Towards Correct and Efficient Persistent Fuzzing: Reducing Instrumentation Overhead and Managing Program State

Sydney K. Earp

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Applications

Matthew D. Hicks, Chair
Daphne Yao
Tijay Chung

May 9, 2025
Blacksburg, Virginia

Keywords: Fuzzing, Software Security, Program Instrumentation

Towards Correct and Efficient Persistent Fuzzing: Reducing Instrumentation Overhead and Managing Program State

Sydney K. Earp

(ABSTRACT)

Coverage-guided fuzzing is a powerful technique for automatically discovering software bugs and security vulnerabilities. This work improves the efficiency and correctness of coverage-guided fuzzers by introducing dynamic removal of redundant coverage instrumentation and implementing a state-managed version of persistent mode. Traditional coverage instrumentation remains active even after code regions have been thoroughly explored, resulting in unnecessary runtime overhead. Meanwhile, persistent fuzzing—used to accelerate input execution by looping the target program in memory—can suffer from state contamination between test cases. To address these challenges, this system dynamically disables coverage tracking for already-covered paths and uses compiler-inserted instrumentation to restore global variables, heap allocations, and file state at the start of each iteration. The result is a fast, self-contained fuzzing framework that improves execution throughput and test isolation, without requiring OS-level support or recompilation.

Towards Correct and Efficient Persistent Fuzzing: Reducing Instrumentation Overhead and Managing Program State

Sydney K. Earp

(GENERAL AUDIENCE ABSTRACT)

Modern software systems are complex, and finding bugs or vulnerabilities in them is difficult and time-consuming. One popular method to automate this process is called fuzzing, where a program is repeatedly tested using random or unexpected inputs to see if it crashes or behaves incorrectly. This research focuses on a particular kind of fuzzing that uses feedback to explore new parts of the program. However, this approach can be inefficient—it often checks the same parts of the program repeatedly and may reuse parts of memory that still contain leftovers from previous tests, which can cause inaccurate results. This work improves the speed and reliability of fuzzing by removing unnecessary checks and ensuring that the program starts fresh for every test. These improvements make fuzzing faster and more trustworthy for finding bugs in real software.

Dedication

To my cat Moonlight, my lifelong study buddy.

Acknowledgments

I would like to thank my advisor, Dr. Matthew Hicks, for his guidance and support, and my friends and labmates in the FoRTE Research Group for their help and camaraderie. Thank you to my parents, whose support and example have been the secret sauce to my journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As software becomes increasingly central to the functioning of the modern world, ensuring its security and reliability has emerged as one of the defining challenges in computing. Software systems are comprised of numerous interconnected components, many of which can harbor exploitable vulnerabilities if left unchecked. Traditional testing methodologies, while still valuable, often fall short of effectively probing the vast and complex execution spaces of present-day software. This limitation has driven the adoption of more dynamic testing approaches, among which *fuzz testing*, or *fuzzing*, stands out as a particularly effective method for uncovering latent vulnerabilities [14, 33].

Fuzzing introduces anomalous, unexpected, or malformed data into systems to elicit unanticipated behaviors. Such behaviors can include crashes (e.g., segmentation faults), hangs (e.g., infinite loops), or semantic anomalies (e.g., incorrect calculations), all of which may indicate underlying bugs or security flaws [30, 32]. Its scalability, automation, and effectiveness have made fuzzing a standard practice in both academic research and industry [1]. Among the various fuzzing paradigms, *coverage-guided fuzzing (CGF)* has proven to be one of the most impactful due to its intelligent approach to test case generation [7, 19]. By leveraging real-time feedback on code coverage, CGF adapts the fuzzing processes to prioritize paths that have not yet been tested, maximizing the efficacy of the testing session.

Despite its advantages, CGF faces two persistent limitations. First, while coverage feedback is essential for guiding the fuzzer, the program instrumentation that monitors coverage re-

mains active throughout the testing process, even after certain areas of the codebase have been thoroughly tested. This redundancy not only consumes unnecessary computational resources but can also slow down the testing process significantly, particularly when instrumentation occurs on hot paths that are exercised repeatedly [15, 16, 17]. Second, to maximize execution throughput, modern fuzzers often employ persistent fuzzing. This execution mode accelerates the fuzzing process by looping the target program in memory to avoid process reinitialization. However, this can lead to *state leakage*—a scenario where remnants from one test case inadvertently influence the outcome of subsequent ones, skewing results and potentially masking vulnerabilities. Despite this shortcoming, persistent mode is frequently used for evaluation of program behavior and benchmark comparisons of fuzzer performance [7, 22, 24].

This thesis addresses these two core inefficiencies through a unified framework that combines *dynamic coverage removal* with *correct and efficient persistent-mode execution*. First, the system identifies and disables coverage instrumentation at runtime once program edges have been fully explored, reducing overhead without requiring recompilation. Second, it ensures reliable execution in persistent mode by restoring global state, heap memory, and file handles between test cases—preserving correctness while avoiding expensive process creation.

To the best of our knowledge, this is the first work to integrate both accurate state restoration for persistent fuzzing and dynamic removal of redundant coverage instrumentation. By addressing both performance and correctness in tandem, this approach enables more scalable, reliable fuzzing and opens new opportunities for applying persistent-mode fuzzing to a broader class of software systems.

The primary contributions of this work are as follows:

- A dynamic instrumentation removal mechanism that disables coverage logging at run-

time for already-covered edges, improving execution throughput without requiring recompilation [15, 16].

- A persistent-mode fuzzing framework with accurate program state restoration, using compiler-inserted instrumentation to reset global variables, heap memory, and file handles between test cases [22, 24].
- A fully integrated, user-space fuzzing infrastructure that operates on Clang-instrumented binaries and avoids kernel-level dependencies such as OS snapshotting [8, 29] or external tracers.
- A comprehensive evaluation across five real-world open-source benchmarks, demonstrating that the combined approach improves fuzzing performance relative to baseline configurations.

Chapter 2

Background

Fuzzing is an automated software testing technique used to uncover software bugs and security vulnerabilities by rapidly executing large volumes of test inputs. Its appeal lies in its scalability and ability to discover unexpected behaviors that may elude manual analysis. The effectiveness of fuzzing is contingent on throughput-how quickly test cases can be generated, executed, and analyzed, as well as input diversity. The faster a fuzzer can iterate, the more likely it is to explore deeper or more complex paths in a program within a reasonable timeframe.

Fuzzing approaches are commonly categorized based on how much knowledge they have about the internal structure of the program under test. In a *black-box* setup, the fuzzer operates without any internal visibility, treating the program as an opaque system and observing only input-output behavior. This model allows for fast input generation but lacks precision in exploring execution paths. On the opposite end, *white-box* fuzzing utilizes full program visibility, typically requiring access to the source code, to guide test generation through techniques such as symbolic execution and constraint solving [11, 23]. While powerful, these approaches are computationally expensive and often impractical at scale. *Grey-box* fuzzing strikes a balance by using lightweight instrumentation to obtain partial execution feedback—most commonly, code coverage [6, 14, 30]. This feedback guides input mutations toward unexplored paths, improving the chances of discovering unique behaviors while maintaining high throughput.

2.1 Coverage-Guided Fuzzing

Coverage-guided fuzzing (CGF), a prevalent variant of grey-box fuzzing, augments the target program with lightweight instrumentation to collect real-time code coverage feedback. This feedback is used to guide test case mutations, prioritizing inputs that trigger new program states. The primary objective of CGF is to maximize exploration of the program under test (PUT) by focusing on inputs that increase coverage. Inputs that exercise previously unexplored code paths are considered valuable and saved to serve as the basis for future mutations, allowing the fuzzer to iteratively build on prior progress [3, 32]. This feedback-driven loop is illustrated in Figure 2.1. By prioritizing untested areas of the program, CGF concentrates effort on promising regions of code, increasing the likelihood of uncovering hidden vulnerabilities and edge-case bugs that traditional testing methods may overlook. CGF has proven effective in exposing security weaknesses in diverse software environments, including web browsers [1, 6, 19, 25] and operating systems [14, 29, 30].

2.1.1 Coverage Instrumentation

Coverage instrumentation in fuzzing involves adding extra code to the target program to monitor what parts of the code are executed during testing. Coverage metrics used by instrumentation are critical for evaluating how thoroughly a program has been tested. Common metrics include function coverage, which tracks whether each function in the program was executed; block coverage, which measures whether each basic block of code within a function was executed; and edge coverage, which records whether each control flow path between blocks has been executed, offering finer granularity than block coverage. Among these, edge coverage is the most widely used because of the detailed insight into the control flow path it provides [16, 27, 28, 32].

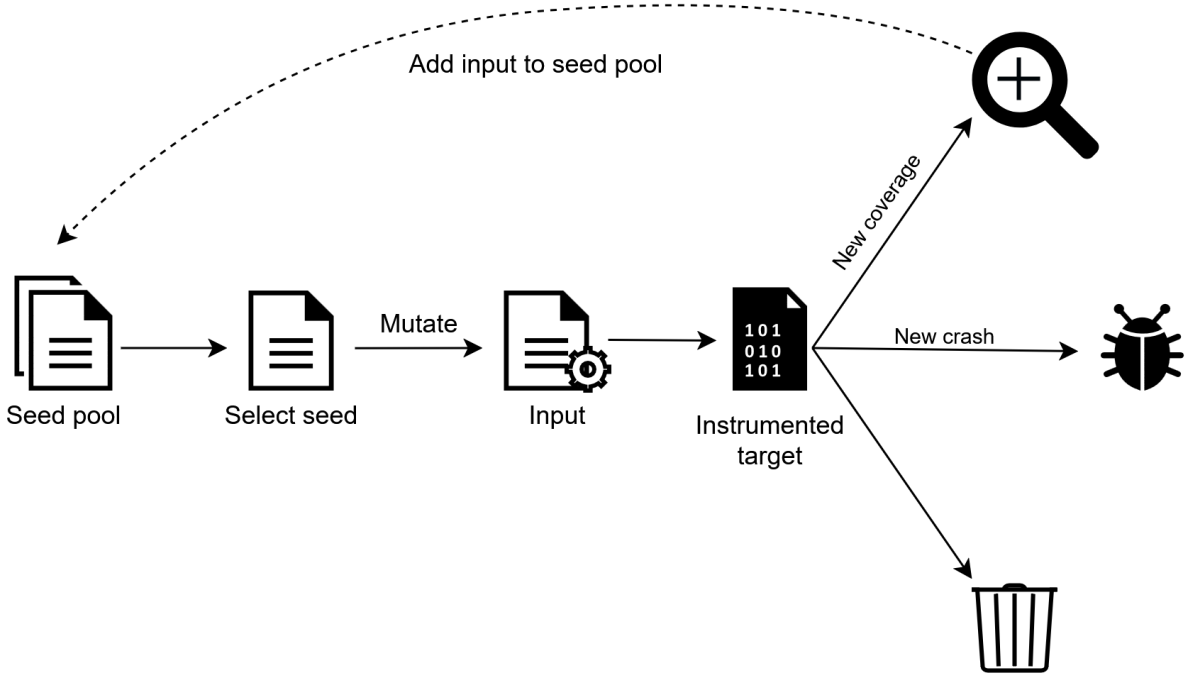

Figure 2.1: Overview of the coverage-guided fuzzing workflow. Coverage feedback from each execution is used to guide input mutation and select new seeds for further exploration.

2.1.2 Hit Counts

Hit counts in fuzzing refer to the number of times specific pieces of code are executed during the fuzzing process. This metric is particularly useful in identifying which parts of the code are more frequently tested and potentially over-tested. By analyzing hit counts, fuzzers can prioritize less frequently executed paths, enhancing the thoroughness of the testing process. This approach can lead to more efficient discovery of latent vulnerabilities by ensuring less explored paths are adequately tested [15, 19, 27, 32].

2.1.3 Performance Metrics

In fuzzing, the primary measure of success is often the discovery of crashes, which indicate potential vulnerabilities. Crashes are considered critical metrics as they typically represent

exploitable conditions within the software. Fuzzers aim to maximize the number of unique crashes discovered during testing, thereby enhancing the software's reliability and security. The effectiveness of a fuzzer is frequently judged by its ability to generate inputs that lead to new, unique crashes, highlighting areas of the code that require further investigation and remediation [13, 32].

2.2 Process Creation

Coverage-guided fuzzing is computationally expensive due to the overhead of process creation, program initialization, and input execution. To address these inefficiencies, fuzzing frameworks have developed strategies to reduce target execution overhead. Two widely used techniques—forkserver-based execution and persistent fuzzing—have been instrumental in improving fuzzing efficiency.

2.2.1 Forkserver

One of the most impactful optimizations is the forkserver execution model, first introduced by AFL [30]. Traditionally, fuzzers restart the target program for each test case, incurring high process creation overhead. The forkserver model mitigates this by keeping an initialized instance of the target program in memory and forking new child processes as needed. This allows new test cases to execute with minimal startup cost, as memory pages are shared between the parent and child processes via copy-on-write memory management [31]. AFL++ [6, 7] refined this approach by optimizing shared memory communication and improving compatibility across different platforms. However, while forkserver execution significantly improves fuzzing throughput, it does not eliminate all execution overhead entirely—each

newly forked process still incurs some startup cost due to process creation and memory page management, particularly in large applications with complex initialization routines.

2.2.2 Persistent Fuzzing

To further improve execution efficiency, persistent fuzzing—also referred to as in-process fuzzing, single-process fuzzing, or loop-based fuzzing—eliminates process forking altogether by keeping the fuzzing target alive in memory and repeatedly executing new test cases within the same process. Instead of terminating after each input, the target runs inside a persistent harness that loops over inputs, processes each one, and returns control to the fuzzer without restarting. This technique is particularly useful for fuzzing libraries, where initialization costs dominate execution time [19]. By avoiding repeated process creation, persistent fuzzing enables fuzzers to execute test cases at a much higher rate than forkserver-based execution. However, state persistence issues arise, where global variables, heap allocations, and file descriptors may persist across test cases, leading to inconsistent fuzzing results. Unlike traditional per-process fuzzing, which relies on the operating system to reset execution state, persistent fuzzing requires explicit state restoration mechanisms to ensure correctness [6].

2.3 Performance Optimizations in Coverage-Guided Fuzzing

Improving the efficiency of CGF has been a central focus of recent research. A number of strategies target different sources of performance bottlenecks:

- **Execution efficiency**: Techniques such as forkserver-based execution and persistent fuzzing reduce the overhead of process creation and initialization. These methods, discussed in the previous section, significantly increase throughput but introduce new

challenges related to state management across executions.

- **Instrumentation overhead**: As fuzzers repeatedly explore the same regions of code, continued coverage tracking becomes redundant. Tools like UnTracer [15] and Hexcite [16] mitigate this by removing or minimizing instrumentation in well-covered areas.

- **Program state management**: Persistent fuzzing can lead to residual state leaking between test cases, affecting correctness. Solutions such as AFL-Snapshot [8], WinFuzz [24], and ClosureX [22] restore a clean baseline between executions using snapshotting or compiler-inserted resets.

The remainder of this section covers previous work in minimizing instrumentation overhead and managing program state, both of which play a critical role in enabling fast and reliable fuzzing.

2.3.1 Reducing Overhead

Coverage-guided fuzzers typically use compiler-inserted instrumentation to collect coverage feedback at runtime, enabling them to steer test case generation toward unexplored regions of code. However, as fuzzing progresses, the fuzzer repeatedly executes well-covered code regions, wasting CPU cycles on redundant coverage collection that no longer provides useful guidance [3, 7, 13, 28].

UnTracer [15] reduces coverage overhead by dynamically switching between instrumented and non-instrumented versions of the target binary. During normal fuzzing, it runs a fast, non-instrumented version for the majority of test cases, falling back to a slower, instrumented version only when a potentially new path is detected. This selective tracing strategy enables previously explored code regions to execute at near-native speed, significantly improving

fuzzing throughput. However, deferring instrumentation carries risks—if a test case appears uninteresting due to the lack of coverage feedback but actually triggers a deeper, subtle behavior, the fuzzer may miss important vulnerabilities.

Hexcite \cite{hexcite} extends prior work by supporting finer-grained coverage metrics, such as edge coverage and loop hit counts, while maintaining high fuzzing throughput. Unlike UnTracer, which relies on dynamic binary switching and supports only basic block coverage, Hexcite statically rewrites binaries to preserve precise control-flow visibility with minimal overhead. By operating at the edge level, it achieves better feedback precision without sacrificing performance, enabling deeper path exploration than block-based or always-instrumented approaches.

Several complementary efforts have focused on reducing coverage overhead in binary-only settings. For example, RetroWrite [5] performs static binary rewriting to insert coverage instrumentation and sanitization hooks without requiring source code access. ZAFL [17] lifts binaries to intermediate representation (IR) to enable compiler-quality transformations, then re-emits them with edge-level instrumentation. These approaches broaden the applicability of CGF to closed-source and stripped binaries while maintaining high-quality instrumentation.

2.3.2 Managing Program State

In traditional per-process fuzzing, each test case runs in a new process, allowing the operating system to automatically reset execution state. However, execution models such as persistent fuzzing, which keep the fuzzing target alive across multiple test cases, require manual state restoration to prevent contamination. Without proper resets, global variables, heap allocations, and file descriptors from previous test cases may persist, leading to unintended side

effects in subsequent executions [22, 24]. To address this, researchers have explored both system-level snapshotting and instrumentation-based state restoration.

Xu et al. [29] proposed a set of operating system primitives to support high-performance fuzzing, including a lightweight snapshot mechanism designed as an alternative to `fork()`. These primitives allow a target process to save and restore its execution state efficiently, enabling fast iteration without the overhead of full process creation. Although initially presented in a fuzzer-agnostic form, they later inspired implementations like AFL-Snapshot-LKM [8], which integrated kernel-level snapshotting into AFL to reduce test case execution overhead without modifying the target binary.

Winnie [12] implements a forkserver-style execution model for Windows by synthesizing fuzzing harnesses and using fast process cloning to avoid repeated initialization. While it does not rely on compiler instrumentation or kernel modifications, it achieves process-level test isolation similar to traditional forkserver fuzzing on Unix-based systems.

WinFuzz [24] introduces a binary-level solution for correct persistent fuzzing in Windows environments. Rather than relying on kernel primitives, it injects snapshot and restore logic directly into compiled binaries using static binary rewriting. This target-embedded technique restores key state components—including the stack, registers, global memory, and heap allocations—between test cases using binary hooks and guard pages. It enables high-speed persistent fuzzing without compromising test isolation.

ClosureX [22] shares the same goal of enabling correct persistent fuzzing but operates at the compiler level. Using LLVM instrumentation, it inserts custom passes during compilation to track and reset global variables, heap allocations, and file descriptors between iterations. This compiler-assisted technique embeds state restoration directly into the source-level control flow, avoiding reliance on kernel features or binary rewriting while providing fine-grained

control over memory and I/O state.

While WinFuzz and ClosureX both embed state-resetting logic into the target program, they operate at different abstraction levels—binary rewriting versus compiler IR instrumentation. More broadly, while kernel-level snapshotting offers fast and accurate resets without modifying program logic, instrumentation-based solutions provide more portable, extensible approaches that can be adapted to a wider range of fuzzing targets. The most appropriate strategy depends on platform constraints, target availability, and performance goals.

Table 2.1: Comparison of related fuzzing systems.

System	**State Reset**	**Coverage Overhead Reduction**	**Runs in User Space**
AFL++ [7]	✗	✗	✓
UnTracer [15]	✗	✓	✓
Hexcite [16]	✗	✓	✓
AFL-Snapshot-LKM [8]	✓	✗	✗
WinFuzz [24]	✓	✗	✓
ClosureX [22]	✓	✗	✓
This Work	✓	✓	✓

Chapter 3

Motivation

While the evolution of fuzzing techniques, particularly coverage-guided fuzzing, has greatly advanced the field of automated software testing, existing state-of-the-art tools like AFL++ [6] still exhibit significant limitations that can compromise the efficiency and effectiveness of the fuzzing process. This section outlines these critical challenges, highlighting the motivation behind this research to enhance both the precision and performance of fuzzing tools.

3.1 Existing Tools and Challenges

AFL++ enhances the foundational features inherited from AFL, such as forkserver and persistent mode, by optimizing their implementation to improve test case execution efficiency and effectiveness. Separately, UnTracer [15] introduces optimized coverage tracing techniques to reduce the overhead of tracing execution. To maintain execution correctness while minimizing overhead, ClosureX [22] and WinFuzz [24] implement target-embedded snapshotting mechanisms for effective state management. However, despite these advancements, several key issues persist, which are central to the focus of this work:

3.1.1 Inefficient Coverage Instrumentation

Coverage-guided fuzzers, including AFL++, rely heavily on instrumentation to track which parts of the code are executed during tests. While techniques such as those used in UnTracer [15] and Hexcite [16] have significantly reduced this overhead by disabling or minimizing coverage instrumentation in already-explored regions, most fuzzing workflows still incur unnecessary runtime cost from instrumentation that remains active even after regions have been thoroughly exercised [3, 13, 28].

LLVM's SanitizerCoverage instrumentation provides support for selectively disabling coverage insertion at compile time through static blocklists and allowlists [27]. However, this approach requires recompiling the target each time coverage needs to be removed, making it impractical for active fuzzing campaigns—especially those targeting large or slow-to-compile software. Furthermore, the granularity of these static lists is limited to the function level, preventing fine-grained control over individual basic blocks or edges.

Reducing this redundant overhead remains an important goal for improving the speed, scalability, and efficiency of modern fuzzers.

3.1.2 Persistent State Management

While persistent mode offers significant performance benefits by reducing the overhead of process initialization for each test case, its implementation often leads to improper management of program states. This mode frequently fails to completely reset the state of the program between test cases, leading to state leakage. These flaws stem from insufficient control over global memory and dynamic allocations, leading to unreliable fuzzing outcomes and compromising the integrity of test results [22, 24]. Despite its prevalence, the correct application of persistent mode that appropriately manages program state is rare. AFL++'s

persistent mode is designed for targets that do not alter program state across executions, which is an uncommon characteristic for many real-world fuzzing targets [6]. As a result, its applicability is limited, reducing the general effectiveness of frameworks that rely on this model.

AFL++'s LKM (Linux Kernel Module) for snapshotting [8] attempts to address this by allowing the fuzzer to quickly restore the program state to a clean snapshot after each test case. However, this approach is tightly coupled with Linux kernel capabilities, which restricts its use to environments where such modifications are permissible or feasible [10].

3.2 Research Focus

The central motivation of this research is to simultaneously address these two prevalent issues:

- **Dynamic Coverage Removal:** Inspired by the initial steps taken by UnTracer and Hexcite [15, 16], this work aims to further develop methods to dynamically disable unnecessary coverage instrumentation, thus reducing the computational overhead associated with redundant data collection. This approach directly enhances the fuzzer's efficiency, allowing it to operate more quickly and extend its operational duration, which indirectly aids in thorough code exploration.

- **Enhanced State Management in Persistent Mode:** Building on concepts from WinFuzz [24], this research aims to refine the implementation of persistent mode by ensuring thorough and reliable state resets between test cases. By improving state management techniques, this work seeks to eliminate the inaccuracies caused by state leakage, thereby enhancing the reliability and accuracy of fuzzing results.

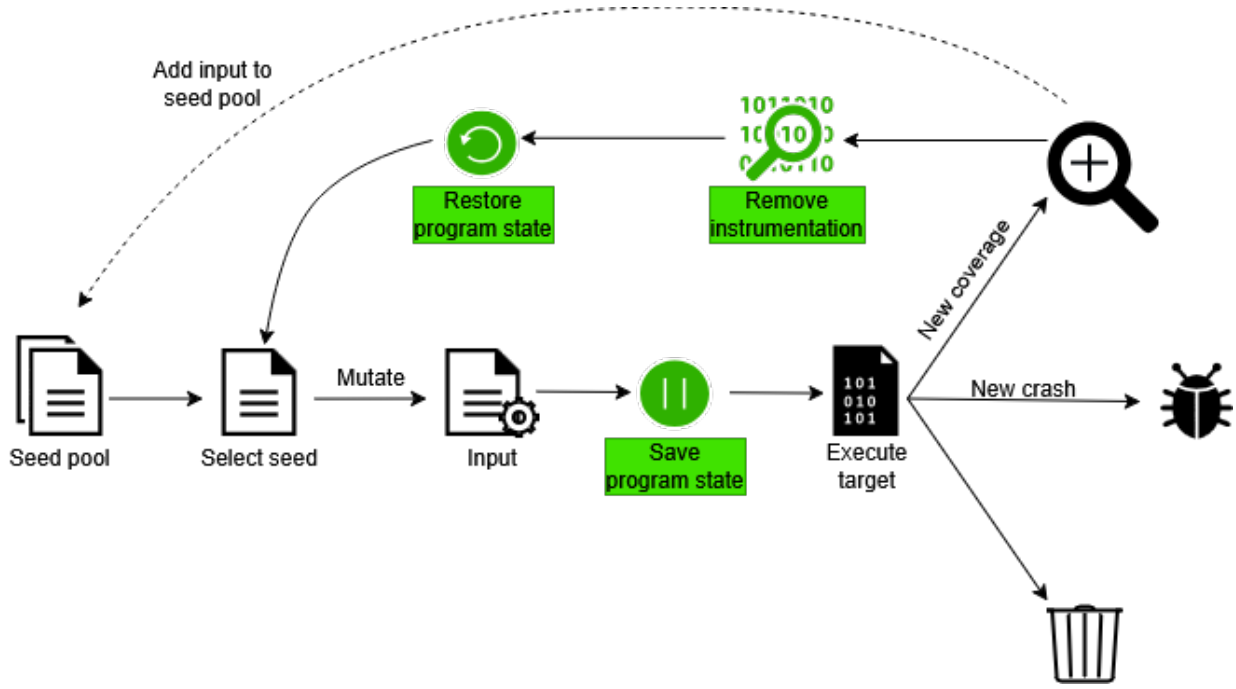

Figure 3.1: The overall fuzzing flow; contributions highlighted in green.

Figure 3.1 highlights the contributions of this work. By focusing on these aspects, this research contributes to the broader goal of making fuzzing not only faster but also more reliable, thereby better serving the needs of modern software development environments where security and performance are critical [4, 14].

Chapter 4

Design

Coverage-guided fuzzers must balance the competing demands of maximizing throughput and maintaining accurate, effective coverage feedback. However, both coverage instrumentation and persistent in-memory fuzzing introduce unique challenges that, if not carefully managed, can significantly degrade performance or corrupt execution state.

This work addresses two major sources of inefficiency in coverage-guided fuzzing: redundant instrumentation overhead, and state contamination during persistent-mode execution. It introduces two synergistic mechanisms—dynamic coverage removal and automated state restoration—that together reduce overhead while maintaining correctness. These enhancements are integrated into a lightweight, extensible framework that builds on existing compiler instrumentation.

4.1 Reducing Coverage Overhead

Coverage instrumentation is essential for guiding input mutation in CGF, but it remains active even after a program edge has already been explored. This results in substantial redundant overhead, especially in tight loops or hot paths that are repeatedly executed.

The design addresses this by dynamically disabling coverage instrumentation once it becomes unnecessary. Instead of relying on static blocklists—which require prior knowledge and

recompilation—the system monitors which edges have been hit during fuzzing and removes the corresponding instrumentation in-place.

To do this safely and efficiently, the design modifies the binary directly on disk between executions. This avoids the need to disable instrumentation at runtime and ensures the modified binary reflects the updated state permanently. The approach relies on relative offsets to identify instrumentation sites but does not depend on the actual layout in memory (e.g., it remains compatible with ASLR or randomized loading).

This dynamic strategy preserves the initial benefits of fine-grained edge coverage but eliminates redundant logging on already-covered paths, reducing overhead over time without sacrificing correctness or requiring recompilation. Specifically, it removes the high-cost function calls associated with Clang's `trace_pc_guard` instrumentation once an edge has been hit. However, it retains lightweight hit count tracking via inline 8-bit counters, which increment a byte in shared memory at each execution. These counters impose minimal overhead and continue to provide frequency-based feedback, allowing the fuzzer to adapt its mutation strategy without the full cost of traditional coverage tracing.

4.2 Correct Persistent Fuzzing

Persistent mode accelerates fuzzing by keeping the target program loaded in memory and reusing the same process across multiple inputs. However, this model introduces challenges around test isolation. If program state from one input leaks into the next, it can cause false positives, missed bugs, or non-deterministic behavior.

To ensure correctness without forfeiting the speed benefits of persistent mode, this design introduces automated state management via static instrumentation and lightweight runtime

support. The key principle is to treat the start of each fuzzing iteration as a soft "reset point" that restores memory and resource state to a known baseline.

This is accomplished through a set of compile-time instrumentation components, implemented as custom LLVM passes:

- **GlobalPass**: Identifies writable global variables and inserts logic to back up and restore their values.
- **DynamicPass**: Instruments heap allocation functions to track memory and enforce cleanup between iterations.
- **CleanupPass**: Hooks file I/O and termination-related functions to prevent resource leaks and premature exits.

These passes introduce lightweight instrumentation that enables runtime hooks to restore state without requiring kernel modifications or external harnesses. The persistent fuzzing loop can then continue indefinitely while maintaining isolation between test cases.

By embedding reset logic directly into the target, this design enables efficient, correct persistent-mode fuzzing across a wide range of real-world programs.

Chapter 5

Implementation

This chapter describes the implementation of a lightweight fuzzing framework that combines dynamic coverage removal with correct persistent-mode execution. To reduce instrumentation overhead, the system monitors which control-flow edges have been covered during fuzzing and patches the corresponding sites in the binary with NOP instructions. To ensure correctness in persistent mode, it uses a set of LLVM passes to restore global variables, heap memory, and external resources between test cases. All mechanisms operate entirely in user space and are compatible with C/C++ targets compiled with Clang's SanitizerCoverage instrumentation [26]. The result is a self-contained infrastructure that improves fuzzing efficiency without sacrificing test isolation or requiring kernel support.

5.1 Coverage Removal

As described in the previous chapter, this work implements a dynamic coverage removal mechanism that eliminates redundant instrumentation during fuzzing. This section details how coverage call sites are identified and patched within the binary at runtime.

5.1.1 Relative Addresses of Coverage Instrumentation

At runtime, Clang's `trace-pc-guard` instrumentation populates a shared memory coverage data array with program counter (PC) addresses for each instrumented site. Each index in the array corresponds to a unique program edge, identified to by it's guard ID number [27]. During the first execution of each edge, the inserted callback function computes the address of the coverage call site, then stores the offset of that address relative to the program's `main` function. This strategy avoids absolute addresses, which may vary due to ASLR or binary layout differences.

To reconstruct the actual location of the coverage site in the binary, the fuzzer retrieves the address of the `main` function using the `nm` utility [2] and adds it to the stored offset. The result is the precise file address of the `call` instruction corresponding to a newly hit edge.

5.1.2 Rewriting Coverage Call Sites

Static analysis of binaries compiled with Clang's `trace_pc_guard` confirmed that coverage instrumentation sites consistently appear as 5-byte `call` instructions to the `__sanitizer_cov_trace_pc_guard` function, implemented in `trace-pc-guard.c`. Each such instruction corresponds to the opcode pattern `E8 xx xx xx xx` in x86 assembly.

To verify the structure, placement, and runtime behavior of these instrumentation sites, a combination of static and dynamic reverse-engineering tools was employed. These included Ghidra [18], `objdump`, Radare2 [21], and GDB [9]. These tools were used throughout development to inspect binary layout, confirm instruction encodings, observe control flow, and ensure that runtime patching did not disrupt program semantics.

To remove these instructions, the fuzzer opens the target binary on disk and overwrites the

corresponding file offset with five NOP (0x90) instructions. This disables the instrumentation logic while preserving control flow and instruction alignment. The fuzzer performs this operation only after the target process has been terminated to ensure safe and consistent modification. The insertion of NOP instructions is exemplified in figure 5.1.

```
30836:   48 8d 3d cb c8 0f 00    lea    0xfc8cb(%rip),%rdi        # 12d108 <__TMC_END__+0x30>
3083d:   e8 de d3 0c 00          call   fdc20 <__sanitizer_cov_trace_pc_guard>
30842:   8b bd 44 fc ff ff       mov    -0x3bc(%rbp),%edi
```

(a) Before: Call to `__sanitizer_cov_trace_pc_guard` is present at address `0x3083d`.

```
30836:   48 8d 3d cb c8 0f 00    lea    0xfc8cb(%rip),%rdi        # 12d108 <__TMC_END__+0x30>
3083d:   90                      nop
3083e:   90                      nop
3083f:   90                      nop
30840:   90                      nop
30841:   90                      nop
30842:   8b bd 44 fc ff ff       mov    -0x3bc(%rbp),%edi
```

(b) After: The `call` instruction is replaced by five `NOP (0x90)` instructions.

Figure 5.1: Replacing coverage instrumentation with NOPs. This transformation disables the coverage tracking logic at runtime while preserving binary layout and control flow.

5.1.3 Coverage Removal Strategy

To prevent corruption or race conditions, coverage instrumentation is removed only between fuzzing executions. In forkserver mode, the target is killed and relaunched after patching. In persistent mode, the binary is modified once and reused across many iterations. This ensures that patching occurs in a controlled, stable environment without modifying a running process.

The removal process is tightly integrated into the fuzzer's feedback loop. After each test case, the fuzzer checks shared memory for newly hit edges. If new coverage is detected, the system computes the file offsets corresponding to the instrumentation call sites and overwrites them with NOPs in a single batch. This batch rewriting model minimizes I/O

overhead and ensures that redundant logging is eliminated efficiently, even when multiple edges are discovered by a single input.

Although live in-memory patching was considered (e.g., using `mprotect` to mark code pages writable), it was ultimately rejected due to complexity, platform-specific behavior, and the need for stability. By using direct on-disk modification and relative offsets, the system remains portable and avoids dependence on process memory layout, preserving compatibility with ASLR.

While the instrumentation removal process itself introduces a small amount of overhead, this cost is quickly recovered over time. In practice, coverage-increasing inputs are exceedingly rare; prior work has shown that most test cases do not lead to new code coverage on common benchmarks [15]. As such, eliminating unnecessary instrumentation after one of these rare events significantly improves throughput for all subsequent executions. The full instrumentation removal loop is illustrated in Algorithm 1.

Algorithm 1 Dynamic Coverage Removal (`Fuzzer`)

Require: P: Path to the target binary

Require: C: Coverage bitmap in shared memory

Setup: Initialize shared memory and pipe communication
$M \leftarrow$ `GET_MAIN_ADDRESS`(P)
`READ_SEED_FILE`()
`EXECUTE_TARGET`(P)
while true **do**
  for all mutations from coverage-increasing seeds **do**
    `SEND_INPUT_TO_TARGET`()
    `WAIT_FOR_TARGET`()
    $Offsets \leftarrow$ `CHECK_NEW_COVERAGE()` from C
    if $Offsets \neq \emptyset$ **then**
      `KILL_TARGET`()
      for all $offset \in Offsets$ **do**
        `SEEK_TO`($M + offset$) in P
        `OVERWRITE_BYTES`(offset, `NOP`, length = 5)
      end for
      `RESTART_TARGET`(P)
    end if
  end for
end while
`CLEANUP_SHM_AND_EXIT`()

5.2 State Management

Persistent fuzzing significantly reduces the overhead of process creation by executing multiple fuzzing iterations within a single long-lived target process. However, the correctness of this optimization depends on ensuring that each iteration begins from a clean state. Without explicit restoration of memory and program state, inputs may interfere with one another, leading to invalid coverage signals, missed bugs, or unstable behavior.

To enable automatic state restoration, the system inserts instrumentation directly into the target during compilation using a set of custom LLVM passes. These passes operate on the program's intermediate representation (IR) and allow the compiler to inject reset logic for global variables, heap memory, and file-related state [20]. This approach integrates seamlessly with existing Clang-based toolchains and avoids the need for runtime instrumentation or external wrappers.

All inserted instrumentation is gated by the persistent-mode option, ensuring that state restoration logic is only active during persistent-mode fuzzing and remains inactive unnecessary.

5.2.1 Global Variable Restoration

Global variables with external linkage are a common source of state pollution in persistent fuzzing. To address this, we introduce `GlobalPass`, an LLVM module pass that performs static state checkpointing for global variables.

At compile time, `GlobalPass` identifies all non-constant, non-static global variables in the target program, excluding standard I/O descriptors and types matching `FILE*`. For each eligible global variable `G`, the pass emits a corresponding backup variable `G_backup` and

inserts instrumentation in the entry block of `target_main()` to restore `G` from `G_backup` at the beginning of every iteration.

The pseudocode for this transformation is shown in Algorithm 2. This ensures that any global variable modified during a fuzzing iteration is reset before the next input executes.

Algorithm 2 Static Global Variable Backup and Restore (`GlobalPass`)

Require: P: LLVM module of the target program

Skip pass if P is a known instrumentation or support module
$F \leftarrow$ `target_main` function in P
if F is not found **then**
    return
end if
$G \leftarrow$ list of global variables in P
for all $GV \in G$ **do**
    if GV is non-constant and external **then**
        if GV is not a standard I/O stream or `FILE*` **then**
            $B \leftarrow$ new global backup variable with zero-initialization
            Add pair (GV, B) to backup list
        end if
    end if
end for
At the entry of `target_main()`, inject `load` from each backup and `store` to corresponding global
return

5.2.2 Heap Allocation Management

Many fuzzing targets perform dynamic memory allocation using standard C library functions (`malloc`, `calloc`, `realloc`, `free`). In the absence of process isolation, memory leaks and dangling pointers from previous inputs can persist and accumulate, causing instability or erroneous behavior.

To prevent this, `DynamicPass` rewrites all calls to the standard heap management functions to use custom wrapper functions instead. These functions log each allocation into a runtime-maintained `std::unordered_map` keyed by pointer address. At the end of each persistent iteration, the system invokes a cleanup routine, `freeAllAllocations()`, which traverses the allocation map and frees any remaining live objects.

This approach guarantees that heap state is restored to an empty baseline before each new input is processed. The detailed instrumentation logic is described in Algorithm 3.

Algorithm 3 Dynamic Memory Interposition and Reset (`DynamicPass`)

Require: P: LLVM module of the target program

Skip pass if P is a support/instrumentation module
$F \leftarrow$ `target_main` function in P
if F is not found **then**
    return
end if
for all `malloc`, `calloc`, `realloc`, `free` **do**
    Replace all calls with custom wrappers (`myMalloc`, etc.)
end for
for all basic blocks BB in F **do**
    if BB ends in a `return` instruction **then**
        Inject a call to `freeAllAllocations()` before return
    end if
end for
return

5.2.3 File and Exit Management

Fuzzing targets often open files, use buffered I/O, or call `exit()` to terminate execution under certain conditions. In persistent mode, these actions can disrupt test isolation by leaking file handles or leaving buffers in an inconsistent state, or halt the fuzzing loop entirely by terminating the process.

To mitigate this, `CleanupPass` rewrites calls to `fopen()`, `fclose()`, and `exit()` to point to instrumented wrappers: `fopen_hook()`, `fclose_hook()`, and `exitHook()`. Alongside their original behavior, these hooks perform three key functions:

- Track all open `FILE*` handles in a runtime set
- Close all unclosed files at the end of each iteration
- Override the behavior of `exit()` to run cleanup logic and return control to the fuzzer

This pass ensures that file descriptors and exit behavior are safely contained, allowing fuzzing to proceed uninterrupted even if the target attempts to terminate or leak file handles. Algorithm 4 outlines the instrumentation strategy.

Algorithm 4 File Handle and Exit Hook Instrumentation (`CleanupPass`)

Require: P: LLVM module of the target program

Skip pass if P is a support module
$F \leftarrow$ `target_main` function in P
if F is not found **then**
    return
end if
for all functions `fopen`, `fclose`, `exit` in P **do**
    Replace all uses with `fopen_hook`, `fclose_hook`, `exitHook`
end for
return

Algorithm 4 outlines the instrumentation strategy.

Chapter 6

Evaluation

This chapter presents a comprehensive evaluation of the proposed fuzzing framework, focusing on execution performance, edge coverage, and the tradeoffs introduced by the hybrid configuration. It begins with an overview of the experimental setup, including the selected benchmarks, system configurations, and evaluation infrastructure. The following sections report performance results in terms of execution throughput, analyze coverage growth and edge discovery, and conclude with a synthesis of the findings.

6.1 Experimental Setup

6.1.1 Benchmarks

To validate the dynamic coverage removal and correct persistent-mode enhancements introduced by this research, a series of experiments were conducted on five real-world open-source benchmark programs. These benchmarks were selected to span a range of software domains, while meeting the following criteria:

- **Instrumentation compatibility:** Each program must compile cleanly with Clang and support LLVM-based coverage instrumentation.
- **Input handling:** The program should accept input from a file or stdin, enabling

integration with the fuzzer's shared-memory-based execution pipeline.

- **Coverage potential:** Each benchmark should expose a sufficiently large and complex control-flow surface (i.e., edge set) to enable meaningful coverage analysis and performance differentiation.

Table 6.1 summarizes the selected benchmarks, along with their accepted input formats and primary functionality. Together, they span parsers, compressors, interpreters, and document processors, providing a realistic evaluation suite.

Table 6.1: Overview of selected benchmark programs

Program	**Input Type**	**Primary Functionality**
cJSON	`.json` file	JSON parser
zlib	`.zlib` raw compressed file	Decompress binary data
Poppler	`.pdf` file	Parse & render PDF page
xmllint (libxml2)	`.xml` file	Parse XML into DOM
Lua (interpreter)	`.lua` script	Execute Lua code

6.1.2 System Configurations

Four fuzzing configurations were evaluated to isolate and assess the contributions of target-embedded state management and dynamic coverage removal:

- **Static**: Baseline configuration using traditional forkserver-based execution with Clang's `trace-pc-guard` instrumentation. This mode includes no persistent mode state management or coverage removal.

- **Dynamic**: Identical to the Static configuration but augmented with dynamic coverage removal. As new edges are discovered, the fuzzer modifies the binary in place by patching out redundant coverage function calls.

- **Persistent**: Implements a correct persistent-mode execution loop. This configuration includes state management via LLVM passes, but does not perform dynamic coverage removal.

- **Hybrid**: Combines correct state-managed persistent-mode execution with dynamic coverage removal.

All four configurations share the same fuzzing infrastructure, mutation strategies, and harnessing interface. This design isolates the effects of execution model and instrumentation strategy by limiting variability introduced by unrelated implementation factors. However, due to the stochastic nature of fuzzing, some variation in outcomes is unavoidable. As such, not all observed differences in throughput or coverage can be definitively attributed to configuration differences alone, though repeated trials and consistent trends help support the conclusions drawn.

6.1.3 Execution Environment

All fuzzing experiments were performed on a virtualized cluster of Linux environments configured specifically for isolation and repeatability. Each fuzzing trial was assigned to a dedicated virtual machine to prevent resource contention and ensure consistent execution characteristics.

- **Operating System:** Ubuntu 22.04 LTS (64-bit)

- **VM Resources:** 1 vCPU and 2 GB RAM per virtual machine

- **Trial Duration:** Each fuzzing run was executed for 24 hours.

- **Repetitions:** 5 independent trials were conducted per configuration per target, totaling 100 trials.
- **Timeouts:** No explicit per-testcase timeout was enforced in the target binaries; instead, hung executions were implicitly bounded by the persistent-mode loop or fork-server timeouts in practice.

The fuzzing infrastructure was fully automated and capable of launching parallel, non-interfering trials across machines. Input, coverage, and execution data were logged continuously for each run and processed offline for analysis.

6.2 Performance Evaluation

This section evaluates the execution throughput of the four fuzzing configurations by analyzing the total number of testcases executed over a 24-hour period. Execution throughput is a key measure of efficiency in fuzzing, as faster iteration enables broader input exploration and accelerates coverage discovery.

6.2.1 Total Executions Across Configurations

Figure 6.1 presents the total number of testcases executed over 24 hours for each target under all four fuzzing configurations. Across most benchmarks, the Hybrid configuration achieves the highest overall throughput, with Persistent mode close behind. These results support the central hypothesis of this work: that combining persistent-mode execution with dynamic removal of redundant instrumentation yields the most efficient fuzzing strategy by reducing both startup overhead and unnecessary runtime logging.

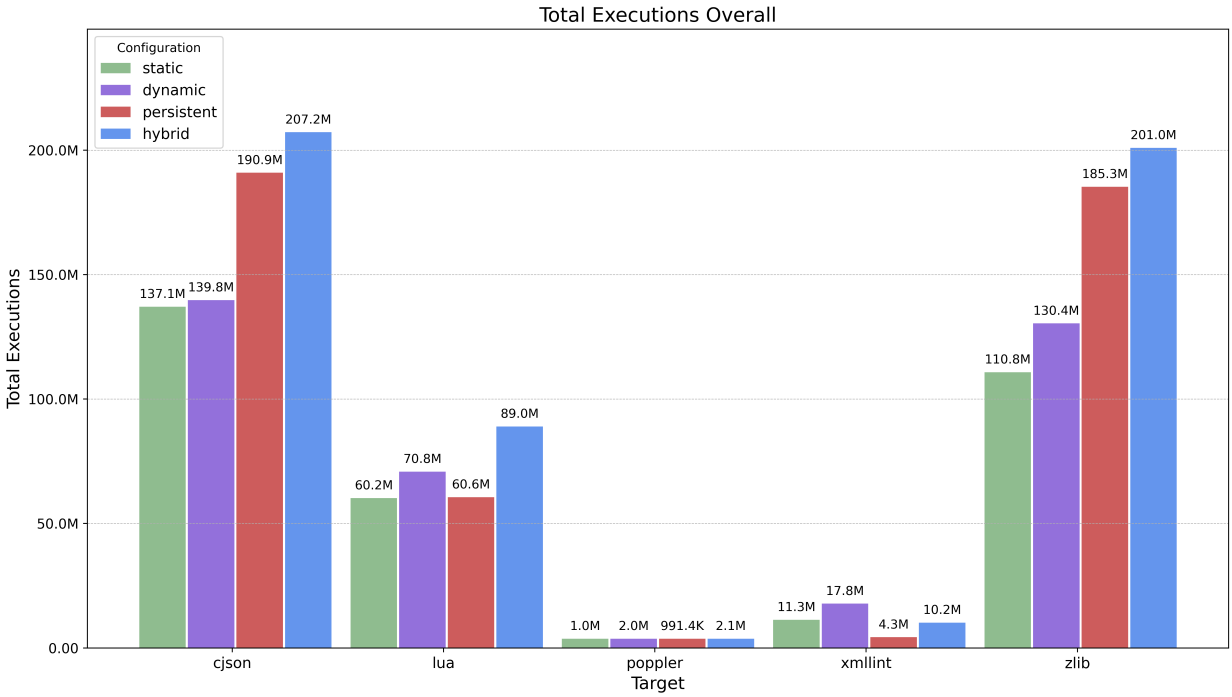

Figure 6.1: Total number of testcases executed over 24 hours for each configuration and target.

The benefits of this combination are most clearly seen in *cJSON* and *zlib*. Both are ideal fuzzing targets with tight harnesses that call directly into well-isolated library functions. These benchmarks lend themselves well to persistent fuzzing, and Hybrid mode achieves slightly better throughput than Persistent alone. Even Dynamic mode shows gains over Static, suggesting that removing redundant instrumentation yields measurable performance improvements, even when the binary must be restarted after patching.

Lua shows a variation on this trend. Hybrid mode again leads, but Dynamic mode outperforms Persistent, suggesting that coverage removal alone contributed more to throughput than in-memory execution. This may reflect recurring coverage overhead in frequently traversed paths, or limitations in how much startup cost persistent mode can avoid in this target

In *Poppler*, a heavyweight PDF rendering engine, throughput is uniformly lower across all configurations. Persistent execution offers little benefit here, and Hybrid only modestly outperforms Dynamic. This suggests that factors such as high setup or memory costs may limit the advantages of in-memory reuse. Even so, Dynamic and Hybrid both improve upon Static, indicating that coverage removal remains beneficial even when persistent mode is less effective.

Finally, *xmllint* presents the most unexpected result. Dynamic mode achieves the highest execution count, followed by Static and Hybrid, with Persistent trailing far behind. This inversion of expected trends may be due to internal buffering, parsing routines, or global state that is difficult to reset between iterations. Unlike the other targets, persistent execution appears to slow down this benchmark rather than speed it up, highlighting the importance of target-specific structure when choosing execution models.

6.2.2 Throughput Over Time

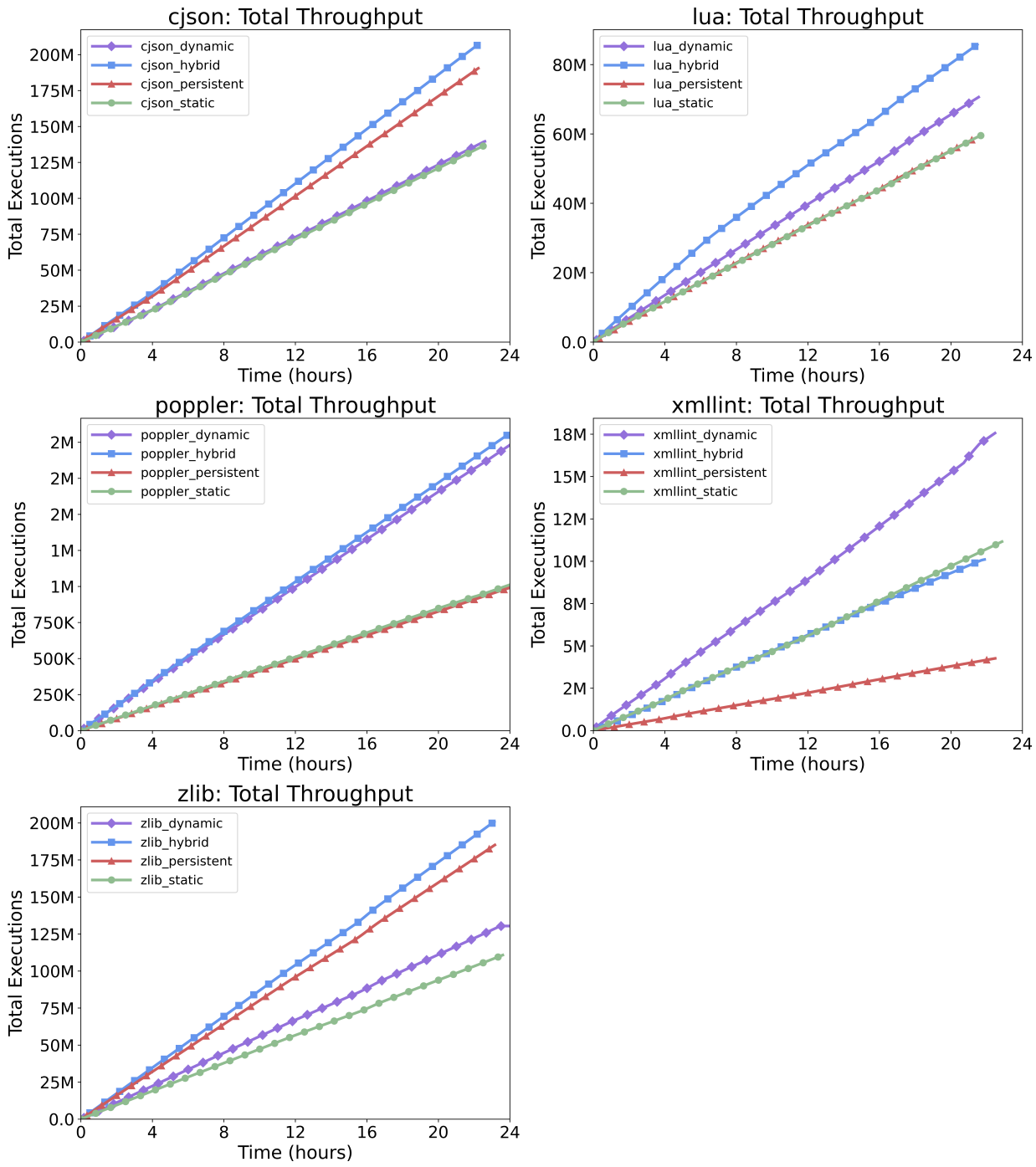

Figure 6.2: Execution trends over time for each target (one subplot per benchmark).

Figure 6.2 shows the progression of testcase execution throughout the fuzzing campaign. As in the total execution results, *cJSON* and *zlib* exhibit clear separation between configurations, with Hybrid and Persistent maintaining strong throughput across the entire 24-hour window. These targets respond well to persistent-mode fuzzing, showing consistent gains from in-memory execution and instrumentation reduction.

In *Lua*, Hybrid starts with only a modest lead but gradually pulls ahead, reflecting the accumulating benefits of removing redundant instrumentation. A similar pattern appears in *Poppler*, where Hybrid and Dynamic track closely while Static and Persistent fall behind early. These trends suggest that coverage instrumentation reduction is more beneficial than persistent execution for loop-heavy or complex programs.

xmllint again deviates from the broader trend. Dynamic mode achieves the fastest execution growth, while Persistent mode lags considerably. The convergence between Hybrid and Static lines suggests that persistent execution may have introduced unanticipated bottlenecks that were not offset by coverage removal. These results suggest that some real-world binaries may exhibit structure or statefulness that undermines the benefits of a simple persistent-mode implementation.

6.2.3 Relative Coverage Comparison

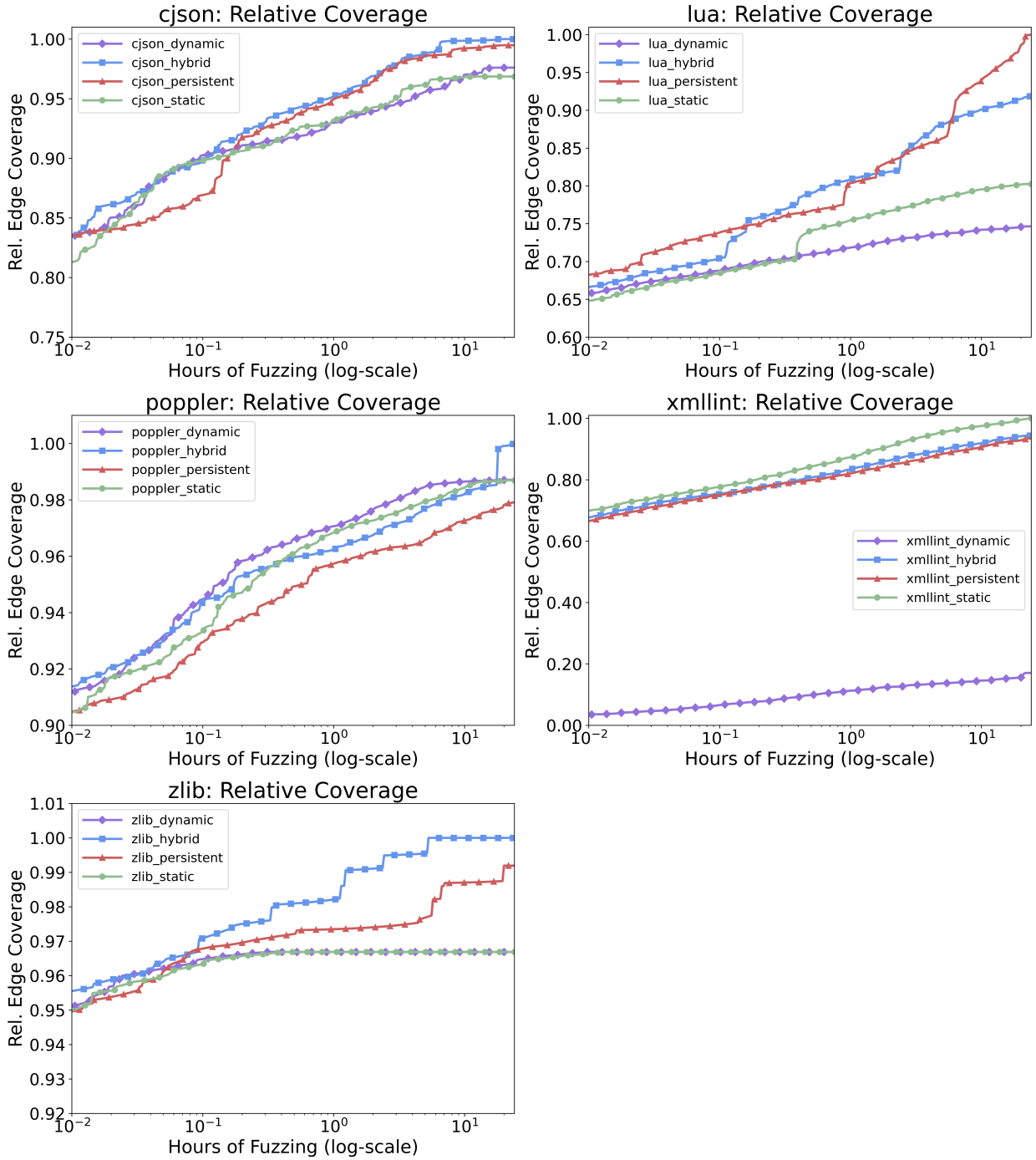

Figure 6.3: Relative edge coverage at the end of the 24-hour fuzzing period for each configuration and target. Values are normalized by the highest observed coverage per target.

Figure 6.3 presents the relative edge coverage across all targets, normalized to the best-performing configuration in each case. Overall, coverage trends closely mirror throughput: configurations that executed more testcases generally discovered more edges, particularly on targets amenable to persistent-mode execution.

On well-behaved targets like *zlib* and *cJSON*, Hybrid mode consistently achieves the highest coverage, with Persistent close behind. These benchmarks feature short input-processing paths and minimal internal state, allowing the high throughput of Hybrid to translate directly into coverage exploration gains. Static and Dynamic modes fall significantly behind, highlighting the cost of tracing every input and repeatedly restarting the process.

Lua follows a similar pattern, with Hybrid and Persistent outperforming the fork-based modes. Interestingly, Persistent mode eventually surpasses Hybrid in edge count, despite executing fewer total testcases. This result suggests that throughput alone does not fully determine coverage outcomes, and other factors—such as program structure or input sensitivity—may influence deeper path discovery.

In *Poppler*, Hybrid ultimately achieves the highest edge coverage, surpassing Dynamic late in the fuzzing campaign. Dynamic and Static perform similarly through most of the run, with both maintaining a clear lead over Persistent. This reinforces the idea that coverage gains here are driven more by instrumentation cost than by execution model.

The most anomalous case is *xmllint*, where Static achieves the highest final coverage and Dynamic the lowest, with Hybrid and Persistent in between. Unlike the other targets, there is no clear performance benefit from either persistent execution or coverage removal. Overall throughput was also low across all configurations, suggesting that *xmllint*'s complexity or initialization behavior may fundamentally limit fuzzing effectiveness under all tested modes.

6.2.4 Summary of Results

The results validate the core hypothesis of this work: combining persistent-mode execution with dynamic coverage removal yields a more efficient fuzzing strategy. Across most targets, the Hybrid configuration achieved the highest execution throughput and competitive or superior coverage, demonstrating that reducing both process overhead and redundant instrumentation has compounding benefits.

On targets like cJSON, zlib, and lua, Hybrid mode consistently performed best, translating execution speed into deeper code coverage. Even where Persistent mode alone was strong, Hybrid provided additional gains through on-the-fly simplification of the binary.

The benefits were less clear on more complex targets. In poppler, persistent execution offered little advantage, but Hybrid's ability to remove instrumentation still led to the highest final coverage. In xmllint, neither persistent execution nor coverage removal significantly helped; Hybrid and Static performed similarly, while Persistent lagged, suggesting that some targets are inherently resistant to in-memory execution models.

Overall, Hybrid mode proved to be the most generally effective strategy. It maintained correctness while accelerating execution, and adapted well across a range of real-world software targets. These findings underscore the value of combining state restoration with runtime-aware instrumentation, and suggest opportunities for further optimization through adaptive or profile-guided switching between fuzzing modes.

Chapter 7

Conclusion

This research advances coverage-guided fuzzing by addressing two key inefficiencies: the state contamination issues in persistent mode and the high runtime overhead from static coverage instrumentation. A hybrid approach is proposed that combines dynamic coverage removal with enhanced state management, enabling more efficient and reliable fuzzing execution.

Dynamic coverage removal is implemented by monitoring newly discovered edges during execution and dynamically patching out the corresponding `trace-pc-guard` instrumentation function calls using NOP instructions. This reduces redundant tracing without requiring recompilation. To support correct execution in persistent mode, a set of custom LLVM passes—GlobalPass, DynamicPass, and CleanupPass—collectively manage program state. GlobalPass restores non-static global variables, DynamicPass tracks and frees heap allocations between iterations, and CleanupPass hooks file I/O and exit functions to maintain test isolation. Together, they ensure that each fuzzing iteration begins from a consistent baseline without requiring OS-level snapshotting.

Evaluation across five real-world targets shows that this system achieves higher execution throughput than traditional coverage-guided fuzzing, including persistent-mode-only configurations. The overhead of dynamic patching and restoration is offset by the elimination of unnecessary instrumentation and re-execution. These results are achieved without requiring OS-level snapshotting or full binary recompilation between runs.

Future work could explore integration with existing fuzzing frameworks such as AFL++ and ways to leverage hardware support for runtime binary modification. Additionally, adaptive strategies could be developed to dynamically toggle persistent execution or instrumentation granularity based on runtime profiling of the target's behavior to optimize when and how coverage tracking is removed. This direction offers a promising foundation for improving the efficiency, adaptability, and robustness of modern fuzzing infrastructure.

Bibliography

[1] Abhishek Arya, Oliver Chang, and Kostya Serebryany. Clusterfuzz: Fuzzing at google scale. In *Proceedings of the 28th USENIX Security Symposium*, pages 1437–1450. USENIX Association, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/arya.

[2] GNU Binutils. nm - list symbols from object files. https://man7.org/linux/man-pages/man1/nm.1.html, 2024. Accessed: 2024-12-18.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, pages 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery. doi: 10.1145/2976749.2978428. URL https://doi.org/10.1145/2976749.2978428.

[4] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 678–689, 2020. doi: 10.1145/3368089.3409748. URL https://doi.org/10.1145/3368089.3409748.

[5] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511, 2020. doi: 10.1109/SP40000.2020.00025. URL https://ieeexplore.ieee.org/document/9152762.

[6] Andrea Fioraldi, Marc Heuse, Dominik Maier, and Heiko Eißfeldt. Afl++: A

community-maintained fork of the american fuzzy lop fuzzer. https://github.com/AFLplusplus/AFLplusplus, 2020. Accessed: 2024-12-18.

[7] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020. URL https://www.usenix.org/conference/woot20/presentation/fioraldi.

[8] Andrea Fioraldi, Marc van Hauser, Joey Jiao, and Heiko Eisfeldt. Afl++ snapshot lkm, 2020. URL https://github.com/AFLplusplus/AFL-Snapshot-LKM. Accessed: 2024-11-01.

[9] Free Software Foundation. Gdb: The gnu project debugger. https://www.gnu.org/software/gdb/, 2024. Accessed: 2024-12-18.

[10] Robert Gawlik, Daniel Gruss, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2020. URL https://www.ndss-symposium.org/ndss2020/ndss-2020-programme/kafl-hardware-assisted-feedback-fuzzing-os-kernels/.

[11] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2008. URL https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/.

[12] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Network and Distributed System Security Symposium (NDSS)*, 2021. URL https://www.ndss-symposium.org/ndss-paper/

winnie-fuzzing-windows-applications-with-harness-synthesis-and-fast-cloning/. Accessed: 2024-12-18.

[13] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018. doi: 10.1145/3243734.3243804. URL https://doi.org/10.1145/3243734.3243804.

[14] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47:2312–2331, 2018. URL https://api.semanticscholar.org/CorpusID:102351047.

[15] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802, 2019. doi: 10.1109/SP.2019.00069.

[16] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*, pages 351–365, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3460120.3484787. URL https://doi.org/10.1145/3460120.3484787.

[17] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1683–1700. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/nagy.

[18] National Security Agency. Ghidra software reverse engineering framework. https://ghidra-sre.org, 2019. Accessed: 2024-12-18.

[19] LLVM Project. Libfuzzer – a library for coverage-guided fuzz testing, 2022. URL https://llvm.org/docs/LibFuzzer.html. Accessed: 2024-12-10.

[20] LLVM Project. Llvm compiler infrastructure documentation. https://llvm.org/docs/, 2024. Accessed: 2024-12-18.

[21] Radare Project. Radare2 reverse engineering framework. https://rada.re/n/, 2024. Accessed: 2024-12-18.

[22] Rishi Ranjan, Ian Paterson, and Matthew Hicks. Closurex: Compiler support for correct persistent fuzzing. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25)*, pages 151–163, New York, NY, USA, 2025. Association for Computing Machinery. doi: 10.1145/3669940.3707281. URL https://doi.org/10.1145/3669940.3707281.

[23] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016*. The Internet Society, 2016. doi: 10.14722/ndss.2016.23368. URL https://www.ndss-symposium.org/ndss2016/driller-augmenting-fuzzing-through-selective-symbolic-execution/. Publisher Copyright: © 2016 Internet Society.

[24] Leo Stone, Rishi Ranjan, Stefan Nagy, and Matthew Hicks. No linux, no problem: Fast and correct windows binary fuzzing via target-embedded snapshotting. In *Proceed-*

ings of the 32nd USENIX Security Symposium (USENIX Security '23), pages 4913–4929, 2023. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/stone. Accessed: 2024-11-01.

[25] Robert Swiecki. Honggfuzz – security oriented software fuzzer, 2020. URL https://github.com/google/honggfuzz. Accessed: 2024-12-10.

[26] The Clang Team. Clang documentation. https://clang.llvm.org/, 2024. Accessed: 2024-12-18.

[27] The Clang Team. Sanitizercoverage documentation. https://clang.llvm.org/docs/SanitizerCoverage.html, 2024. Accessed: 2024-12-18.

[28] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, Chaoyang District, Beijing, 2019. USENIX Association. URL https://www.usenix.org/conference/raid2019/presentation/wang-jinghan.

[29] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 2313–2328, New York, NY, USA, 2017. Association for Computing Machinery. doi: 10.1145/3133956.3134046. URL https://dl.acm.org/doi/10.1145/3133956.3134046.

[30] Michal Zalewski. American fuzzy lop (afl). https://github.com/google/AFL, 2013. Accessed: 2024-12-18.

[31] Michal Zalewski. Fuzzing random programs without execve(), October 2014. URL

https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html. lcamtuf's old blog. Accessed: 2024-11-01.

[32] Michal Zalewski. Technical whitepaper on afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2017. Accessed: 2024-12-18.

[33] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):230:1–230:36, September 2022. doi: 10.1145/3512345. URL https://doi.org/10.1145/3512345.