ClosureX: Transforming Source Code for Correct Persistent Fuzzing

Rishi Ranjan

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
Na Meng
Thang Hoang

May 07, 2024
Blacksburg, Virginia

Keywords: Fuzzing, Program Instrumentation, Security

ClosureX: Transforming Source Code for Correct Persistent Fuzzing

Rishi Ranjan

(ABSTRACT)

Fuzzing is a popular technique which has been adopted for automated vulnerability research for software hardening. Research reveals that increasing fuzzing throughput directly increases bug discovery rate. Given fuzzing revolves around executing a large number of test cases, test case execution rate is the dominant component of overall fuzzing throughput. To increase test case execution rate, researchers provide techniques that reduce the amount of time spent performing work that is independent of specific test case data. The highest performance approach is persistent fuzzing, which reuses a single process for all test cases by looping back to the start instead of exiting. This eliminates all process initialization and tear-down costs. Unfortunately, persistent fuzzing leads to semantically inconsistent program states because process state changes from one test case remains for subsequent test cases. This semantic inconsistency results in both missed crashes and false crashes, undermining fuzzing effectiveness. I observe that existing fuzzing execution mechanisms exist on a continuum, based on the amount of state that gets discarded and restored between test cases. I present a fuzzing execution mechanism that sits at a new spot on this state restoration continuum, where only test-case-execution-specific state is reset. This fine-grain state restoration provides near-persistent performance with the correctness of heavyweight state restoration. I construct ClosureX as a set of LLVM compiler passes that integrate with AFL++. Our evaluation on ten popular open-source fuzzing targets show that ClosureX maintains semantic correctness all while increasing test case execution rate by over 3.5x, on

average, compared to AFL++. ClosureX also finds bugs more consistently and 1.9x faster than AFL++, with ClosureX discovering 15 0-day bugs (4 CVEs).

ClosureX: Transforming Source Code for Correct Persistent Fuzzing

Rishi Ranjan

(GENERAL AUDIENCE ABSTRACT)

Fuzzing is a technique of automated vulnerability research which tries to find bugs in programs by generating randomised inputs and feeding it to the program under test. It then monitors the program execution to identify any crashing inputs which can be later triaged by a human in order to conretely identify any bugs, as well as perform root-cause analysis. In this work, I introduce a new program state restoration technique to achieve correctness in persistent mode, the fastest execution mechanism in fuzzing.

Dedication

I dedicate this work to my parents, my brother and my girlfriend, who have been my rock throughout this journey.

Acknowledgments

I would like to express my deepest gratitude towards my advisor Dr. Matthew Hicks, for his support, encouragement and guidance throughout my research. I would also like to thank my lab mates at FoRTE Research group for their friendship and support throughout my time at Virgnia Tech. I am grateful to my parents, my brother and my girlfriend Radhika for believing in me through academic journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Modern day applications are growing in complexity and functionality at a rapid rate. As code bases continue evolving to meet increasing functional demand, the responsibility of producing secure applications also increases, which can only be fulfilled by intensive testing and validation of code bases. [21]. With this growing complexity, leading tech companies have been focusing on fuzzing to increase software security as an automated supplement to traditional verification methods [9, 16, 24].

Fuzzing is an automated means for discovering vulnerabilities in applications is the industry standard for security testing, and has found applicability in all varieties of software. Fuzzing operates by generating random test cases based on a set of seed inputs provided and running the test case against the program under test to find bugs (as signaled by a crash). Even if the test cases fail to find a bug, a key aspect of fuzzing is adding interesting test cases to the seed pool. By doing this, fuzzing makes incremental progress towards the discovery of crash-inducing test cases. The most popular metric used to determine if a test case is interesting is whether it exercises some as-of-yet unseen portion of the program; this is called coverage-guided fuzzing. Code coverage has improved the ability of fuzzing to explore complex programs, and helps fuzzing incrementally get to a bug in the program under test. [10, 17].

Roughly, there are two ways to increase test case execution rate: decrease program execution time and decrease the time spent executing support code (e.g., process creation, initialization, management, and tear down). Recent work has nearly eliminated fuzzing-related overhead incurred during the execution of program code by baking-in coverage tracking [5] and deferring coverage tracking to the uncommon case of an interesting test case [17, 19]. These breakthroughs emphasize the problem of time wasted executing support code as that remains the central source of fuzzing related execution overhead.

Minimizing support code overhead centers on identifying and eliminating all sources of invariant code execution: i.e., the same code executes and produces the same software-level effects independent of the actual test case contents. At one extreme, the fuzzer can execute a target in a completely new process for each test case. This is a very coarse grain approach that destroys all program state and recreates the entire process from the start for every test case. This achieves correctness at the expense of performance. At the opposite extreme is persistent fuzzing that uses a single process for all test cases, ignoring any changes to software-level state across test cases. This sacrifices correctness for increased performance. The current approach that maintains correctness while improving performance is the forkserver [8]. The forkserver hooks a process (i.e., the starting point of each test case's execution) after process initialization and uses Linux's `fork` system call to spawn a new process. The operating system uses copy-on-write to avoid cloning the entire process state while still maintaining the correctness of per-process execution. While the forkserver is both performant and correct, I observe that it is still more heavyweight than required for fuzzing.

I posit that a correct version of persistent fuzzing is possible. I observe that it is possible to reuse a single process across test cases as long as we preserve the abstraction of each test case getting its own process. Preserving this abstraction requires a form of rollback

recovery [7] where, at the completion of a test case's execution, I rollback to the starting state in preparation for the next test case. I design and implement a correct persistent fuzzer called ClosureX. ClosureX logs changes to program state during execution (e.g., which global variables were modified) and restores them before executing the next test case. By doing this fine-grain, execution-specific state tracking and restoration, ClosureX removes initialization, tear-down, and most support code execution, while preserving correctness.

To evaluate ClosureX, I fuzz a diverse set of popular open-source programs and compare ClosureX's performance with the most popular whitebox fuzzer, AFL++. Our evaluation shows that ClosureX increases test case execution rate by over 3.5x and discovers bugs 1.9x faster all while maintaining semantic correctness. Across my evaluation, ClosureX finds 15 unique bugs across 4 programs, resulting in 4 CVEs, and 7 patches.

ClosureX embodies several technical contributions:

- I show that there is a fine-grain, execution specific way to track and undo state changes to create a correct version of persistent fuzzing.
- I design and implement a set of LLVM passes to reset the state of a program to its starting state after execution.
- I integrate ClosureX with the leading fuzzer AFL++.
- I prove the semantic correctness of ClosureX.
- I find and report 15 0-day bugs in popular open-source programs.
- I open source ClosureX for use and extension by the broader software testing community.

1.2 Background

Having access to the source code of an application offers the most effective and efficient fuzzing due to the ability to directly analyze, modify, and integrate with application source code. Previous work leverages this level of program access to provide low-overhead code coverage tracing [5] to improve performance and sanitizers [5, 22, 27] to improve fuzzing effectiveness. Despite these improvements, with the help of recent advancements in binary rewriting [18] and coverage tracing [10, 17, 19] binary-only fuzzing has recently started to perform better and catch up to source-available fuzzing in both performance and effectiveness.

ClosureX is a source-available fuzzing technique that better leverages source code availability to improve fuzzing performance.

1.2.1 Compiler Optimizations

LLVM is a language and platform agnostic compiler architecture, which allows users to implement and integrate their own program analysis and transformation passes at compile-time [12]. LLVM works by transforming the source code into an intermediate representation called LLVM IR. The LLVM IR is a strongly-typed assembly-like intermediate representation with an added catch of infinite virtual registers. The language and platform agnostic optimizations are applied directly on the LLVM IR. LLVM IR introduces an infinite amount of virtual registers in order to leverage Static Single-Assignment (SSA). SSA is a technique which ensures that every register is defined only once in the IR. Any later definitions of a variable lead to introduction of a new virtual register in the IR. SSA helps in seamlessly applying compile-time optimizations to the IR as it simplifies the conditions for such optimizations. Moreover, LLVM allows developers to implement their own compile-time passes in

order to analyze and modify the LLVM IR. These passes are then scheduled by LLVM in the middle-end of the compilation pipeline. This allows for scalable program instrumentation, analysis and optimization.

Most modern fuzzer today rely on LLVM-based instrumentation in order to modify/instrument programs for fuzzing, especially to collect code coverage from the program during execution. [5, 8, 13, 22, 23, 27]. LibFuzzer and AFL++ both utilize LLVM's trace-pc-guard instrumentation [5, 13, 34]. It allows capturing coverage at different granularities, like function-level, block-level, edge-level, or even hitcount-level. Fuzzing also involves program modification for sanitization and optimization [22, 27].

ClosureX is a set of LLVM passes that integrate with the fuzzer AFL++.

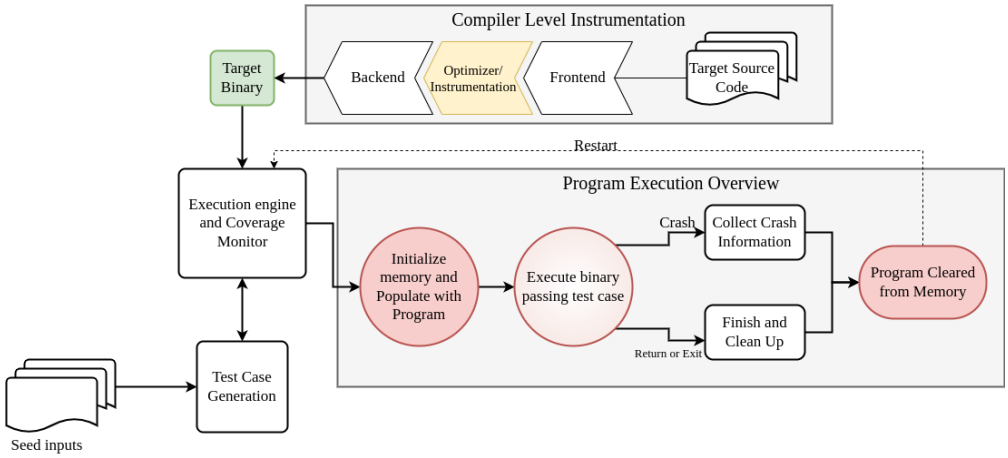

Figure 1.1: Coverage guided fuzzing loop showcasing various steps involved. Red-colored steps represent the current performance bottlenecks for source-available fuzzing, thus the focus of this thesis.

1.2.2 Fuzzing Overview

Fuzzing can be understood as the process of feeding test cases into a program under test and monitoring the resulting execution. Fuzzing leverage information from the program execution in order to guide its test-case generation process in order to find bugs and triage

crash-inducing test cases to uncover associated bugs. The process of fuzzing consists of four main components [34]:

1. Program instrumentation
2. Test case generation
3. Program execution and monitoring
4. Feedback analysis

Programs, depending upon the availability of their source-code, are fuzzer using either binary-only techniques or source-available techniques. The two broad categories differ mainly in their ways of gaining insights from a program execution. Binary-only fuzzing techniques perform analysis of a compiled program, and instrumentation to measure program execution is often injected into the program using binary rewriting techniques at block-level. [18, 30]. A popular way to do this has been in the form of insertion of callback routines [8] and system-wide tracing using signals [17, 19] at the entry instruction of the block to gain code-coverage. However, source-available techniques which work using compile time instrumentation are the most performant way of gainig code-coverage from a program execution. These instrumentations are implemented using either GCC [26] or LLVM [12, 23, 27]. The biggest performance boost comes from further optimizations that can be performed by the compiler after code coverage instrumentation. It also allows for achieving a more granular coverage metric like edge or hitcount based coverage with near-native application execution performance. This has led to source-available fuzzing to be adopted by industry leaders into their software development lifecycle for continuous testing of their applications [9, 16, 24].

Test case generation, involves using an initial seed corpus to generate test cases that differ just enough to explore new parts of a program. Ideally, test case generation will change portions of the seed test case that govern conditional statements, exposing unseen program behavior. The instrumentation inserted into a program to calculate its code coverage during

an execution produces a bitmap structure which holds information about the parts of the program executed. Coverage-guided fuzzing is the field which utilizes this coverage information in order to generate new test cases to feed to the program. Fuzzers can generate test cases in two different ways: generational [3, 28] and mutational [2, 5, 8, 13, 24, 34]. Generational fuzzers generate test cases from scratch and are often paired with an abstract model of acceptable test case as a part of grammar-based fuzzing [25, 29]. In contrast, mutational fuzzers generate new test cases by applying random-influenced mutations to a previously-executed and saved test case.

Feedback analysis is the technique of processing the feedback obtained from executing the target with a test-case and determining whether the test-case is "interesting" and should partake in further mutation cycles. The parameters to determine this depends on the type of feedback obtained, e.g., in case of code-coverage, any test-case that executes an unexplored block or edge in the target in considered interesting. Crash triage is the process of isolating a crash found by the fuzzer, and determining its root-cause.

ClosureX is a coverage-guided mutational fuzzer.

1.2.3 Fuzzing Performance

Figure 1.1 demonstrates that program execution for every test case is handled by an external system, usually a part of the fuzzer. For pre-AFL fuzzers, this means executing the program again after the current execution of the program terminates via the `fork()` system call, which creates a fresh process, followed by a call to the `execve()` system call which loads the program under test into the newly created process. AFL-based fuzzers utilise a program called the forkserver for controlling the program execution. The execution engine is not only responsible for transmitting code-coverage feedback from the target to the fuzzer, but it also

handles the passing of newly generated test cases from the fuzzer to a new target instance. Moreover the execution engine monitors the memory and CPU usage of the executing process, as well as checks it for any unusual behaviour like crashes or timeouts. The execution engine also makes sure to keep the target running within fixed system resources, in order to allow the rest of the system to function normally during fuzzing. [5, 8].

Key Insight: Process creation, initialization, management, and tear-down costs are significant burden in a fuzzing use case.

1.2.4 AFL++ Design

AFL++ is a popular fork of the original AFL fuzzer, which is regularly updated with new advancements in fuzzing. [33]. [5]. AFL's became the defacto fuzzer due to its performance (thanks to a set of performance increasing "hacks") and its extensibility. AFL-derived fuzzers often have the same flow, which is represented in Figure 1.1. AFL++ adds its own execution engine, called the forkserver which greatly increases the program execution throughput and makes the system robust. The forkserver aims to get rid of the large overheads associated with process creation, and it achieves this by using the `fork()` system call on linux for lightweight process cloning and compile-time instrumentation. Forkserver achieves its performance by eliminating the need to repeatedly load a new instance of a program using `execve()` system call. It essentially works by loading a target binary once into a new process, then allowing it to complete its initialization and suspending it right before `main`. This suspended binary is then forked for every test case using fast copy on write technique, thereby skipping the program loading step for every test case. [32].

AFL's forkserver is also responsible for handling newly generated test cases by the fuzzer. It uses Inter Process Communication techniques like pipes and sets up the test case before

forking the suspended target for execution. The program then generates a coverage bitmap, which is accessible to the forkserver since it lies in a shared memory segment between them. After every iteration of the fuzzing loop, the forkserver collects this bitmap and passes it onto the fuzzer for execution measurement. The fuzzer then utilises this bitmap to determine whether the test case was able to exercise any new code paths and should it be saved in the fuzzing queue for further mutation.

Key Insight: AFL++ forkserver still incurs some process initialization and management overhead as well as process tear-down overhead. Additionally, copy-on-write works at the page level, making it coarse-grain.

ClosureX correctly reuses a single process across all test cases and restores program state at fine-grain level, minimizing all support code overhead.

1.2.5 Persistent Fuzzing

AFL++ provides a test case execution mode through LLVM instrumentation called persistent mode. Persistent fuzzing, unlike traditional `fork`-based fuzzing, reuses a single process for all test cases. Using only a single process for the entire fuzzing campaign improves the test case execution rate at the cost of removing inter-test-case isolation, i.e., the effects on program and process state from one test case carry over to all subsequent test cases. This tradeoff allows persistent fuzzing to be the fastest method for library or API fuzzing [4, 11], but fails for application fuzzing due to exposing semantically inconsistent states. Semantically inconsistent states occur when a test cases starts in a program state not possible given the program's code due to the latent effects of a previous test case's execution. Semantically inconsistent execution leads to the program accumulating state, known as *stale state* (shown in Figure 1.2). To understand the impact of *stale state*, consider an example where the

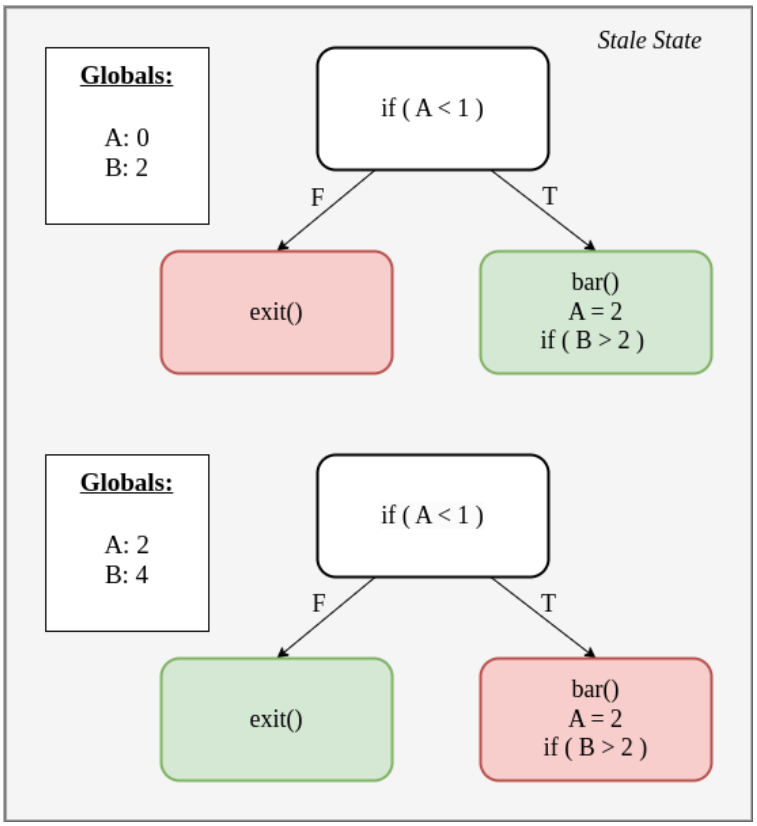

Figure 1.2: In this example, the global variable A leads to a stale state in the program. After the first iteration of the program, A is set to 2 and the `true` branch of the program is never exercised, eliminating deeper code coverage and thus wasting subsequent fuzzing cycles.

program under test has a global variable, which is set during the initial iteration, and in further iterations, this value leads to inconsistent execution of the program for a given test case. Semantically inconsistent executions result in missed crashes and synthetic crashes that waste developer time in the triage step.

To prevent semantically inconsistent executions and their knock-on effects, developers must leverage a deep understanding of the code and any potential state modifications and manually guard against such changes or revert them should they occur. LibFuzzer, which is LLVM's persistent fuzzing tool [13, 23] uses compile-time instrumentation and manually designed library harness to provide persistent fuzzing for library code. Unfortunately, the knowledge and time investment required by LibFuzzer does not scale to full applications.

ClosureX provides automatic support for whole-application persistent fuzzing, achieving high performance while ensuring semantic correctness.

1.3 Motivation

Faster execution of the target is one of the most important criteria for efficient and scalable fuzzing. In order to discover bugs or reach deeper parts of program code, a fuzzer needs to execute a large number of test cases. As shown by previous work, a fuzzer spends more than 90% of its time on test case execution [17]. Thus, any improvement in the execution speed of the target directly improves the fuzzer's exploration rate. There exist multiple techniques for target execution during fuzzing, each with its own performance and correctness tradeoff. In this section, I analyze the different execution mechanisms used in fuzzers in terms of performance and correctness.

1.3.1 Limitations of Existing Fuzzing Execution Mechanisms

Process creation based execution. The most primitive execution mechanism, which is the go-to technique of target execution in Windows-based fuzzers, is process creation. In this technique, a new process, its related data structures in kernel, and all its memory pages are created from scratch for every test case using operating system API calls. Creating a new process for each test case provides a consistent starting state for all test cases, but incurs a huge overhead due to time spent creating, initializing, managing, and tearing down the process.

Fork-based execution. The most common execution mechanism adopted by Linux-based fuzzers is the fork-based execution mechanism. `fork` is a "lightweight" process cloning API on

Linux (not available on Windows). It achieves an improvement over costly process creation by utilising copy-on-write to avoid duplicating entire process state for each test case. For every given test case, the fuzzer performs the `fork` system call to clone a new process from an existing one to run the target. Copy-on-write only copies memory pages as the child process modifies state. However, this still incurs an additional cost of loading the target into memory, setting up the correct execution context and finally performing the tear-down for every execution of the target. All of these initialization and tear-down steps are common across every execution of the target and the target process spends a significant amount of time performing them.

The forkserver-based execution. To deal with the high overhead of target initialization, AFL-based Linux fuzzers employ a forkserver, a significant improvement over simple fork-based execution. A forkserver, forks and loads the target into memory, pausing at the beginning of the `main` function of the target. The forkserver then waits for request from the fuzzer, and when it is received, forks itself and creates the target process. This ensures that the process initialization steps are done—only once—in the forkserver, which then forks itself to create new processes for each test case. Even though a forkserver does away with the process initialization costs, it still suffers from page-level duplication/management costs and process tear-down costs, as well as the kernel-level cost of process duplication.

Persistent fuzzing Persistent fuzzing (also called in-process fuzzing) is the most performant mode of target execution. It uses a single process to execute all test cases.[1] This eliminates process creation, initialization, management, and tear-down overhead. Unfortunately, persistent fuzzing is inherently incorrect. This is due to the stale state that is incurred and not cleaned up between test cases. The remedy is target specific and requires deep knowledge

[1] In practice, developers set a limit on the number of test cases to use a single process for before creating a fresh process as a hack to deal with cross-test-case state contamination. Our analysis reveals that this number can be as low as one to maintain correctness.

and correct reasoning about the target and the impact of test cases on program state.

Kernel-based snapshotting

An even more operating-system-specific improvement on the forkserver is kernel-based snapshotting [31]. Kernel-based snapshotting reduces the amount of state that needs to be reset when the fuzzer calls `fork` to create a new target process. This provides a modest performance improvement but only works for operating systems that already support the `fork` process creation mechanism and requires kernel modification. Real world adoption of fuzzing advancements shows that any fuzzing execution approach, while being fast and effective, must also be portable and maintainable across platforms. To explore the deployability tradeoff of leading fuzzing execution mechanisms, we investigate if AFL++ and kernel-based snapshotting run on various operating systems and versions. Table 1.1 shows that AFL++ enjoys wide support (so will any tool that extends it), but kernel-based snapshotting has limited applicability.

	Kernel-based snapshot	AFL++
Linux 4.8.10	✓	✓
Linux 5.15.0		✓
Linux 6.2.0		✓
Windows		✓
MacOS		✓

Table 1.1: Comparison of kernel-based snapshot and AFL++ portability. ClosureX extends AFL++ in a kernel agnostic way, bring high-performance fuzzing to all platforms.

1.3.2 Automated, Correct, and Performant Persistent Fuzzing

From my analysis of different target execution mechanisms in fuzzing, I conclude that each of them suffers from a roadblock which hinders the fuzzing throughput. For the most prim-

itive execution mechanism i.e., process creation, the limitation is the heavy cost of process creation. Fork-based execution builds on this approach by using copy-on-write for process cloning but has large initialization and tear-down costs. While forkserver-based execution circumvents fork's initialization cost by performing it only once, it still has heavy tear-down costs. Persistent fuzzing, on the other hand performs no state-restoration and tear-down, but leads to semantically inconsistent execution.

My observation is that there is a gap between no state-restoration of persistent mode execution mechanisms and general-purpose state resetting of process-level approaches. A more fine-grained—yet automatic—approach is needed to reset program state after execution which combines the performance of persistent fuzzing with the correctness of per-process fuzzing and is compatible across targets and operating systems.

Chapter 2

Design

2.1 Design

Persistent mode fuzzing is the fastest execution mechanism for fuzzing, but it lacks state tracking and restoration ability that leaves it incorrect with significant manual intervention. To fill this gap, we propose ClosureX. ClosureX is a compile-time instrumentation tool which transforms an out-of-the-box target program with two capabilities:

- Persistent fuzzing loop
- Program state tracking and restoration between persistent loop iterations

2.1.1 Persistent Fuzzing Loop

All executable C programs have a primary function named `main`. The C runtime calls `main` after initialization to start target execution. After execution, the `main` function returns and the operating system reaps the process's resources. Persistent mode differs from this approach by redirecting execution on exit to the start of `main`. Since this is all done without the overhead of creating a new process or killing the old process, this execution mechanism is more performant.

ClosureX transforms the program's source-code at compile-time and inserts the persistent

mode functionality into the compiled program automatically. ClosureX then incorporates state tracking and restoration functionality as required by the program's code (e.g., ClosureX adds hooks for dynamic memory tracking and restoration if the program uses the heap). The user has to make no additional modifications to the source code or be aware of persistent mode fuzzing at all for ClosureX integration.

2.1.2 State Restoration For Correct Execution

Whenever a program executes, it modifies program state during its execution. Without restoring software state semantically impossible states occur that hide crashes and create false crashes. To ensure semantically correct starting state for every test case, I identify the major sources of program state which we address and restore at the end of every execution.[1]

Program stack. The stack is the address space in the process where the local variables, as well as subroutines' information are stored during program run time. This is a static memory section of the process and a stack pointer is used to increase or decrease the size of the stack during execution. When a function is called, the caller stores the return address and the base pointer of its own stack frame on the stack. The callee function then allocates space on the stack by reducing the stack pointer appropriately. The saved return address and caller's stack pointer value are used during function return to restore the stack back to its caller. Thus, the stack holds important program state during run time.

When a process terminates, the kernel frees up its memory pages, which also include stack-related memory pages. In a new process the stack starts with `main`'s frame active and return information pointing to the C runtime. However, persistent mode fuzzing revolves around

[1] ClosureX is an extensible platform for correct persistent fuzzing. The types of state addressed in this section are what we saw with a diverse set of real-world programs. I believe that developers can use these state tracking and restoration patterns to handle any target-specific state not explicitly accounted for here.

iterating over relevant code in the same process for performance. Incorrect restoration of the stack after every iteration leads to the program running out of space due to stale stack frames present on the stack, or functions using incorrect information present on the stack i.e., semantically incorrect execution.

To deal with this, there are two cases that ClosureX must handle: normal return flows and `exit` flows. In normal flows, execution returns back to ClosureX's persistent loop via a series of `return` statements. This naturally clears the stack. In an `exit` flow, execution arrives at ClosureX's persistent loop with latent information on the stack. To address this, ClosureX restores the stack pointer and register values to what they were during the first test case's execution. This ensures that, no matter the way the target completes execution (ignoring the much desired crash), the stack will appear to the following test case just as it did for the first test case.

Program heap. The heap is a dynamic memory section which is used to fulfill the memory requests by the program at run time. These requests are made using calls to `malloc`, `calloc`, `realloc` etc. The memory obtained by these calls should later be returned to the C runtime using `free` calls. However, most of the end user programs that I explore have memory leaks—because they rely on the operating system to reap any outstanding dynamic memory. Thus, these memory leaks are not harmful in such isolated executions due to the kernel taking care of it. However, when running the program in persistent mode, where the same process is used to perform a large number of executions of the program, the memory leaks very quickly add up and lead to the process going out of memory. This is a drain on performance as well as a source of false crashes.

The naive solution is to hook all dynamic memory management function calls and track all open dynamic memory allocations. Upon the target's completion, ClosureX uses `free` to close all open allocations. While this solution works, there must be some nuance: I identify

that memory allocations done during a process lifetime originate from different sources. The C standard library `libc` as well as other dynamically loaded libraries also allocate dynamic memory for their functioning. If such memory is freed after every iteration, it can lead to undefined behaviour in the target. Thus, I only free the dynamic memory allocated by the target after each iteration.

Global memory. Global memory is used by programs to store data that is accessible to all subroutines. This is a static memory region as it is present in the executable itself. Global memory, thus is an important source of program state—and one most harmed by persistent fuzzing. After the execution, the kernel cleans up all the process-related memory pages which also includes global memory pages. During per-process execution, the target always starts with a non-contaminated global memory state as it is loaded directly from the executable. However in-process fuzzing uses the same process for a large number of target executions and contaminated global variables from previous execution lead to both missed and false crashes.

Restoring global memory to a previous known good state requires a fine-grained approach for maximal performance. A naive approach restores every page of the global memory to a previously taken snapshot, even if the page has or will not be written to. ClosureX takes a finer grain approach and narrows down the restoration to only the relevant sections of the global memory. ClosureX creates a snapshot of potentially modifiable global memory region of the target and restores it after every target iteration This is done at a byte-level rather than a page-level which ensures efficiency for my approach.

File descriptors. In Linux, every open file is associated with a file descriptor, which is handled by the kernel on behalf of the process. The kernel sets an upper limit on the number of file descriptors a process can use. When a process exits, the kernel closes all the open file descriptors, thus cleaning up all the possible zombie file handles for the process. If a process

opens a very large number of file handles, it encounters a "too many open files" error. In persistent mode fuzzing, even a single file descriptor left open after an iteration leads to the process running out of file descriptors very quickly. Thus, open file descriptors are a source of stale program state and need to be handled by ClosureX to prevent false crashes.

We observe that during fuzzing, the file handles opened by the target can be categorized into two sets. First is the handles which are opened once during the initialization of the target i.e., they're part of the initialization process. Closing and reopening such handles after every iteration leads to inefficiency and a better approach is to reset the handle to its starting position. The second set of file handles are the ones opened by the target during the fuzzing relevant execution. For correct execution, ClosureX tracks such file handles during the target's execution and closes them between iterations.

Exit flows Programs which deal with parsing any input structure, often terminate the process when they encounter any ill-formed input. This is done by calling the `exit` system call. This is a common occurrence for persistent fuzzing since a majority of the inputs mutated during fuzzing are not well-formed (this is the dark side of the underconstrained nature of fuzzing). This leads to premature process termination even in persistent fuzzing loop.

Similar to dynamic memory management, I distinguish between exit calls originating from the target and other loaded libraries like `libc`. The exit calls performed by the libraries are considered critical which need handling by the fuzzer. Thus such exit calls are not tracked by ClosureX. When I encounter an exit call from the target, the control-flow is transferred back to the persistent loop, which performs all the state restoration operations mentioned above and starts execution of the next test case.

2.2 Implementation

I implement ClosureX as a collection of five compile-time passes for the LLVM architecture mentioned in Table 2.1. I choose LLVM due to its modularity and ease of development, but my approach can be extended to any compiler toolchain like GCC. My instrumentation is compatible with any coverage-tracking instrumentation, and to benchmark the fuzzing performance, I compile the target with AFL's `Sanitizer Coverage Guard` on top of ClosureX's instrumentation.

ClosureX Pass	Functionality
RenameMainPass	Rename Main function of target
HeapPass	Manage target's heap memory
FilePass	Manage file descriptors of the target
GlobalPass	Move global's into a seperate memory section
ExitPass	Instrument exit calls in the target

Table 2.1: ClosureX LLVM Passes

2.2.1 ClosureX Harness

To implement correct persistent fuzzing, I inject a harness to loop over the target code for some large number of iterations. My harness initializes the target and stores relevant program state like global section before target execution. After every iteration of the target, my harness restores the state of the target program. Listing 2.1 shows the target prior to harness injection, and Listing 2.2 depicts target post harness injection.

Listing 2.1: Target code prior to ClosureX harness injection

```
main(argc, argv) {

    // Input parsing related code
    // Body of program
}
```

Listing 2.2: Target injected with the ClosureX harness at compile time.

```
start_main(argc, argv) {
    // Input parsing related code
    // Body of program
}
// Buffer for saved context
jmp_buf longjump_buf;
main(argc, argv) {
  global copy = copy_global_sections();
  while(i != iterations_count) {
    if (setjmp(longjmp_buf)) {
        // Target called exit()
    } else {
    // This is the main function of target
        start_main(argc, argv);
    }
    // Stack already restored; restore everything else
    restore_global_sections();
    reset_heap_memory();
    close_open_file_handles();
    ++i;
  }
}
```

2.2.2 Renaming `main`

To add in-memory looping functionality to the target, I rename the `main` function of the target to `start_main`. ClosureX's stub harness then calls `start_main` from its own main function in a loop. I perform this function renaming using our LLVM Module pass `RenameMainPass`. `RenameMainPass` operates on the module and tries to locate a function called `main` in the module. If `main` is found, `RenameMainPass` renames the function using `setName` method call on the `Function` object.

2.2.3 Resetting the Program Stack

ClosureX's stub harness implicitly resets the stack of the target. Since ClosureX's harness's `main` function replaces the target's `main` function, the calls to the target's `main` function are semantically correct. The arguments passed to the target's `main` function are the same as command-line arguments, i.e., `argv` and `argc`. In the case of normal execution end i.e., return to the harness from the target's `main` function, the series of `return` statements which eventually flow back to the harness unwind the stack automatically.

2.2.4 Dealing with Abnormal Program Exits

Apart from returning normally after the execution of the target's main function, a target can use the `exit` system call to exit the process immediately—without unwinding the stack. This is a typical behaviour in the case of failed file parsing.

To deal with this, ClosureX uses complex control-flow transfer in Linux which are performed using the `setjmp` and `longjmp` function calls. I set a jump target at the beginning of our harness's `main` function using the `setjmp` function, which also stores the values present in

general purpose registers at the jump target. During compile-time, I replace all the calls to `exit` system calls in the target code with our function wrapper `exitHook`. This is done by `ExitPass` using the `replaceAllUsesWith` method. The `exitHook` function wrapper calls `longjmp` to our previously set jump target with the exit status. The `longjmp` function transfers the control flow back to my harness's execution loop and also restores the saved values into the general purpose registers, including the stack and base pointer registers. This rewinds the stack back to harness's main function, thus restoring the stack. ClosureX hooks `exit` calls only in the instrumented source code of the program, and thus this has no effect on any exception handling code or exit calls in any externally loaded libraries like standard library libc.

2.2.5 Resetting Global Memory

As shown in Figure 2.1, to reset the global memory of the target, my compile-time pass `GlobalPass` first identifies the potentially modifiable global variables in the target, and then moves all such variables into a new section in the target, which we name as

`closure_global_section`. In order to identify potentially modifiable variables in an LLVM module, I iterate over all the global variables in a module and call `isConstant` method on it which returns true if it is a constant and false otherwise. If the variable is not a constant, I move it to the new section using the `setSection` API call in LLVM. My approach greatly reduces the size of the global section by excluding entities such as immutable strings from `closure_global_section`.

During the fuzzing campaign, the address of this new `closure_global_section` is passed as the environment variable `CLOSURE_GLOBAL_SECTION_ADDR`, and the size is passed as the environment variable `CLOSURE_GLOBAL_SECTION_SIZE`. The address and size of `closure_global_section`

can be easily obtained using an ELF parsing tool like `readelf`. We can observe in Figure 2.2, before the execution of target's `main` function, the new section is copied into a buffer by the ClosureX harness. After every iteration, this buffer is used to restore the global section, thus restoring the modified global memory of the target.

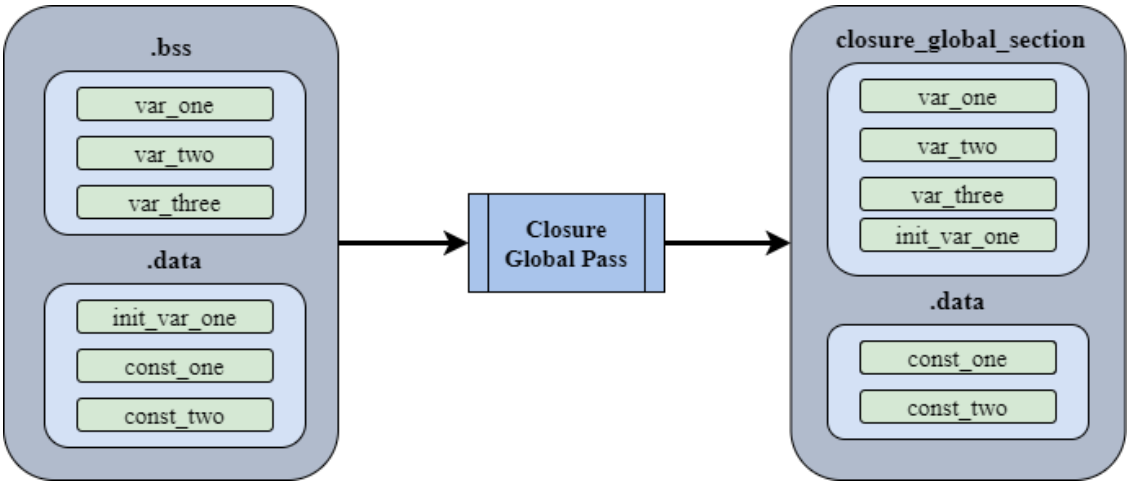

Figure 2.1: The transformation performed by ClosureX's Global pass

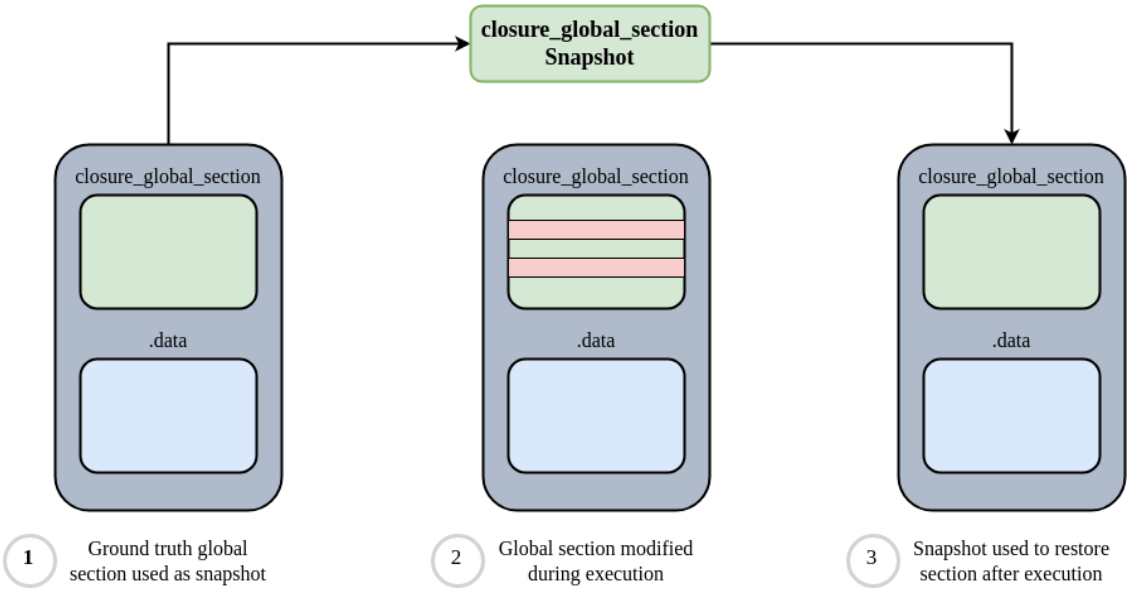

Figure 2.2: ClosureX global resetting procedure

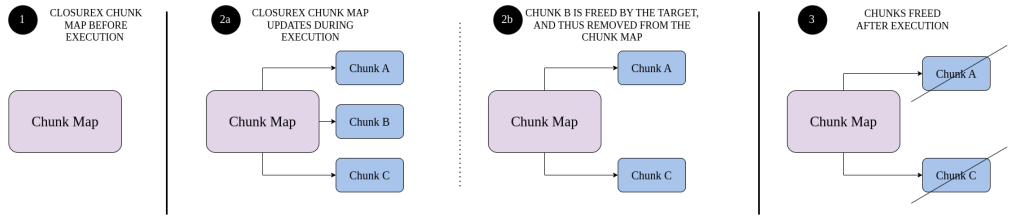

Figure 2.3: ClosureX heap resetting procedure during runtime

2.2.6 Resetting Dynamic Memory

To manage the dynamic memory of the target, I deal with two sources of error. Not only do we free all the memory leaks by the target, I also ensure that no allocated memory already freed by the target is freed by our mechanism. To achieve this, I follow an approach of tracking all the memory allocations and deallocations done by the target during its execution. ClosureX's `HeapPass` replaces all the instances of following functions by its own function wrappers.

- `malloc` → `myMalloc`
- `realloc` → `myRealloc`
- `calloc` → `myCalloc`
- `free` → `myFree`

`HeapPass` also operates on an LLVM module. For every module, I declare a new function for `myMalloc`, `myCalloc`, `myRealloc` and `myFree` in the module using the `getOrInsertFunction` method in the module. These declarations are resolved during the linking-phase of the compilation, as they're defined in ClosureX's harness. I then perform the above mentioned replacements using the `replaceAllUsesWith` method on the `Function` object for `malloc`, `calloc` etc. Figure 2.3 depicts our approach. When the target requests memory using malloc family functions, my wrapper stores the pointer returned by malloc family functions into a

hash map structure. Whenever a pointer is freed by the target, it is removed from the hash map structure by our custom free function `myFree`.

After the target's main function is executed and the control flow returns to ClosureX's harness, I iterate over all the pointers still stored into our hash map and free them. This ensures that the target starts with a clean dynamic memory heap, and our harness can fuzz the target in persistent mode for a very large number of iterations (e.g., 1,000,000 iterations). Figure 2.3 illustrates this technique.

2.2.7 Closing Open File Handles

ClosureX deals with all the open file descriptors in the target with function replacement techniques similar to resetting dynamic memory of the target. I replace file opening routine `fopen` with `fopen_hook` which tracks and stores every open file handle into a hash map. I replace calls to `fclose` with `fclose_hook` which deletes the file handle from the hash map when it is closed. This is done by our `FilePass` Function Pass. After every execution of target's main function, the harness iterates over the hash map and closes all the open file handles. This pattern described is useful for managing other resource handles which can be acquired on Linux like sockets and shared memory.

Chapter 3

Results

3.1 Evaluation

The goal of ClosureX is to provide developers the speed of persistent fuzzing with the correctness of per-process fuzzing. Thus, the evaluation systematically explores those two criteria.

1. **Correctness:** Is ClosureX's version of persistent fuzzing correct? Is test case execution the same no matter if it was executed in a fresh process versus being executed in the same process after many other test cases?

2. **Performance:** What is the improvement that ClosureX's instrumentation achieves over per-process fuzzing in terms of test case execution rate and, most importantly, bug finding.

3.1.1 Evaluation Setup

I compare ClosureX's correct persistent mode execution with the execution mechanism of the most popular Linux fuzzer AFL++. **The ideal outcome is ClosureX finds the same crashes and coverage as AFL++ (since the underlying fuzzer is the same and ClosureX maintains correctness), but much faster than AFL++.** Finding new

crashes faster is important as it reduces the resources needed for a given exploration rate or decreases the vulnerability time-window of a program. ClosureX instrumented targets are fuzzed with AFL++ and no modifications are needed in AFL++ to support ClosureX's instrumentation. Both ClosureX and AFL++ use the same coverage tracing and seed mutation mechanisms. Doing this isolates the impact of ClosureX's persistent execution mechanism.

All experiments are performed on Microsoft Azure cloud instances of type `Standard_DS1_v2`, with 3.5 GB memory and a single CPU core running Ubuntu 20.04 LTS. In order to account for randomness and add statistical rigour to our results, we perform five, 24-hour trials of each benchmark/fuzzer configuration.

Benchmark Selection: The evaluation consists of ten diverse, open-source benchmark programs, common to fuzzing platform evaluations [15, 19, 24]. The benchmarks selected for the evaluation satisfy the following criteria:

- Extensive fuzzing: Each of the benchmark has been part of multiple peer-reviewed academic work or well fuzzed open-source fuzzing benchmarks. This enables cross comparison as well as making any bugs we uncover significant as they have not been discovered previously.
- Diversity: benchmarks vary in complexity and task.
- Rich variety of input formats: Table 3.1 shows that the benchmarks have a diverse set of input formats.

Benchmark	Input Format	Executable Size
bsdtar	tar	4.7 M
libpcap	pcap	2.4 M
gpmf-parser	mp4 (GoPro)	720 K
libbpf	bpf object	1.9 M
freetype	ttf	4.6 M
giftext	gif	232 K
zlib	zlib archive	260 K
libdwarf	ELF	2.8 K
c-blosc2	bframe	12 M
md4c	markdown	652 K

Table 3.1: Evaluation benchmarks.

3.1.2 ClosureX Correctness

To validate that ClosureX correctly restores program state between test cases, I show that irrespective of the order or number of test cases preceding a test-case-of-interest, it is equivalent to running that test case in isolation, with a fresh process. I divide this into two kinds of equivalence with fresh process execution:

1. Dataflow equivalence: program state updates must be equivalent for both ClosureX execution and fresh process execution.
2. Control-flow equivalence: program execution path must be equivalent for both ClosureX execution and fresh process execution.

To determine whether ClosureX correctly restores state, I use the entire test case queue accumulated during the fuzzing campaign of a target. This ensures that we cover all seen program behaviors during fuzzing. For each input in the queue, I compare two state snapshots from the target. The first snapshot is taken after executing the target in a fresh process. I take the second snapshot after executing, in ClosureX's persistent mode, the test case after 1000 iterations of test cases randomly selected from the queue. If ClosureX correctly restores state, then the two snapshots is identical.

Dataflow equivalence

There are three sources of program state that ClosureX must manage in our benchmark corpus. The program stack is controlled by well-formed function calls in ClosureX's harness, as well as the arguments passed to the target's main function is correct at every execution. This ensures that the stack is fresh for every iteration.

ClosureX frees all the stale heap objects after every iteration of the target. To verify that the dynamic memory of the target is correctly reset by ClosureX, I use valgrind [20], a debugging and profiling tool, to check for dynamic memory related inconsistencies like double-free or use-after-free. I find no such occurrences for any queue input in any of our benchmarks. I also compare the memory usage of the target during its execution to a fresh process's execution. Excluding ClosureX's own memory, the memory usage of the target is identical to a fresh process execution, showcasing that the dynamically allocated memory was freed correctly.

To verify global state equivalence, I compare the global state in the first snapshot (taken after execution of a queue input in a fresh process) to the second snapshot. If the two global states match, I establish dataflow equivalence between ClosureX's persistent mode execution and a correct fresh process execution. To handle non-determinism in program execution, which happens naturally even outside of fuzzing, I execute a target in a fresh process multiple times, and compare the global state for each execution. This captures ground-truth non-determinism: we consider the bytes that vary between these fresh executions non-deterministic. The reason for the existence of such non-deterministic bytes could be storing address of some data on the heap, or data generated or processed through the PRNGs.

All the targets have the same snapshot as a fresh process execution for all the fuzzing queue

entries. This establishes that ClosureX is dataflow equivalent to a fresh process execution.

Control-flow equivalence

I take a similar approach to dataflow equivalence to verify the execution path equivalence to a fresh process execution. For every queue test case, I first get the ground-truth path sensitive edge-coverage by running the target with the queue input in a fresh process. Then I iterate over ClosureX persistent mode and execute 1000 other randomly-selected queue inputs. That same process is then used to execute the test case of interest and its path sensitive edge-coverage is recorded. I then compare this coverage data with the ground truth coverage for the queue input. If these two coverages are equal, ClosureX persistent mode and fresh process execution are control-flow equivalent for this test case. I repeat the same process for all the test cases in the queue. I find that no queue entry leads to an inconsistency between ClosureX execution and a fresh process execution.

As in dataflow equivalency, we exclude test cases that induce a non-deterministic execution path on the target, i.e., their execution path differs across multiple fresh process executions. To determine the ground-truth non-deterministic queue test cases, I run each test case 10 times within a fresh process and collect the path-sensitive edge-coverage. The test cases which generate a discrepancy among these fresh executions are then flagged as non-deterministic. I only notice such non-deterministic queue inputs in the freetype benchmark. I suspect that a PRNG affected control-flow sequence like a conditional or a switch-case causes this behaviour in freetype.

I find no inconsistency in control-flow equivalence compared to fresh process execution. Given no deviation in data- or control-flow behavior, I claim that ClosureX ensures semantic consistency and test cases behave as if they were executed in a fresh process.

3.1.3 Test Case Execution Rate

Previous work shows that the test case throughput is the most significant part of efficient and successful fuzzing campaign [17]. ClosureX enables the target to isolate its initialization and tear-down procedures from the main fuzzing loop, thus improving the target execution speed and the test case throughput. In order to measure this performance gain, I compare ClosureX's persistent execution mechanism with AFL++'s forkserver execution mechanism, which is the fastest correct execution mechanism present in AFL++.

I run five 24 hour trials for our benchmarks and report the average number of test cases executed by both fuzzers, as well as the improvement ClosureX achieves over AFL++. Table 3.2 summarizes our results. On an average, ClosureX executes more than **3.5x** test cases than AFL++.

Benchmark	**ClosureX**	**AFL++**	**Speedup**	**ρ value**
bsdtar	379M	93M	4.09	0.0079
libpcap	565M	201M	2.81	0.0079
gpmf-parser	456M	193M	2.36	0.0079
libbpf	884M	212M	4.6	0.0079
freetype	493M	168M	2.94	0.0079
giftext	1118M	233M	4.79	0.0079
zlib	1117M	293M	4.00	0.0079
libdwarf	913M	265M	3.44	0.0079
c-blosc2	905M	213M	4.24	0.0079
md4c	538M	215M	2.50	0.0079
Average			3.53	0.0079

Table 3.2: Average number of test cases executed in 24 hours along with the statistical significance ρ value for the Mann-Whitney U-test. $\rho < 0.05$ indicates statistical significance [14].

3.1.4 Code Coverage Improvement

Given that ClosureX is equivalent to per-process execution, just much faster, ClosureX and AFL++ should cover the same portions of code (ignoring randomness), but ClosureX should do it much faster. Code coverage is an important metric of coverage-guided fuzzing. It guides the fuzzer to test deeper and more complex logic in the target. The coverage mechanism is same as AFL++, which has its own hitcount-based edge coverage collection implementation loosely based on LLVM's Sanitizer Coverage Guards.

Table 3.3 summarizes the average code coverage for ClosureX and AFL++ across benchmarks. On an average, ClosureX covers **7.8%** more edges. Mann-Whitney U-test indicates statistically significant improvement across five of our benchmarks.

Benchmark	ClosureX	AFL++	% Improvement	ρ value
bsdtar	18.13%	13.80%	31.4	0.031
libpcap	15.64%	15.22%	2.76	0.547
gpmf-parser	14.43%	13.82%	4.41	0.222
libbpf	6.45%	6.34%	1.8	0.015
freetype	16.30%	16.11%	1.17	0.150
giftext	27.57%	27.21%	1.32	0.003
zlib	36.22%	35.91%	0.87	0.111
libdwarf	5.20%	5.04%	3.34	0.020
c-blosc2	2.12%	1.76%	20.41	0.093
md4c	82.17%	82.08%	0.11	0.010
Average			7.8	0.079

Table 3.3: Edge Coverage percentage of ClosureX and AFL++ for different benchmarks, along with the ρ value for Mann-Whitney U-test. $\rho < 0.05$ indicates statistically significant results.

3.1.5 Time-to-bug Improvement

Even though faster execution and quicker code coverage demonstrate the efficiency of a fuzzer, its practical value comes from finding bugs—and finding them faster. In order to showcase the practical value of **CLOSUREX**, I evaluate AFL++ and **CLOSUREX**, on their speed and effectiveness in finding bugs. As a part of the performance and coverage evaluation, I find 15 0-day bugs in the latest source-code across 4 different benchmarks. Table 3.4 summarizes the bugs discovered. When **CLOSUREX** and AFL++ find the same bugs, **CLOSUREX** does so **1.9x** faster. **CLOSUREX** also finds bugs more consistently than AFL++, with 25% more trials finding bugs.

Benchmark	CLOSUREX	AFL++	Bug Type
c-blosc2	7148 (4)	11896 (2)	Null Ptr Deref.
c-blosc2	25358 (4)	12471 (2)	Null Ptr Deref.
c-blosc2	7442 (4)	18299 (2)	Null Ptr Deref.
c-blosc2	22957 (1)	16097 (1)	Null Ptr Deref.
gpmf-parser	798 (5)	1386 (5)	Division by Zero
gpmf-parser	43178 (3)	2365 (1)	Unaddressable Access
gpmf-parser	2368 (5)	20340 (5)	Unaddressable Access
gpmf-parser	33028 (5)	22721 (5)	Division by Zero
gpmf-parser	6144 (5)	6238 (4)	Invalid Write
gpmf-parser	14025 (2)	5493 (2)	Invalid Read
libbpf	61 (5)	91 (5)	Null Ptr Deref.
libbpf	515 (5)	552 (5)	Null Ptr Deref.
libbpf	2278 (5)	9964 (4)	Null Ptr Deref.
md4c	8489 (5)	22254 (4)	Memcpy with negative size
md4c	44714 (3)	54342 (2)	Array out of bounds access

Table 3.4: Time (sec) for AFL++ and **CLOSUREX** to find bugs (number of trials which found that bug) for each fuzzer.

0-days and bug disclosure All the bugs reported above in Table 3.4 are 0-day bugs. I

reported all the bugs to their respective developers. As of writing, bugs in `c-blosc2` and `gpmf-parser` have been fixed. The developers of `libbpf` acknowledged the bugs. I have not heard back from the `md4c` developers. I have been assigned 4 CVEs.

One of the bugs found during my fuzzing campaign is a null pointer dereferencing bug in libbpf. libbpf is a Linux kernel library used to load and run bpf object files into the Linux kernel in a sandboxed environment. The library tries to parse the relocation section of the crashing ELF object, and reads the data of relocation section, which returns a NULL pointer. When this NULL pointer is accessed in the library, it causes a crash.

Chapter 4

Discussion

4.1 Discussion

4.1.1 Supporting Other Operating Systems

While I have implemented ClosureX on the Linux operating system, our approach is operating system agnostic, and requires very little engineering effort to support other operating systems. To demonstrate this, consider the Windows operating system. ClosureX depends only on LLVM APIs and does not require any operating system level primitives to function. LLVM officially supports the Windows operating system and the ClosureX compile-time passes are readily ported to Windows. To extend ClosureX's support to Windows, other Windows specific memory management functions like `HeapAlloc` and `VirtualAlloc` will also need to be replaced in a similar way to `malloc` and `free`. To deal with `exit` calls in the target, Windows API function `SetThreadContext` and `GetThreadContext` can be used instead of the `setjmp` and `longjmp` calls used in ClosureX. My technique of dealing with file management calls in target are directly portable to Windows. Finally, my technique of resetting global state of a target by moving it to a new section is also directly portable to Windows. I expect similar or better results on the Windows operating system due to its lack of a copy-on-write based `fork` system call, which further deteriorates the performance of per-process fuzzing.

4.1.2 Supporting Other Architectures

ClosureX has been tested on Intel x86_64 architecture, but since neither my compile-time passes, nor my persistent-mode harness use any architecture-specific functionality, it can work out-of-the-box on other architectures like ARM and RISC-V. The only prerequisite for ClosureX to support an architecture is that LLVM has backend support for that.

4.1.3 Other Forms of Latent Program State

My benchmark targets only have four sources of program state that require handling: local memory (stack), global memory, dynamic memory (heap), and file descriptors. Thus, ClosureX currently has restoration support for these program states. While the benchmarks used in my experiments have only have the mentioned sources of residual program state, I acknowledge that there could be other such sources in more complex targets. For example, in multi-threaded processes, if the main thread spawns multiple threads, which continue executing after the main thread exits, the operating system kills all such spawned threads. However, in persistent mode execution, the main thread never exits but loops back and such child threads continue executing until they're killed manually. To deal with such multi-threaded programs, I use a similar approach to dynamic memory resetting: we track all the threads that are created during an execution and kill any threads that remain after the main thread completes execution (this would be when they are killed by the operating system).

4.1.4 Extending ClosureX

I open-source ClosureX and its artifacts on Github, and our design and implementation makes it compatible with all kinds of fuzzers [1]. I imagine the fuzzing community will

employ and extend ClosureX, using our foundational support for state tracking and reset patterns which work for most programs and utilise it for their target's special requirements.

4.2 Related Work

ClosureX is orthogonal to a lot of other fuzzing improvements, and can be combined together to achieve an even higher performance. The lightweight and easily extendable nature of ClosureX makes it complementary to different fuzzing improvement techniques.

4.2.1 Kernel-based Snapshotting

Kernel-based snapshotting is an execution technique of restoring the state of the process to a prior taken "snapshot". Work done in "Designing new operating primitives" implements a new system call `snapshot`, which stores the important aspects of program state like process pages, `brk` values and restores them after every execution [31]. This technique is also implemented as a Linux Kernel Module in AFL++ [6]. Implementing this technique for closed-source kernels like Windows or MacOS is infeasible. Even though Linux is an open-source operating system with a rich API for tooling, such a snapshot process still requires a significant engineering effort to maintain it for the ever-changing Linux kernel. The same issue is also present in AFL-LKM [6]. This makes maintenance and portability of kernel-based snapshotting challenging.

4.2.2 Less Instrumentation

Previous work shows that an effective way of increasing the program execution throughput is by reducing the number of instrumentation points in the program dynamically. This is

done using dynamic binary rewriting techniques. [17, 18, 19]. ClosureX is orthogonal to any such coverage collection mechanism and such systems can compilment ClosureX very well. Coverage guided tracing technique of UnTracer can be combined with ClosureX, and will lead to even better throughput for persistent fuzzing, similar to WINNIE [10]. I expect an even greater increase in test case execution by removing the tracing overheads from ClosureX.

4.2.3 Faster Execution for Multi-core Fuzzing

Multi-core fuzzing benefits largely from the improvements of ClosureX. Managing the system resource is a large part of parallelizing fuzzing efforts [9, 31, 34]. ClosureX instrumentation allows running parallel instances of the target program, which consume a manageable amount of resources, as well as require no external engine for monitoring, thus removing kernel and os-related contentions completely. Without ClosureX, resource starvation impacts the performance of all fuzzing efforts leading to worse throughput and performance. I expect to unlock scalar improvements with ClosureX instrumentation. Similar scalability improvements like dual-filesystem for better in-target testcase operations, as well as in-memory testcase logging for faster testcase sharing between fuzzer instances are compatible with ClosureX and will greatly improve the scalability of multi-core fuzzing [31].

Chapter 5

Conclusions

The fuzzer AFL++ is highly robust and offers a flexible infrastructure that enhances the efficiency of fuzzing methods. Advancements in reducing time spent tracing code coverage have further improved AFL's performance, leaving fuzzing-support-code execution time as a significant source of run-time overhead in fuzzing. We show that through compiler-level program instrumentation, it is possible to create naturally restartable programs that allow an entire fuzzing campaign's worth of test cases to execute within a single process. Doing so simplifies fuzzing, increases performance and bug finding speed dramatically, and makes high-performance fuzzing independent of operating system features.

Since source-available fuzzing is the most powerful fuzzing methodology, improving the performance is simple and easy to implement, and has a great potential for impact once implemented. The evaluation results indicate more opportunities for fuzzer improvements through more advanced, fuzzing-specific compiler analysis.

Bibliography

[1] Anonymous Anonymous. URL anon.ymous.

[2] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741, 2015. doi: 10.1109/SP.2015.50.

[3] Chen Chen, Han Xu, and Baojiang Cui. Psofuzzer: A target-oriented software vulnerability detection technology based on particle swarm optimization. *Applied Sciences*, 11(3), 2021. ISSN 2076-3417. doi: 10.3390/app11031095. URL https://www.mdpi.com/2076-3417/11/3/1095.

[4] Andrea Fioraldi. Frida api fuzzer, 2020. URL https://github.com/andreafioraldi/frida-fuzzer.

[5] 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.

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

[7] Erol Gelenbe. A model of roll-back recovery with multiple checkpoints. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 251–255, Washington, DC, USA, 1976. IEEE Computer Society Press.

[8] Google. Afl, 2017. URL URL:https://github.com/google/AFL.

[9] Google. cluster-fuzz, 2022. URL https://github.com/google/clusterfuzz.

[10] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. Winnie : Fuzzing windows applications with harness synthesis and fast cloning. In *NDSS*, 2021.

[11] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. 2018. doi: 10.48550/ARXIV.1808.09700. URL https://arxiv.org/abs/1808.09700.

[12] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi: 10.1109/CGO.2004.1281665.

[13] LLVM. Libfuzzer – a library for coverage-guided fuzz testing., 2022. URL URL:https://llvm.org/docs/LibFuzzer.html.

[14] Patrick E. McKnight and Julius Najab. *Mann-Whitney U Test*, pages 1–1. John Wiley & Sons, Ltd, 2010. ISBN 9780470479216. doi: https://doi.org/10.1002/9780470479216.corpsy0524. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470479216.corpsy0524.

[15] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: An open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1393–1403, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3473932. URL https://doi.org/10.1145/3468264.3473932.

[16] Microsoft. Onefuzz, 2022. URL https://github.com/microsoft/onefuzz.

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

[18] 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.

[19] 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, page 351–365, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384544. doi: 10.1145/3460120.3484787. URL https://doi.org/10.1145/3460120.3484787.

[20] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250746. URL https://doi.org/10.1145/1250734.1250746.

[21] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. *Program Comprehension and Code Complexity Metrics: A Replication Package of an FMRI Study*, page 168–169. IEEE Press, 2021. URL https://doi.org/10.1109/ICSE-Companion52605.2021.00071.

[22] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[23] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157, 2016. doi: 10.1109/SecDev.2016.043.

[24] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. Vancouver, BC, August 2017. USENIX Association.

[25] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 244–256, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384599. doi: 10.1145/3460319.3464814. URL https://doi.org/10.1145/3460319.3464814.

[26] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Scotts Valley, CA, 2009. ISBN 144141276X.

[27] The Clang Team. Sanitizercoverage, 2019. URL URL:https://clang.llvm.org/docs/SanitizerCoverage.html.

[28] Peach Tech. "peach fuzzing platform", 2021. URL https://www.peach.tech/.

[29] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594, 2017. doi: 10.1109/SP.2017.23.

[30] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Comput. Surv.*, 52(3), jun 2019. ISSN 0360-0300. doi: 10.1145/3316415. URL https://doi.org/10.1145/3316415.

[31] 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, page 2313–2328, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134046. URL https://doi.org/10.1145/3133956.3134046.

[32] M. Zalewski. Fuzzing random programs without execve(), 2014. URL https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html.

[33] M. Zalewski. american fuzzy lop (2.41b), 2017. URL http://lcamtuf.coredump.cx/afl/.

[34] M. Zalewski. Technical whitepaper on afl-fuzz. 2017. URL URL:https://lcamtuf.coredump.cx/afl/technical_details.txt.