Closure: Transforming Source Code for Faster Fuzzing

Ian G. Paterson

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

Masters of Science
in
Computer Science and Applications

Matthew D. Hicks, Chair
Kirk W. Cameron
Eli Tilevich

May 9th, 2022
Blacksburg, Virginia

Keywords: AFL, LLVM, Fuzzing, Code Modification

Closure: Transforming Source Code for Faster Fuzzing

Ian G. Paterson

(ABSTRACT)

Fuzzing, the method of generating inputs to run on a target program while monitoring its execution, is a widely adopted and pragmatic methodology for bug hunting as a means of software hardening. Technical improvements in throughput have shown to be critical to increasing the rate at which new bugs can be discovered time and time again. Persistent fuzzing, which keeps the fuzz target alive via looping, provides increased throughput at the cost for manual development of harnesses to account for invalid states and coverage of the programs code base, while relying on forking to reset the state accrued by looping over the same piece of code multiple times. *Stale state* can lead to wasted fuzzing efforts as certain areas of code may be conditionally ignored due to a *stale* global. We propose Closure, a toolset which enables programs to run at persistent speeds while avoiding the downsides of *stale state* and other bottlenecks associated with persistent fuzzing.

Closure: Transforming Source Code for Faster Fuzzing

Ian G. Paterson

(GENERAL AUDIENCE ABSTRACT)

The process of program testing to find bugs is becoming increasingly automated. A current method called "Fuzzing", is a widely adopted means for finding bugs and is required in the life cycle of program development by major companies and the US Government. I look at current improvements in fuzzing, and expand the use case of the cutting edge method called persistent fuzzing to a wider array of applications with my tool Closure. With Closure, fuzzing practitioners can experience faster fuzzing performance with less manual effort.

Dedication

I dedicate this Thesis to my Parents, Beth and Eric Paterson, and my Fiance Erin Kelly, who have supported me every step of the way.

Acknowledgments

I would like to thank my Advisor Dr. Matthew Hicks, and my friends and lab mates of the FoRTE Research Group for their help and feedback throughout the last year.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern applications are continuing to grow at an alarming rate. As code bases continue to grow to fit the increasing demand for functionality in computing systems, so too does the technical overhead required to properly test and validate these code bases [18]. With this growing complexity, companies such as Microsoft and Google are turning to fuzzing to increase software hardening over traditional methods [7, 8, 14].

Fuzzing, an automated means for discovering bugs in applications has become the industry standard for security hardening from commercial B2B systems to mission critical DoD applications. In short, fuzzing is the process of generating input based on a given seed and running the input against a program to find bugs. Recent research has focused on the field of fuzzing known as coverage guided fuzzing. Coverage is shown to improve the richness in test generation, but quality tests have shown to be less important then high throughput of test cases in practice [10, 15].

Recent work focuses on reducing coverage tracing overhead [16, 17], to provide high levels of throughput. I see that when doing this one can achieve highly efficient fuzzing. This methodology is focused towards binary only fuzzing, and does not provide the same profound improvement in source available fuzzing scenarios. Source available fuzzing as of recently has started to stagnate, reaching a plateau of performance improvements. Focusing on process creation and tear down, improvements such as the ForkServer aim to reduce the overhead of spawning new instances by exploiting an initialized binary to fork new instances, opposed to

the traditional method of executing a new instance per test case. In process fuzzing, which aims to keep the program alive within a loop avoiding overhead and limitations associated with system calls altogether, relies on manual harness, and a specialized execution control and monitoring system which monitors wasted fuzzing efforts. This shows that rate of test case execution is most important for fuzzing, as long as the fuzzer minimizes executions which are semantically inconsistent with the source code.

To address these concerns for persistent source available fuzzing, I propose Closure, a tool set consisting of an agnostic stub harness and a suite of LLVM passes to leverage compile time instrumentation to enable self managing applications. Closure fills the gaps in persistent instrumentation enabling applications to run in a loop while avoiding memory leak, stale state, file access errors, and exits which lead to Semantically Inconsistent Execution (SIE) and other performance impacting overheads.

To evaluate the performance of Closure, I build 10 commonly fuzzed programs in a persistent manner with Closure tooling and compare it to the leading non persistent system, AFL++'s ForkServer. I look at the correctness of each run, ensuring that crashes are accurate to the program and not due to persistent fuzzing. I observe that with Closure, applications execute up to 600% more test cases within the same time frame. I see that by increasing the number of test cases executed while managing the programs state, I successfully cover more of the program map, while eliminating false positives due to SIE simplifying the triage process.

In this paper I show that I have contributed the following.

1. A set of LLVM passes and a abstract stub harness with which a program is able to run in AFL++'s persistent mode without experiencing Semantically Inconsistent Execution (SIE).

2. A modified ForkServer, called LoopServer, which manages the flow of test cases to the

application running persistently without shutting it down after so many iterations.

3. A new method for replacing fopen and fopen64 with fmemopen enabling new methodology for implementing shared memory test cases in complex applications.

4. Provide a comprehensive system for automatically harnessing applications to be fuzzed in a stateful and persistent manner.

Chapter 2

Background

Source available fuzzing is the most efficient means for fuzzing applications in terms of executions per second when fully instrumented. Source available fuzzing achieves high throughput while maintaining coverage accuracy due to the implementation of compiler level instrumentation. In fuzzing, it is shown that the number of test cases run is more effective then better test generation regardless of fuzzing platform [10, 15]. Binary only fuzzing has recently started to overtake source available fuzzing by manipulating the instrumentation and the way that binary's are dynamically rewritten. I look toward compiler driven code modification to instrument binary's at compile time to improve throughput of test cases.

2.1 LLVM, LLVM IR, and Compiler Optimizations

LLVM a compiler infrastructure built around the C language, provides a system to write complex compiler optimizations and program analysis passes [11]. These passes allow developers to implement complex tooling into programs at the compiler level allowing for a language agnostic toolset. LLVM works by converting the source code into a low level virtual machine representation of the compiled program in the form of LLVM IR files, or LLVM bytecode (.bc files). These files represent the compilers intermediate representation which defines the semantic definition of the application abstracted from its source code. This intermediate representation mimics common assembly code with the trade off for infinite virtual

registers. The way that virtual registers are utilized is a feature of Single-Static Assignment (SSA), which is a methodology for structuring code during parsing to ensure every instance of a register is used only once. This allows optimizations to be easily applied as most optimizations rely on SSA as a requirement in traditional compilers. Additionally, developers are able to implement tooling in the form of LLVM optimizations (opt) passes written in C++, which process and modify either LLVM bitcode and LLVM IR. This allows for simple code modification and large scale code analysis.

LLVM code modification is now the state of the art methodology for implementing instrumentation and tooling for fuzzers [5, 6, 12, 20, 21, 24]. LibFuzzer and AFL++ both implement LLVM level instrumentation in the form of pcguard instrumentation [5, 12, 31]. Pcguard is used to measure the state of the program at certain points in execution, and is used for calculating the code coverage during execution to feed run time information to analysis and test generation. Other forms of instrumentation are in the form of sanitation and optimization [20, 24]. I extend the existing instrumentation of AFL++ with my own abstract stub harness, and additional LLVM passes for performance increasing code transformation.

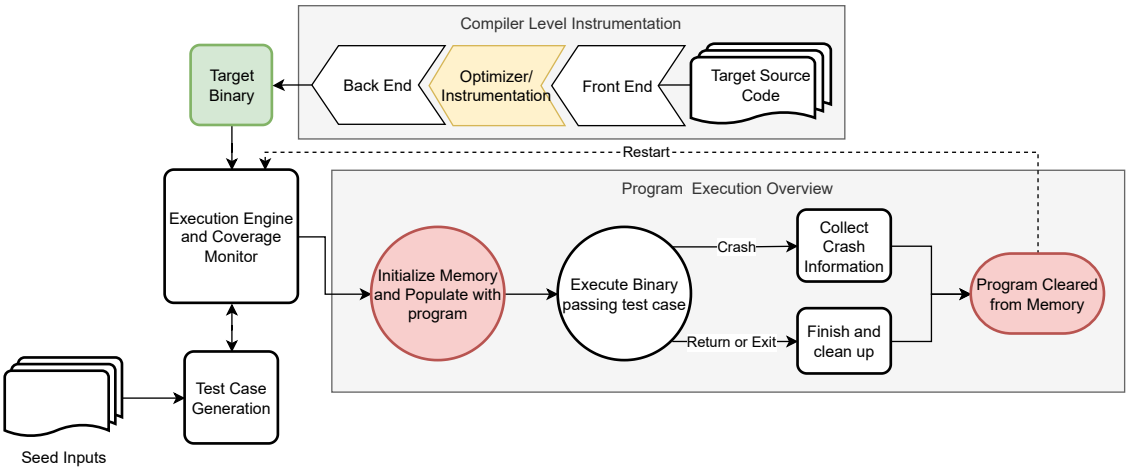


Figure 2.1: Basic example of the fuzzing loop highlighting program execution and source code instrumentation. Red colored steps symbolize points of execution which impact overhead.

2.2 Fuzzing Overview

Fuzzing, at its core, is the process of feeding inputs into an application or harnessed library, and measuring the result of its execution. The result of its execution gives us insight into the behavior of the application with the end goal being to find and triage bugs which would typically be non trivial to solve. The process of fuzzing is broken down into 4 main components. [31]

1. Program coverage instrumentation
2. Program Execution and Monitoring
3. Test Case Generation
4. Feedback Analysis and Triage

In practice, applications are fuzzed in either binary only or source code available methods. These 2 different modes means that I have 2 different ways to measure the execution of the application. Binary only fuzzing requires analysis of the binary programs to disambiguate structure and inject coverage tooling at the basic block level utilizing binary rewriting [26]. This is done traditionally in the form of callbacks and system signals at the entry to each basic block [15] to determine coverage by either block coverage, and inferred edges [17]. Another way of performing binary fuzzing is to emulate the architecture using a tool such as QEMU, and analyzing the execution of the program through process tracing [32]. This method gives increased precision, but is typically slower then binary rewriting [4] while enabling richer analysis.

Test generation, is the process of taking an initial seed corpus and using it to generate similar inputs. The goal of generating these inputs is to explore new portions of the fuzzed

program by triggering different conditional statements in the code based on input. The program instrumentation previously mentioned is used to generate a file structure similar to a map which stores information based on the execution of the currently fuzzed program. The process of generating new inputs based on the current program coverage makes up the field of coverage-guided fuzzing. Test case generation falls into 2 main styles generational [3, 25], and mutational [2, 5, 6, 8, 12, 31].

The most optimized way of performing code coverage is compiler level instrumentation utilizing compiler optimization style code modifications to inject specialized coverage tooling either via GCC [23] or LLVM [11, 21, 24]. When doing this, instrumentation is applied inline within the assembly code enabling high efficiency code coverage. Inlining instrumentation allows us to increase granularity of coverage monitoring in a programatically optimized manner. A common example of inlined instrumentation is using shared memory bitmaps where coverage is stored in a memory segment accessible by both the monitor and the fuzzed target. At runtime, tracing of the application is faster in source available systems in comparison to closed source. Fuzzing source code is most frequently the common use case of fuzzing within industry. Microsoft, Google, and DoD all require fuzzing within their SDLC for validating the security of their tools and applications [7, 8, 14].

2.3 Fuzzing Performance

To improve the performance and throughput of test cases, programs are modified or harnessed to improve the rate at which the fuzzer explores the targeted code. Compilers are frequently used to modify the code for optimal fuzzing effort[19, 22]. To improve performance of pre-compiled binaries, tracing overhead is addressed by dynamically instrumenting the basic blocks of the program by removing instrumentation at runtime to improve throughput

while maintaining the coverage, called coverage guided tracing [15, 16, 17]. Source available and API fuzzers improve throughput by reducing the number of executions by looping over the target code areas as seen in LibFuzzer harnesses [12]. LibFuzzer harnesses are shown to be capable of high throughput and efficiency, but designing new harnesses which effectively explore the entirety of a program is an arduous and manual task. Work into automating this process has shown to be successful but require initial library or program consumers to generate effective fuzz harnesses [9].

As seen in Figure 2.1, execution is managed by an external program responsible for starting new iterations of the program. In the basic fuzzing implementation, this would be simply executing the program again after the program has finished via `fork()` to create a new process instance, and `execve()` to launch a new instance of the fuzz target. In the case of AFL, the tool which controls the execution of the target program is called the fork server (which I will explore further in section 2.4). The goal of an execution engine is to not just relay coverage information between the test generation processes, but also to pass the new inputs from the test generation into a new instance of the program. The final job of the execution engine is to monitor the system as it runs accounting for memory usage, cpu usage, timeouts and crashes. Frequently, this is not OS specific, some systems with less aggressive garbage collection or limited resources (IE file descriptor limits) will experience OS bottlenecks such as memory leak or file access errors at a higher rate then others. This is remedied by use of a controller which monitors system resources and keeps the fuzzing target within certain resource allocations (as seen by the ForkServer with a target memory limit) [5, 6].

Key Insight: Process construction and tear down costs are resource expensive and increase latency. Finding ways to reduce the cost of process construction is an optimization that applies across all scenarios.

2.4 AFL design and AFL++

AFL++ is the community maintained branch of the AFL fuzzer [30] which combines multiple state of the art breakthroughs in the field of fuzzing [5]. AFL [6, 31], focused on creating an ecosystem of "hacks" for fuzzing that was both robust and easily extensible. AFL breaks down its fuzzing into methodology matching the process flow shown in Figure 2.1. AFL ties this flow into a robust system with improvements to the rate of program execution by implementing their fork server. The goal of the fork server is to eliminate overheads associated with starting a new instance of a program, by relying more heavily on the `fork()` command and better instrumentation. The core idea is to start the target binary, suspend its execution before running, and then pass new test cases into a newly forked instance of the suspended task. Doing this allows the system to exploit Copy on write to spawn new instances of the fuzzing target from a pre-started instance eliminating the call to `execve()` [29].

On top of creating new instances of the fuzz target, the fork server also contains the necessary tooling to handle new test cases from the test generation portion of the fuzzer. Using IPC, it makes sure that the input file is correctly set up before starting a new iteration of the fuzzing loop. Throughout execution, the program is logging explored points in code by writing coverage data to a shared memory bitmap. The usage of the bitmap in this fashion is called coverage tracing and it holds the basic edge and basic block information of the current execution. This bitmap is then used by test generation to determine which test cases to keep in the corpus in the event that a new portion of previously unexplored code was found.

AFL++ performs its test generation by congregating years of research into fast and optimized generation and mutation methods, program performance optimizations, and coverage

instrumentation improvements across all modes of fuzzing [1, 28]. Whether test cases are generated based on code coverage or random mutation, throughput of test cases is not a common overhead of AFL (unless dealing with large seed corpus or files over 1 KB). AFL manages to provide test cases at a rate that allows for thousands of executions per second. With this test generation system in place and separated from program execution, designing execution flows that are accurate quickly becomes the bottleneck when using AFL++. In process fuzzing addresses this problem.

2.5 In process fuzzing

LibFuzzer, LLVM's in process fuzzing tool, [12, 21] uses compiler driven instrumentation and manually designed library harnesses to provide whats known as persistent fuzzing. Persistent fuzzing, unlike traditional fuzzing, aims to not let the process ever exit to eliminate the need for the fork command. Since this works in fuzzing, it also makes file IO unnecessary, allowing for direct data input through shared memory cutting out expensive overheads. These combined greatly improve the rate of execution in comparison to forking new iterations and file input. LibFuzzer, is the current state of the art for source-code fuzzing. Despite its speed, implementation of LibFuzzer is complex and requires a deep understanding of the source application or library to construct effective fuzzing harnesses. This method does not account for state, and requires the developer to create a system that explores code in a complex way while also accounting for state in between iterations. This leads to memory leak as well as state errors that are not possible during normal execution.

AFL++ provides another implementation to an in process and in memory fuzzer in the form of LLVM persistent instrumentation mode. This mode is intended for library fuzzing similarly to libfuzzer but allows for the developer to create a test harness which loops a target

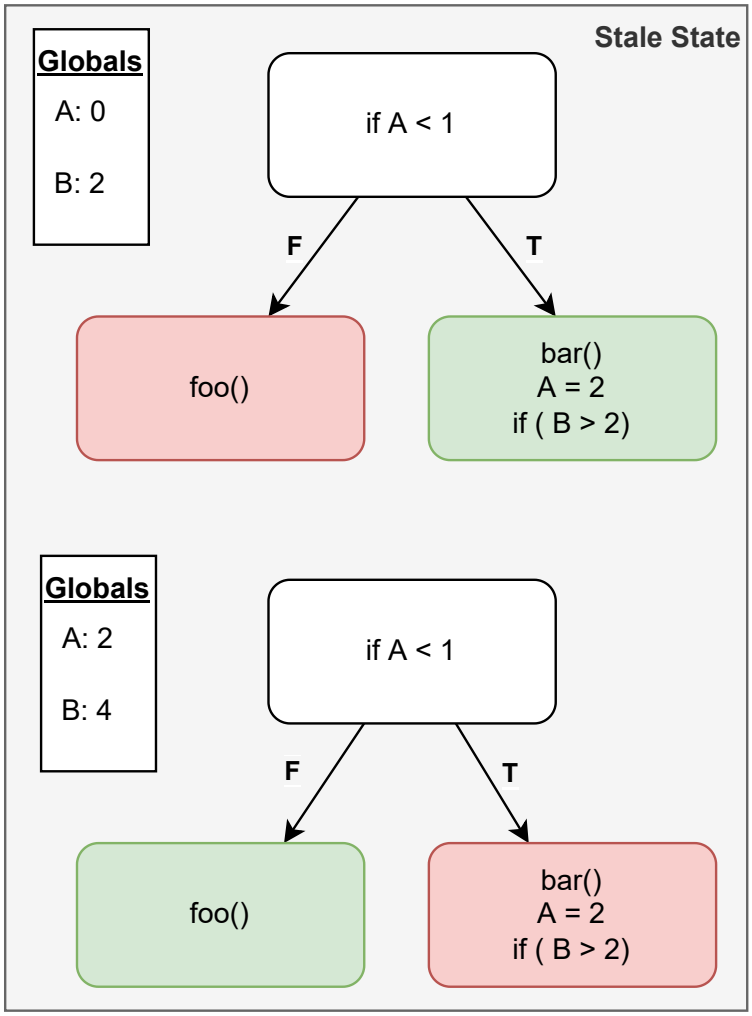


Figure 2.2: Example of stale state scenario, in this situation, after the first iteration, execution is locked out of the true branch which could hold deeper points of code to explore with a certain input and without accounting for this fuzzing effort is wasted.

piece of code a set number of iterations before restarting the harness. This methodology does provide increased throughput, however it does not account for complex states that applications have, and leads to increased false positives and Semantically Inconsistent Execution (SIE). To prevent this from happening a developer needs to set a lower number of iterations [5], but this comes with the trade off of increased fork occurrence which reduces throughput. The impact of semantically inconsistent execution comes in the underlying idea of *stale state* (shown in Figure 2.2). *Stale state*, is the scenario where a program has a logically defining global which is set on an initial iteration, locking out semantically consistent execution from occurring at any subsequent iteration, changing the effectiveness of running a single test case. I show how this manifests in an application such as NASM in section 3.

Solution: With Closure, I eliminate the lost fuzzing effort associated with Semantically Inconsistent Execution, and enable persistent fuzzing with semantic accuracy without requiring a developer to spend additional time creating harnesses.

Chapter 3

Motivation: Common Overheads in Persistent Fuzzing

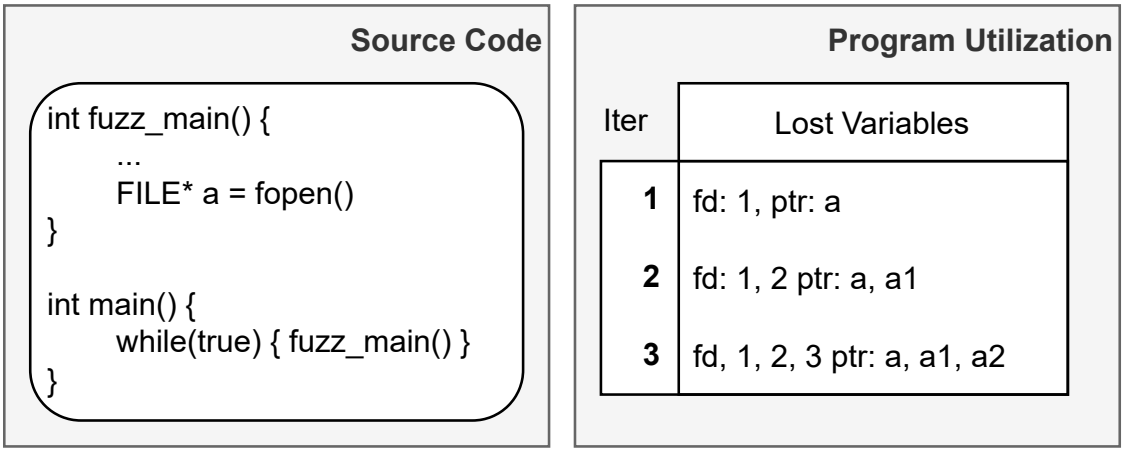
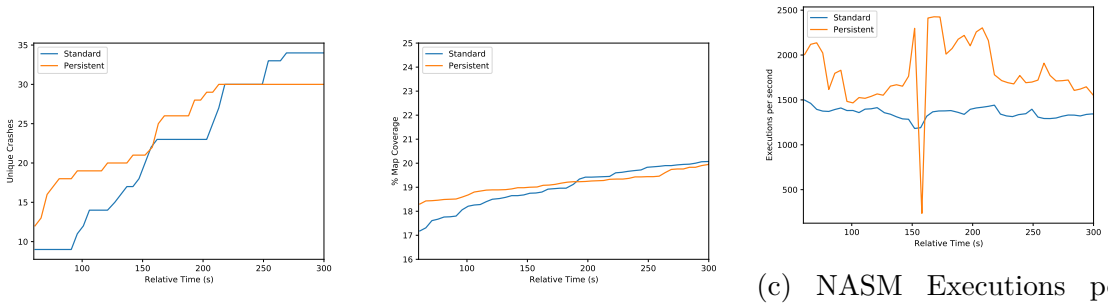


Figure 3.1: Example of Lost variables and file descriptors. This is a common scenario to have occur when a program is running persistently. Fopen, returns both a file pointer, and generates a system level file descriptor. If the file pointer or file descriptor are not closed, both become dangling variables at the end of the fuzzed code's execution.

Persistent fuzzing is the most effective means for performing high throughput fuzzing by removing the reliance on the `fork()` call. `Fork()` conveniently resets state of the program in between iterations. This means that one can achieve high throughput in persistent mode, but lose useful state resetting which ensures semantically consistent execution. I explore the correctness of this methodology to isolate common issues that arise when keeping complex applications alive through multiple iterations. From a theoretical viewpoint, I predict common scenarios which could lead to Semantically Inconsistent Executions (SIE).



(a) NASM unique crashes (b) NASM Map Coverage (c) NASM Executions per second

Figure 3.2: NASM 5 minute test to discover underlying issues with persistent fuzzing of complex applications.

- **Stale state** due to logically significant globals (2.2)
- **Memory Leak** due to pointers being malloc'd without freeing (3.1)
- **File descriptor constraints** due to files being opened with being closed (3.1)

I expect that in the common scenario, applications will see one if not all four of these common bottlenecks. This does show that the bottlenecks would be inconsistent across all applications and some may experience issues more frequently then others depending on the nature of the machine.

To explore the nature of running programs persistently I designed a system to harness standard C/C++ applications with AFL++'s persistent tooling [5]. To determine if the other semantics breaking scenarios exist, I ran the program NASM with my new harness until I observed signs of semantically inconsistent execution. Once I determined this, I ran the standard fuzzed version for the same period of time to see if the rate of execution was increased and if the persistent instance showed other signs of invalid fuzzing effort. NASM was chosen due to its known high complexity and wide acceptance by the fuzzing community as a valuable benchmark. Within this 5 minute run, Figure 3.2, NASM was able to run 170% more executions when in persistent mode, but also found more then

2317x more crashes. Despite the number of unique crashes being higher in the standard instrumentation, the total number of crashes that occurred in the persistent tooling was over 126,000, making up more then 22% of all run test cases as opposed to the .02% which is seen in the standard instrumentation. This shows applications when ran in a persistent mode frequently experience state changes which caused for false positive crashes and inconsistent state transfers from the non-persistent mode of fuzzing. This is further corroborated by the lower map coverage seen in the persistent version despite the increased rate of execution 6.2c.

I explore this behavior across multiple open source applications and libraries, and observed memory leak and file descriptor issues which break the application into these stale state scenarios which increases the rate at which I experience semantically inconsistent executions. Despite this, AFL++'s stability metric was frequently not indicative of these issues as programs such as cert-basic persistently run maintaining 100% stability, while accruing large number of timeouts and crashes. Looking deeper, I notice that these occurrences of stale state are not trivially isolated as complexity varies despite stemming from the same root causes. Stale state issues such as file descriptor overheads are frequently due to underlying constraints within the Operating System which the fuzzer is running in. In the event that the program starts a new iteration before a file descriptor was properly cleared, the operating system will eventually stop allocating additional descriptors leading to increased errors. In the common case of a 1000 iteration persistent run on a system with a 64 file descriptor limit will experience stale state and wasted fuzzing efforts after less then 1% of the total runtime of the persistent iteration. With this in mind, I observe that stability as a metric does not reflect the effectiveness of the fuzzer requiring more robust methods for determining the correctness of the crashes being generated.

Chapter 4

Closure

To solve the issues of Semantically Inconsistent Execution (SIE), I propose Closure. Closure transforms source code during compilation so that it manages its own state while maintaining the performance of persistent fuzzing, eliminating the prevalence of SIE. With Closure I produce self-managing programs.

4.1 Closure Self-Managing Programs

As seen in Figure 4.1, common programs consist of a `main()` function which holds the target code to be fuzzed. When being ran by the fork server, the program runs as intended with the process being torn down between each iteration. As seen previously, when run persistently, skipping this tear down or resetting step leads to SIE. With Closure, applications are capable of running indefinitely without experiencing SIE, giving the throughput improvements of persistent fuzzing with the accuracy of state from fork execution. The goal of this design is to shift the reliance of fork based resetting to an in-process methodology.

Closure works by applying 3 transformations on top of the target source code. These transformations are as follows harnessing, instrumentation, and function swapping.

Target Harnessing is the process where I attach my stub over the targets `main` function injecting the entirety of Closures tooling into the target. Making use of Link Time Opti-

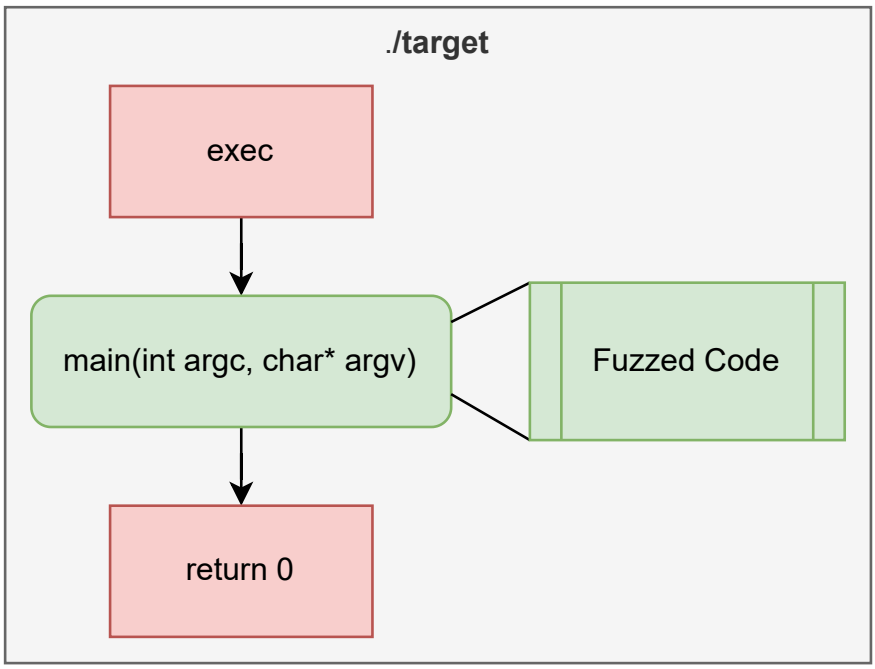


Figure 4.1: Standard execution. A good execution will return 0, a crash will occur inside of fuzzed code or an exit will be experienced in fuzzed code.

mization (LTO) functionality in modern compilers LLVM or GCC, I link together multiple complex objects into a single complex object. This allows us to perform large scale optimizations and transformations over the entire program. While the target program is in the IR form, I isolate the `main` function. Once found, I rename `main` → `start_main` so that it does not interfere with the harnesses main function, allowing us to link the target program to the harness. This works because I rename the original main function to a known function name that I define in the harness. This then becomes the only traced call inside of the main fuzzing loop of the harness (4.2 ①).

Within this fuzzing loop I design a set of logic which enables us to point to an initial state and know if I have reached this point in execution via a ”would be exit”, or by normal program exiting. With this I control the flow of test cases and communicate with the fuzzing engine when I are ready for the next test case (4.2 ②).

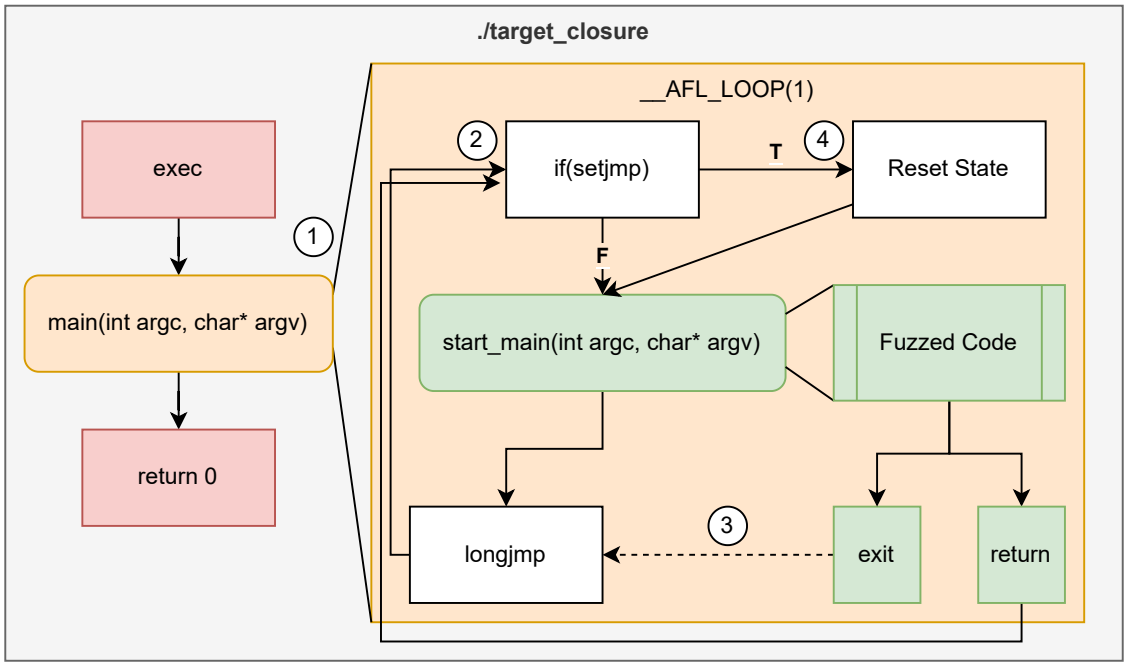


Figure 4.2: Closure execution. The target programs execution (in green) remains consistent from Figure 4.1, with Closures additional tooling attached on top of the target. If the setjmp has been reached by means of longjmp it will reset any stale state. Normal execution should not exit unless the practitioner forces the application to close or a crash occurs.

Target Instrumenting, the program is instrumented using global analysis to instrument global resetting and isolating the jump buffer. Once the program has been harnessed, I perform a control flow analysis to determine where the programs exits are. Using the harness's jump buffer I instrument each exit in the program with a call to longjmp (4.2 ③) which allows us to keep the program in memory instead of closing and relying on an additional fork to start the next test scenario. Using setjmp/longjmp also gives us program counter resetting, and stack resetting helping maintain program state. It is trivial to put the program inside of a loop but an exit will still stop execution regardless of this modification. Using longjmp and setjmp in this manner allows us to avoid all scenarios which are not indicative of crashing from triggering process tear down and creation.

To instrument global resetting I design a global undo logging system that is injected via compiler transformations. Using the result of a global analysis and a control flow analysis I isolate the location right before the target code is fuzzed, and which globals are non-constant. I do this because instrumenting globals which are static is not necessary as their values will not change. I then generate shadow copy's of each non-constant global with the same initialization data using compiler transformations to inject new globals. I then instrument the top of the basic block which calls start_main, to add load and store instructions to reset each global back to its initial state stored inside of the shadow copy. By doing this I eliminate any needed global logging at the start of each iteration while maintaining an in process method to efficiently reset global state between test cases.

Function swapping, functions (IE: `malloc`, `free`, `fopen`) are replaced with the respective functions linked in with the harness to provide in program pointer and file tracking. With this transformation I am able to add lightweight tooling at the source code level on top of existing functions so I add tracking logic to the target program. To monitor an open file descriptor I look for the current input file coming from AFL, and clone it as a sentinel value

before returning the file pointer from fopen. I then link this cloned file descriptor to the original so they close at the same time regardless of which one is actively closed or where they are closed in execution. I then check in between iterations if the sentinel is still alive, which I then close from within my harness at the reset state step (4.2 ④).

To achieve dynamic memory mapping, I follow the design from file descriptors, however I monitor pointers by storing them in a global hash map as opposed to a global sentinel. For malloc and calloc, I call the generic function, and insert the returned pointer into the hash map before returning it to the original call site. Realloc not only allows one to change the size of the pointer, but also changes the pointer location. To handle this I first remove the pointer from the logging hash map and then replace it with the newly generated pointer in the hash map before returning the new pointer back to the call site. When free is called, I remove the freed pointer from the hash map before freeing the actual pointer. In the event of an exit or poor programming, pointers may persist through iterations. To ensure memory is completely cleared in between each iteration I loop over the hash map to clear all remaining allocated pointers before starting the next iteration (4.2 ④).

With this design, a program is able to run persistently within the fuzzing loop while managing its own resource utilization. This shifts the overhead of resource management from the fuzzer itself to the fuzzing target and replaces fork based state resetting for in process state resetting via Closure.

Chapter 5

Closure Implementation

I implement Closure as a series of 5 LLVM passes, a C stub harness with the required cover functions for the common case, and a script for processing the resulting IR file of compilation via gclang. The series of passes is laid out in Table 1. These passes are module passes which perform whole program modifications which do not break the semantics of the application.

LLVM Closure Passes		
Pass Name	Pass Functionality	Req?
Main Pass	Renames main	Yes
Heap Pass	Heap Management	No
File Pass	File Sentinel	No
Global Pass	Global Undo Logging	No
JmpExit	Long Jump from Exits	Yes

Table 5.1: OPT passes to use on programs to attach Closure Tooling. Some applications or harnesses are not going to experience these concerns and not all are necessary for achieving Semantically Consistent persistent execution

5.1 Closure pre-processing and Compilation

I show the Closure compilation flow in Figure 5.1. Using `gclang` a front end for LLVM from the GLLVM suite[13], I compile the program as usual passing any recommended optimizations or linked libraries. Using gllvm produces the full executable of the program which then is used to generate the targets LLVM bit code, including all compile time linked libraries giving us a full IR representation of the application via `get-bc` (another GLLVM

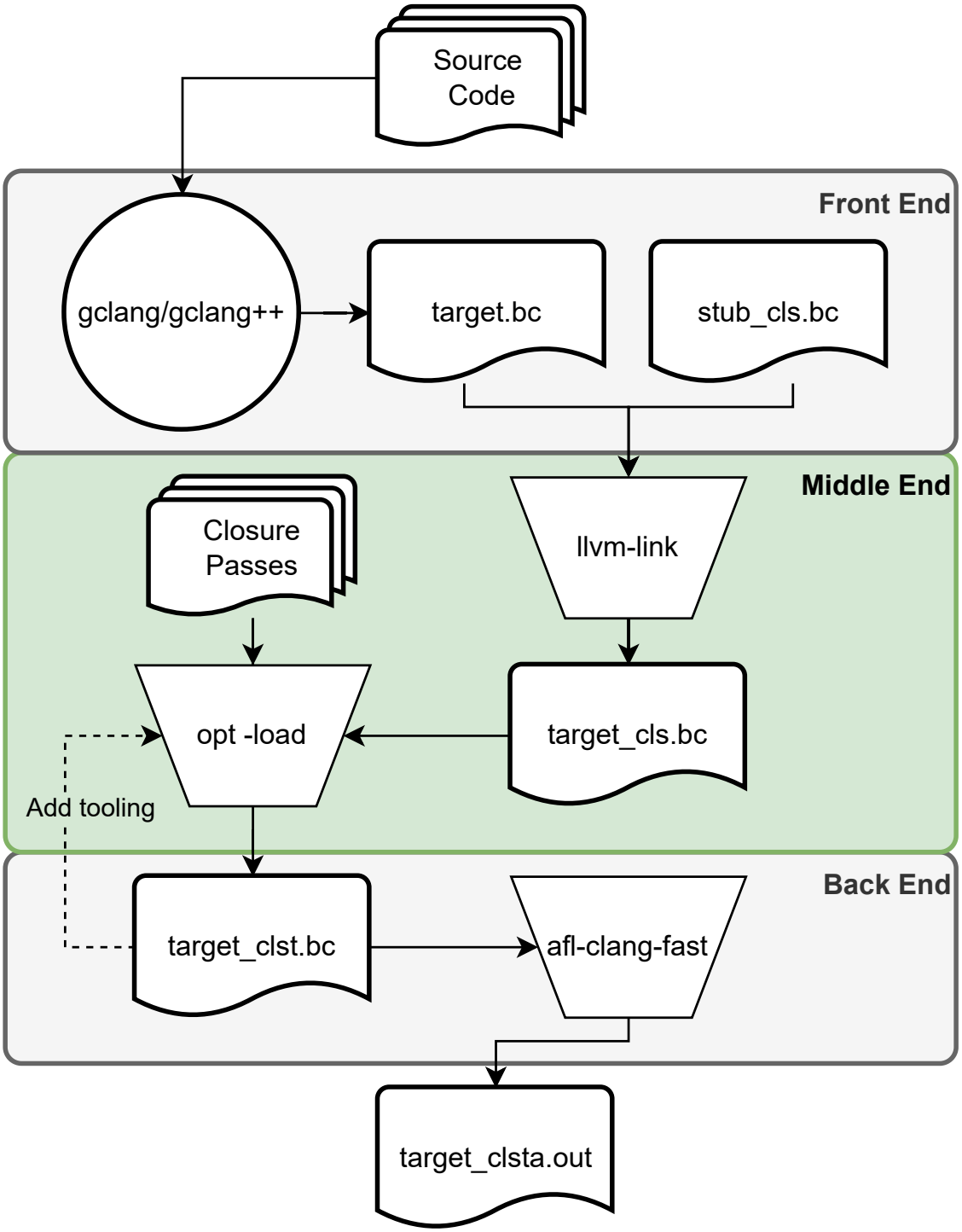


Figure 5.1: Closure compilation flow. The front end is when Closure's stub is harnessed over the target program. The middle end is where Closure links the stub, and modifies the code to use the linked utilities in place of system calls. The back end is responsible for injecting AFL's code coverage tooling into the target while avoiding instrumentation of the harness

tool). Next I perform the harnessing step, which uses a LLVM module pass to rename the `main()` function of the target programs bit code, and attaches my stub harness onto the program via `llvm-link`. Once the stub is linked, I run `opt` with the remaining passes to replace the targeted C function calls with the functions found in the stub harness. Finally each program is recompiled via `afl-clang-fast` with the `AFL_LLVM_DENYLIST` file telling it to avoid instrumenting the stub harnessed tools so I only fuzz the fuzz target. If I do not exclude the harness's source code from the code coverage instrumentation, then I will experience significant stability drops which impedes the process of determining which test cases triggered exploration of a previously unexplored portion of the programs coverage map. This will also increase the map size of the application leading to inaccurate map coverage metrics.

5.2 Stub Harness

Listing 5.1: A simplified Stub example

```
1  #include <stdio.h>
2  // all other includes here
3  jmp_buf __jmp_buf;
4  __AFL_FUZZ_INIT() :
5  //tooling and cover functions go here
6  int main(int argc, char* argv) {
7      // initialize deferred forkserver and shm if used
8      __AFL_INIT();
9      while(__AFL_LOOP(1)){
10         if(setjmp(__jmp_buf)) {
11             // recovered from exit scenario
12         }
13         else {
```

```
14              // run renamed main from target program inside of the afl loop
15              __start_main(argc, argv);
16          }
17          // Reset state
18      }
19      return 0:
20  }
```

Listing 1 outlines the general structure of my Closure stub. This stub consists of the AFL instrumentation which would be built into a typical library harness which gives us access to LLVM persistent mode, and its shared memory functionality. The stub also consists of a variety of cover functions which are explained in their respective sections coming up. With these functions I am attaching additional program functionality to the application, which are injected by LLVM passes over the original function calls. This system allows us to attach Closures tooling in a manner that does not impede the current functionality of the existing AFL architecture.

I modify the injected code from the AFL++ library for `__AFL_LOOP()` so that it loops indefinitely by short circuiting the iteration counting logic. If additional tooling is not applied via the provided passes, this will run as it would in standard AFL++ persistent mode but will have low stability and accrue memory leak and state corruption. I provide a baseline tool set for implementing a persistent fuzzing system for C applications. I also provide a template which allows for shared memory implementation via fopen, which when extended could be used across other complex file handling systems.

5.3 Renaming Main and Harnessing the Target

To perform my task of renaming the main method of the target application, I use the main pass which grabs the targets `main` function, and then renames the function to the reflect the function called inside of the harness. To handle non-standard applications, an engineer would need to change the search name of the name within the LLVM pass so that it grabs the starting function of the desired program.

Once the program has been modified so it no longer contains a main function, I treat it as if the program is a library. I then utilize `llvm-link`, a tool for joining LLVM IR object files together, to join the program as a library to my harness creating a single unified program containing the target within its fuzzing loop. Since all of the cover functions and global variables are visible to the application I perform the instrumentation and function swap steps to fully tool Closure over the target program.

5.4 Instrumenting Important Code Points

To instrument the global undo logging and exits I use the Global pass and JmpExit passes. Each of these passes utilize global analysis of the harnessed target program to find the respective variables.

Global undo logging For the Globals, I follow my design for resetting globals and implement the procedure as an LLVM pass. First I find the parent block to the start of the fuzzed portion of the program, which once found I keep as a record. I then loop over the global list, creating a copy of each global one at a time which I call a shadow copy. Once I generate the shadow copy, I then instrument a `store` and `load` instruction at the recorded entry block location to reset the global to an initial point before execution of the application. This will

load the shadow copy into memory, and then allow us to store its contents back into the original global before starting each execution ensuring global state is consistent between iterations. I instrument the entry block as opposed to the exit block as an unexpected longjmp will ignore the reset step at this point. With my compiler level logging I can capture the pre-initialized state eliminating any initial logging step before the first iteration.

Jump Exiting To implement the exit avoidance, I make use of a global variable longjmp buffer which I attach with my harness. The LLVM pass first finds all locations of exit functions in the code. Once I have all locations of exits in the program, I then use the longjmp buffer to add a longjmp instruction before each exit, which when triggered during run time forces execution back to the setjmp point set within the harness. This transformation can be seen in Listing 2 and 3.

Listing 5.2: Example of exit block in LLVM IR

```
422 :
    call void @exit
    unreachable
```

Listing 5.3: Exit block after being modified via the Exit Avoidance pass

```
422 :
    call void @longjmp ([1 x %struct.jmp_buf_tag]* @__jmp_buf, i32 1)
    call void @exit
    unreachable
```

5.5 Swapping Cover Functions

I implement the file sentinel and dynamic heap memory management as a series of cover functions and harness them over their respective C functions using LLVM passes.

File Sentinel For setting up the file sentinel I first check the file name that is passed to fopen for if it contains the /.cur_input file that AFL uses as its test case file. Then I open the file using fopen and store it in a temporary variable. If the file name matches the name of the current input then I duplicate the file pointer using the dup3() function with the FD_CLOEXEC flag to link the duplicate to the file descriptor which, is being used by the target program. This ensures that if either one of the file descriptors are closed then the linked file descriptor will also close. Once I have cloned the file descriptor into the file sentinel, I then return the temporary file pointer. At the end of each iteration I check if the file sentinel is still open by using fnctl() to check the liveness of the file descriptor. I then close the file sentinel in the event that the program has finished execution without closing the test case file. An example of fopen and file sentinel closing can be seen in Listings 4 and 5.

Listing 5.4: Closure fopen cover function example

```
1 FILE* __fopen(const char* file, const char* mode) {
2     FILE* fp;
3     int input = strstr(file, captionpos"captionpos.captionposcur_inputcaptionpos")
          != NULL;
4     fp = fopen(file, mode)
5     if (input) {
6         __file_sentinel = dup3(fileno(fp), FD_CLOEXEC);
7     }
8     return fp;
9 }
```

Listing 5.5: Close if open command that runs regardless of fopen or fopen64

```
1 void close_if_open() {
2     int i = fnctl(__file_sentinel, F_GETFD);
3     if(i){ close(i) }
```

```
4 }
```

I instrument these functions then with the Closure file pass which first finds all cases of fopen in the program and renames each instance to the cover function name ___fopen. Doing this in some cases overwrites the fopen call inside of the cover function which creates a recursive loop, so I navigate back to the cover function to confirm that the fopen call is still valid. In the event that a program utilizes fopen64 (fopen with O_LARGEFILE flag passed by default), such as sam2p, I add an additional coverage function and instrument it via the same pass. Another option that I have built into my fopen cover functions is an environment variable AFL_INPUT_SHM, which when set to 1, replaces fopen and file access, with AFL's shared memory test case passing and handing it through fmemopen as opposed to fopen. This allows us to greatly reduce file overhead associated with fuzzing.

Heap Management is implemented by the same means as the file sentinel. Each of the respective functions for malloc, calloc, realloc, and free are constructed following my design(examples in Listing 6 and Listing 7). My LLVM pass which handles swapping the C functions for the cover functions first finds all instances of dynamic memory operations, and then replaces each call with my cover functions. With this I can log all memory allocations within a hash map. I have utilized a pointer map to store every void pointer before returning the created pointer to its original call site. When freeing memory I first remove the pointer from the map, and then free the pointer from memory. If I reach the end of an iteration, I check the map for leftover pointers, and then iterate over the remaining pointers as I clear dynamic memory and the logging map. With this system applications can keep a running log of their dynamic memory utilization, and prevents memory leak.

Listing 5.6: Closure malloc cover function example

```
1 void* __cls_malloc(size_t size){
2     void* temp = malloc(size);
```

```
3     HH.add(map, temp); // add to hash map
4     return temp;
5 }
```

Listing 5.7: Closure free cover function example

```
1 __cls_free(void* ptr){
2     HH.clear(ptr); // remove from hash map
3     free(ptr);
4 }
```

5.6 Closure LoopServer

The loop server is my modified forkserver for controlling the execution of test cases through AFL++. I modify the existing logic inside of the forkserver, to keep the program open indefinitely until a crash or hang has been found. This is done by a short circuit in front of the check for iteration count in the loop. As seen in Figure 5.2, the loop server keeps the program in a state where it will only be reforked in the event that the instance crashes, or timeouts. This means that I am limiting the forking overhead to unavoidable scenarios which improve the throughput of test cases in the common case. I compare my LoopServers performance to its parent, the forkserver, highlighting the basic fork based resetting to the performance of a fully persistent instance with Closure resetting of the same application.

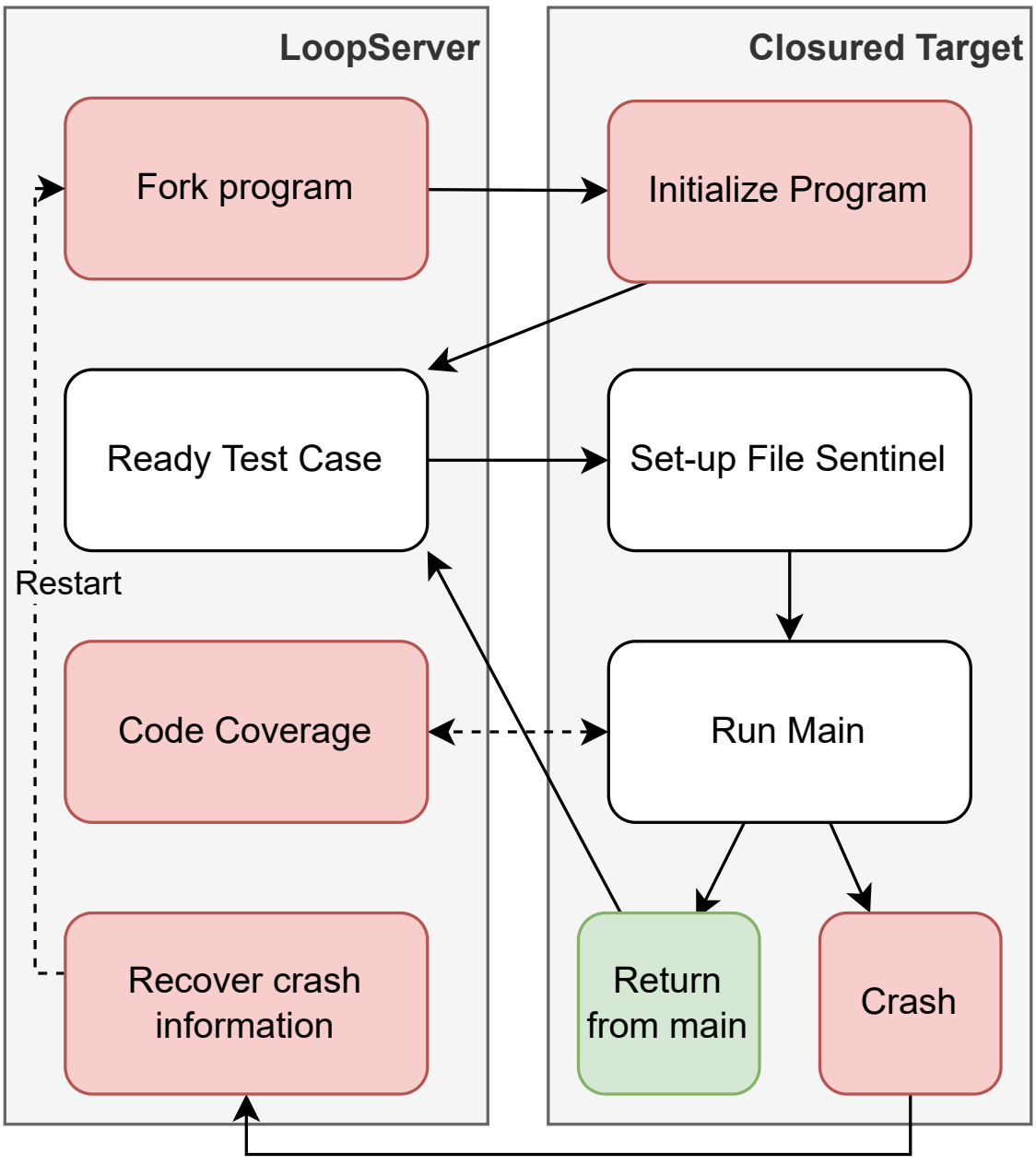


Figure 5.2: LoopServer Orchestration with Fuzzing Target. Areas colored red are high overhead points in execution, by limiting the frequency of these scenarios I effectively increase throughput

Chapter 6

Evaluation

To evaluate the performance of Closure, I answer the following questions.

1. Does Closure improve the rate of test case execution over standard fork based fuzzing?
2. Does Closure achieve fork level accuracy of state resetting between test case execution?
3. Does shared memory test case input via fmemopen in Closure further increase rate of test case execution?

6.1 Evaluation Method

To evaluate the performance of Closure, I run benchmarks across 10 open source code bases, table 6.1. For libraries, I utilized any pre-existing programs which are bundled in the repository (IE. cert-basic for libksba). To generate each executable for testing, I followed the methodology outlined in section 5.1. Once generated, the I re-compile the IR using AFL-clang-fast to instrument a standard version of the application without Closure. This version will utilize AFL++'s standard forking for execution. To create the Closure version, I first process the LLVM IR file with Closure which creates the resulting Closure instrumented IR. Finally, this is compiled the same way as the raw version using afl-clang-fast. This version will run in the LoopServer of the persistent mode that I modified. I compile both

Closure Test Suite			
Program	Library	Purpose	Closure
MJS	MJS	Interpreter	Full/SHM
Exiv2	exiv2	Photo Metadata	Full / No SHM
nasm	nasm	ASM Processing	Full/SHM
mp42ts	bento4	Video Processing	Full/ No SHM
sam2p	sam2p	Image Conversion	Full/SHM
jpec	jpec	JPEG Encoding	Full/ No SHM
bsdtar	libarchive	Archive	Full/ No SHM
cert-basic	libksba	Networking	Full/SHM
readelf	binutils	ELF reader	Full/ No SHM
djpeg	libjpeg	JPEG editor	Full/ SHM

Table 6.1: Test Programs and their respective tooling levels for running Closure. Full: signifies a program with all passes applied. SHM/NO-SHM denotes if it was possible to implement shared memory IO.

the standard instrumentation and the Closure instrumentation from the same IR to ensure that the fuzzing target is consistent between the 2 toolings. I observe that by utilizing the AFL_LLVM_DENYLIST, the number of instrumented points in the source code is consistent between the Closure IR and the Standard IR.

In my testing, applications were run in fully persistent mode limiting fork calls to crashes and timeouts. As recommended in the in the AFL manual [5], if applications were experiencing a large number of timeouts I adjusted the timeout duration to accommodate longer running applications (applies to sam2p/djpeg). This inherently slows down the rate of execution, but allows for the program to be more thoroughly explored. I tested both versions of each application (standard and Closure instrumented) inside of a Ubuntu20.04 VM consisting of 12-cores, and 12gb of memory. I gathered the number of executions, crashes, timeouts, and AFL's stability metrics to analyze the performance of applications tooled with Closure persistent fuzzing compared to standard fork based fuzzing. I prove that with Closure, I increase throughput of test cases while mitigating Semantically Inconsistent Execution (SIE) due to persistent fuzzing.

6.2 Closure Performance

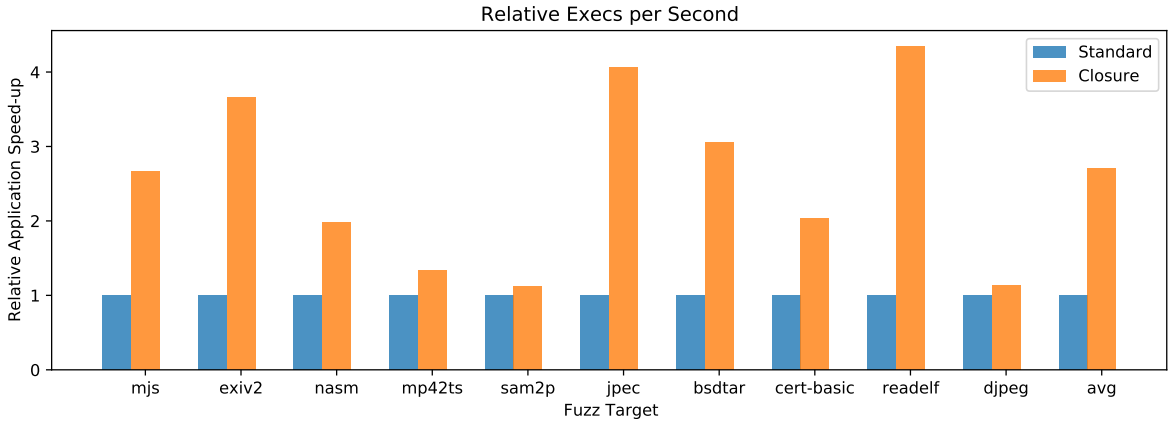
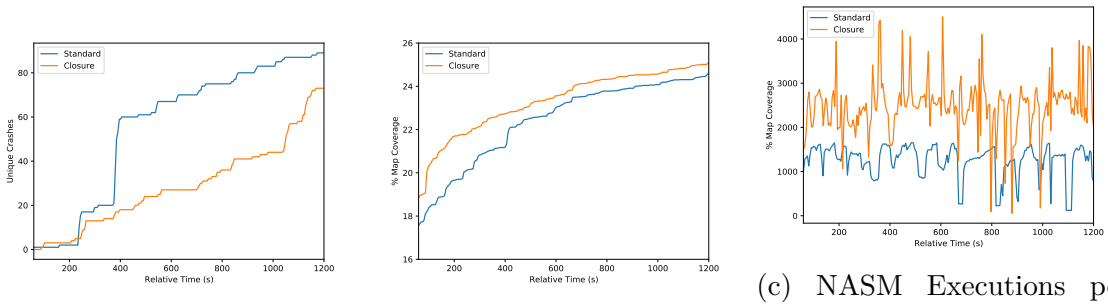


Figure 6.1: Relative execution performance. It is seen that Closure on average performs 270% faster then AFL forkserver alone while using standard file input.

I analyze the performance of Closure to fork based fuzzing by comparing the results of the raw instrumentation binary, and the Closure tooled binary in a 20 minute test in `afl-fuzz`. I choose to use a short time scale as this is typically when the fuzzer is running the slowest so the improvements will be more accurate to a high coverage finding point in the fuzzing process. Additionally, performance improvements are optimal after the fuzzer has gotten up to speed, so highlighting initial performance shows the worst case scenario for persistent fuzzing.

As seen in Figure 6.1, the average performance improvements is 270%. This is seen across programs of varying use cases and complexities. For example, applications which are library based tended to perform better then more complex applications such as nasm. Applications such as MJS and exiv2 showed my best results where they experience greater than 200% despite being full applications. I suspect this to be due to the initialization steps of certain applications leading to additional overhead each iteration which overshadows the performance increase from removing the forking overhead.



(a) NASM Unique Crashes (b) NASM Map Coverage (c) NASM Executions per second

Figure 6.2: NASM 20 minute test with Closure comparison to verify that execution is faster and semantically accurate

6.3 Closure Correctness

In Figure 6.2, I see that by maintaining the state and controlling stability of the application, I see direct increase in the number of edges that are found due to the increased rate of execution. In comparison to persistent mode without Closure tooling, the number of edges was much lower despite the higher rate of execution due to the persistent state corruption which would keep the fuzzer focused in the same area of the code. Stability is a metric provided by AFL++ [6, 31]. This metric has a main usage inside of the LLVM persistent fuzzing tooling as a means of seeing the change in state based on a single input in a given point in the code. The stability of the program is determined based on edge coverage, and if a test case takes the same path on the same input. If a test case traverses a edges inconsistently with the initial test then this impacts the stability. AFL++ mentions drop in stability does not immediately signify runtime problems, making this metric unfit for persistent application fuzzing. I show that by attaching a state controlling harness I successfully fuzz complex open source applications while maintaining 98% stability on average proving that invalid program state drastically impacts the accuracy of fuzzing efforts.

As seen in table 6.2 I limit the number of false positives from deadlock, memory leak, file

	Standard					Closure				
Target	S	C	Touts	cov %	res	S	C	Touts	cov %	res
MJS	100%	8	5	33.25%	1.00	99%	62	13	34.63%	2.67
exiv2	100%	0	0	10.96%	1.00	99%	0	0	10.96%	3.66
nasm	100%	89	38	24.56%	1.00	99%	73	67	25.07%	1.98
mp4ts	100%	83	110	12.98%	1.00	100%	101	105	13.00%	1.34
sam2p	100%	0	76	10.05%	1.00	92%	0	75	10.09%	1.12
jpec	100%	0	5	82.52%	1.00	100%	7	2	83.05%	4.06
bsdtar	100%	0	1	1.68%	1.00	100%	0	1	1.68%	3.06
cert-basic	100%	6	6	38.64%	1.00	100%	3	6	53.63%	2.04
readelf	100%	0	0	6.80%	1.00	99%	0	1	10.15%	4.34
djpeg	100%	0	2	4.34%	1.00	100%	0	7	7.69%	1.13

Table 6.2: Stability metrics for each test program in 20 minute test. **S**tability is the AFL++ metric at the end of runtime. **C**rashes is the number of crashes at end of test. **Touts** is the number of timeouts at end of test. **coverage %** is the percent of code covered at end of test. **Res** is the relative execution speed (from Figure 6.1)

access, or bad state. I show that by maintaining common state issues in persistent fuzzing I am able to achieve higher rates of fuzzing execution. I do observe that in some scenarios that this does not work and breaks the program as seen in Sam2p, however reducing the applications Closure instrumentation and timeout/memory limits allows us to alleviate the occurrence of false positives.

6.4 Closure Fmemopen

To analyze the file overhead, I look at a few edge case scenarios. For this I isolate programs in the benchmarks which utilize the fopen command. I focus on fopen due to it's behavior as a wrapper for the open command. I utilize the file pass to replace the fopens with my cover function as mentioned previously. I then run the program replacing any file IO with fmemopen, which treats a buffered memory space as a file. This happens to work well with the AFL++'s persistent fuzzing tooling since it then passes the data for the next test case through shared memory removing the overhead associated with flushing testcases to the

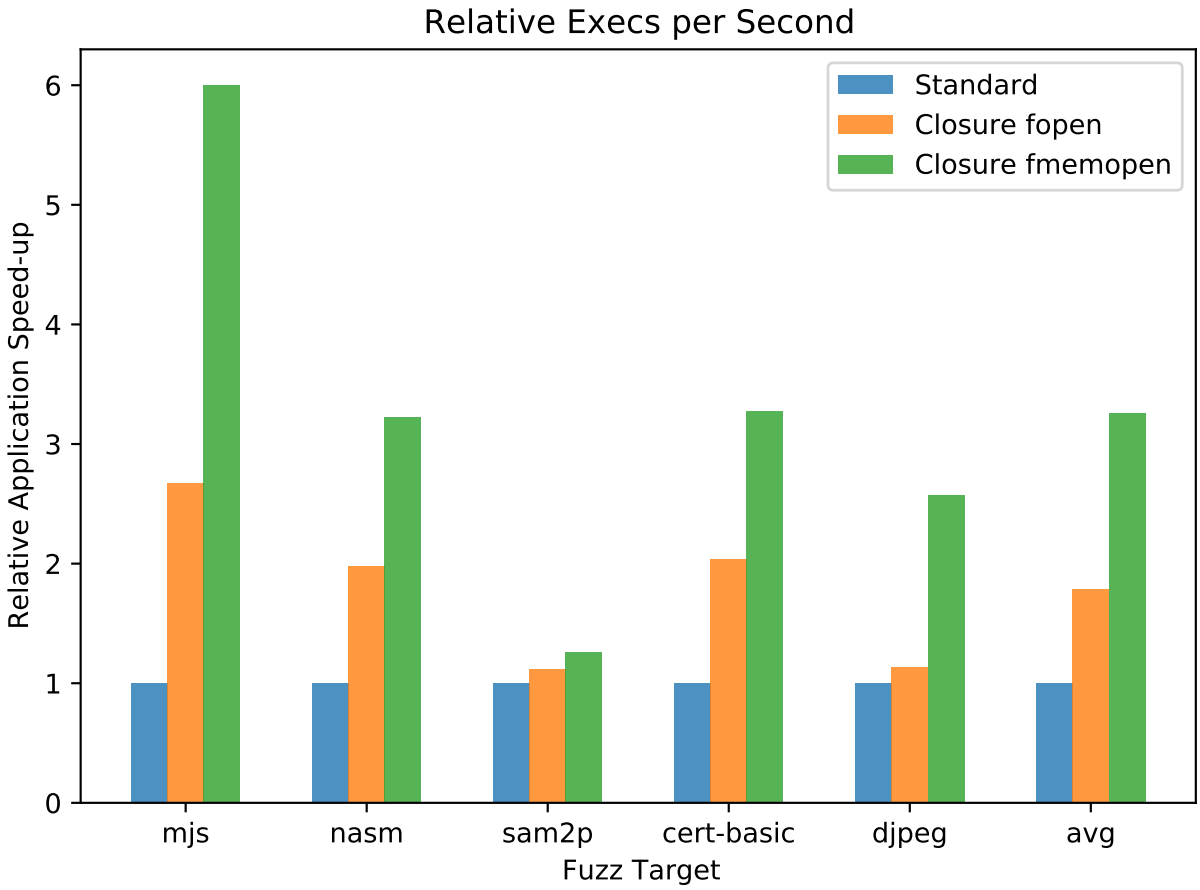


Figure 6.3: Relative execution performance focusing on the file access overhead between fopen and fmemopen.

cur_input file. I show that on average, by replacing calls to fopen, with calls to fmemopen I further increase throughput up to 600% in the case of MJS, with an average of 331% increase as seen in Figure 6.3.

Chapter 7

Discussion

7.1 Big Programs, Large Initialization Step

While I see performance improvements in NASM, sam2p, and djpeg, the improvements that I see do not reflect the best case scenario for Closure. My research shows that this is due to applications specific file processing which occurs inside of the applications initialization step. To provide a greater speed up for applications that exhibit this behavior a more ore aggressive form of code motion would be needed to isolate and pull initialization steps outside of the target program into the stub harness at a deferred initialization point inside of the fuzzing loop. Implementing this is non-trivial and requires additional research into flow analysis and methodologies for hoisting large blocks of complex code outside of its current context while maintaining semantic accuracy. This would allow us to eliminate initialization steps or file access which occur before the test case is loaded, and begin fuzzing at that point. With this I would simplify the fuzzing loop while simultaneously improving throughput.

7.2 Windows Compatibility

Windows and its support by LLVM is in its infancy. Due to the way that windows handles dynamically linked files, LLVM does not fully allow for LLVM IR files to be linked together and then compiled in the same manner that is possible on UNIX based systems. To properly

support windows, work into a custom stub would be necessary focusing on winapi. This is due to the fine differences between DOS and UNIX systems, and the way that windows specific C runtimes are more effective for windows scenarios then the standard C runtime functions. I can attribute this to how standard C functions call the winapi system level variants, which in some scenarios breaks functionality of windows compiled programs.

7.3 Closure and C++

Due to the differences between C and C++, this system does not have full support for the C++ library. C++ makes use of objects with their own constructors and destructors for new instances of an object. These both simplify down to malloc, calloc, realloc, and free based on the context of what is being done to an object at any given time. Expanding this is an engineering concern that would require extending and modifying the functions inside of the harness and adjusting the compiler passes for the correct function swaps.

Chapter 8

Related Work

I have shown that Closure by itself improves performance of fuzzing. I see that these related ideas would work well with Closure to unlock even higher throughput of test case execution then Closure by itself.

8.1 Less Instrumentation

As seen in the work by Nagy and Hicks. [15, 16, 17], by reducing the number of instrumented points in the program via dynamic rewriting of binaries, there is an increase the throughput of the targeted applications to effectively no overhead in the common case. I see that the implementation of Closure in this system would be trivial since 2 forkservers are present. In this implementation of Untracer like tooling I would see the benefits of coverage guided tracing to reduce the code coverage overhead, with the high throughput seen in persistent fuzzing as similarly seen in WINNIE [10]. I would expect that by removing tracing overheads from Closure, I would see even greater increase in test case execution.

8.2 Faster Execution

As mentioned previously, the Linux Kernel Module for AFL [27], provides state resetting functionality in the form of their `snapshot()` primitive. This primitive provides rich state

resetting with higher input then the fork call. This improvement relies on a modification to the kernel to inject this primitive into the OS so that it can be used. Utilizing a similar system, I would be able to provide the initialization step discussed in section 7.1. Using a snapshot like primitive to capture the state of the program after initialization, giving us an optimal reset point that I can instrument via LLVM passes.

Multi-processor fuzzing would benefit greatly from the improvements of Closure. Being able to parallelize fuzzing efforts is greatly dependent on managing OS resources [7, 27, 31]. By instrumenting the program with Closure, I would have multiple self managing instances running in parallel maintaining optimal resource utilization. Without Closure, resource starvation would impact the performance of all fuzzing efforts leading to reduced speeds, I expect that instrumenting with Closure would enable scalar improvements.

Chapter 9

Conclusion

I have shown that the state of the art fuzzer AFL++ is robust and provides an easily extensible infrastructure to improve the performance of bug hunting. I show that through high level program tooling, I extend the functionality of basic C programs in a way that allows them to stay persistent in memory, while providing accuracy and efficiency while improving the rate of test case execution by 200-600%. Seeing as source available fuzzing is the most efficient pre-existing methodology, improving the performance is simple and easy to implement. The evaluation results indicate that more fuzzers should adopt Closure to enable semantically consistent persistent fuzzing. I show that by analyzing source code with compilers, it is possible to create system and program agnostic solutions which improve fuzzing efforts.

Bibliography

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[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] 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/SP40000.2020.00009.

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

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

[8] Google. oss-fuzz, 2022. URL https://github.com/google/oss-fuzz.

[9] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou.

[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] 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.

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

[13] Ian A Mason. gllvm: Whole program llvm in go, 2021. URL URL:https://github.com/SRI-CSL/gllvm.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, 2018. doi: 10.1109/SP.2018.00056.

[20] 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.

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

[22] Laurent Simon and Akash Verma. Improving fuzzing through controlled compilation.

In *2020 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 34–52, 2020. doi: 10.1109/EuroSP48549.2020.00011.

[23] 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.

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

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

[26] 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.

[27] 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.

[28] Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. Pathafl: Path-coverage assisted fuzzing. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 598–609, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367509. doi: 10.1145/3320269.3384736. URL https://doi.org/10.1145/3320269.3384736.

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

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

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

[32] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access*, 6:37302–37313, 2018. doi: 10.1109/ACCESS.2018.2851237.