7			
		2	20	26	3			
PROJ	10	0	3	6	1	1	7	2
		1	1	5	4			
		2	1	0	9			
Total	**97**	–	–	–	–	**28 (28.8%)**	**54 (55.6%)**	**15 (15.4%)**

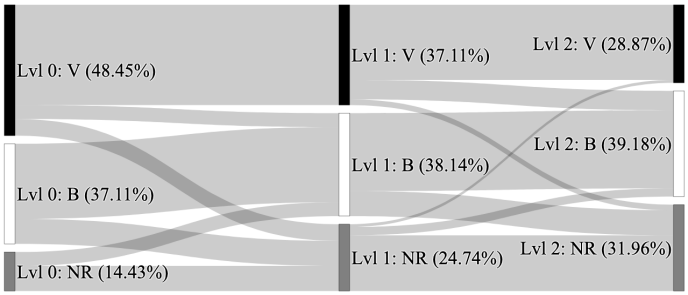


Fig. 4: Sankey diagram showing classification transitions of false-alarm locations across slice levels on dataset-FP.

7.3 RQ2: How effective is SNIPTEST at filtering out false alarms across slice levels?

Goal and Approach. While RQ1 focused on the ability of SNIPTEST to detect true vulnerabilities, RQ2 complements the analysis by measuring its effectiveness in eliminating false alarms, which is crucial for prioritizing bug reports to triage. As before, we analyze how these false alarms behave across increasing slice levels.

Results. As we can see from Table 3, across the 97 false alarms in our dataset-FP, SNIPTEST classifies 54 locations (55.6%) as Possible False Positives (PFP), indicating that only benign execution paths are observed across slice levels 0–2. **SNIPTEST produces Possible False Positive evidence for a considerable portion of confirmed false alarms at minimal slice levels.** We note that 36 out of these 54 cases (37.1% of all locations) are already classified as benign at level 0, without requiring expansion to larger slices. This shows that SNIPTEST can provide early prioritization evidence without expanding every warning to higher levels.

Benign classifications at lower levels remain stable or conservative at higher levels. Figure 4 visualizes how location classifications evolve across slice levels. In our experiments, locations classified as benign at lower levels (B) did not subsequently exhibit vulnerable behavior (V) at higher levels. The observed transitions support using lower-level benign outcomes as heuristic evidence for being a Possible False Positive. We do observe a small number of locations (16) that transition from benign (B) to unreachable (NR) at level 2, mainly due to added environmental complexity in larger slices (e.g., file I/O dependencies). However, conservatively, we classify these cases as PFP because they had previously demonstrated benign behavior when reached in smaller slices. For example in the case of PROJ, we have 7 PFPs which arise from 6 benign locations at Level 0 and an additional benign location at Level 1 (previously vulnerable at Level 0). All of these become unreachable at Level 2 but still are considered as a Possible False Positive due to benign behaviour in smaller slices.

Incomplete slice context can occasionally mislead classification, but higher levels improve classification accuracy. From Table 3, we also observe that 28 suspicious code locations (28.8%) are misclassified as vulnerable (PTP) across all levels, despite belonging to the false alarms dataset-FP. This misclassification is primarily attributed to the Fixreverter framework [60], where injected bugs alter guard conditions outside the slice context. While expanding slices beyond level 2 could potentially resolve these cases, it would introduce significant computational cost. Conversely, we observe that 12 false vulnerabilities initially showing vulnerable behavior at lower levels are correctly reclassified as benign at higher levels (5 at level 1 and 7 at level 2), indicating that moderate slice expansion can recover correct classification.

7.4 RQ3: How accurately does SNIPTEST reproduce the actual execution paths of true vulnerabilities in stack traces?

Goal and Approach. Beyond detecting vulnerabilities or filtering false alarms, the focus of RQ3 is on fidelity–whether SNIPTEST triggers vulnerabilities through realistic calling contexts. This fidelity is crucial for assessing the semantic relevance of the vulnerability trigger. Since many vulnerabilities reside deep in nested call chains, reproducing a vulnerability from a function in the stack trace of an exploit suggests that SNIPTEST follows a meaningful path. Such fidelity not only validates the detection but also helps developers understand the bug's triggering context, potential exploit vectors, and severity. Moreover, SNIPTEST may uncover alternate calling paths not present in the original trace, revealing new avenues for exploitation.

To evaluate this, for each true vulnerability in dataset-TP, we compare the entry function of the slice fuzzed at level k with the k^{th} function in the stack trace generated by executing the test suite on the buggy version of the project. We count a match when SNIPTEST triggers the vulnerability from a slice rooted at a function that appears in the actual bug stack trace.

Listing 1 illustrates the crash stack trace of a heap buffer overflow from the libxml2 project:

```
#0 xmlGetPredefinedEntity  /libxml2/entities.c:496:13
#1 xmlSAX2GetEntity        /libxml2/SAX2.c:872:8
#2 xmlParseEntityRef       /libxml2/parser.c:9740:12
#3 LLVMFuzzerTestOneInput  /libxml2/fuzzer.cc:87:10
```

Listing 1: Bug stacktrace of a heap buffer overflow.

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

TABLE 4: Alignment between ground-truth exploit stack traces and vulnerable paths identified by SNIPTEST. **V** = Bug oracle alerted at the vulnerable location in at least one slice.

Repository	Level	# True Vulns.	V	Aligns
libxml2	0	38	30	30 (78.9%)
	1	38	26	19 (50.0%)
	2	38	20	11 (28.9%)
zstd	0	49	47	47 (95.9%)
	1	49	38	31 (63.2%)
	2	49	29	25 (51.0%)
PROJ	0	10	9	9 (90.0%)
	1	10	7	5 (50.0%)
	2	10	4	3 (30.0%)

Although the vulnerability lies in `xmlGetPredefinedEntity`, it is only reachable via multiple intermediate calls. To test whether SNIPTEST replicates this path, we fuzz slices rooted at higher-level functions. For example, a Level 1 slice rooted at `xmlSAX2GetEntity` may trigger the bug and thus replicate the second frame in the trace. Similarly, a Level 2 slice rooted at `xmlParseEntityRef` may reproduce the bug from the third stack frame. These cases indicate that SNIPTEST can successfully trace backward and isolate vulnerable paths that align with real crash behavior by only fuzzing partial code slices.

To obtain the ground truth, we use the project's existing test suite to execute the injected vulnerability, then collect the resulting stack trace using dynamic sanitizers such as ASAN [52] and UBSAN [16]. We then compare this against the stack trace reported by SNIPTEST when it triggers a vulnerability. We specifically compare the top three frames to determine if the slice entry function corresponds to one of the real call frames. We also ensure that the type of vulnerability reported (e.g., heap overflow, use-after-free) is consistent between both traces.

Results. Table 4 presents the evaluation results for RQ3. For each project, the table reports: (1) the number of true vulnerabilities with available ground-truth stack traces, (2) the number of cases where location exhibits vulnerable behavior (V) at given slice level, and (3) the number of cases where the slice entry function aligns with a function in the ground-truth stack trace (Aligns). Below, we present the main observations.

Tight alignment at shallow slices. From Table 4, we observe that at Level 0, SNIPTEST replicates the ground-truth stack trace in 88.6% of cases across all projects (on average). Alignment is particularly strong for zstd (95.9%) and PROJ (90%). This confirms that small, minimal slices often preserve enough execution context to detect true vulnerabilities.

Alignment decreases as slice levels increase. We also observe that the average alignment drops from 88.6% at Level 0 to 56.7% at Level 1 and 40.2% at Level 2, due to increased slice complexity, e.g., added control flow, environmental dependencies, and stricter input constraints, making it harder to reproduce ground-truth stack traces during fuzzing. Additionally, since SNIPTEST relies on a static call graph, it may miss dynamic edges (e.g., indirect calls), leading to deviations from actual call paths. Still, SNIPTEST often triggers vulnerabilities through alternative paths, maintaining detection effectiveness despite imperfect alignment.

TABLE 5: Effect of optimizations (O1-O5) on analysis-time across Levels 0-2 in SNIPTEST. Times are reported in days.

Project	Dataset	No Optimizations		With Optimizations	
		Slice	Fuzz	Slice	Fuzz
libxml2	True Vulns.	0.650	23.370	0.032	0.644
	False Alarms	0.650	23.370	0.032	0.535
zstd	True Vulns.	0.021	5.598	0.003	0.284
	False Alarms	0.021	5.598	0.003	0.223
PROJ	True Vulns.	0.096	0.489	0.019	0.179
	False Alarms	0.096	0.489	0.019	0.152
Total	–	**1.534**	**58.914**	**0.108**	**2.017**

TABLE 6: Effect of optimizations (O3-O4) on number of slices fuzzed by SNIPTEST. **%** = % of slices skipped for fuzzing due to optimizations. *Level 0 results are omitted as they are unaffected by fuzzing optimizations (O3-O4).*

Project	Dataset	Level 1			Level 2		
		Before	After	%	Before	After	%
libxml2	True Vulns.	694	208	70.0	1736	837	51.7
	False Alarms	694	71	89.7	1736	182	89.5
zstd	True Vulns.	102	66	35.2	162	86	46.9
	False Alarms	102	42	58.8	162	50	69.1
PROJ	True Vulns.	25	18	28.0	26	24	7.6
	False Alarms	25	13	48.0	26	9	65.3

7.5 RQ4: How efficient is SNIPTEST with its slice-level optimizations?

Goal and Approach. This RQ examines how the optimizations described in Section 6 impact the overall analysis time of SNIPTEST, aiming to improve its scalability without sacrificing classification accuracy. Unlike previous RQs, where SNIPTEST was run without optimizations to measure baseline effectiveness, this RQ focuses on practical efficiency.

We re-run SNIPTEST on the vulnerability dataset-TP and dataset-FP while enabling all optimizations (O1–O6), including slice caching, fuzzing termination heuristics, and parallelization across 20 CPU cores. We separately measure (1) slice creation time, (2) fuzzing time, and (3) the number of slices analyzed, and compare them against the unoptimized single core baseline of SNIPTEST.

Results. Table 5 discusses the effect of optimizations on SNIPTEST analysis time. **SNIPTEST slice optimizations (O1-O2) reduce slice creation time by 92.9% on average across projects. Additionally, SNIPTEST fuzzing optimizations accelerated by parallelization (O3-O5) reduce fuzzing time by 96.5% across projects.**

Pruning code unnecessary for vulnerability execution (O6) improves reachability without directly affecting runtime. Though it doesn't reduce slicing or fuzzing time, it increases the likelihood of reaching vulnerabilities during fuzzing, yielding a 35% increase in vulnerabilities reached across slice levels in our experiments.

Optimizations reduce the number of slices fuzzed by 70.7% on average across projects. Table 6 shows that

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

optimizations (O3-O4) reduce the number of slices fuzzed by 70.7% overall. Specifically, slice analysis is optimized by 54.8% on dataset-TP and achieves an 86.6% reduction on dataset-FP. This trend confirms that skipping higher-level slice analysis (O4) is more effective on false alarms which are not triggered at shallow levels.

7.6 RQ5: How does SNIPTEST compare with directed fuzzers in efficiency and target reachability?

Goal and Approach. This RQ compares SNIPTEST against distance-based directed fuzzer AFLGo in terms of (1) code reachability and (2) analysis time. AFLGo [24], operates on the full program using coverage-guided mutations to reach targets, while SNIPTEST tests isolated slices around the targets only. We compare SNIPTEST with AFLGo, a commonly adopted baseline in comparative studies [27], [35], [43], to improve comparability with prior results.

We compare SNIPTEST with AFLGo configured in two modes: unseeded (UDF) and seeded (SDF) fuzzing. The seeded mode uses curated test cases from the project's test suite, excluding inputs that immediately crash or hit targets. The test cases in these projects are extensive, reflecting substantial manual effort to test program features, which provides an advantage to the fuzzer. SNIPTEST instead performs fuzzing without seeds across all slices. All fuzzers are given a 7-day budget and 20 cores, matching the parallel setup of SNIPTEST.

For SNIPTEST, we enable all optimizations and record the total time to analyze all slices. For AFLGo, we report the time to trigger the final unique vulnerability; if no vulnerabilities are triggered, the full 7 day budget is recorded. This setup ensures a fair comparison of how quickly each approach produces results.

Results. **SNIPTEST validates more code locations than unseeded directed fuzzers.** As shown in Table 7, SNIPTEST validates 107 target locations, outperforming the unseeded directed fuzzer (40) and closely matching the seeded fuzzer (112). On large and input-constrained projects such as `zstd`, the unseeded fuzzer fails to reach any true or false vulnerability locations, whereas SNIPTEST validates 29 true vulnerabilities and 26 false alarms respectively, which demonstrates that SNIPTEST's slice-based strategy is less affected by complex program entry points.

SNIPTEST is substantially faster in efficiency than both seeded and unseeded directed fuzzers. SNIPTEST completes its entire analysis in 2.09 days across all datasets, compared to 22.23 days for UDF and 11.68 days for SDF, i.e, SNIPTEST achieves a speedup of $10.6\times$ over the unseeded directed fuzzer and $5.5\times$ over the seeded directed fuzzer. This highlights the efficiency of SNIPTEST's parallelized and targeted slice-based analysis.

SNIPTEST uniquely validates vulnerabilities that directed fuzzers miss. Table 8 shows that SNIPTEST uniquely validates 86 cases missed by the unseeded fuzzer and 51 missed by the seeded fuzzer, thanks to its ability to analyze localized code without complex runtime setup. While seeded fuzzer uniquely validates 56 cases, these often depend on global state (e.g., file I/O or environment variables), which is harder to simulate in isolated slices.

TABLE 7: Comparison of SNIPTEST (ST), unseeded directed fuzzing (UDF), and seeded directed fuzzing (SDF). **Val.** = Validated cases; **TT** = total time (days).

Project	Dataset	# Cases	SNIPTEST (ST)		UDF		SDF	
			Val.	TT	Val.	TT	Val.	TT
libxml2	True Vulns.	38	20	0.67	18	2.50	24	2.16
	False Alarms	38	21	0.56	20	2.63	24	2.22
zstd	True Vulns.	49	29	0.28	0	7.00	33	2.20
	False Alarms	49	26	0.22	0	7.00	28	2.63
PROJ	True Vulns.	10	4	0.19	1	1.43	1	1.19
	False Alarms	10	7	0.17	1	1.67	2	1.28
Total	–	**194**	**107**	**2.09**	**40**	**22.23**	**112**	**11.68**

TABLE 8: Overlap of validated cases between SNIPTEST (ST), the seeded directed fuzzer (SDF) and unseeded directed fuzzer (UDF).

Project	Dataset	# Cases	SNIPTEST vs. UDF			SNIPTEST vs. SDF		
			Both ST ∩ UDF	Only ST	Only UDF	Both ST ∩ SDF	Only ST	Only SDF
libxml2	True Vulns.	38	10	10	8	13	7	11
	False Alarms	38	11	10	9	11	10	13
zstd	True Vulns.	49	0	29	0	20	9	13
	False Alarms	49	0	26	0	12	14	16
PROJ	True Vulns.	10	0	4	1	0	4	1
	False Alarms	10	0	7	1	0	7	2
Total	–	**194**	**21**	**86**	**19**	**56**	**51**	**56**

7.7 RQ6: How does SNIPTEST compare with LLM based approaches for classifying vulnerabilities?

Goal and Approach. This RQ compares two different forms of warning triage, SNIPTEST and LLM4SA [55]. SNIPTEST represents the execution-based setting: it generates sliced binaries, fuzzes them, and uses sanitizer feedback to observe whether a warning is triggered, reached without failure, or not reached. LLM4SA represents the prediction-based setting: it reasons statically over code context from learned buggy patterns and predicts whether a warning is likely true or false without executing the program. Therefore, this comparison should not be interpreted as a direct tool-equivalence comparison, but as a study of the trade-offs between runtime evidence and static prediction for warning classification.

For SNIPTEST we enable all optimizations and compare results on dataset-TP and dataset-FP. We share the project wise classification results in Table 9. The results demonstrate that SNIPTEST outperforms LLM4SA across all accuracy metrics for each project. Furthermore, we show the confusion matrices for SNIPTEST and LLM4SA in Table 10. The confusion matrix indicates that LLM4SA is biased toward classifying warnings as false alarms (predicted 51 of cases in dataset-TP as false alarm and 47 of cases in dataset-FP as false alarm), which explains its comparatively better performance on the dataset-FP than on the dataset-TP. Overall, SNIPTEST outperforms LLM4SA, with an F1 score of 0.791 versus 0.360.

Moreover, from Table 9 we found that LLM4SA does not exhibit a substantial distinguishing power between the same code locations in dataset-TP and dataset-FP. This

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

TABLE 9: Comparison of SNIPTEST and LLM4SA across projects.

Project	Dataset	Total	SnipTest		LLM4SA	
			TP	FP	TP	FP
libxml2	True Vulns.	38	20	0	12	15
	False Alarms	38	7	21	14	14
zstd	True Vulns.	49	29	0	6	32
	False Alarms	49	20	26	8	30
PROJ	True Vulns.	10	4	0	4	4
	False Alarms	10	1	7	5	3

TABLE 10: Confusion matrices for SNIPTEST and LLM4SA

	SnipTest		LLM4SA	
	Pred True	**Pred False**	**Pred True**	**Pred False**
Real True	53	0	22	51
Real False	28	54	27	47

behavior arises from the construction of the datasets using the RevBugBench framework [60], in which different guard conditions are manipulated: stronger guards yield the dataset-FP (false alarms), whereas weaker guards result in the dataset-TP (true bugs). LLM4SA reasons about static analysis warnings by extracting a small code snippet from the vulnerable function. However, the guard that determines whether the vulnerability is exercised may reside in a transitive caller, preventing the LLM from reliably distinguishing true bugs from false alarms. In contrast, SNIPTEST incrementally expands code slices (up to three levels in our current work) based on reachability and vulnerability outcomes, thereby incorporating relevant guard conditions into the analysis.

These results should be interpreted in light of the differences in task formulation. SNIPTEST benefits from concrete runtime evidence produced by fuzzing and sanitizers, while LLM4SA is limited to the static context provided to the model. Conversely, LLM4SA does not require slice construction, compilation, or fuzzing infrastructure. Thus, our results do not imply that execution-based triage universally replaces prediction-based approaches. Rather, they show that, in our benchmark, sliced execution provides stronger evidence for distinguishing true vulnerabilities from false alarms, particularly when the decisive guard conditions are located outside the local code context available to the LLM.

7.8 RQ7: How does SNIPTEST compare with slice based approach FUZZSLICE?

Goal and Approach. This RQ compares SNIPTEST with the slice-based approach FUZZSLICE. SNIPTEST builds upon FUZZSLICE as its low-level slice-compilation component, but the two systems operate at fundamentally different units of reasoning: FUZZSLICE analyzes a single Level-0 slice, whereas SNIPTEST evaluates a family of caller-rooted slices organized by call-graph distance. In SNIPTEST, slices are fuzzed independently, observations are aggregated within each level, and the sequence of level outcomes is then synthesized into warning-level evidence. We compare FUZZSLICE's single-slice output against SNIPTEST's full multi-level framework, reflecting their respective units of analysis.

TABLE 11: Comparison of SNIPTEST and FUZZSLICE across slice levels. **V** = Bug oracle is triggered at the location in at least one slice at the level, **B** = Location is reached in at least one slice but the bug oracle is not alerted, **NR** = Location is not reached in any slice at the level.

Project	Dataset	# Cases	Slice	SnipTest			FuzzSlice		
				V	B	NR	V	B	NR
libxml2	True Vulns.	38	0	30	0	8	20	0	18
			1	26	0	12	-	-	-
			2	20	0	18	-	-	-
	False Alarms	38	0	14	17	7	13	16	9
			1	11	14	13	-	-	-
			2	7	12	19	-	-	-
zstd	True Vulns.	49	0	47	0	2	34	0	15
			1	38	0	11	-	-	-
			2	29	0	20	-	-	-
	False Alarms	49	0	30	13	6	30	9	10
			1	24	18	7	-	-	-
			2	20	26	3	-	-	-
PROJ	True Vulns.	10	0	9	0	1	6	0	4
			1	7	0	3	-	-	-
			2	4	0	6	-	-	-
	False Alarms	10	0	3	6	1	3	6	1
			1	1	5	4	-	-	-
			2	1	0	9	-	-	-

We present the project-wise classification results in Table 11. As shown in the table, FUZZSLICE is evaluated only at the Level 0 setting, since it cannot be directly configured to generate higher-level slices. Both approaches exhibit comparable performance on PROJ. However, on projects such as libxml2 and zstd, SNIPTEST achieves higher precision across both datasets. On the dataset of reached bugs, SNIPTEST achieves a precision, recall, and F1 score of 0.654, 1.0, and 0.791, respectively, by leveraging all three slice levels for warning classification. In comparison, FUZZSLICE, using only the Level 0 slice for classification, achieves corresponding scores of 0.566, 1.0, and 0.722 on its reached bugs. Both techniques have high recall as true vulnerabilities in dataset-TP are easily exploited by the fuzzer at the Level 0 slice when reached. SNIPTEST achieves improvement in F1 score over FUZZSLICE, primarily due to the inclusion of additional code context across slice levels, which enables more accurate classification of benign locations.

Finally, at Level 0 alone, SNIPTEST reaches and classifies 169 out of 194 locations, whereas FUZZSLICE reaches and classifies 137. This corresponds to SNIPTEST reaching and classifying 32 additional locations, or 23.3% more locations than FUZZSLICE at Level 0 indicating improved wrapper generation for SNIPTEST as described in Approach Section 5.

8 SNIPTEST ON OPEN SOURCE PROJECTS

In the previous section, we evaluated SNIPTEST on a synthetic dataset from RevBugbench framework [60]. In this section, we demonstrate the practical application of SNIPTEST by deploying our cloud-based SNIPTEST setup on two real-world open-source projects: Vim [12] and libpcap [9]. Vim is a widely used text editor, and libpcap is a common Linux utility for network packet capture.

Using the static analysis tool Flawfinder [4], we generated 2,328 warnings for Vim and 1,284 for libpcap. From

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

these, we selected warnings corresponding to high-impact vulnerabilities, such as CWE-120 (buffer overflow) and CWE-476 (null pointer dereference), which were not covered by native tests. This process yielded 24 warnings in Vim and 9 in libpcap, which we then attempted to validate using SNIPTEST.

We fuzzed the associated multi-level code slices (levels 0–2) containing the sampled warnings. Out of the 33 candidate warnings, only four produced consistent crashes across all slice levels. For these, we submitted issues and pull requests to the affected projects. Within a week, this effort led to the confirmation of three new vulnerabilities across both projects: two buffer overruns and one null pointer dereference [3], [10], [11]. The fourth case, an out-of-bounds write in Vim, was disputed by developers, who argued in an email thread that the behavior was intentional and not a bug. We discuss each identified vulnerability and its fix below:

1) **Buffer overflow in libpcap [3].** Libpcap is a portable API for low-level network monitoring and traffic analysis across Unix-like systems and Windows (via WinPcap). A heap buffer overflow was identified in the function utf_16le_to_utf_8_truncated, which is responsible for converting Windows error messages from UTF-16 to UTF-8. During this conversion, the function incorrectly counts the number of characters written when determining how much data to truncate, resulting in a heap buffer overwritten at the end of the destination buffer. This vulnerability was assigned CVE-2025-11964 [13]. The developer's fix corrects the character counting logic used during conversion. The developer released a new version of the software patching the vulnerability and appreciating the authors for spotting the issue in the release notes [13].
2) **Buffer overflow in Vim [11].** Vim is a highly configurable text editor designed for efficient text creation and modification. A buffer overflow was discovered in Vim within the function cs_pathcomponents, which truncates file paths based on a user-supplied integer value (e.g., displaying only the last N directories of a path). The truncation logic contains an error in string pointer manipulation—specifically, a double decrement—leading to a buffer underflow. The issue was fixed by correcting the pointer logic to perform a single decrement instead of two.
3) **Null dereference in Vim [10].** A null pointer dereference was identified in Vim in the function cs_find_common, which is executed when the find command is invoked within cscope. During execution, the function attempts to create a temporary file. If this operation fails (e.g., due to insufficient disk space, filename conflicts, or inadequate permissions), the resulting null pointer may still be accessed later in the function, causing a crash. The fix applied for the bug ensures that the function exits early if temporary file creation fails, preventing dereference of a null pointer.

The remaining warnings either did not produce crashes at all slice levels or were not reachable (categorized as PFP or NR). While our analysis does not constitute a full-scale evaluation of several static analysis warnings, we intended to demonstrate the practical utility of SNIPTEST in identifying real vulnerabilities in widely-used open-source software.

9 Limitations and Future Work

We now discuss an exhaustive list of factors in the SNIPTEST framework that can create false positives and false negatives. We also discuss potential mitigation strategies.

1) **Incomplete minimal context for bugs (false positives only)**. A reported bug may be classified as a true vulnerability or a false alarm based on a guard condition that resides several functions earlier in the call hierarchy. In our current implementation, SNIPTEST limits analysis to at most three slice levels. Extending the analysis to additional slice levels could incorporate such transitive guard conditions and help avoid misclassifying false alarms as true vulnerabilities.
2) **Insufficient fuzzing time relative to slice size (false negatives or unreached code)**. In some cases, the allocated fuzzing time is insufficient to adequately explore larger code slices. In our evaluation, the time budget is 5/7.5/10 minutes for Levels 0, 1 and 2. These limits are empirically chosen and may be inadequate for certain complex slices. Moreover, large test cases (e.g., functions with many or large arguments) and inefficient mutation strategies can further hinder exploration. Such limitations may result in unreached code or false negatives - where reachable bugs are not exploited within the allotted time. For example, Table 12 shows 5 repetitions of SNIPTEST on our dataset. We observed 2 cases in libxml2 (dataset-TP) crashing at all slice levels on some iterations and not being reached in others. However in dataset-FP, the same cases can be reached at higher slice levels, classifying them as benign.
3) **Function pointers, file I/O, and runtime environment variables (false positives, false negatives or unreached code).** SNIPTEST currently assigns function pointers to NULL and does not randomly bind them to functions within the slice. In addition, it does not model file-based I/O or runtime environment variables that influence program behavior. These limitations can lead to both false positives, false negatives or unreached code. Addressing these aspects is left for future work.

Finally, we note that the cross-level classification rule in SNIPTEST is heuristic. Although it is empirically motivated by the observed stability of warning behavior across slice levels, it does not provide formal soundness or completeness guarantees. A warning may still be misclassified if the generated slices omit relevant program context, if required environmental state is not modeled, or if the fuzzer fails to explore the necessary inputs within the time budget. We therefore interpret output of SNIPTEST as evidence for prioritization rather than definitive validation.

10 Conclusion

We presented SNIPTEST, an execution-based warning-triage framework built atop the FUZZSLICE function-slice compilation primitive. SNIPTEST organizes slices into progressively broader context levels and aggregates warning behavior

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

TABLE 12: SNIPTEST results across projects over repeated runs

#	All TP	All PFP	All NR
#1 (True Vulns.)	53	0	44
#2 (True Vulns.)	53	0	44
#3 (True Vulns.)	51	0	46
#4 (True Vulns.)	53	0	44
#5 (True Vulns.)	52	0	45
#1 (False alarms)	28	54	15
#2 (False alarms)	28	54	15
#3 (False alarms)	28	54	15
#4 (False alarms)	28	54	15
#5 (False alarms)	28	54	15

both within and across these levels. This methodology yields evidence about whether a warning persists as caller-imposed constraints are gradually restored. In our benchmark, SNIPTEST produced Possible True Positive evidence for 53 of 97 confirmed vulnerabilities and Possible False Positive evidence for 54 of 97 confirmed false alarms when analyzing across three slice levels. It also generated Possible True Positive evidence for 28 false alarms, underscoring that insufficient context remains an important limitation. SNIPTEST should therefore be viewed as a prioritization aid that complements developer review. Our evaluation further demonstrates that sliced execution can substantially reduce fuzzing time and expose previously unknown defects, as evidenced by our disclosure of CVE-2025-11964. These results support multi-level sliced fuzzing as a practical approach for gathering warning-triage evidence—rather than as definitive proof of vulnerability or benignness.

REFERENCES

[1] "american fuzzy lop," Mar. 2023, [Online; accessed 10. Mar. 2023]. [Online]. Available: https://lcamtuf.coredump.cx/afl

[2] "libFuzzer – a library for coverage-guided fuzz testing. — LLVM 17.0.0git documentation," Jan. 2023, [Online; accessed 29. Jan. 2023]. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[3] "Fix a copy-and-pasteo in utf_16le_to_utf_8_truncated(). · the-tcpdump-group/libpcap@aebfca1," Sep. 2025, [Online; accessed 8. Sep. 2025]. [Online]. Available: https://github.com/the-tcpdump-group/libpcap/commit/aebfca1aea2fc8c177760a26e8f4de27b51d1b3b

[4] "Flawfinder Home Page," Sep. 2025, [Online; accessed 9. Sep. 2025]. [Online]. Available: https://dwheeler.com/flawfinder

[5] "Infer Static Analyzer | Infer | Infer," Apr. 2025, [Online; accessed 27. May 2025]. [Online]. Available: https://fbinfer.com

[6] "Introduction - Tree-sitter," Apr. 2025, [Online; accessed 27. May 2025]. [Online]. Available: https://tree-sitter.github.io/tree-sitter

[7] "Juliet C/C++ 1.3 - NIST Software Assurance Reference Dataset," May 2025, [Online; accessed 27. May 2025]. [Online]. Available: https://samate.nist.gov/SARD/test-suites/112

[8] "LeakSanitizer — Clang 21.0.0git documentation," Jan. 2025, [Online; accessed 27. May 2025]. [Online]. Available: https://clang.llvm.org/docs/LeakSanitizer.html

[9] "libpcap," Sep. 2025, [Online; accessed 9. Sep. 2025]. [Online]. Available: https://github.com/the-tcpdump-group/libpcap

[10] "Null Pointer dereference in cs_find_common · Issue #18225 · vim/vim," Sep. 2025, [Online; accessed 8. Sep. 2025]. [Online]. Available: https://github.com/vim/vim/issues/18225

[11] "patch 9.1.1680: MS-Windows: possible buffer-under run in if_cscope · vim/vim@191d778," Sep. 2025, [Online; accessed 8. Sep. 2025]. [Online]. Available: https://github.com/vim/vim/commit/191d77872d8938fce28abaec061d4d6cde88cf48

[12] "vim," Sep. 2025, [Online; accessed 9. Sep. 2025]. [Online]. Available: https://github.com/vim/vim

[13] "Home | TCPDUMP & LIBPCAP," Jan. 2026, [Online; accessed 11. Jan. 2026]. [Online]. Available: https://www.tcpdump.org/index.html#latest-releases

[14] "LeakSanitizer — Clang 23.0.0git documentation," Jan. 2026, [Online; accessed 13. Jan. 2026]. [Online]. Available: https://clang.llvm.org/docs/LeakSanitizer.html

[15] "MemorySanitizer — Clang 23.0.0git documentation," Jan. 2026, [Online; accessed 13. Jan. 2026]. [Online]. Available: https://clang.llvm.org/docs/MemorySanitizer.html

[16] "UndefinedBehaviorSanitizer — Clang 23.0.0git documentation," Jan. 2026, [Online; accessed 13. Jan. 2026]. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[17] B. Aloraini and M. Nagappan, "Evaluating State-of-the-Art Free and Open Source Static Analysis Tools Against Buffer Errors in Android Apps," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sep. 2017, pp. 295–306.

[18] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, Aug. 2008.

[19] D. Binkley and M. Harman, "A large-scale empirical study of forward and backward static slice size and context sensitivity," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, pp. 22–26.

[20] ——, "Forward slices are smaller than backward slices," in *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*. IEEE, pp. 2005–01.

[21] D. W. Binkley and K. B. Gallagher, "Program slicing," *Advances in computers*, vol. 43, pp. 1–50, 1996.

[22] bminor, "binutils-gdb," May 2024, [Memory leak in binutils]. [Online]. Available: https://github.com/bminor/binutils-gdb/commit/7ed1acafa0b5d135342f9dcc0eb0387dff95005a

[23] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2020.

[24] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2329–2344.

[25] M. Böhme and B. Falk, "Fuzzing: on the exponential cost of vulnerability discovery," in *ESEC/FSE 2020: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 713–724.

[26] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 2329–2344.

[27] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a Desired Directed Grey-box Fuzzer," in *CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 2095–2108.

[28] L. Chen, Q. Cai, Z. Ma, Y. Wang, H. Hu, M. Shen, Y. Liu, S. Guo, H. Duan, K. Jiang, and Z. Xue, "SFuzz: Slice-based Fuzzing for Real-Time Operating Systems," in *CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 485–498.

[29] C. Csallner and Y. Smaragdakis, "Check'n'crash: combining static checking and testing," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 422–431.

[30] J. Eyolfson, L. Tan, and P. Lam, "Correlations between bugginess and time-based commit characteristics," *Empirical Software Engineering*, vol. 19, pp. 1009–1039, 2014.

[31] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[32] U. Garg, G. Sikka, and L. K. Awasthi, "Empirical analysis of attack graphs for mitigating critical paths and vulnerabilities," *Computers & Security*, vol. 77, pp. 349–359, 2018.

[33] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: improving actionable alert ranking," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 152–161.

[34] M. Harman and S. Danicic, "Using program slicing to simplify testing," *Software Testing, Verification and Reliability*, vol. 5, no. 3, pp. 143–162, 1995.

[35] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2022, pp. 36–50.

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3725696

[36] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "Path-sensitive backward slicing," in *Static Analysis: 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings 19*. Springer, 2012, pp. 231–247.

[37] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding Kernel Race Bugs through Fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019, pp. 754–768.

[38] A. Kallingal Joshy, X. Chen, B. Steenhoek, and W. Le, "Validating static warnings via testing code fragments," in *ISSTA 2021: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, Jul. 2021, pp. 540–552.

[39] ——, "Validating static warnings via testing code fragments," in *ISSTA 2021: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, Jul. 2021, pp. 540–552.

[40] H. J. Kang, K. L. Aw, and D. Lo, "Detecting false alarms from automatic static analysis tools: how far are we?" in *ICSE '22: Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 698–709.

[41] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "{DAFL}: Directed grey-box fuzzing guided by data dependency," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4931–4948.

[42] B. Korel and J. Laski, "Dynamic program slicing," *Information processing letters*, vol. 29, no. 3, pp. 155–163, 1988.

[43] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3559–3576. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu

[44] Y. Lei, X. Mao, and T. Y. Chen, "Backward-slice-based statistical fault localization without test oracles," in *2013 13th International Conference on Quality Software*. IEEE, 2013, pp. 212–221.

[45] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[46] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecur.*, vol. 1, no. 1, pp. 1–13, Dec. 2018.

[47] P. Louridas, "Static code analysis," *Ieee Software*, vol. 23, no. 4, pp. 58–61, 2006.

[48] A. Murali, N. Mathews, M. Alfadel, M. Nagappan, and M. Xu, "Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[49] M. Nadeem, B. J. Williams, and E. B. Allen, "High false positive detection of security vulnerabilities: a case study," in *ACM-SE '12: Proceedings of the 50th Annual Southeast Regional Conference*. New York, NY, USA: Association for Computing Machinery, Mar. 2012, pp. 359–360.

[50] J. Park, I. Lim, and S. Ryu, "Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, May 2016, pp. 61–70. [Online]. Available: https://ieeexplore.ieee.org/document/7883289

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

[52] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," 2012.

[53] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation*, 2007, pp. 112–122.

[54] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, pp. 1419–1457, 2020.

[55] C. Wen, Y. Cai, B. Zhang, J. Su, Z. Xu, D. Liu, S. Qin, Z. Ming, and T. Cong, "Automatically inspecting thousands of static bug warnings with large language model: How far are we?" *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 7, pp. 1–34, 2024.

[56] ——, "Automatically inspecting thousands of static bug warnings with large language model: How far are we?" *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 7, pp. 1–34, 2024.

[57] V. Wüstholz and M. Christakis, "Targeted Greybox Fuzzing with Static Lookahead Analysis," *arXiv*, May 2019.

[58] X. Yang, J. Chen, R. Yedida, Z. Yu, and T. Menzies, "Learning to recognize actionable static code warnings (is intrinsically easy)," *Empirical Software Engineering*, vol. 26, no. 3, p. 56, 2021.

[59] R. Yedida, H. J. Kang, H. Tu, X. Yang, D. Lo, and T. Menzies, "How to find actionable static analysis warnings: A case study with findbugs," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2856–2872, 2023.

[60] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, "{FIXREVERTER}: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing," pp. 3699–3715, 2022, [Online; accessed 7. Jan. 2025]. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-zenong

[61] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE transactions on software engineering*, vol. 32, no. 4, pp. 240–253, 2006.



Aniruddhan Murali is a Ph.D. graduate from the Cheriton School of Computer Science at the University of Waterloo, Canada. His research interests include fuzzing, code slicing, vulnerability detection, and automatic bug fixing. You can find more about him here.



Noble Saji Mathews is a Research Associate in the David R. Cheriton School of Computer Science at the University of Waterloo. His research interests include artificial intelligence and software engineering. Noble received his Masters in Computer Science from the University of Waterloo. You can find more about him here.



Mahmoud Alfadel is an Assistant Professor with the Department of Computer Science, University of Calgary. His research interests include mining software repositories, software ecosystems, open-source security, and release engineering.



Meiyappan Nagappan is an Associate Professor at the Cheriton School of Computer Science, University of Waterloo. He has worked on empirical software engineering to address software development concerns and currently researches the impact of large language models on software development.



Meng Xu is an Assistant Professor in the Cheriton School of Computer Science at the University of Waterloo, Canada. His research is in the area of system and software security, with a focus on delivering high-quality solutions to practical security programs, especially in finding and patching vulnerabilities in critical computer systems. This usually includes research and development of automated program analysis / testing / verification tools that facilitate the security reasoning of critical programs.

Authorized licensed use limited to: University of Calgary. Downloaded on October 23,2026 at 22:13:48 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.