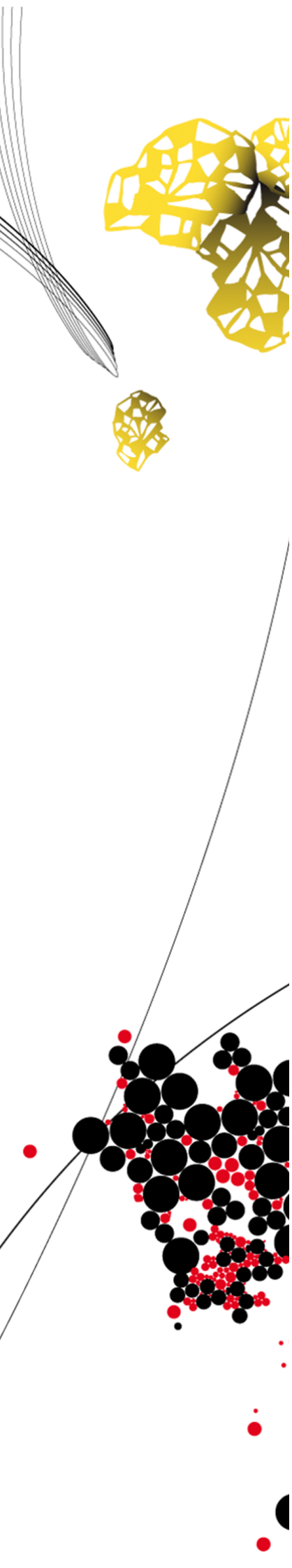


UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Fuzzing: A Comparison of Fuzzing Tools

Suhas Belle Lakshminarayan

M.Sc Thesis
March 2023

Supervisors:
dr. ir. M. Ottavi
dr. ir. A. Continella
dr. ir. N. Alachiotis
B. Endres Forlin
R. Wols

Computer Architecture for
Embedded Systems (CAES) Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

It is critical to detect bugs or vulnerabilities in software because they can serve as an entry point for an attacker, potentially leading to serious consequences. These bugs or vulnerabilities could be the result of a programming error in the design of the software or program. Manually locating all bugs or vulnerabilities in the field of software security is an error-prone and complex task. These efforts can be reduced by a technique known as Fuzzing or Fuzz Testing, which is based on the ability to detect bugs or vulnerabilities by generating inputs of various types (valid, invalid, malformed, etc.) that are fed into the software and tested repeatedly.

There are several Fuzzing Tools (Fuzzers) available that frequently succeed in identifying vulnerabilities. This work demonstrates the complete operation of three fuzzers, namely American Fuzzy Lop (AFL), LibFuzzer, and Angora Fuzzer, as well as a comparison of these fuzzers with program metrics such as code coverage, types of bugs or vulnerabilities detected, number of bugs detected, and execution speed, which in turn measures the fuzzer's performance.

Contents

Chapter 1

Introduction

In 2017 the WannaCry ransomware attack [1] by WannaCry ransomware cryptoworm affected over 300,000 computers with Windows operating system in 150 countries, causing major disruptions and financial depletion in billions of dollars. The attackers encrypted the data and demanded ransom payments in Bitcoin Cryptocurrency form, which affected critical industries such as finance, energy, and healthcare. Spanish mobile company Telefonica was one of the first companies to be affected. The NHS hospitals and surgeries across the UK were also affected by a loss of 92 million pounds due to this attack. Identifying software vulnerabilities or bugs by manually analyzing them is a significant research problem in software security. Due to the complexity, it is hard to verify the software as it is an error-prone task manually. A vulnerability [2] is a weakness or gap in a system's security [3] measures that attackers can exploit to breach the system's security policies. Unpatched vulnerabilities can lead to severe consequences, as demonstrated by high-profile attacks such as WannaCry.

To reduce these efforts, a new technique was introduced by Miller [4] in 1988 called **Fuzzing** or **Fuzz Testing**, which can be used to detect software bugs or vulnerabilities. The main idea behind this technique is the generation of inputs of all sorts (Valid, Invalid, Malformed, etc.), which are fed into the program or software being fuzzed, hoping to trigger software bugs and vulnerabilities. These software bugs and vulnerabilities might be caused by a programmer's mistake in designing the software or program. Some of the most common vulnerabilities in targeted binaries are stack and heap-based overflows, integer overflows, double-free and use-after-free bugs [5]. These vulnerabilities can be classified based on the CVE (Common Vulnerabilities and Exposures) list, and CVSS (Common Vulnerability Scoring System) score based on the bug's severity. Therefore, it is essential to find these software bugs or vulnerabilities as this could be the entry point for an attacker. From these software bugs or vulnerabilities, the attacker can generate an exploit (e.g., a tool or script) that can cause data loss, financial loss, or damage to the asset, i.e., the

software or program being fuzzed.

In other words, Fuzzing is a technique that simulates attacks on the software or program being fuzzed by repeatedly feeding the targeted program with malformed or semi-valid inputs or test cases. For this reason, the fuzzing tools (also called fuzzers) will often succeed in finding out more and more vulnerabilities. Multiple fuzzing tools are available, and the bugs or vulnerabilities found during a fuzzing session on a targeted program might be different in all fuzzers. Some of the fuzzers focus on a particular technique that can trigger new bugs and improve performance. For example, Angora Fuzzer uses principled search, and for the fuzzer to solve all constraints in the targeted program, a minimum of one conditional statement is needed. This cannot be performed the same way in other fuzzers, so the bugs found will be different. The fuzzer stores the bugs and the input which caused it for further use.

This report mainly focuses on understanding the working of three fuzzers: American Fuzzy Lop, LibFuzzer, and Angora Fuzzer. The fuzzers are compared by executing a set of experiments on targeted programs. Program metrics, namely code coverage, types of bugs or vulnerabilities detected, the number of bugs detected, and execution speed, are determined for all experiments, which can be used to measure the performance of the fuzzer. The bugs or vulnerabilities in American Fuzzy Lop can be categorized using an extension tool called *Exploitable GDB*, and the same was ported and partially implemented in LLDB to categorize LibFuzzer crashes.

1.1 Company Background

Marc Witteman, who has been working in the field of chip security since 1993, founded Riscure [6] in 2001. Riscure began serving customers from all over the world from its headquarters in Delft, The Netherlands. Riscure assesses the security of software, chip technology, and embedded/connected devices designed to operate safely in any environment. Riscure is the industry's leading security testing lab for pay-TV solutions. Riscure is also an international market leader in providing test equipment for chip technology side channel and fault injection robustness. Manufacturers, government agencies, and security testing laboratories worldwide use Riscure's equipment.

1.2 Motivation

Finding software bugs or vulnerabilities manually in a targeted program is challenging, and fuzzing tools make this process easier. The current research work focuses on understanding three state-of-the-art fuzzers. The idea behind this research is to analyze the complete working of all the fuzzers. In addition, the analysis helps understand different mutation/generation strategies used by fuzzers to obtain more code coverage.

The current research's state-of-the-art fuzzers differ from the several existing works [7] [8] [9]. This work shows how bugs/vulnerabilities can be introduced into a targeted program to compare the working of fuzzing tools.

1.3 Research Question

It is important to understand the working of the fuzzing tools in order to compare them. This research aims to compare the fuzzers with the following research question:

- **How can we compare different state-of-the-art fuzzers, which fuzzers are good at detecting security bugs/vulnerabilities and how can they be categorized?**

1.4 Report organization

The rest of the report is organized as follows: Chapter 2 presents the background knowledge needed to understand the report. Chapter 3 describes the design and working of fuzzing tools in detail and describes the design or porting of the crash triaging plugin from GDB to LLDB. Chapter 4 describes the targeted programs used for this research and the results obtained fuzzing these programs. Chapter 5 compares the three fuzzing tools used in this research with limitations and recommendations concluding the report.

Chapter 2

Background

This chapter discusses and presents the background for understanding this work. In the following subsections, we present a classification containing different types of fuzzing and real-world benchmarks used in industry and academia.

Fuzzing [10] [11] is an automated software testing technique that feeds a computer program with valid, unexpected, invalid, or random data as input. This computer program is then monitored for errors such as crashes, failed built-in code assertions, or potential memory leaks.

Fuzzing is mainly used as an automated technique to expose software bugs and vulnerabilities in security-critical programs that might be exploited with malicious intent. This technique saves the programmer a lot of time and energy by checking the entire targeted program with all possible inputs generated by the fuzzer. With these newly generated inputs, if the fuzzer found an input that caused a crash, using mutation strategies, the fuzzer generates more inputs that can reproduce the findings. As the fuzzer generates all sorts of inputs, Fuzzing can also help cover unexpected edge cases. Fuzzing also allows us to achieve better code coverage as the fuzzers run the software under test and always provide inputs that can be used to reproduce a bug.

The steps followed by a fuzzer on a targeted program are as shown in Fig 2.1.

- The targeted program or system is first identified
- Using this targeted program, the inputs needed for the program are identified
- The inputs identified in the previous step are mutated in order to generate more inputs for fuzzing
- The newly generated inputs are then used on the targeted program and is tested
- Once the above steps are done, the fuzzer monitors the system behaviour to

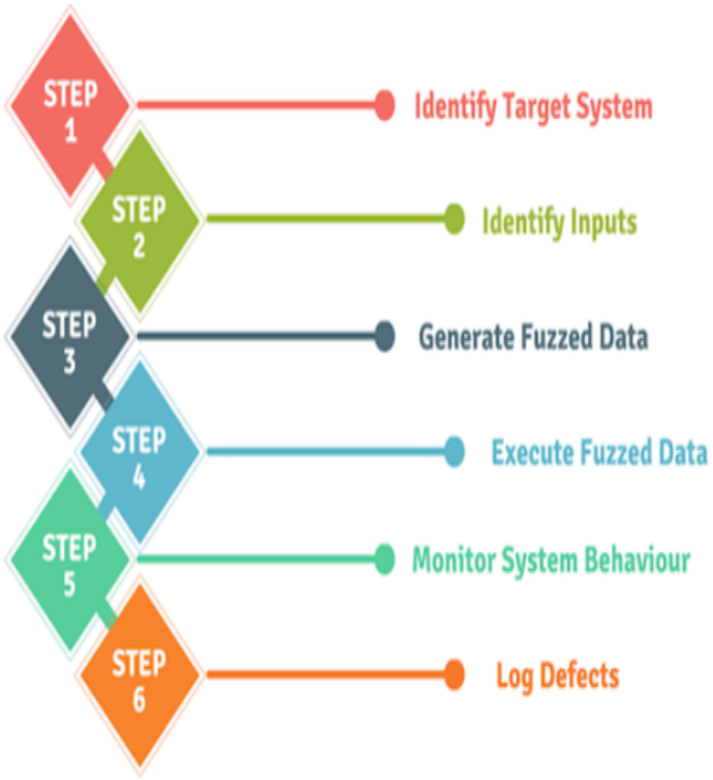


Figure 2.1: Steps followed for Fuzzing

check for software bugs or security-critical vulnerabilities which is also called as **crashes** according to the fuzzer

- If a bug is found or if there is a crash in the fuzzing session, it will be logged and stored as output which we can use to see what input caused that crash

2.0.1 Types of Fuzzing

There are three main categories of fuzzing techniques [12], which are discussed below.

Black-box Fuzzing

In this method of fuzzing, inputs are generated without the knowledge of the targeted program. This can be further classified into two variants, namely *Mutational* and *Generational* black-box fuzzing. In mutational black-box fuzzing, one or more seed inputs are needed for fuzzing to start. This seed input is mutated, and new inputs are generated. At random locations in the input, random mutations are applied to generate more inputs. For example, a single bit may be flipped in the input

to generate more inputs. Likewise, in generational black-box fuzzing, the inputs are generated from scratch. New inputs are generated based on the structural specification of input. Peach fuzzer is an example of a black-box fuzzer. In a black-box fuzzing scenario, the inner works and expected input format are unknown, as shown in Fig 2.2.

Figure 2.2: Black-box Fuzzing

Grey-box Fuzzing

This fuzzing method leverages program instrumentation. The targeted program is injected with a piece of code right after every conditional jump to get lightweight feedback, which can be used to control the fuzzer. During compile time, several control locations are instrumented in the program. The program is provided with an input/seed *corpus* (Set of inputs stored into files individually and provided as an entry point to the targeted program). This seed input is mutated in order to generate new inputs for fuzzing. If the new inputs cover new control locations, the code coverage automatically improves, and these inputs are added to the seed *corpus*. Fig 2.3 shows how grey-box fuzzers are implemented in which some of the inner workings and input structure can be known.

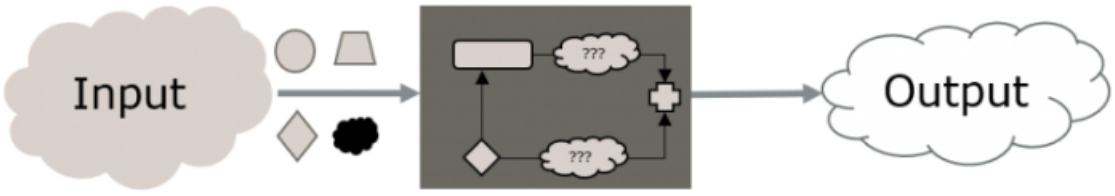


Figure 2.3: Grey-box Fuzzing

White-box Fuzzing

This method of fuzzing is based on **Symbolic Execution** technique. Interesting paths in the program can be found using symbolic execution as it uses program

analysis and constraint solvers. The constraint solvers used here are Satisfiability Modulo Theory (SMT) solvers. The path condition is calculated with a seed input, and the input is mutated. The modified path condition is provided to a constraint solver, which generates new inputs. The main advantage of this technique is the generation of input by traversing into a new path while checking the previous inputs for path conditions. Fig 2.4 shows how white-box fuzzers are implemented in which the complete inner working and the input structure are known.

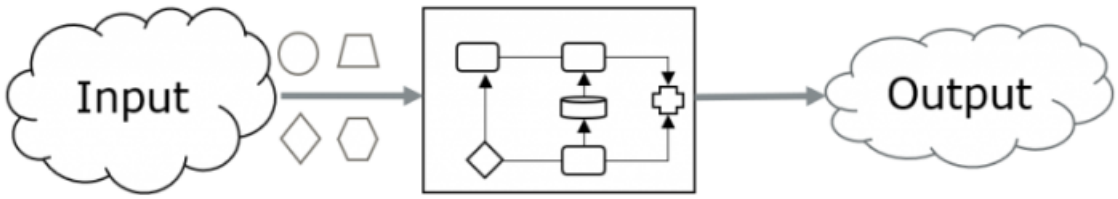


Figure 2.4: White-box Fuzzing

2.0.2 True Code

True Code [13] is an automated code testing solution designed by Riscure for testing embedded systems with the main focus being **SAST** (Static Application Security Testing) and **DAST** (Dynamic Application Security Testing). True Code uses LibFuzzer as the fuzzing tool to fuzz a targeted program or software.

SAST - Static Application Security Testing

SAST is a white-box testing technique used to secure the software by reviewing the source code to identify sources of bugs or vulnerabilities. Identifying security-related bugs or vulnerabilities is crucial, and True Code provides code checkers for this. The Fault Injection sensitive checks are specific to situations where both software and hardware are combined. These checks work entirely on C and partially on C++, with the checks being performed on a piece or the entire codebase.

DAST - Dynamic Application Security Testing

DAST is a black-box testing technique to secure the software or program by performing attacks on the software or program to identify security-related bugs and vulnerabilities. The checks for bugs or vulnerabilities are done by True Code during run-time using the following methods:

- Fault Injection Simulation

- Fuzzing

Fault injection [14] is a technique that can stress test a computing system and understand its behavior. This technique can be achieved using hardware or software or a hybrid approach. Some of the widely studied hardware fault injections include applying extreme temperatures, high voltages, and so on. For fault injection and fuzzing, the user must create a harness tree for the function under analysis. The stubbed function will replace the original function to speed up the analysis, bypass the hardware interfaces, and limit the call depth. Currently, both methods are fully compatible with the C codebase.

Chapter 3

Implementation

In this chapter, the in-depth analysis of three fuzzers, namely **American Fuzzy Lop** [15], **LibFuzzer** [16], and **Angora Fuzzer** [17] is presented to understand the working and the strategies used by the fuzzers. The following state-of-the-art fuzzers used in this research are chosen, considering LibFuzzer as the main fuzzer since it is the fuzzing engine used in TrueCode by Riscure [13]. Furthermore, American Fuzzy Lop is used due to the design similarities with LibFuzzer. Whereas, Angora Fuzzer in existing work [18] was tested in comparison with only American Fuzzy Lop, and the designers of Angora claim the fuzzer to be the best state-of-the-art fuzzer.

3.1 Sanitizers

As explained in Chapter 2, the main idea behind fuzzing is to feed randomly generated inputs to the targeted program to trigger a bug or vulnerability. However, we can accelerate this process using **Sanitizers**. The compilers used to build fuzzer instrumented binaries can be combined with sanitizers where the targeted program is instrumented with extra code checks for illegal conditions like buffer overflow, integer overflow, etc. These sanitizers are further categorized as follows.

3.1.1 Address Sanitizer(ASAN)

Address Sanitizer [19] is a fast error detector focused on memory-related bugs. ASAN [20] [21] can be set as a flag in all fuzzers, consisting of a compiler instrumentation module and a run-time library. The program will exit with a non-zero exit code, and the error message is printed if a memory-related bug is detected. On the first detected error, ASAN exits. The following types of bugs or vulnerabilities can be detected using ASAN.

- Out-of-bounds accesses to heap, stack and globals

- Use-after-free
- Use-after-return
- Use-after-scope
- Double-free, invalid free
- Memory leaks (experimental)

ASAN can also be set with an option of **Leak Sanitizer** (*ASAN_OPTIONS=detect_leaks=1*). This Leak Sanitizer is a run-time memory leak detector which detects leaks with memory error when combined with ASAN.

3.1.2 Undefined Behavior Sanitizer(UBSAN)

Undefined Behavior Sanitizer [22] is an error detector to identify undefined behavior. UBSAN can also be set as a flag for the LibFuzzer compiler and is still an experimental feature in AFL and Angora. When the flag is set, the targeted program is modified at compile time to detect variety of undefined behavior during program execution. Some types of bugs or vulnerabilities detected by using UBSAN are as follows.

- Array subscript out of bounds
- Bit-wise shifts that are out of bounds for their datatype
- De-referencing misaligned or null pointers
- Signed integer overflow
- Conversion from, to, or between floating-point types which would overflow the destination

3.1.3 Memory Sanitizer(MBSAN)

Memory Sanitizer [23] is an error detector of uninitialized reads. MBSAN is an experimental feature that still needs to be fully implemented with all the fuzzers. It consists of a run-time library and a compiler instrumentation module.

3.1.4 Data Flow Sanitizer(DFSAN)

Data Flow Sanitizer [24] is a generalized dynamic flow analysis, not a tool, unlike other sanitizers designed to detect a particular bug. DFSAN instead provides a generic dynamic data flow analysis framework using which we can detect application-specific issues within our code. This sanitizer is an experimental feature like MBSAN and is not completely implemented in all fuzzers.

3.2 American Fuzzy Lop

American Fuzzy Lop (AFL) is a mutation-based, security-oriented Greybox fuzzer released in 2013 [15]. AFL works on discovering clean and new test cases that triggers a new internal state by the use of a novel type of compile-time instrumentation and genetic algorithms. These new states increase the code coverage in the targeted binary. Firstly, the usability of this fuzzer allows anyone with fuzzing knowledge to run the fuzzer out-of-the-box on multiple programs without any knowledge of the domain and the targeted program itself. Secondly, the uncovering of vulnerabilities is done by AFL with shallow effort for the security analyst.

AFL works with an instrumentation-guided genetic algorithm. The overall algorithm can be added up as follows.

- The initial user-supplied test case is added to the queue
- The next input file in queue is taken
- Without modifying the behavior of the program, try to trim the test cases to the smallest size
- File mutation using multiple fuzzing strategies
- As a result of mutated inputs, if the instrumentation records a new state transition, the mutated output is added to the queue as a new entry
- Go to 2 and repeat the steps

AFL generates new inputs by modifying a seed input/initial input. AFL leverages coverage feedback to understand how to reach deeper into a program. The program's source code is compiled to build a fuzzer-instrumented executable binary. It is also possible to run AFL on un-instrumented binaries using a virtual machine or a dynamic instrumentation tool. AFL is implemented with Address Sanitizer 3.1.1, and Undefined Behavior 3.1.2 is still an experimental feature. Hence, AFL will be successful in detecting only memory-related bugs/vulnerabilities. The sub-section

mentions how AFL generates new inputs to reach deeper into the target program and thus increase code coverage.

3.2.1 Methods/Strategies used by the Fuzzer for input mutation and generation

AFL works in such a way that it has a rare feedback loop where one can measure what changes to the input file result in discovering new branches in the code and which ones are a waste of time. Below are the steps or strategies used by AFL [25] [26] to mutate and generate new inputs for fuzzing.

Walking Bit Flips

The input provided to the fuzzer is first bit flipped sequentially, resulting in new input. The bits in the input can always be flipped by one bit in this method. In a single row, there can be one to four bits flipped. For a large corpus [25] of input files, the observed results are

- Flipping a single bit: Approx. 70 new paths per one million generated inputs
- Flipping two bits in a row: Approx. 20 additional paths per million generated inputs
- Flipping four bits in a row: Approx. 10 additional paths per million inputs

```
1   /* Single walking bit. */
2   stage_short = "flip1";
3   stage_max = len << 3;
4   stage_name = "bitflip 1/1";
5
6   stage_val_type = STAGE_VAL_NONE;
7   orig_hit_cnt = queued_paths + unique_crashes;
8   prev_cksum = queue_cur -> exec_cksum;
9   for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {
10      stage_cur_byte = stage_cur >> 3;
11      FLIP_BIT(out_buf , stage_cur);
12      if (common_fuzz_stuff(argv, out_buf, len)) goto abandon_entry;
13          FLIP_BIT(out_buf , stage_cur);
14
15  /* Two walking bits. */
16  stage_name = "bitflip 2/1";
17  stage_short = "flip2";
18  stage_max = (len << 3) - 1;
19
```

```
20      orig_hit_cnt = new_hit_cnt;
21      for (stage_cur = 0; stage_cur <stage_max; stage_cur++) {
22          stage_cur_byte = stage_cur >> 3;
23          FLIP_BIT(out_buf, stage_cur);
24          FLIP_BIT(out_buf, stage_cur + 1);
25
26          if (common_fuzz_stuff(argv, out_buf, len)) goto abandon_entry;
27
28              FLIP_BIT(out_buf, stage_cur);
29              FLIP_BIT(out_buf, stage_cur + 1);
30
31      /* Four walking bits. */
32      stage_name = "bitflip 4/1";
33      stage_short = "flip4";
34      stage_max = (len << 3) - 3;
35
36      orig_hit_cnt = new_hit_cnt;
37      for (stage_cur = 0; stage_cur <stage_max; stage_cur++) {
38          stage_cur_byte = stage_cur >> 3;
39          FLIP_BIT(out_buf, stage_cur);
40          FLIP_BIT(out_buf, stage_cur + 1);
41          FLIP_BIT(out_buf, stage_cur + 2);
42          FLIP_BIT(out_buf, stage_cur + 3);
43
44          if (common_fuzz_stuff(argv, out_buf, len)) goto abandon_entry;
45
46          FLIP_BIT(out_buf, stage_cur);
47          FLIP_BIT(out_buf, stage_cur + 1);
48          FLIP_BIT(out_buf, stage_cur + 2);
49          FLIP_BIT(out_buf, stage_cur + 3);
```

Listing 3.1: Walking Bit Flips

Walking Byte Flips

In this mutation method, the bits in the input are flipped by one byte. Like walking bit flip, 8, 16, or 32-bit wide bitflips are done in walking byte flip [25]. With this method, AFL can discover approximately 30 additional paths per million input in addition to the path discovered with walking bit flips.

```
1       /* Walking byte. */
2       stage_name = "bitflip 8/8";
3       stage_short = "flip8";
4       stage_max = len;
5
6       orig _hit_cnt = new_hit_cnt;
7
```

```
8     /* Two walking bytes. */
9     stage_name = "bitflip 16/8";
10    stage_short = "flip16";
11    stage_cur = 0;
12    stage_max = len - 1;
13
14    orig_hit_cnt = new_hit_cnt;
15
16    /* Four walking bytes. */
17    stage_name = "bitflip 32/8";
18    stage_short = "flip32";
19    stage_cur = 0;
20    stage_max = len - 3;
21
22    orig_hit_cnt = new_hit_cnt;
```

Listing 3.2: Walking Byte Flips

Simple Arithmetic

This mutation method is divided into three stages. Initially, the fuzzer performs addition and subtraction on all the bytes in the input to generate new inputs. Next, the input is either incremented or decremented by inspecting 16-bit values. If the most significant byte is not modified, the results will be similar to stage one. Finally, the input is added or subtracted in the third stage by inspecting 32-bit values.

```
1     /* 8-bit arithmetic. */
2
3     stage_name = "arith 8/8";
4     stage_short = "arith8";
5     stage_cur = 0;
6     stage_max = 2 * len * ARITH_MAX;
7
8     stage_val_type = STAGE_VAL_LE;
9
10    orig_hit_cnt = new_hit_cnt;
11
12    /* 16-bit arithmetic, both endians. */
13
14    stage_name = "arith 16/8";
15    stage_short = "arith16";
16    stage_cur = 0;
17    stage_max = 4 * (len - 1) * ARITH_MAX;
18
19    orig_hit_cnt = new_hit_cnt;
20
```

```
21      /* 32-bit arithmetic, both endians. */
22
23      stage_name = "arith 32/8";
24      stage_short = "arith32";
25      stage_cur = 0;
26      stage_max = 4 * (len - 3) * ARITH_MAX;
27
28      orig_hit_cnt = new_hit_cnt;
```

Listing 3.3: Simple Arithmetic

Known Integers

In this mutation method, certain sets of integers are hard-coded. Then, the integers are chosen to trigger a new edge condition in the targeted program. For example, some integer values are -1, 256, 1024, MAX INT-1, and MAX INT+1 [25]. When this method is used, the fuzzer overwrites the input with one of the mentioned integer values.

Stacked Tweaks

Once all the previous methods are used, and inputs are generated, the fuzzer enters a loop to generate more inputs by performing the below-mentioned operations.

- Single bit-flips
- Attempts to set ”interesting” bytes, words, or dwords (both endians)
- Addition or subtraction of small integers to bytes, words, or dwords (both endians)
- Completely random single-byte sets
- Block deletion
- Block duplication via overwrite or insertion
- Block memset

Test Case Splicing

The input is generated in this method by considering two different inputs. These two inputs are then spliced at a random location, and the stacked tweaks method is run on this newly generated input. As a result, this method requires a large input corpus and usually discovers 20% more execution paths not triggered by other mutation methods.

3.2.2 Fuzzer Usage

To start with, the fuzzer needs to be installed with some necessary dependencies or packages. The brief workflow of AFL is mentioned below.

- The targeted software or program is compiled with AFL's compiler to obtain the instrumented binary
- The fuzzing needs to be started with a test corpus (Testcase directory)
- On the instrumented binary, run AFL
- Analyze the results obtained

For the fuzzer to function correctly, we must ensure at least one seed file or initial input in the **testcase_dir** that is expected by the targeted software or program. One has to keep in mind the two rules when designing the initial test case, which are

- Try to keep the initial input file small. Keeping it under 1kB is ideal, but not strictly necessary
- Multiple test cases can be used only if they are different from each other functionally

The targeted program should first be compiled with the AFL compiler. For example, if the software or program that needs to be fuzzed is written in C, then **afl-gcc** compiler has to be used, and similarly, **afl-g++** for C++ programs. This compilation instruments the targeted software or program with additional code, and this added code allows the fuzzer to communicate with the compiled binary and generate new inputs to discover new code paths. This entire process is called **instrumentation**.

A new test case directory and a "findings" or "output" directory are needed before we start fuzzing the targeted software or program. AFL makes three sub-directories inside the findings directory, namely **crashes**, that contains the test cases which caused the application to crash, **hangs**, that holds the test cases which caused the application to hang, and **queue** that contains the test cases which the fuzzer still needs to test the application with. Finally, we use the command below to start the fuzzing session.

```
$ ./afl-fuzz -i testcase_dir -o findings_dir /path/to/program [..params..]
```

If the program takes input from a file, @@ can be used to mark the location in the target's command line where the input file name should be placed [27]. The fuzzer will substitute this automatically even if @@ is not explicitly mentioned in the command.

```
$ ./afl-fuzz -i testcase_dir -o findings_dir /path/to/program @@
```

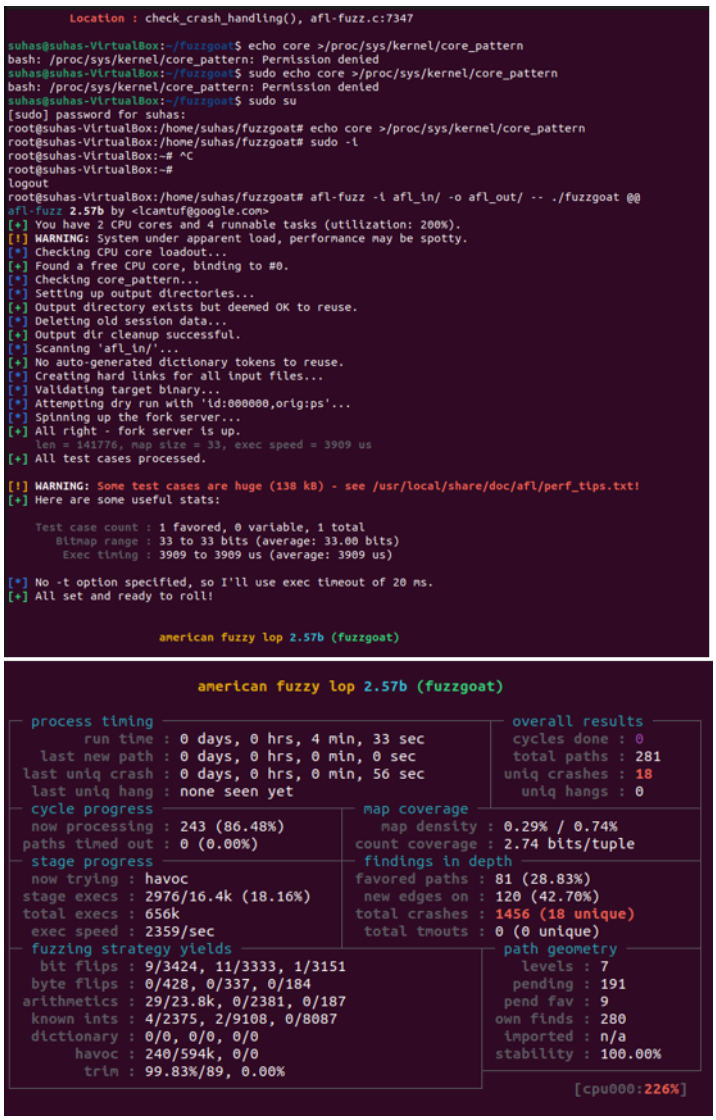


Figure 3.1: American Fuzzy Lop (AFL)

Figure 3.1 shows the UI of the AFL fuzzer. The fuzzing process can be interrupted using Ctrl-C.

3.2.3 Crash Triaging Tool - Exploitable GDB

The crashes obtained after a successful fuzzing session using AFL can be triaged using a plugin called **Exploitable**. The Exploitable plugin [28] is a GDB extension that classifies the bugs or crashes based on severity. This plugin examines the state of a crashed software or program and summarizes how difficult it would be for an attacker to generate exploits to gain control of the program or software. Software

developers can use the plugin to prioritize bugs and address the most critical ones [29].

Figure 3.2 shows a simple workflow of the Exploitable plugin. The plugin iterates over a list of ordered **rules** (lib/rules.py) to generate **Classification** (lib/classifier.py). **Tag** is added to the classification by the exploitable plugin when the state of application running in GDB is matched with a rule. Depending on the command parameters, classification obtained as a result of exploitable invocation can either be printed to GDB's stdout or saved to a pickle file.

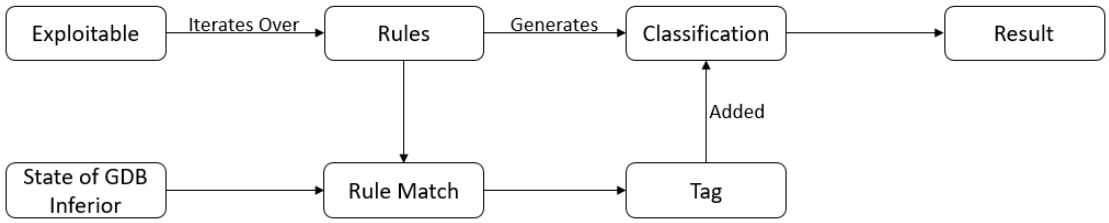


Figure 3.2: Simple Workflow of Exploitable Plugin

lib/classifier.py

The file **lib/classifier.py** has some essential classes, namely, Tag, Classification, and Classifier. The class Tag defines a partial description of the state of a GDB. This tag is obtained if there is a matching rule. These tags are compared by type and ordered by rank. The class Classification defines how exploitable the current state in the GDB inferior or targeted program is. The classifier object returns an instance of this object. The result obtained by this is as shown in Figure 3.3.

```
result = ["Description: {}".format(self.desc),
          "Short description: {}".format(self.tags[0]),
          "Hash: {}.{}".format(self.hash.major, self.hash.minor),
          "Exploitability Classification: {}".format(self.category),
          "Explanation: {}".format(self.explanation)]
```

Figure 3.3: Result of Exploitable plugin

lib/rules.py

This file specifies the list of rules which are used to classify the state of a GDB inferior. These rules are based on the CVE list and the CVSS score, categorized and ordered from most exploitable to least exploitable. The four different categories are

- EXPLOITABLE
- PROBABLY_EXPLOITABLE
- PROBABLY_NOT_EXPLOITABLE
- UNKNOWN

lib/analyzers/x86.py

Analyzers are used to match the rules and determine the severity of the bugs based on the category. The class analyzers contains methods which corresponds to the rules defined in lib/rules.py , that analyzes a targeted program to determine properties of the particular target. If the analyzer method returns True, the rule is said to match to the state of the GDB inferior's state, otherwise it's not a match.

lib/gdb_wrapper

A GDB wrapper is a collection of Python objects that wrap and extend the GDB Python API.

Figure 3.4 shows the output of a sample test. From the result of the plugin we can see that the crash is considered to be **EXPLOITABLE** as there is a possible stack corruption. The plugin also gives a short explanation on what the actual problem is or what actually caused the crash.

Figure 3.4: Exploitable Output

3.3 LibFuzzer

LibFuzzer [16] is an in-process, coverage-guided Grey-box fuzzing engine. The library under test is linked to fuzzer and fed with mutated inputs via **target function** (A fuzzing entry point). New inputs are generated by the fuzzer keeping a track of the reached areas of code to maximize code coverage. The address sanitizer 3.1.1, undefined behavior sanitizer 3.1.2 is implemented in LibFuzzer and can be used as flags to compile the targeted program, while memory sanitizer 3.1.3, and data flow sanitizer 3.1.4 are still experimental features. LibFuzzer will successfully detect memory-related and undefined behavior bugs/vulnerabilities.

Fuzz Target

Implementation of fuzz target for the use of fuzzer on a library under test is the initial step. A function which accepts an array of bytes and uses the API under test to do something interesting with these bytes is a fuzz target. The fuzz target can be implemented as shown below

```
// fuzz_target.cc
extern "C" int LLVMFuzzerTestOneInput (const uint8_t *Data, size_t Size){
    DoSomethingInterestingWithMyAPI(Data, Size);
    return 0;
}
```

While implementing the fuzz target, some of the important steps that needs to be followed are mentioned below.

- Fuzz target is executed multiple times by the fuzzer inside the same process with unique inputs
- Any kind of input must be accepted
- The fuzz target must not *exit()* on any input
- Avoiding greater complexity, logging, or excessive memory consumption improves the fuzzing speed

Corpus

A corpus of sample inputs for library under test is very much needed for the fuzzer to work efficiently. This corpus must usually be filled with all sorts of inputs which are both valid and invalid for the library under test. Random mutations are applied

to these sample inputs by LibFuzzer to generate new inputs that are stored in the corpus. If a new path is discovered in the code under test, the respective mutated input is added to the input corpus and maybe used later. LibFuzzer works without any initial input unlike AFL but due to that reason, if the library under test accepts structured, complex inputs the efficiency of fuzzing will reduce. This corpus also acts as a check to make sure the fuzz target or entry point still works and the inputs in the corpus run without any errors.

Fuzzer Usage

To use the LibFuzzer, we first need to install **Clang** compiler. A fuzzer binary has to be generated using this compiler on the targeted program or software. To build this fuzzer binary, certain flags has to be used with the Clang compiler

- **-fsanitize=fuzzer**(required) provides in-process coverage information to libFuzzer and links with the libFuzzer runtime
- **-fsanitize=address**(recommended) enables the Address Sanitizer
- **-g**(recommended) enables debug information which makes it easier to read the error messages(if any)

```
clang -g -fsanitize=fuzzer,address path/to/program
```

The above command shows the example usage of the Clang compiler with flags, which in turn generates a output fuzzer binary of the targeted program. The following command can be used to run the fuzzer.

```
./a.out
```

```
suhas@suhas-VirtualBox:~/fuzztriage/03-zstd$ ./fuzzer-03
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1495252732
INFO: Loaded 1 modules   (3 inline 8-bit counters): 3 [0x55847ac16fd0, 0x55847ac16fd3),
INFO: Loaded 1 PC tables (3 PCs): 3 [0x55847ac16fd8,0x55847ac17008),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 2 ft: 2 corp: 1/1b exec/s: 0 rss: 27Mb
#262144 pulse  cov: 2 ft: 2 corp: 1/1b lim: 2611 exec/s: 131072 rss: 27Mb
#524288 pulse  cov: 2 ft: 2 corp: 1/1b lim: 4096 exec/s: 131072 rss: 27Mb
#1048576        pulse  cov: 2 ft: 2 corp: 1/1b lim: 4096 exec/s: 116508 rss: 27Mb
#2097152        pulse  cov: 2 ft: 2 corp: 1/1b lim: 4096 exec/s: 123361 rss: 27Mb
#4194304        pulse  cov: 2 ft: 2 corp: 1/1b lim: 4096 exec/s: 119837 rss: 27Mb
#8388608        pulse  cov: 2 ft: 2 corp: 1/1b lim: 4096 exec/s: 110376 rss: 27Mb
```

Figure 3.5: LibFuzzer

Figure 3.5 shows the output of LibFuzzer. Information about fuzzer options and configuration is printed out initially as output which includes the present random seed. The output lines below the information about the fuzzer reports the following statistics (when non-zero) [16]

- **cov** gives the total number of code blocks or edges covered
- **ft** LibFuzzer uses different signals to evaluate the code coverage: edge coverage, edge counters, etc. These signals combined are called features
- **corp** Number of entries in the current test corpus and the size of it in bytes
- **lim** Current limit on the length of new entries in the corpus
- **exec/s** Number of fuzzer iterations per second
- **rss** Current memory consumption

3.3.1 Methods/Strategies used by the Fuzzer for input mutation and generation

3.6 shows the fuzzing loop of LibFuzzer which is similar to almost all the other mutational fuzzers. The execution starts with an initial seed input, new inputs are generated by selecting and applying a mutator. These inputs will then be tested on the fuzz target, and we check whether the coverage increased or not. If code coverage is increased by the mutated input, the input is added to the seed corpus which can be used further. The mutators in the fuzzer are of random nature and this makes

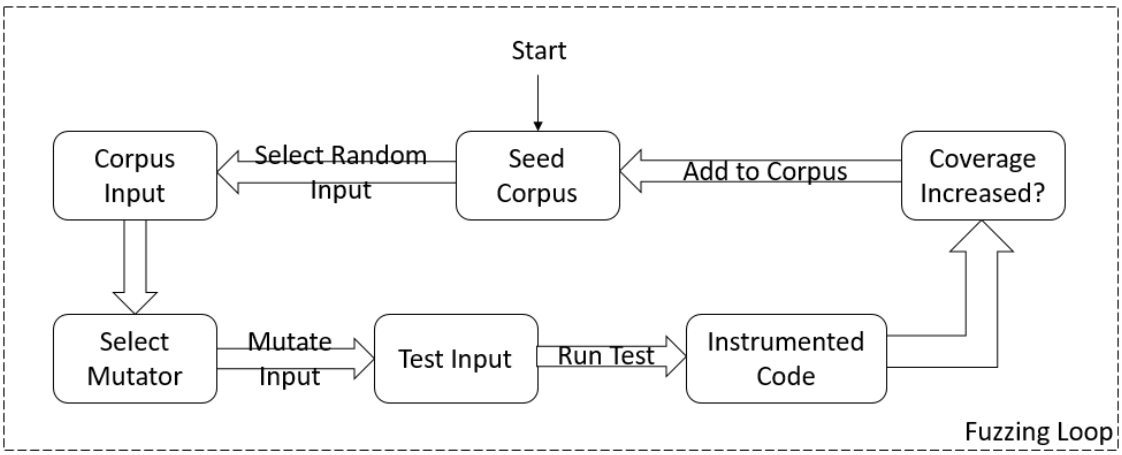


Figure 3.6: LibFuzzer Fuzzing Loop

powerful local search for useful inputs, but as a consequence, to obtain additional

coverage the tests have to be run multiple times. The table 3.1 [7] below shows the list of LibFuzzer mutation operations.

The mutation strategies of LibFuzzer are almost similar to AFL. The inputs gener-

Mutator	**Description**
EraseBytes	Size reduction by removing a random byte
InsertByte	Increase size by one random byte
InsertRepeatedBytes	Increase size by adding at least 3 random bytes
ChangeBit	Randomly flip a bit
ChangeByte	Randomly flip a byte
ShuffleBytes	Rearrange the input bytes randomly
ChangeASCIIInteger	Find ASCII integer in data, perform random math ops and overwrite into input
ChangeBinaryInteger	Find Binary integer in data, perform random math ops and overwrite into input
CopyPart	Return part of input
CrossOver	Recombine random inputs in corpus
AddWordPersist AutoDict	Replace part of input with the previous input that increased the coverage
AddWordTemp AutoDict	Replace part of the input with one that increased code coverage previously
AddWord FromTORC	Replace part of input with a recently performed comparison.

Table 3.1: Mutation Strategies used by LibFuzzer

ated after applying all these mutation strategies does not guarantee that it always increases the code coverage. At times, the fuzzer might generate similar inputs that do not explore any unexplored branches in the library or the code under test.

3.3.2 Crash Triaging Tool - Exploitable LLDB

The exploitable plugin for triaging AFL crashes uses GDB as the debugger. The main focus here is to change the debugger from GDB to LLDB in-order to use this plugin to triage crashes obtained after a libFuzzer fuzzing session. LLDB is a debugger component of the LLVM project and supports more operating systems and fuzzers. The porting of the plugin from GDB to LLDB was achieved by analyzing the list of things that are being used by GDB and below-mentioned are the list of things used.

- Getting the OS name (show osabi)

- Getting the target architecture (show architecture)
- Getting the program counter (print $pc)
- Getting the stack pointer (print $sp)
- Getting information about the signal (print $_siginfo)
- Getting the address that caused a segmentation fault (print $_siginfo, _sifields, _sigfault, _si_addr)
- Getting the last executed assembly instruction, using the program counter
- Getting the process ID
- Getting the stackframe of the crash (Used to check stack buffer overflow and heap errors)
- Using information from the process address space

The working of this exploitable plugin using LLDB is similar to the GDB version 3.2.3. LLDB Python API [30] available for public use is used to port all the GDB functions to LLDB functions. LLDB debugger is necessary as it would be impossible to get the values of program counter, stack frame and process ID. These values are needed to write the libraries explained in 3.2.3 using LLDB Python API. Out of all the rules written according to the CVE list and CVSS score in Exploitable GDB, majority of them were successfully implemented in LLDB.

3.4 Angora Fuzzer

Angora [18] is a mutational-based coverage-guided fuzzer whose primary goal is to solve path constraints without the use of symbolic execution to increase branch coverage. Angora considers the conditional statement in the targeted program or software as a constraint and then mutates the inputs based on the conditional statement. Whereas all the other fuzzers blindly mutate the input to generate new inputs, which may or may not increase the code coverage. All the unexplored branches of the targeted software or program are tracked by Angora in order to solve path constraints on these branches. Using sanitizers 3.1 is an experimental feature in Angora Fuzzer. Figure 3.7 shows the critical techniques used by Angora to solve path constraints, and these techniques are explained below.

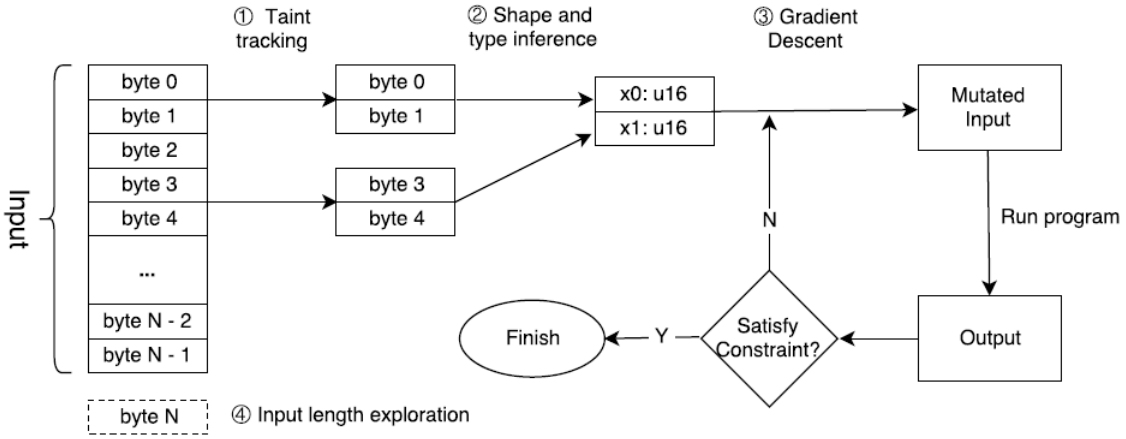


Figure 3.7: Angora Fuzz Loop

3.4.1 Context-sensitive branch coverage

The branch coverage in AFL is the count of the number of times each branch is executed. Using lightweight instrumentation, AFL gets the branch coverage information by instrumenting the targeted program at each branch point during compile time. To record the number of executions of a branch in different runs, AFL uses a global branch coverage table. Similarly, for each run AFL keeps record in a path trace table of the number of times each branch in all conditional statements is executed. AFL checks whether a new internal state of the program is triggered by a newly generated input by the fuzzer by comparing the branch coverage table and path trace table. Unfortunately, the branches of AFL are context-insensitive, i.e., if the same branch is executed in a different context, AFL fails to distinguish these executions, in turn failing to notice new internal states of the program. To overcome this limitation,

in Angora context sensitive branch coverage is used. Angora during instrumentation assigns random ID for each call site [18]. For recursive function calls, the ID is pushed to the stack by Angora and outputs two unique values, where the number of unique branches are at most doubled.

3.4.2 Scalable byte-level taint tracking

The objective of Angora [18] is to execute previously unexplored branches in a targeted program by generating necessary inputs. The fuzzer does this by analyzing how the predicate of the branch is affected by identifying the specific bytes in the input. To achieve this level of precision, Angora uses byte-level taint tracking. However, this technique can be computationally intensive, mainly applied to each byte. As a result, AFL avoids using it. For most program runs, taint tracking is optional. By performing taint tracking on a single input, it is possible to determine which byte offsets influence the execution of conditional statements. Once this information is recorded, the program can be run, and input can be mutated without taint tracking, allowing Angora to perform efficiently as AFL in input execution.

3.4.3 Search based on Gradient Descent

Angora modifies the input to satisfy path constraints rather than using the resource-intensive symbolic execution method [18]. Gradient Descent algorithm is implemented in Angora to solve path constraints and this algorithm is widely used in machine learning. Angora considers the predicate that determines whether a branch should be executed on a black-box function *f(x)* as a constraint, where *x* is a vector of the input values that are used in the predicate, and *f()* represents the computation from the program start to predicate. Three types of constraints can be placed on *f(x)*

```
1    f(x) < 0;
2    f(x) <= 0;
3    f(x) == 0;
```

Gradient descent is iterative and is used to find the minimum of the function f(x). Unfortunately, there is no analytic form of f(x) in fuzzing. Hence, to compute each directional derivative, the program must be run twice with the original and modified input. The input is modified by adding or/and subtracting with a small value of 1.

3.4.4 Type and shape inference

Shape inference is used to determine the bytes in the input which are always together and in the program used as a single value [18]. Similarly, type inference

is used to determine the type of input value. This helps in better performance of the gradient descent algorithm. Finally, dynamic taint analysis is used to solve the problem of type and shape inference.

3.4.5 Input length exploration

During taint analysis, Angora associates the destination memory in *read* function calls with byte offsets in input [18]. If a return value is used in a conditional statement and the constraints of the same are not satisfied, the input length is increased by Angora such that *read* call can get all the requested bytes.

3.4.6 Fuzzer Usage

Angora Clang compiler is needed to generate fuzzer binaries required for fuzzing a targeted program. The compiler then generates a regular fuzzer binary and a taint tracked fuzzer binary. Initial seeds/inputs are needed for Angora [17] to function. The fuzzer can then be invoked with the command below

```
./angora_fuzzer -i input -o output -t path/to/taint/program – path/to/fast/program [argv]
```

To understand how Angora fuzzer uses the techniques explained previously we can consider the following program as an example.

```
1  #include "stdint.h"
2  #include "stdio.h"
3  #include "stdlib.h"
4  #include "string.h"
5
6  int main(int argc, char **argv) {
7
8   if (argc < 2)
9      return 0;
10
11 FILE *fp;
12 char buf[255];
13 size_t ret;
14
15 fp = fopen(argv[1], "rb");
16
17 if (!fp) {
18     printf("st err\n");
19     return 0;
20 }
21
```

```
22      int len = 20;
23      ret = fread(buf, sizeof *buf, len, fp);
24      fclose(fp);
25      if (ret < len) {
26          printf("input fail \n");
27          return 0;
28      }
29
30      uint16_t x = 0;
31      int32_t y = 0;
32      int32_t z = 0;
33      uint32_t a = 0;
34
35      memcpy(&x, buf + 1, 2);   // x 1 - 2
36      memcpy(&y, buf + 4, 4);   // y 4 - 7
37      memcpy(&z, buf + 10, 4); // 10 - 13
38      memcpy(&a, buf + 14, 4); // 14 - 17
39      if (x > 12300 && x < 12350 && z < -100000000 && z > -100000005 && z
        != -100000003 && y >= 987654321 && y <= 987654325 && a == 123456789) {
40
41          printf("hey, you hit it \n");
42          abort();
43      }
44    return 7;
45  }
```

Listing 3.4: Simple Example Program

The program first opens a file to get inputs needed. Once the file is opened, the values of the same are stored in a buffer. From this buffer, the values of x, y, z, and a are obtained. Then it runs a conditional statement. When this program needs to be fuzzed with Angora, it has to be compiled first with *angora-clang* to generate the fuzzer binaries. Figure 3.8 shows how Angora evaluates the conditional statement.

For every condition in the conditional statement of the program being fuzzed, a fuzz loop runs every time to solve constraints. In the above example program, the first condition in the conditional statement is $x < 12300$. When the fuzz loop starts, *arg1* is assigned with the value obtained from the input file, and *arg2* is assigned with the right-hand value of the first condition. These arguments' types are checked to determine whether they are *signed* or *unsigned*. If signed, the values of *arg1* and *arg2* are translated and then assigned to *a* and *b*. The values are assigned directly to *a* and *b* if unsigned. These values are checked with the constraints defined in *output.rs* file of the fuzzer, and the output is calculated based on that. If this calculated output is zero, then the first condition is explored. If not (output != 0), Gradient Descent is applied. First, Angora increments *a* by one and *b = arg2* to check for

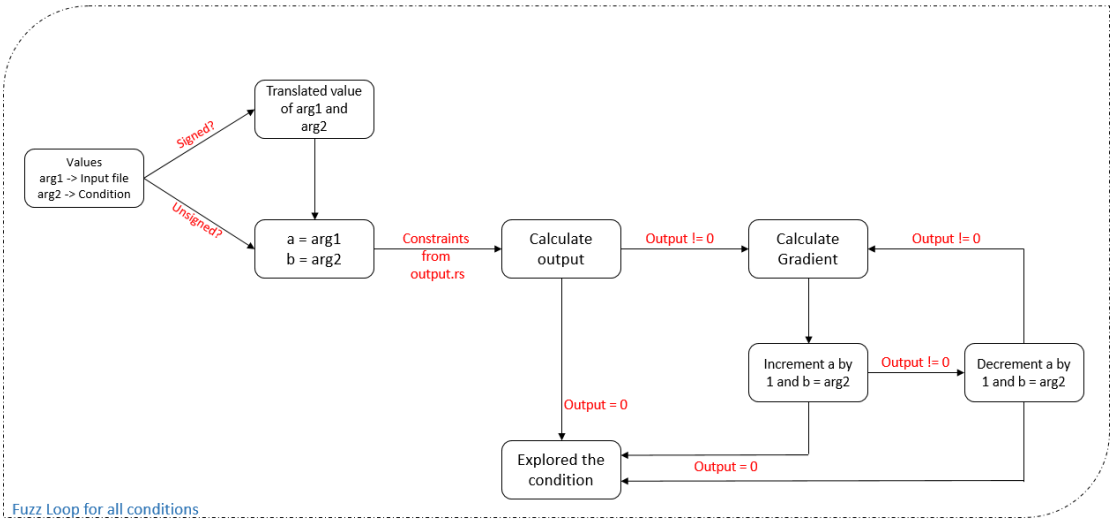


Figure 3.8: Simple Workflow of Angora

output. If zero, the condition is explored, and if not zero, then *a* is decremented by one and *b = arg2* to check for output. This fuzz loop is executed for all conditions and stops when all conditions are explored. Figure 3.9 shows the output of Angora Fuzzer.

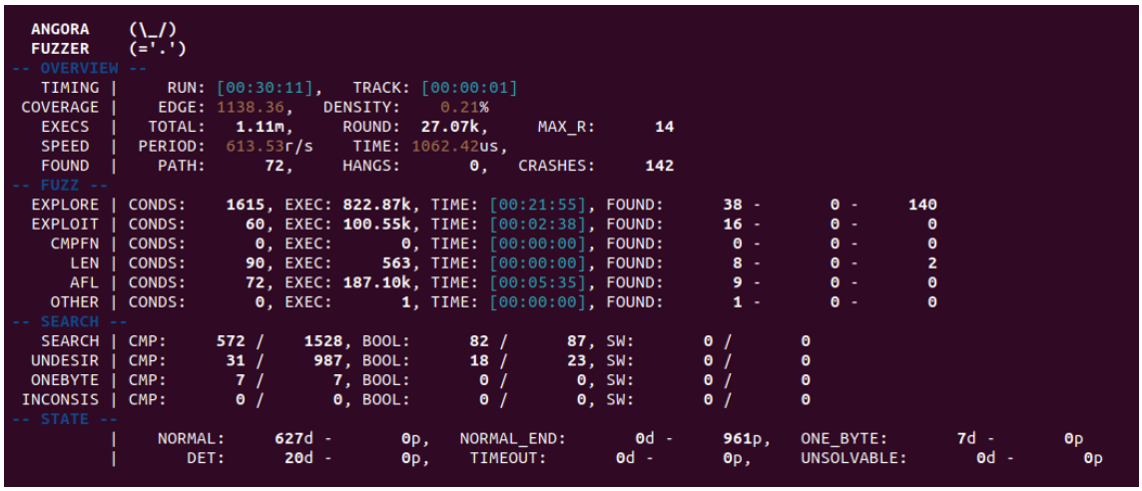


Figure 3.9: Angora Fuzzer

Chapter 4

Experiments and Discussion

This chapter describes the experiments conducted on targeted programs to evaluate the fuzzers. The obtained results are then used to compare the fuzzers.

4.1 Fuzz Test

The dependencies needed to fuzz a program with all three fuzzers as explained in fuzzer usage 3.2.2, 3.3, 3.4.6 sections in Chapter 3 has to be installed initially. The targeted program is then written in C-language as shown in 43. This program first opens a file to get input from the file and stores the contents of the file in a buffer, as shown in lines 25 - 35. The contents from this buffer are then copied to *a* and **b**. When the *calculate* function is called on line 40, the function enters a conditional statement and performs a few things as shown in line 8 - 17.

The program 43 is designed to test the fuzzers by manually injecting bugs or vulnerabilities as shown in lines 11, 12, and 16. The bugs on lines 12, and 16 are memory-related bugs/vulnerabilities as explained in Section 3.1, whereas the bug on line 11 is undefined behavior bug/vulnerability. The program is then compiled with the dependencies installed according to the fuzzer and further explained in the sub-sections below.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <stdint.h>
5
6   int __attribute__((noinline)) calculate(int x, int y)
7   {
8       if(y >= x)
9       {
10    int op1 = x + y + 100000000000;
11    char* buff1=(char*)malloc(op1); /*Test2 integer overflow*/
```

```
12      return (x / y);  /*Test1 Divide by zero*/
13          }
14          else
15          {
16      return (y / x);
17          }
18      }
19
20      int main(int argc, char **argv) {
21      if (argc < 2) {
22          fprintf(stderr, "usage: %s <input-file>\n", argv[0]);
23          return 0;
24      }
25      FILE *fp;
26      char buf[255];
27      size_t ret;
28
29      fp = fopen(argv[1], "rb");
30      if (!fp) {
31          printf("Cannot open file\n");
32          return 0;
33      }
34      ret = fread(buf, sizeof *buf, 255, fp);
35      fclose(fp);
36
37      int a, b, output;
38      memcpy(&a, buf, sizeof (int));
39      memcpy(&b, buf + sizeof(int), sizeof(int));
40      output = calculate(a, b);
41      return 0;
42      }
```

Listing 4.1: Buggy Fuzz Test Program

4.1.1 Fuzzing with American Fuzzy Lop(AFL)

The targeted *fuzztest*.c program 43 is compiled with *afl-gcc* compiler to get AFL instrumented binary. During this compilation process, multiple flags and sanitizers 3.1 can be set, which will help identify different types of bugs or vulnerabilities when fuzzing.

```
AFL_HARDEN=1 AFL_USE_ASAN=1 afl-gcc -ggdb -O0 fuzztest.c -o fuzztest
```

The above command is executed, and the resulting AFL fuzzer binary is named *fuzztest*. From the command, we can observe that the Address Sanitizer is enabled

to detect memory-related bugs using *AFL_USE_ASAN=1* as a flag for the compiler. Furthermore, the *AFL_HARDEN=1* flag causes the CC wrapper to enable code hardening options automatically to detect simple memory bugs. To fuzz with AFL, as explained in section 3.2.2 of Chapter 3, we first need to create an input and output directory. From program lines 25 - 35, we can see that the program needs inputs from a file. For this reason, we need to create an initial input and place it in the input directory. The fuzzer can then be invoked with the following command.

```
afl-fuzz -i input -o output – ./fuzztest @@
```

For the first test run, only one bug was introduced in the code inside the calculate function 6, as shown in 5

```
1  int __attribute__((noinline)) calculate(int x, int y)
2  {
3   return (x / y);  /*Test1 Divide by zero*/
4  }
```

Listing 4.2: Calculate function

When fuzzed with AFL, this program 5 detects the Divide by Zero 3 and considers this a crash. The fuzzing session was interrupted after three minutes, and the resulting AFL UI is shown in Fig 4.1. The same crash is detected multiple times with the random inputs that are generated by the fuzzer. Only one crash of this particular type is considered unique, and the input that caused this is stored in the output folder.

The input that caused the crash can be used as the initial input, and when the fuzzer is invoked via GDB, the resulting output is as shown in Fig 4.2. The purpose of running the fuzzer out of GDB is to debug the crash entirely, and we can see the divide by zero error introduced to test the fuzzer.

For the second test run, the complete fuzztest program 43 is compiled with the same command used for the first test run. The program is then fuzzed with AFL and detects two unique crashes. The input that caused the crash is used as initial input to the fuzzer invoked via GDB, and the resulting output is as shown in Fig 4.3 and Fig 4.4.

4.1.2 Fuzzing with LibFuzzer

The targeted program 43 should be modified to a Fuzz Target, as explained in Section 3.3 of Chapter 3. The resulting modified code is as shown in 29. The functioning of the modified program is similar to the one explained in Section 4.1 except that the fuzzer no longer needs an initial input which has to be provided by the user. In ad-

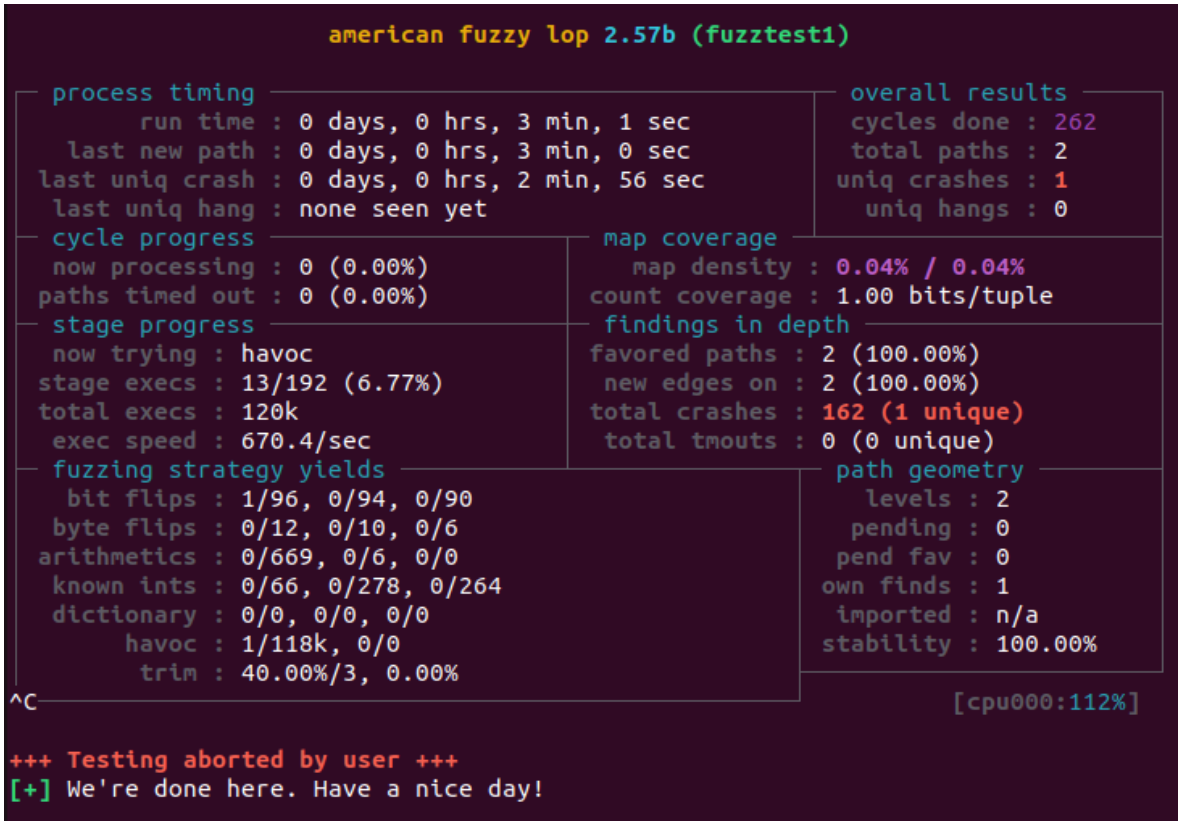


Figure 4.1: AFL 1 Bug

dition, the main function in program 43 is changed to the LLVMFuzzerTestOneInput function, which assigns the values to *a* and *b* and calls the calculate function.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <stdint.h>
5
6   int __attribute__((noinline)) calculate(int x, int y)
7   {
8       if(y >= x)
9       {
10          int op1 = x + y + 10000000;
11          char* buff1=(char*)malloc(op1);  /*Test2 integer overflow*/
12          return (x / y);  /*Test1 Divide by zero*/
13      }
14      else
15      {
16          return (y + x);
17      }
18  }
19  int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size){
20      if (size < 2 * sizeof(int))
```

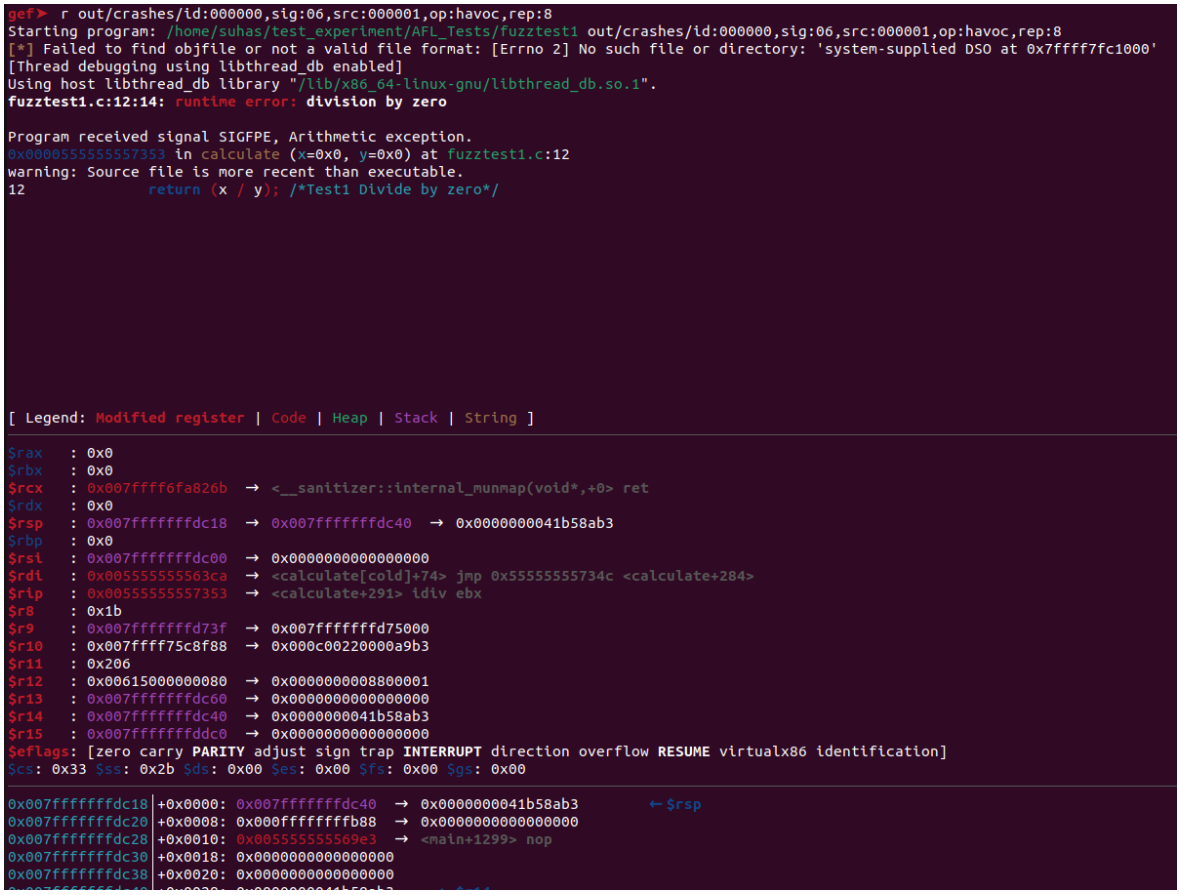


Figure 4.2: AFL 1 Bug using GDB

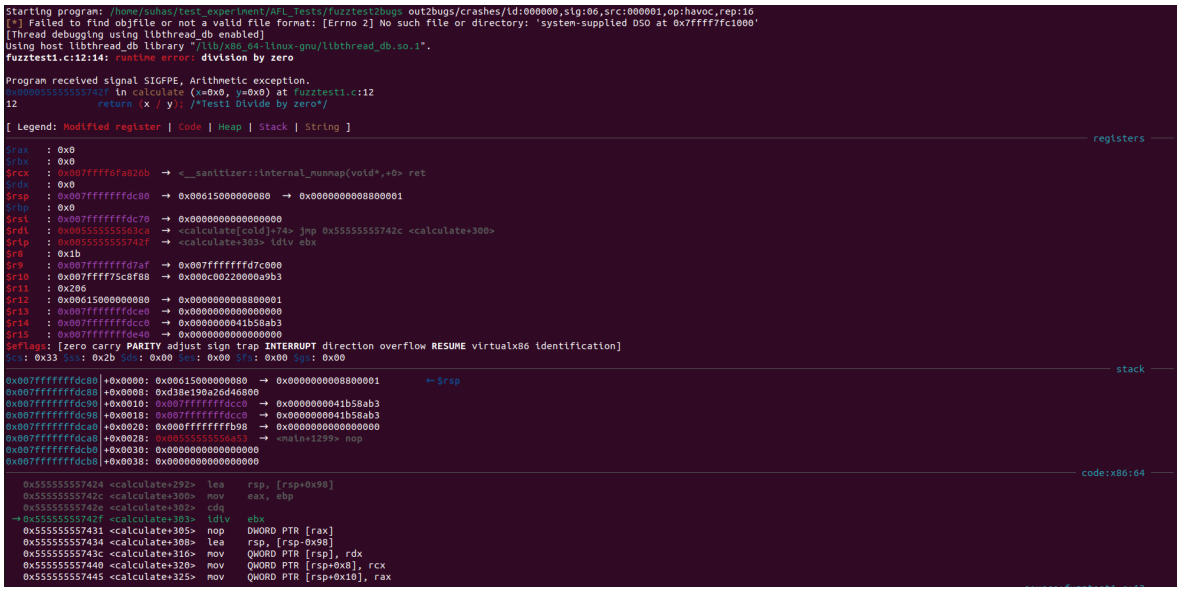


Figure 4.3: AFL 1st Bug using GDB

```
21          {
22              return 0;
```

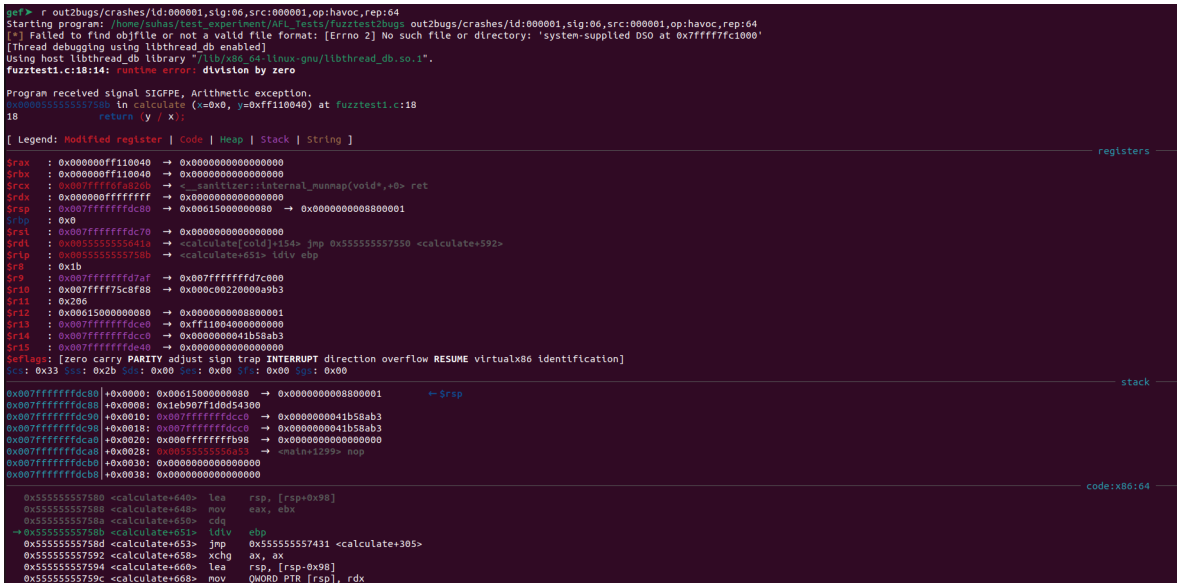


Figure 4.4: AFL 2nd Bug using GDB

```
23          }
24      const int a = *((const int *)(data) + 0);
25      const int b = *((const int *)(data) + 1);
26      const int result = calculate(a,b);
27      return 0;
28  }
```

Listing 4.3: Buggy Fuzz Test Program for LibFuzzer

For the first test run, the calculate function in program 29 was written the same as 5. The program is then compiled using *Clang* compiler with flags and sanitizers 3.1 to build a Libfuzzer instrumented binary. The command used to compile is shown below.

```
clang -fsanitize=fuzzer,address,undefined -g fuzztest.c -o fuzztest
```

The flag *-fsanitize=fuzzer* performs necessary instrumentation and links the Libfuzzer's *main()* symbol, and the remaining flags *-fsanitize=address,undefined* enables the address sanitizer and undefined behavior sanitizer to detect memory-related bugs and undefined behavior. An input directory may be created to store the input that caused the crash. The generated fuzzer binary is run *./fuzztest corpus/* to start fuzzing with LibFuzzer, where the fuzzer generates random input and stops when the first crash is detected. The detected crash is stored in the current directory where the fuzzer binary is run. For this particular program 29 with the calculate function 5, the Divide by Zero error on line 3 is detected by the fuzzer. Fuzzer, when run with an input directory named *corpus/* crashes at the error and prints the output as shown in Fig 4.5.

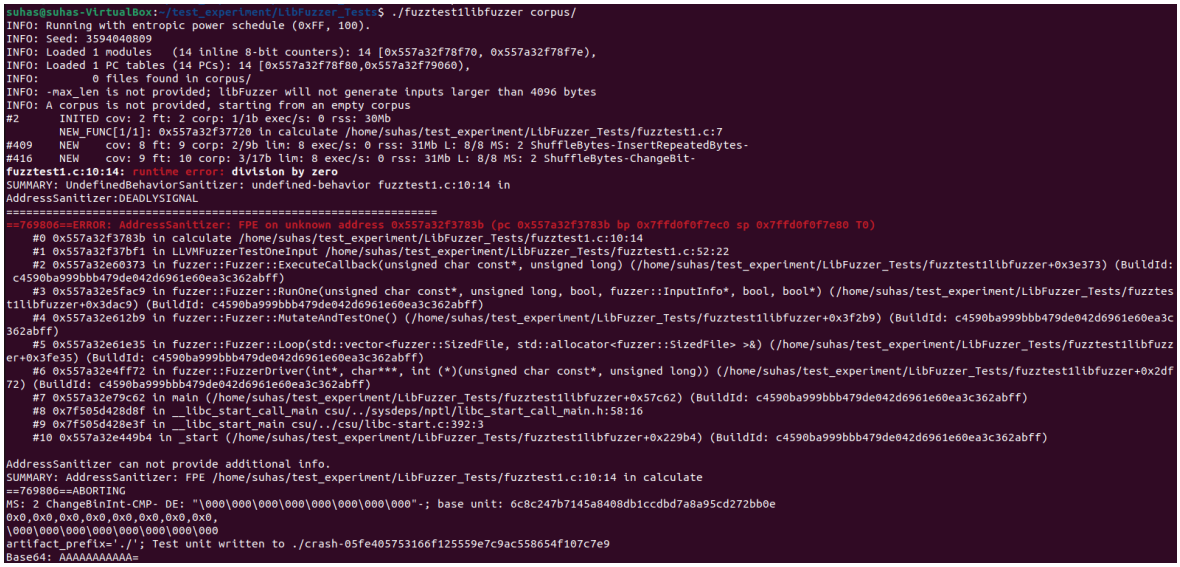


Figure 4.5: LibFuzzer 1st Bug

During the second test run, the program 29 is compiled the same way as the first test run. The generated fuzzer binary, when run with an input folder, will stop at the first crash, thereby detecting the Divide By Zero error in line 12. If the fuzzer has to detect the second bug 11, the fuzzer has to be run with the command below where *-ignore_crashes=1* is a flag that ignores the first crash, and *-detect_leaks=0* enables the Leak Sanitizer. The resulting output is shown in Fig 4.6.

```
./fuzztest corpus/ -ignore_crashes=1 -detect_leaks=0
```

4.1.3 Fuzzing with Angora Fuzzer

The targeted program 43 is compiled using *angora-clang* compiler with flags and sanitizers to obtain fast and taint binaries of the fuzzer. The command used for compiling is shown below.

```
USE_FAST=1 ANGORA_USE_ASAN=1 angora-clang fuzztest.c -lz -o fuzztest.fast
USE_TRACK=1 angora-clang fuzztest.c -lz -o fuzztest.taint
```

The *ANGORA_USE_ASAN=1* enables the address sanitizer 3.1. First, input and output directories should be created to store the initial input and crashes obtained after a fuzzing session. Then, using the command below, the fuzzer is invoked with both fast and taint binaries.

```
./angora_fuzzer -i input -o output -t ./fuzztest.taint – ./fuzztest.fast @@
```

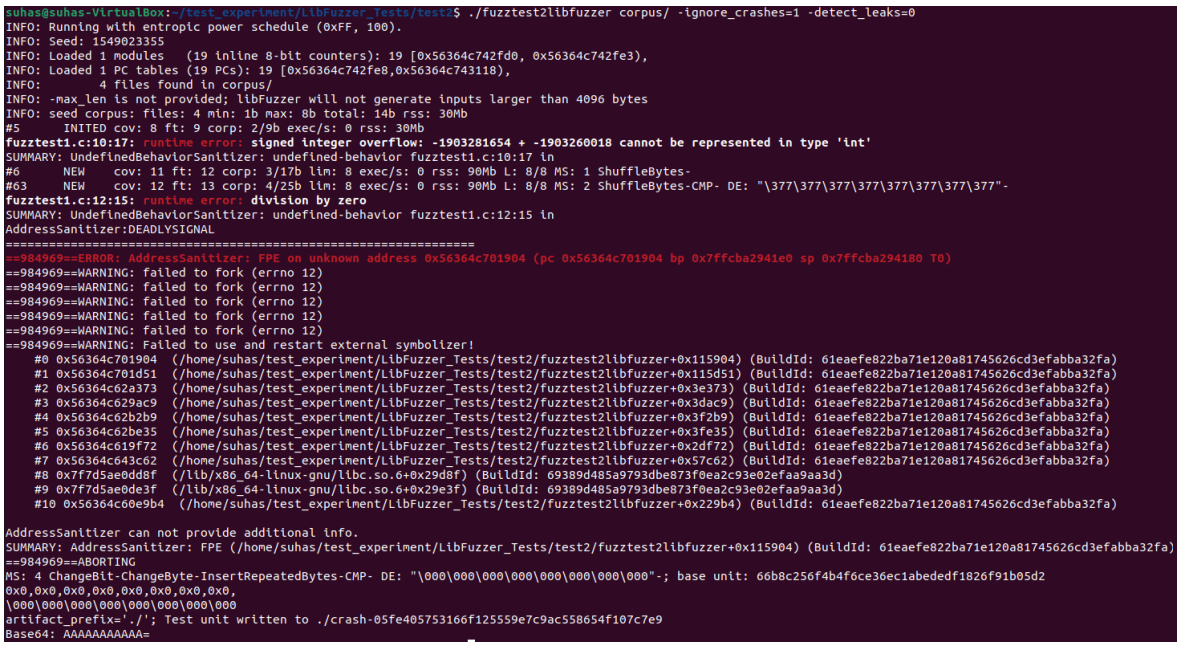


Figure 4.6: LibFuzzer 2 Bugs

Angora fuzzer fails to fuzz the targeted program and does not detect any bugs as it tries to solve constraints, and in the program 43, there is only one conditional statement.

4.2 FuzzGoat - A Buggy JSON Parser

This set of experiments is implemented on an open-source, intentionally insecure JSON parser [31]. Using JavaScript Object Notation(JSON) format, the JSON parser reads and writes entries. The program contains approx.1300 lines of code and named *fuzzgoat*, which is injected with several memory corruption vulnerabilities where the exact number is unknown to test the fuzzers.

4.2.1 Fuzzing with American Fuzzy Lop(AFL)

The *fuzzgoat.c* program is compiled with similar flags and sanitizers using the same command as in sub-section 3.3. The fuzzing session was done for 30 minutes, and the fuzzer detected 25 unique crashes, and the output UI of AFL is shown in Fig 4.7.

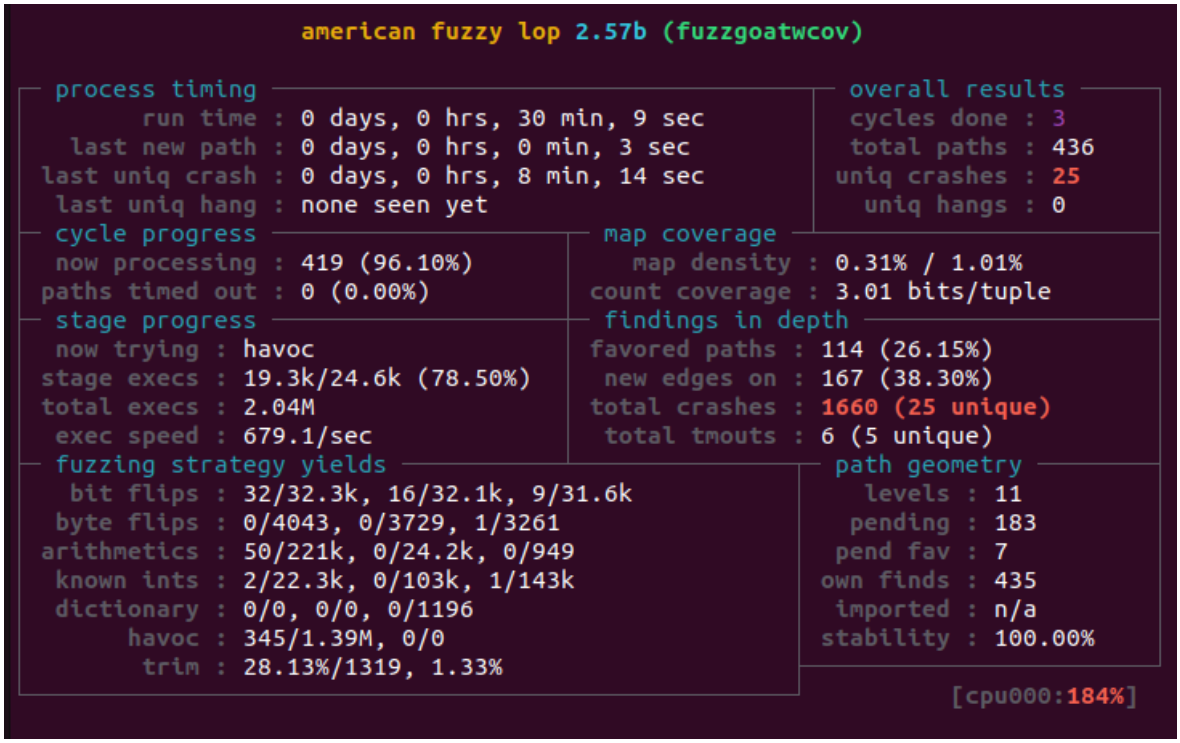


Figure 4.7: FuzzGoat - Fuzzing with AFL

4.2.2 Fuzzing with LibFuzzer

For fuzzing with LibFuzzer, the program is compiled with flags and sanitizers as in sub-section 4.1.2. The instrumented binary is run with an input folder, flags for ignoring the first crash to prevent the fuzzer from exiting, and enabling leak sanitizer. The fuzzing session was done for 30 minutes. The results are as shown in Fig 4.8 and are further explained in Section 4.4

4.2.3 Fuzzing with Angora Fuzzer

The fast and taint binaries are obtained by compiling the *fuzzgoat.c* program with flags and sanitizers similar to the previous experiment in sub-section 4.1.3. The fuzzer is then run with the generated binaries, and the fuzzing session is interrupted after 30 minutes. From the output UI, as shown in Fig 4.9, we can observe that Angora Fuzzer again fails to detect any bugs or vulnerabilities in a buggy JSON parser program.

```
fuzzgoat.c:529:65: runtime error: applying non-zero offset 2 to null pointer
fuzzgoat.c:298:29: runtime error: load of null pointer of type 'char'
==3608809==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x55f876f59e55 bp 0x7fff8ddcfd10 sp 0x7fff8ddcf560 T0)
#4754443: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5628 time: 1044s job: 5629 dft_time: 0
fuzzgoat.c:529:65: runtime error: applying non-zero offset 2 to null pointer
==3608813==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x603000003952 at pc 0x5563c868497e bp 0x7ffc84c59790 sp 0x7ffc84c59788
#4754886: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5629 time: 1044s job: 5630 dft_time: 0
==3608817==ERROR: AddressSanitizer: attempting free on address which was not malloc()-ed: 0x60200000646f in thread T0
#4756656: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5630 time: 1044s job: 5631 dft_time: 0
fuzzgoat.c:529:65: runtime error: applying non-zero offset 2 to null pointer
fuzzgoat.c:298:29: runtime error: load of null pointer of type 'char'
==3608822==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x56431ed8be55 bp 0x7ffd5c9c6970 sp 0x7ffd5c9c61c0 T0)
#4756767: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5631 time: 1045s job: 5632 dft_time: 0
fuzzgoat.c:529:65: runtime error: applying non-zero offset 11 to null pointer
==3608826==ERROR: AddressSanitizer: attempting free on address which was not malloc()-ed: 0x602000002bef in thread T0
#4757159: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5632 time: 1045s job: 5633 dft_time: 0
==3608830==ERROR: AddressSanitizer: attempting free on address which was not malloc()-ed: 0x602000005fcf in thread T0
#4758305: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5633 time: 1045s job: 5634 dft_time: 0
==3608834==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000c599 at pc 0x55e1a524497e bp 0x7ffd8e22db50 sp 0x7ffd8e22db48
#4758665: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5634 time: 1045s job: 5635 dft_time: 0
fuzzgoat.c:529:65: runtime error: applying non-zero offset 2 to null pointer
==3608838==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000005bfc at pc 0x55976b46b97e bp 0x7ffc50630890 sp 0x7ffc50630888
#4758709: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5635 time: 1045s job: 5636 dft_time: 0
fuzzgoat.c:529:65: runtime error: applying non-zero offset 2 to null pointer
==3608842==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6020000017b8 at pc 0x55fe4ac6497e bp 0x7ffef1bf1fb0 sp 0x7ffef1bf1fa8
#4758755: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5636 time: 1045s job: 5637 dft_time: 0
fuzzgoat.c:529:65: runtime error: applying non-zero offset 11 to null pointer
fuzzgoat.c:298:29: runtime error: load of null pointer of type 'char'
==3608847==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x55ea15f72e55 bp 0x7fff58dd6710 sp 0x7fff58dd5f60 T0)
#4759113: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5637 time: 1046s job: 5638 dft_time: 0
fuzzgoat.c:529:65: runtime error: applying non-zero offset 11 to null pointer
fuzzgoat.c:298:29: runtime error: load of null pointer of type 'char'
==3608851==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x56344f491e55 bp 0x7ffdfa0508b0 sp 0x7ffdfa050100 T0)
#4761795: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5638 time: 1046s job: 5639 dft_time: 0
==3608855==ERROR: AddressSanitizer: attempting free on address which was not malloc()-ed: 0x60200001a76f in thread T0
#4762767: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5639 time: 1046s job: 5640 dft_time: 0
fuzzgoat.c:529:65: runtime error: applying non-zero offset 2 to null pointer
==3608859==ERROR: AddressSanitizer: attempting free on address which was not malloc()-ed: 0x60200000acef in thread T0
#4762812: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5640 time: 1046s job: 5641 dft_time: 0
fuzzgoat.c:529:65: runtime error: applying non-zero offset 2 to null pointer
==3608863==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000001698 at pc 0x55d1b389597e bp 0x7ffd79b911d0 sp 0x7ffd79b911c8
#4763055: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5641 time: 1046s job: 5642 dft_time: 0
fuzzgoat.c:529:65: runtime error: applying non-zero offset 2 to null pointer
fuzzgoat.c:298:29: runtime error: load of null pointer of type 'char'
==3608867==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x56128e71de55 bp 0x7ffd7fb92b50 sp 0x7ffd7fb923a0 T0)
#4763438: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5642 time: 1046s job: 5643 dft_time: 0
fuzzgoat.c:529:65: runtime error: applying non-zero offset 2 to null pointer
==3608872==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000005619 at pc 0x55defa5bf97e bp 0x7ffc9d0bfd10 sp 0x7ffc9d0bfd08
#4763478: cov: 654 ft: 2171 corp: 367 exec/s 0 oom/timeout/crash: 0/0/5643 time: 1047s job: 5644 dft_time: 0
```

Figure 4.8: FuzzGoat - Fuzzing with LibFuzzer

```
  ANGORA    (\_/)
  FUZZER    (='o') .o
-- OVERVIEW --
   TIMING |     RUN: [00:30:12],   TRACK: [00:00:00]
 COVERAGE |    EDGE:   95.85%,   DENSITY:    0.05%
    EXECS |   TOTAL: 149.68k,     ROUND:    6101,     MAX_R:    5527
    SPEED |  PERIOD:   82.61r/s    TIME:  530.37us,
    FOUND |    PATH:     230,     HANGS:       0,   CRASHES:       0
-- FUZZ --
  EXPLORE | CONDS:     618, EXEC: 130.70k, TIME: [00:26:19], FOUND:     117 -       0 -       0
  EXPLOIT | CONDS:      40, EXEC:  18.76k, TIME: [00:03:48], FOUND:      94 -       0 -       0
    CMPFN | CONDS:       0, EXEC:       0, TIME: [00:00:00], FOUND:       0 -       0 -       0
      LEN | CONDS:      33, EXEC:     222, TIME: [00:00:02], FOUND:      18 -       0 -       0
      AFL | CONDS:       0, EXEC:       0, TIME: [00:00:00], FOUND:       0 -       0 -       0
    OTHER | CONDS:       0, EXEC:       1, TIME: [00:00:00], FOUND:       1 -       0 -       0
-- SEARCH --
   SEARCH | CMP:     192 /     206, BOOL:      16 /      17, SW:     395 /     395
  UNDESIR | CMP:      11 /      25, BOOL:      16 /      17, SW:       0 /       0
  ONEBYTE | CMP:     190 /     190, BOOL:      16 /      16, SW:     395 /     395
 INCONSIS | CMP:       0 /       0, BOOL:       0 /       0, SW:       0 /       0
-- STATE --
          |    NORMAL:       2d -       0p,   NORMAL_END:       0d -       1p,   ONE_BYTE:     601d -       0p
          |       DET:       0d -       0p,    TIMEOUT:       0d -       0p,    UNSOLVABLE:       0d -      14p
```

Figure 4.9: FuzzGoat - Fuzzing with Angora Fuzzer

4.3 Test Experiment for Angora Fuzzer

As the results of previous experiments for Angora Fuzzer were not successful, we consider the *mini2.c* program, which is given as a test in the repository of fuzzer [17].

```
1 /*
```

```
  Test:
  Nested ‘if‘ conditional statements.
  It is difficult for other fuzzers, but it is easy for Angora.
*/
#include "stdint.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

int main(int argc, char **argv) {
  if (argc < 2)
    return 0;
  FILE *fp;
  char buf[255];
  size_t ret;
  fp = fopen(argv[1], "rb");
  if (!fp) {
    printf("st err\n");
    return 0;
  }
  int len = 20;
  ret = fread(buf, sizeof *buf, len, fp);
  fclose(fp);
  if (ret < len) {
    printf("input fail \n");
    return 0;
  }
  uint16_t x = 0;
  int32_t y = 0;
  int32_t z = 0;
  uint32_t a = 0;
  memcpy(&x, buf + 1, 2);  // x 0 - 1
  memcpy(&y, buf + 4, 4);  // y 4 - 7
  memcpy(&z, buf + 10, 4); // 10 - 13
  memcpy(&a, buf + 14, 4); // 14 - 17
  if (x - 12300 > 0 && x - 12350 < 0 && z + 100000000 < 0 && z +
   100000005 > 0 && z + 100000003 != 0 && y - 987654321 >= 0 && y -
   987654325 <= 0 && a - 123456789 == 0) {
    printf("hey, you hit it \n");
    // abort();
    /* _exit(6); */
    }
    return 0;
}
```

Listing 4.4: Mini2

The working of this program is similar to the one explained in Section 3.4.6. First, the

fuzzer tries to solve all eight constraints of the conditional statement, as shown in Fig 4.10. Then, inputs are generated by the fuzzer for every condition in the program line 37, and once all the constraints are solved, the fuzzer exits with a success message **Solve all constraints!!**.

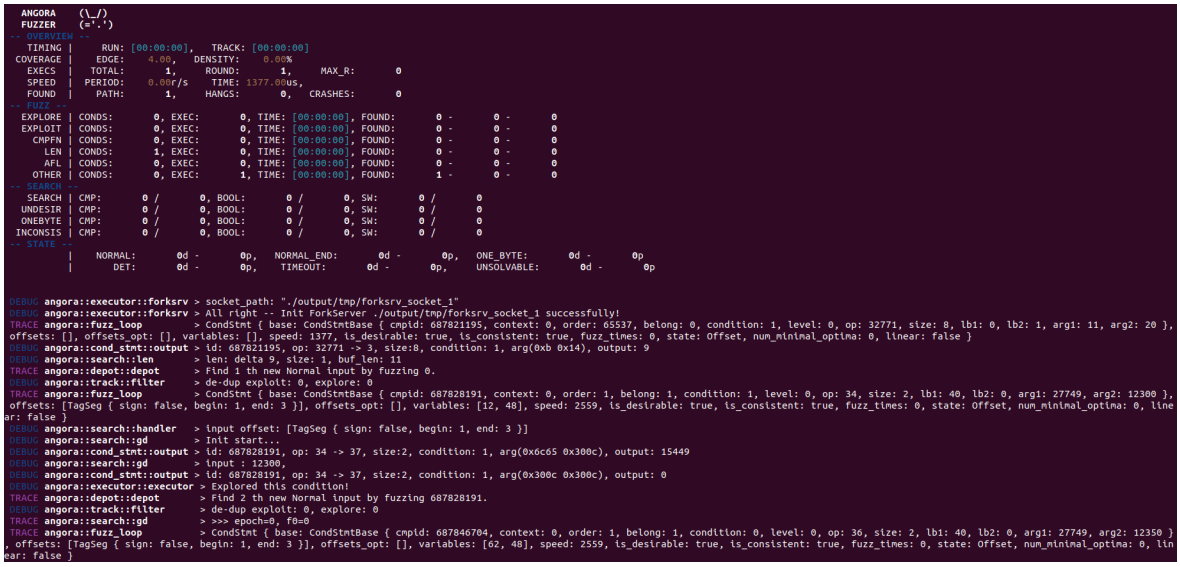


Figure 4.10: Angora Fuzzer Test Experiment output

4.4 Comparison of Fuzzers

This section explains the different program metrics that are used on *fuzztest* 43 and *fuzzgoat* 4.2 experiments to evaluate the results and compare all the fuzzers.

4.4.1 Code Coverage

Code coverage in fuzzing is a testing metric determining the number of lines, functions, and branches validated successfully under a fuzzing session, which helps analyze how well the targeted program is verified. Several tools and compiler flags can be used to generate code coverage. The subsections below explain the usage and results of *fuzzgoat* 4.2 with all the fuzzers.

American Fuzzy Lop Coverage

AFL has an extension tool called *afl-cov* [32] [33] to measure the code coverage. The input files generated by the fuzzer for the targeted binary are used by *afl-cov* to produce code coverage results. *afl-cov* interprets code coverage from one input

to the next to determine which new lines and functions are hit with every newly generated input by AFL. *afl-cov* requires the following dependencies to be installed.

- afl-fuzz
- gcov, lcov, genhtml
- python

The code coverage of *fuzzgoat* 4.2 is measured by creating a copy of the complete project at two different directories, namely */path/to/afl-fuzz-output/* and */path/to/gcov-project*. The directory */path/to/afl-fuzz-output/* contains fuzzer instrumented binary, and the */path/to/gcov-project* directory contains fuzzer binary compiled for gcov profiling support. The command to compile *fuzzgoat* with gcov profiling support is shown below.

```
AFL_HARDEN=1 AFL_USE_ASAN=1 afl-gcc -fprofile-arcs -ftest-coverage -ggdb
-O0 fuzzgoat.c -o fuzzgoatwcov
```

Next, *afl-cov* is started in *–live* mode before starting the fuzzing with fuzzer instrumented binary. */path/to/afl-fuzz-output/* should be specified as a command line argument for *afl-cov* and commands to execute with arguments is shown below. If we wish to generate code coverage for an existing output directory of AFL, omit the *–live* argument.

```
$ cd /path/to/project-gcov/
$ afl-cov -d /path/to/afl-fuzz-output/ –live –coverage-cmd
"cat AFL_FILE | LD_LIBRARY_PATH=./lib/.libs ./bin/.libs/somebin -a -b -c"
–code-dir .
```

The *AFL_FILE* string in the command corresponds to the inputs generated by AFL, that are stored in the *queue/* directory in */path/to/project-fuzz*. *afl-cov* automatically substitutes this *AFL_FILE* with the ID of newly generated inputs *queue/id:NNNNNNN** in the order in which it is being generated while building the code coverage.

With *afl-cov* running, start a fuzzing session with AFL using the command below in a new terminal.

```
$ cd /path/to/fuzzgoat $ afl-fuzz -i input -o output – ./fuzzgoat @@
```

The terminal in which *afl-cov* is running starts measuring the code coverage once the fuzzer generates more inputs in the other terminal. The resulting terminal output is shown in Fig 4.11.

In the */path/to/afl-fuzz-output* directory, *afl-cov* creates a few directories to store the coverage result. These directories are

```
*** Imported 3 new test cases from: out//queue

    [+] AFL test case: id:000413,src:000406,op:havoc,rep:4 (0 / 416), cycle: 0
        lines......: 81.5% (454 of 557 lines)
        functions..: 92.3% (12 of 13 functions)
        branches...: 70.1% (296 of 422 branches)
    [+] Final lcov web report: out//cov/web/index.html
    [+] AFL test case: id:000414,src:000406,op:havoc,rep:2 (1 / 416), cycle: 0
        lines......: 81.5% (454 of 557 lines)
        functions..: 92.3% (12 of 13 functions)
        branches...: 70.1% (296 of 422 branches)
    [+] Final lcov web report: out//cov/web/index.html
    [+] AFL test case: id:000415,src:000406,op:havoc,rep:2 (2 / 416), cycle: 0
        lines......: 81.5% (454 of 557 lines)
        functions..: 92.3% (12 of 13 functions)
        branches...: 70.1% (296 of 422 branches)
    [+] Final lcov web report: out//cov/web/index.html

*** Imported 1 new test cases from: out//queue

    [+] AFL test case: id:000416,src:000406,op:havoc,rep:2,+cov (0 / 417), cycle: 0
        lines......: 81.5% (454 of 557 lines)
        functions..: 92.3% (12 of 13 functions)
        branches...: 70.1% (296 of 422 branches)
    [+] Final lcov web report: out//cov/web/index.html
    [-] No new AFL test cases, sleeping for 60 seconds

*** Imported 15 new test cases from: out//queue

    [+] AFL test case: id:000417,src:000406,op:havoc,rep:2,+cov (0 / 432), cycle: 0
        lines......: 81.5% (454 of 557 lines)
        functions..: 92.3% (12 of 13 functions)
        branches...: 70.1% (296 of 422 branches)
    [+] Final lcov web report: out//cov/web/index.html
    [+] AFL test case: id:000418,src:000406,op:havoc,rep:2,+cov (1 / 432), cycle: 0
        lines......: 81.5% (454 of 557 lines)
        functions..: 92.3% (12 of 13 functions)
        branches...: 70.1% (296 of 422 branches)
    [+] Final lcov web report: out//cov/web/index.html
    [+] AFL test case: id:000419,src:000406,op:havoc,rep:2,+cov (2 / 432), cycle: 0
        lines......: 81.5% (454 of 557 lines)
        functions..: 92.3% (12 of 13 functions)
        branches...: 70.1% (296 of 422 branches)
    [+] Final lcov web report: out//cov/web/index.html
    [+] AFL test case: id:000420,src:000407,op:flip2,pos:11 (3 / 432), cycle: 0
        lines......: 81.5% (454 of 557 lines)
        functions..: 92.3% (12 of 13 functions)
        branches...: 70.1% (296 of 422 branches)
    [+] Final lcov web report: out//cov/web/index.html
```

Figure 4.11: AFL Code Coverage

- cov/diff/ - Contains code coverage results when *queue/id:NNNNN** input file causes the fuzzer to execute a new location of code.
- cov/web/ - Contains code coverage results produced by genhtml in web format.
- cov/lcov/ - Contains raw code coverage data produced by *lcov*
- cov/zero-cov/ - Contains data for each input on functions that are never exe-

cuted by the fuzzer.

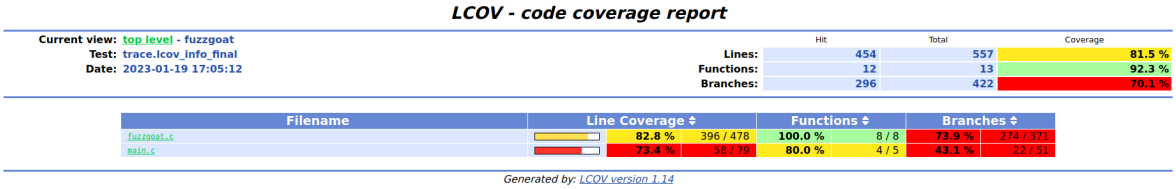


Figure 4.12: Code Coverage Web Report - AFL

From the code coverage reports generated by *afl-cov* shown in Fig 4.12, we can observe that the inputs generated by the fuzzer was successful in covering 12 out of 13 functions, 454 out of 557 lines, and 296 out of 422 branches, resulting in a coverage of 92.3%, 81.5%, and 70.1% respectively on the targeted program *FuzzGoat* 4.2

LibFuzzer Coverage

To measure the code coverage of *fuzzgoat* with LibFuzzer 4.2.2, the program should be compiled with coverage enabled [34], which is set as a flag during compilation, and the command used is shown below.

```
clang -fsanitize=fuzzer,address,undefined -fprofile-instr-generate -fcoverage-mapping fuzzgoat.c -o fuzzgoatwithcov
```

The next step is to run the fuzzer instrumented binary, and when the program exits, *raw profile* is written to the path specified by *LLVM_PROFILE_FILE* environment variable. It is written to *default.profraw* in the current directory if no variable exists. Below mentioned command is used to perform this step.

```
LLVM_PROFILE_FILE="fuzzgoat.profraw" ./fuzzgoatwithcov
```

The raw profiles created in the previous step should be indexed to generate code coverage reports. The *merge* tool in *llvm-profdata* combines multiple raw profiles and indexes them simultaneously. The command used for this step is shown below.

```
llvm-profdata merge -sparse fuzzgoat.profraw -o fuzzgoat.profdata
```

Finally, a file-level code coverage report can be generated with statistics as shown in Fig 4.13, using the command below.

```
llvm-cov report ./fuzzgoatwithcov -instr-profile=fuzzgoat.profdata
```

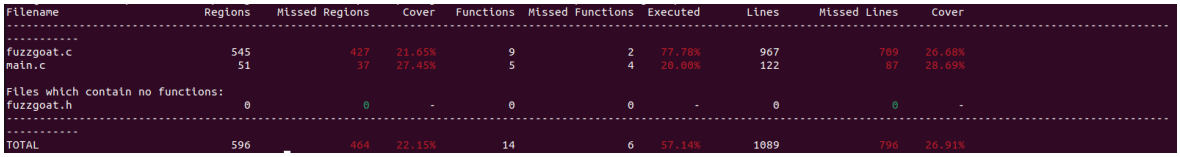
```
suhas@suhas-VirtualBox:~/experiment/libfuzzer-cov$ llvm-cov report ../fuzzgoat/fuzzgoatlibFuzzerwcov -instr-profile=test2.profdata
Filename                      Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed Lines     Cover    Branches   Missed Branches     Cover
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
fuzzgoat.c                        547                94    82.82%          9                 0   100.00%         702                137    80.48%         434                 87    79.95%
main.c                             39                11    71.79%          6                 1    83.33%          75                 23    69.33%          34                  9    73.53%

Files which contain no functions:
fuzzgoat.h                          0                 0         -          0                 0         -           0                  0         -           0                  0         -
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
TOTAL                             586               105    82.08%         15                 1    93.33%         777                160    79.41%         468                 96    79.49%
```

Figure 4.13: Code Coverage Report - LibFuzzer

The mutation strategies and sanitizers flags used by LibFuzzer successfully cover 481 regions out of 581, 14 out of 15 functions, 617 out of 777 lines, and 372 out of 468 branches, resulting in a coverage of 82.08%, 93.33%, 79.41%, and 79.46% respectively on the targeted program *FuzzGoat* 4.2 as shown in Fig 4.13.

Angora Fuzzer Coverage

The steps followed to measure the code coverage of *fuzzgoat* using Angora Fuzzer 4.2.3 is similar to LibFuzzer code coverage 4.4.1. Therefore, the program *fuzzgoat*



Filename	Regions	Missed Regions	Cover	Functions	Missed Functions	Executed	Lines	Missed Lines	Cover
fuzzgoat.c	545	427	21.65%	9	2	77.78%	967	709	26.68%
main.c	51	37	27.45%	5	4	20.00%	122	87	28.69%
Files which contain no functions:									
fuzzgoat.h	0	0	-	0	0	-	0	0	-
TOTAL	596	464	22.15%	14	6	57.14%	1089	796	26.91%

Figure 4.14: Code Coverage Report - Angora Fuzzer

should be compiled for fast and taint instrumented binaries with coverage enabled. Fig 4.9 shows the output, and we can observe that no crashes were detected.

Angora fuzzer fails to solve the path constraints, due to which it failed to detect any crashes in the buggy JSON parser *FuzzGoat* 4.2. The fuzzer was successful only in covering 132 out of 596 regions, 10 out of 14 functions, and 293 out of 1089 lines, resulting in a code coverage of 22.15%, 57.14%, and 26.91% respectively, as shown in Fig 4.14.

4.4.2 Types of bugs or vulnerabilities detected

The types of bugs or vulnerabilities detected by a fuzzer entirely depend on the use of Sanitizers 3.1. To evaluate this metric, we consider the fuzzing of *fuzztest* 43 program.

From the results of sub-sections 4.1.1, 4.1.2, and 4.1.3 in Section 4.1, we can observe that LibFuzzer detects two types of bugs or vulnerabilities. In contrast, Angora detects no bugs, and AFL detects only one memory-related bugs.

4.4.3 Number of bugs detected

This metric is dependent on the types of bugs or vulnerabilities detected. The metric can be evaluated considering *fuzztest* 43 program. From Fig 4.1, the bugs detected are divided into the total number and unique crashes. Unique crashes are counted based on a particular type of bug or vulnerability, and the duplicates are added to the total number. In LibFuzzer, fuzzing stops when a crash is detected, and if *-ignore_crashes=1* flag is set, multiple crashes are duplicates. So, the total number of bugs detected could be a wrong count. Whereas for the Angora fuzzer, with the main idea being solving constraints, the fuzzer might not detect any bugs and still solve constraints as explained in the test experiment for Angora fuzzer 4.3.

4.4.4 Execution Speed

Execution Speed is the speed at which the fuzzers perform mutation strategies to generate inputs for fuzzing. The execution speed of AFL and LibFuzzer is similar

as both these fuzzers use almost identical mutation strategies as explained in subsections 3.2.1, and 3.3.1 of Chapter 3. The execution speed of Angora cannot be determined as the primary goal of the fuzzer is to solve path constraints, and the size of constraints can vary from small to large.

Finally, the results are used to compare AFL, LibFuzzer, and Angora Fuzzer based on the above-mentioned metrics. The code coverage report was generated only for *FuzzGoat* 4.2 buggy JSON parser program. The open source *FuzzGoat* program has approx.1300 lines of code with bugs or vulnerabilities injected into it. In contrast, the *FuzzTest* 4.1 was designed with a limited number of bugs or vulnerabilities, and generating a code coverage report for a small codebase would not result in a code coverage that can be used to compare the three fuzzers.

The code coverage results obtained for all three fuzzers with the *FuzzGoat* being the targeted program is tabulated and shown in Table 4.1. The table shows that LibFuzzer successfully covers slightly more lines, functions, and branches than American Fuzzy Lop. At the same time, the code coverage results for Angora Fuzzer are shallow as the fuzzer did not solve all constraints and failed to detect any crashes. The newly generated inputs by the fuzzers mutation strategy should increase the coverage, and the Angora fuzzer fails to do so.

	American Fuzzy Lop	**LibFuzzer**	**Angora Fuzzer**
Code Coverage	**Lines:** 81.5% **Functions:** 92.3% **Branches:** 70.1%	**Lines:** 82.08% **Functions:** 93.33% **Branches:** 79.49%	**Lines:** 22.15% **Functions:** 57.14%

Table 4.1: Code Coverage Results for Fuzzers with FuzzGoat

The rest of the metrics explained above is used to compare all the fuzzers using *FuzzTest* 4.1 as the targeted program, and this is tabulated as shown in Table 4.2. The metric *Types of bugs or vulnerabilities detected* shows that LibFuzzer succeeds in finding two different types of bugs, namely Divide By Zero on line 12, and Signed Integer Overflow on line 11. In contrast, AFL detects only the Divide by Zero on line 12, and line 16. On the other hand, the Angora fuzzer fails to detect any bugs or vulnerabilities. Similarly, for metric *Number of bugs detected*, both AFL and LibFuzzer detect two bugs in total, and Angora detects 0 bugs. Lastly, from the metric *Execution speed*, we can observe that LibFuzzer took approx. Two seconds to detect both bugs by generating the necessary inputs for them. AFL took approx. Five seconds

to detect the same type of bug or vulnerability in which the Angora fuzzer fails to fuzz the targeted program.

	American Fuzzy Lop	**LibFuzzer**	**Angora Fuzzer**
Types of bugs or Vulnerabilities detected	1	2	0
Number of bugs detected	2	2	0
Execution speed	approx. 5seconds	approx. 2seconds	NA

Table 4.2: Evaluation of Fuzzers with FuzzTest

Chapter 5

Conclusions and recommendations

In this chapter, the project's conclusion is presented over the proposed question by research and how this work was achieved to answer the research question, therefore concluding this work. Furthermore, recommendations are discussed as a way forward for this research work.

5.1 Conclusions

This research work presented a comparison of fuzzing tools. The approach for this comparison was to understand the necessary background in fuzzing initially, then understand the working of the fuzzing tools, and run a set of experiments on targeted programs to obtain the results needed for comparison.

In this section, the research question proposed by the introduction of this work is restated and answered, which is as follows.

- **How can we compare different state-of-the-art fuzzers, which fuzzers are good at detecting security bugs/vulnerabilities and how can they be categorized?**

This question can be answered by comparing the three fuzzers, namely, American Fuzzy Lop 3.2, LibFuzzer 3.3, and Angora Fuzzer 3.4. To compare the fuzzers, program metrics code coverage, types of bugs or vulnerabilities detected, number of bugs detected, and execution speed 4.4 was considered. The results for these metrics were obtained by running experiments with all the fuzzers using *FuzzTest* 4.1, and *FuzzGoat* 4.2 as targeted programs. From the analysis of results in Section 4.4 of Chapter 4, we can observe that **LibFuzzer** performs better than American Fuzzy Lop, and Angora Fuzzer.

This work also presents a way to categorize bugs or vulnerabilities in American Fuzzy Lop and LibFuzzer. The crash triaging tool built for AFL 3.2.3 was modi-

fied and designed for LibFuzzer 3.3.2, and this tool can be used to categorize the security-related bugs or vulnerabilities.

As a final thought and conclusion, this work has found a satisfactory answer to the research question proposed, and the recommendations or future work will be discussed in the next section.

5.2 Limitations

The Angora fuzzer 3.4 failed to run on the targeted program 4.1, due to which the results of comparison with the other two fuzzers were not satisfactory. The fuzzer seemed to work perfectly with the example programs provided in the fuzzer repository and the LAVA-M data set [35] used in the documentation [18]. Angora's input generation/mutation strategies were analyzed to understand the workflow of the fuzzer, and it was nearly impossible to fuzz different targeted programs to test the fuzzer with the resource available for the fuzzer.

5.3 Recommendations

This work presented the design of *Exploitable LLDB*, which was ported from GDB to LLDB. In *Exploitable GDB*, there are 21 rules using which the crashes can be categorized. In *Exploitable LLDB*, 14 of the 21 rules were implemented. The rest of the rules needed the use of the stack, and implementing this is a complex process in LLDB and is open research.

The work also presented a complete understanding of the Angora Fuzzer, but the fuzzer failed to fuzz the targeted programs. The lack of documentation and support for the tool resulted in this failure. For future work, we can get support from the designers of the fuzzer, and the test experiments may succeed, which in turn might change the results of the comparison.

Bibliography

[1] "Wannacry ransomware attack," Feb 2023. [Online]. Available: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

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

[3] D. Lewis, "Exploit vs vulnerability: What's the difference?" Nov 2020. [Online]. Available: https://sectigostore.com/blog/exploit-vs-vulnerability-whats-the-difference/

[4] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[5] J. Fell, "A review of fuzzing tools and methods," *PenTest Magazine*, 2017.

[6] "Driving your security forward," Feb 2023. [Online]. Available: https://www.riscure.com/

[7] A. Pramanik and A. Tayade, "Study and comparison of general purpose fuzzers," *University of Wisconsin-Madison*, 2017.

[8] J. Metzman, L. Szekeres, L. M. R. Simon, R. T. Sprabery, and A. 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*, New York, NY, USA, 2021.

[9] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2777–2794. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei

[10] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[11] "Fuzzing," Feb 2023. [Online]. Available: https://en.wikipedia.org/wiki/Fuzzing

[12] M. Boehme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. PP, 08 2020.

[13] "True code - automated embedded software security checks," Feb 2023. [Online]. Available: https://www.riscure.com/security-tools/true-code/

[14] H. Ziade, R. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, pp. 171–186, 01 2004.

[15] "American fuzzy lop (2.52b)." [Online]. Available: https://lcamtuf.coredump.cx/afl/

[16] "Libfuzzer – a library for coverage-guided fuzz testing.¶." [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[17] AngoraFuzzer, "angorafuzzer," May 2019. [Online]. Available: https://github.com/AngoraFuzzer/Angora

[18] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

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

[20] "Using asan with afl." [Online]. Available: https://afl-1.readthedocs.io/en/latest/notes_for_asan.html

[21] "Asan clang." [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html

[22] "Undefinedbehaviorsanitizer." [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[23] "Memorysanitizer." [Online]. Available: https://clang.llvm.org/docs/MemorySanitizer.html

[24] "Dataflowsanitizer." [Online]. Available: https://clang.llvm.org/docs/DataFlowSanitizer.html

[25] "Lcamtuf's old blog." [Online]. Available: https://lcamtuf.blogspot.com/2014/08/

[26] Google, “Afl/afl-fuzz.c at master · google/afl,” Jun 2021. [Online]. Available: https://github.com/google/AFL/blob/master/afl-fuzz.c

[27] [Online]. Available: https://afl-1.readthedocs.io/_/downloads/en/latest/pdf/

[28] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, “Revery: From proof-of-concept to exploitable,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1914–1927.

[29] Jfoote, “The ’exploitable’ gdb plugin.” [Online]. Available: https://github.com/jfoote/exploitable

[30] “Lldb python api.” [Online]. Available: https://lldb.llvm.org/python_api.html

[31] fuzzstati0n, “Fuzzstati0n/fuzzgoat: A vulnerable c program for testing fuzzers.” [Online]. Available: https://github.com/fuzzstati0n/fuzzgoat

[32] Mrash, “Mrash/afl-cov: Produce code coverage results with gcov from afl-fuzz test cases.” [Online]. Available: https://github.com/mrash/afl-cov

[33] “Cipherdyne.org.” [Online]. Available: http://www.cypherdyne.org/afl-cov/

[34] “Source-based code coverage.” [Online]. Available: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

[35] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, “Lava: Large-scale automated vulnerability addition,” 05 2016.