# Waffle: A Whitebox AFL-based Fuzzer for discovering Exhaustive Executions

by

Behnam Bojnordi Arbab

**Previous Degrees (i.e. Degree, University, Year)**
**Bachelor of Computer Engineering, Ferdowsi University of Mashhad, 2015**

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF**

**Master of Computer Science**

In the Graduate Academic Unit of Computer Science

| | |
|---|---|
| Supervisor(s): | **Ali Ghorbani**, Ph.D., Faculty of Computer Science |
| Examining Board: | **Arash Habibi Lashkari**, Ph.D., Faculty of Computer Science, Chair |
| | **Saqib Hakak**, Ph.D., Faculty of Computer Science |
| External Examiner: | **Donglei Du**, Ph.D., Faculty of Management, UNB |

**THE UNIVERSITY OF NEW BRUNSWICK**

**December 2021**

# Abstract

Fuzz testing helps software security researchers investigate the existing vulnerabilities within programs in an automated fashion. AFL is a whitebox coverage-based fuzzer leveraging a genetic algorithm (GA) to search for vulnerabilities inside a program. The inputs to the program, which may affect the program's execution paths are the chromosomes of GA and the content of the files that make up the genes. AFL investigates code coverages for the program's executions on each input, and the findings with new coverage information are selected for more testing. This technique guides the fuzzer to discover more regions of code. Waffle, is an AFL-based fuzzer searching for executions with higher resource usages, such as execution time. Waffle searches for files that not only discover new regions of code but also require more resources to complete a run. Waffle modifies the instrumentation and fuzzing modules of AFL, with the intention of storing resource/time-consuming executions. To confirm the correctness of the modifications, the binaries are assessed, and the fuzzing procedure is monitored from a status screen. Finally, the performance of Waffle is compared to AFL-based fuzzers, and it is shown that Waffle discovers exhaustive executions effectively.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Introduction

Our daily lives are tangled with the vast usage of software in many different aspects. Nowadays, most of the population in the modern world have a mobile phone in their hands, with hundreds or thousands of small to large software installed on their devices. Vehicles use software to monitor the sensors and react to different situations which help save and ease many lives. Instead of using mechanical tools and devices, the software can now do impossible calculations and suggest the most reliable solutions to different problems affecting our lives. Society communicates easier than before, and a world without the software would be unimaginable unless we choose less productivity and accept incomparable expenses. In a glance, huge loads of data are transmitted every second, and the services serve various customers using their technology. But the reliance of humanity on software establishes a weakness that hackers target. Thereby, security researchers and developers are responsible for stabilizing the field of software security.

Software development has shown to be challenging, as unseen mistakes happen due to wrong approaches, or even worse, basically, because of possible human errors.

The bugs may remain hidden for many years, and critical software must be investigated. The hackers are always looking for any vulnerability that helps them get past the protected areas. To investigate the bugs and prevent their existence, different phases of testing are applied before and after the accomplishment of products. One of the techniques for revealing bugs is to monitor the execution of a program and evaluate the correctness of the software's execution. A manual approach would be understanding source code and generated binaries and pointing out the present deficiencies and faults. This cumbersome task of reading the codes suggests the development of software security tools to speed up the testing. Fuzzer is an **automated software security testing** tool, which helps researchers find vulnerabilities, supported by the processing power of the computers.

## 1.2 Summary of contributions

In this thesis, we introduce a fuzzer that searches for states of a program with a high load on available resources. The main contributions of our work are as follows:

- Waffle: Waffle is a whitebox performance-guided AFL-based fuzzer for guiding the corpus-generation based on coverage and performance features. Waffle stands for What An Amazing AFL (WAAAFL)! The introduced fuzzer is designed to collect the usages of any type of instruction in a run. Waffle can focus on a set of one or more instructions and searches for the inputs which maximize the occurrences of the targeted instructions. The coverage information derived from AFL remains the exploration methodology of AFL, and the performance guidance exploits the discovered regions by finding excessive resource usages.

- Injecting fuzzing-dependent instructions for monitoring the execution of a targeted program, so that a run with a high resource usage is distinguishable.

- An algorithm for increasing the resource usages during testing of the program, which leads to the discovery of exhaustive executions.
- Testing our work with state-of-the-art fuzzers and evaluating the benefit of using our fuzzer.

## 1.3 Thesis Organization

In Chapter 2, some of the related works in the area of fuzz testing are reviewed; software vulnerabilities and their occurrences are discussed, and a graphical representation of software is explained for understanding the execution of the program and unfolding the causes of vulnerabilities. Next, a classification of fuzzers and their usages are described. We wrap up the second chapter by introducing a fuzzer which is the base of our work. In Chapter 3, the proposed fuzzer is introduced. We explain how the solution is applied to a program with possible vulnerabilities. To wrap up the 3rd chapter, a sample program is tested, and the performance of the fuzz testing tool is analyzed. The performance of the implementations is evaluated in Chapter 4. A frontier benchmarking tool for fuzz testing is introduced, and the reports are presented afterward. We wrap up the thesis and conclude in Chapter 5. In addition, some future works are suggested for continuing this work.

# Chapter 2

# Background

## 2.1 Introduction

Fuzzing (short for fuzz testing) is a tool for "finding implementation bugs using malformed/semi-malformed data injection in an automated fashion". Since the late '80s, fuzz testing has proved to be a powerful tool for finding errors in a program. For instance, American Fuzzy Lopper (AFL) has found more than a total of 330 vulnerabilities from 2013 to 2017 in more than 70 different programs [4]. The research on fuzz testing has found its place in software security testing. Liang et al. [1], illustrates the growth of the primary studies from publishers such as *ACM digital library*, *Elsevier ScienceDirect*, *IEEEXplore digital library*, *Springer online library*, *Wiley InterScience*, *USENIX*, and *Semantic scholar*. The queries for the literature reviews were "fuzz testing", "fuzzing", "fuzzer", "random testing", or "swarm testing" as the keywords of the titles. Figure 2.1 presents the results of the mentioned study.

A *run* is a sequence of instructions that connects the start and termination of a program. A successful run (execution) behaves as the program is intended to run. An exception is a signal that is thrown, indicating an unexpected behavior. If

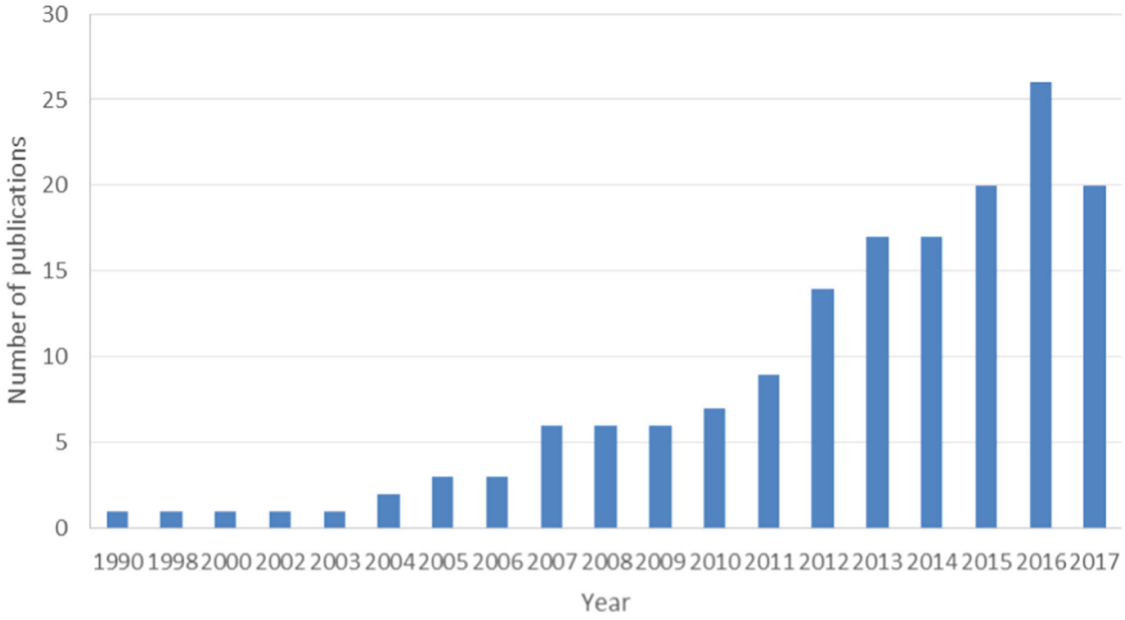



Figure 2.1: Fuzzing papers published between January 1st, 1990 and June 30 2017. [1]

the exception is not caught before the program's termination, the operating system receives an unfinished task with the exception's descriptions. From the OS's perspective, the program has crashed and could not finish its execution properly. A software vulnerability is an unexpected state of the program that is failed to be handled. Different states of a program occur by different inputs that a program takes. The inputs such as *environment variables*, *file paths*, or other program's *arguments* are mainly selected to search for the vulnerabilities.

Fuzz testing is the repetitive execution of a target program with different inputs. Fuzz testing takes two main actions in the fuzzing procedure: the fuzzer *generates test cases* for the target program, and each generated (fuzzed) test case is then passed to the program for *execution*. A fuzzer gathers information out of each execution. A *whitebox* fuzzer has access to the source code of the target program. *Analyzing the source code*, *monitoring the execution*, and *validating the returned value of execution*, are of the capabilities of a whitebox fuzzer. Oppositely, a *blackbox* fuzzer does not have any access to the source code, cannot analyze the execution, and does not check

the result of the execution. Instead, a blackbox fuzzer focuses on executing more instances of the program blindly. Fuzzers with at least one property from each of the whitebox and blackbox fuzzers are in the category of *greybox* fuzzers.

The common strategies for fuzzing new test cases include *genetic algorithms*, *coverage-based (coverage-guided) strategies*, *performance fuzzing*, *symbolic execution*, *taint-based analysis*, etc. Genetic algorithms (GA) are *evolutionary algorithms* for generating solutions to *search* and *optimization problems*. GA has a population of solutions that their evaluations affect their survivability for the next generation. Inspired by the biological operations, GA processes the selected (survived) population and applies *mutations* and other modifications on them, resulting in a new generation of the population [5, 6]. Coverage-guided strategy is a genetic algorithm that utilizes *concrete analysis* of the *execution-path* of a program. A concrete analysis investigates the runtime information of an executive program, and the graph of the executed instructions (execution-path) can be collected through this analysis. Symbolic executions determine the constraints that change the execution-paths [7]. Performance fuzzing is a coverage-based technique that generates *pathological inputs*. "Pathological inputs are those inputs which exhibit worst-case algorithmic complexity in different components of the program" [8]. A taint-based analysis of a program tracks back the variables that cause a state of the executing program. This approach can detect vulnerabilities with no false positives [9].

American Fuzzy Lopper (AFL) [10] is a coverage-based greybox fuzzer, that is originally considering the number of times each *basic block* of execution is visited. Each basic block is a sequence of instructions with no branches except the entry (jump in) and exit (jump out) of the sequence. AFL is published with two default tools for collecting the runtime information: *LLVM* [11] and *QEMU* [12]. "The LLVM Project is a collection of modular and reusable compiler and toolchain tech-

nologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project." AFL acts as a **whitebox** fuzzer in *llvm-mode*. In **llvm-mode**, AFL provides the recipe for compiling the target program with coverage information. The resulting compiler adds *static instrumentations* (**SI**) to the program. Instrumentation is the process of injecting logging instructions into the program, and SI refers to the instrumentations applied on a binary before execution. The added instructions store the code coverage and AFL can use them in Fuzzing. QEMU (Quick EMUlator) is an open-source emulator and virtualizer that helps AFL with *dynamic instrumentation* (**DI**). In DI, the instructions are inserted in runtime, and an emulator such as QEMU helps AFL with discovering the code coverage [13]. *DynamoRIO* [14], *Frida* [15], and *PIN* [16] are some other examples of pioneer DI tools.

In this chapter, we begin with reviewing the previous works that lead to this thesis 2.2. Next, we describe the implementation of AFL and its llvm-mode in 2.3. We wrap up this chapter with conclusions.

## 2.2 Literature Review

Fuzzing searches for software vulnerabilities. "Vulnerability" has different definitions under various organizations and researches. For instance, *International Organization for Standardization (ISO)* defines vulnerability as "A weakness of an asset or group of assets that can be exploited by one or more threats, where an asset is anything that has value to the organization, its business operations, and their continuity, including information resources that support the organization's mission." [17] Yet, the definition needs more details for software.

### 2.2.1 Software vulnerability

According to the *Open Web Application Security Project (OWASP)*: "A vulnerability is a hole or a weakness in the application, which can be a design flaw or an implementation bug, that allows attackers to cause harm to the stakeholders of an application. Stakeholders include the application owner, application users, and other entities that rely on the application." The existence of software vulnerabilities may be compromised and may become an attack target for hackers; this makes the software unreliable for its users.

To exploit a vulnerability, an attacker calls an execution containing the bug and redirects the program's flow with appropriate inputs. An exploitable vulnerability may escalate privileges, leak information, modify/destroy protected data, stop services, execute malicious code, etc. [18]. An analysis of the program may prevent the exposure of vulnerabilities and stop further harm. Various techniques are used to identify weaknesses and vulnerabilities of software. Techniques such as *static analysis*, fuzzing, taint analysis, and symbolic execution, etc. are the most common techniques that can be used cooperatively for error detection [19]. The static analysis evaluates the source code or binary to expose the vulnerabilities without executing the program.

### 2.2.2 Control Flow Graph

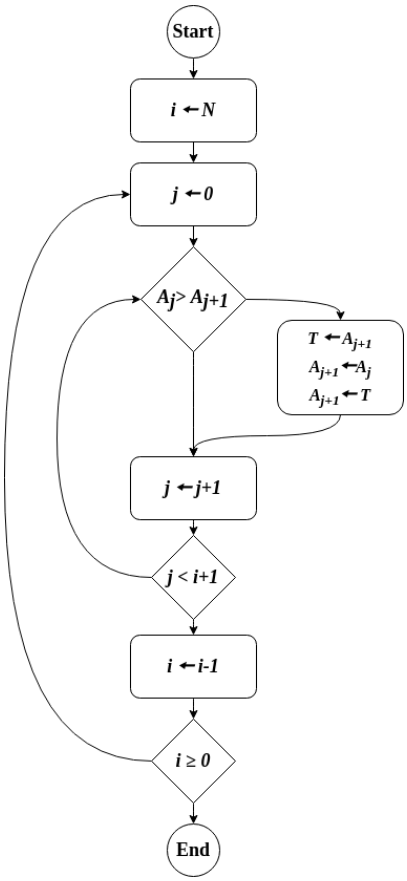
To investigate a program for bugs, a model that helps the research is **Control Flow Graph (CFG)**. CFG is a directed graph whose nodes are the basic blocks of the program, and its edges are the flow path of the execution between two consecutive basic blocks. For instance, the Figure 2.2a illustrates the CFG for *bubblesort* algorithm (Algorithm 1). The branches in CFG split after a *conditional instruction* and a relevant *jump instruction*.

**Algorithm 1:** Pseudocode of bubblesort on array $A$ of size $N$

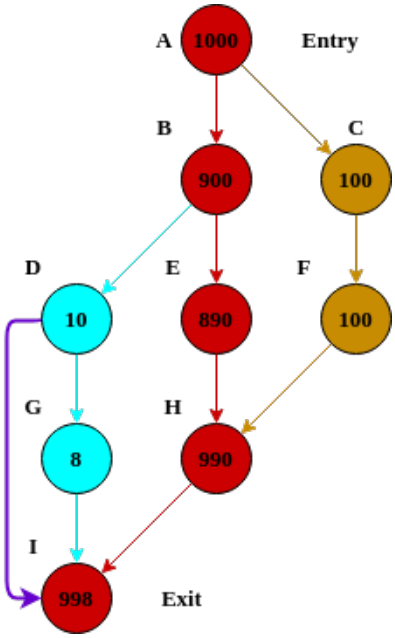
```
   Input: A, N
 1 i ← N;
 2 do
 3 |   j ← 0;
 4 |   do
 5 |   |   if A_j > A_{j+1} then
 6 |   |   |   SWAP(A_j, A_{j+1});
 7 |   |   j ← j + 1;
 8 |   while j < i + 1;
 9 |   i ← i − 1;
10 while i >= 0;
```

An execution processes a path (sequence) of instructions from an entry to any exit location of the program. For instance, consider Figure 2.2b as a CFG illustrating the executed paths of 1000 trials of the program's execution. The numbers in the basic blocks indicate the number of times each basic block is visited. A path such as $A \rightarrow B \rightarrow E \rightarrow H \rightarrow I$ has been explored more than other execution paths. Basic block $D$ is visited occasionally and the edge $D \rightarrow I$ directly goes to the Exit, representing bugs in basic block $D$. These 1000 trials have discovered 9 separable basic blocks, but it does not imply that there are no other basic blocks or edges revealed after more trials. Code coverage measures the number of basic blocks which could be reached in an experiment of trials.

Denial of service (DoS) is a category of vulnerabilities through a network that prevents services from correctly responding to the users. "There are many ways to make a service unavailable for legitimate users by manipulating network packets, programming, logical, or resources handling vulnerabilities, among others. If a service receives a very large number of requests, it may cease to be available to legitimate users. In the same way, a service may stop if a programming vulnerability is exploited, or the way the service handles resources it uses" [20]. In the software domain, the vulnerability is either due to an early termination through a crash, or




(a) Control Flow Graph of bubble-sort algorithm [Algorithm 1]




(b) CFG Heatmap of Algorithm 1. In 1000 runs, different paths are discovered.

Figure 2.2: Control Flow Graph

the program terminates with a timeout.

The vulnerabilities arise after the target program executes with a triggering input. Miller introduced fuzz testing to examine the vulnerabilities of a collection of Unix utilities [21]. The results showed that a random fuzzing on different versions of the utilities could discover bugs in 28% of the targets. The automation in testing programs helps the researchers with validating the reliability of a program. The early fuzzers mimic a procedure of searching for the bugs by starting with *identification* of the target program and its inputs. Next, the fuzzing loop initiates, and the program is run with fuzzed inputs as long as the fuzz testing is not terminated. Figure 2.3 depicts the fuzz testing procedure defined by Sutton et al. [2]. Based on the

definitions, a standard fuzzer consists of:

- Target identification
- Inputs identification
- Fuzzed data generation
- Execution of target with fuzzed data
- Exceptions monitoring
- Exploitability determination

**Target** is a software or a combination of executables and hardware [22]. A targeted software is any program that a machine can execute. Fuzzer needs to know the command for executing the target program and the inputs (arguments) of the program. **Inputs** are a set of environmental variables, file formats, and any other parameters that affect the execution. The initial seeds of the inputs can guide the fuzzer for finding more complex test cases, yet, it is not mandatory to provide seeds, and a fuzzer can generate valid inputs *out of thin air* [23]. After the initial

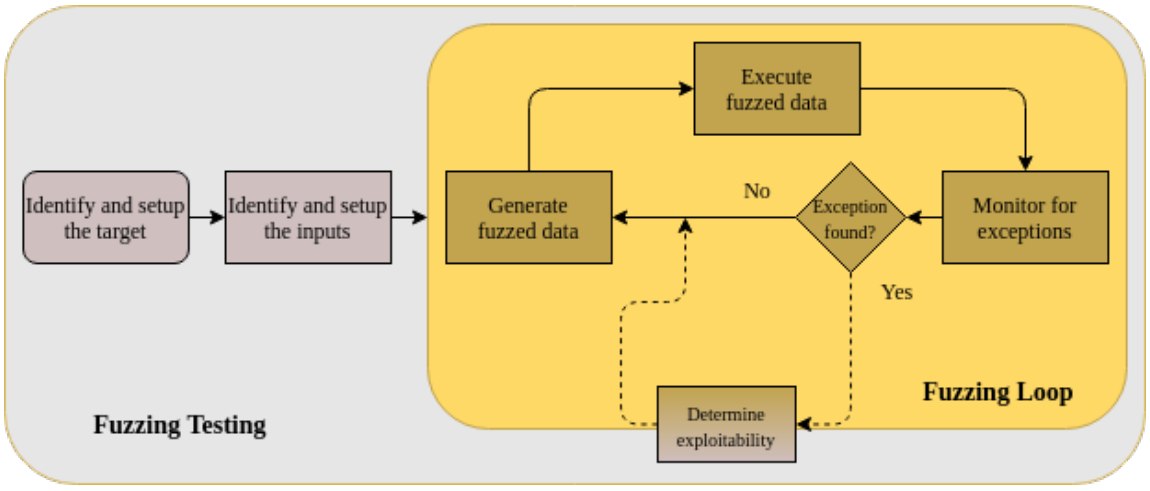



Figure 2.3: Fuzzing phases. Inspired by the definition of Sutton et al. [2]

setup, the **fuzzing loop** begins iterating. In each iteration the fuzzer **executes** the target with the **provided test cases**. Fuzzer then proceeds to detect exceptions returned from the executions and considers the executed input responsible for causing a **vulnerability**. The vulnerability can then be analyzed for **exploitability** in the last stage. An exploitable vulnerability can compromise the system and initiate an anomaly.

The categories for fuzzers with different **program awareness** and different **techniques for fuzzing** the inputs help the community find different applications of the fuzzers for discovering more vulnerabilities. For instance, a developer uses a whitebox fuzzer to assess the immunity of the program (source-code accessible) against malicious activities. On the other hand, an attacker may use a blackbox fuzzer to attack a remote program blindly. A researcher may use a coverage-based fuzzer to consider the execution paths as a variable to reach more regions of the code and detect more crashes; hence, another researcher may use a performance fuzzer to reveal the test cases causing performance issues.

### 2.2.3 Program awareness

The *colorful* representation of fuzzers depends on the amount of information collected from a symbolic/concrete execution. A blackbox fuzzer does not gather any information from the execution. In contrast, whitebox fuzzers have all the required access to the program's source code, and greybox fuzzing covers the gray area between the mentioned types.

#### 2.2.3.1 Blackbox fuzzing

Blackbox fuzz testing is a general method of testing an application without struggling with the analysis of the program itself. The target of an analysis executes after calling the proper APIs, and the errors are expected to occur in the procedure of

trying various inputs. Blackbox fuzzing is an effective technique despite its simplicity [24].

The introduced fuzzer by Miller [21] was of the very first naive blackbox fuzzers. It runs the fuzzing for different lengths of inputs for each target (of the total 88 Unix utilities) and expects a **crash**, **hang**, or a **succeed** after the execution of the program. Each input is then fuzzed with a random mutation to generate new test cases. One of the **downsides** of blackbox fuzzing is that the program may face branches with *magic values*, constraining the variables to a specific set of values; for instance, as shown in Listing 2.1, the chance of satisfying the equation `magic_string=="M4G!C"` and taking the `succeed()` path is almost zero. In [25] and [26] a set of network protocols are fuzzed in a blackbox manner, but as the target is specified, the performance is enhanced drastically. Any application on the web may be considered a black-boxed program as well, so as [27] and [28] have targeted web applications and found ways to attack some the websites, looking for different vulnerabilities, such as XSS.

```
1  string magic_string = random_string();
2  if(magic_string == "M4G!C")
3      return succeed();
4  else
5      return failed();
```

Listing 2.1: Magic Value: `M4G!C` is a magic value

A blackbox fuzzer is unaware of the program's structure and cannot monitor its execution. The **benefit** of using a blackbox fuzzer is the speed of test case generation; the genuine compiled target program is being tested and the fuzzer does not put an effort into processing the inputs and executions. In addition, a blackbox fuzzer is featured to target external programs by using the standard interfaces of those programs. For instance, IoTFuzzer [29] is an Internet of Things (IoT) blackbox fuzzer, "which aims at finding memory corruption vulnerabilities in IoT devices without access to their firmware images." In recent research by Mansur et al. [30],

they introduce a blackbox fuzzing method for detecting bugs in Satisfiability Modulo Theories (SMT) problems. As a result, blackbox fuzzing suggests a general solution in diverse domains. On the other hand, one of **drawbacks** of using blackbox fuzzing is that it finds *shallow* bugs. A shallow vulnerability is an error that appears in the early discovered basic blocks in the CFG of the program. The reason behind this disadvantage is that blackbox fuzzing is **blind** in understanding the execution, and cannot analyze the CFG.

#### 2.2.3.2 Whitebox fuzzing

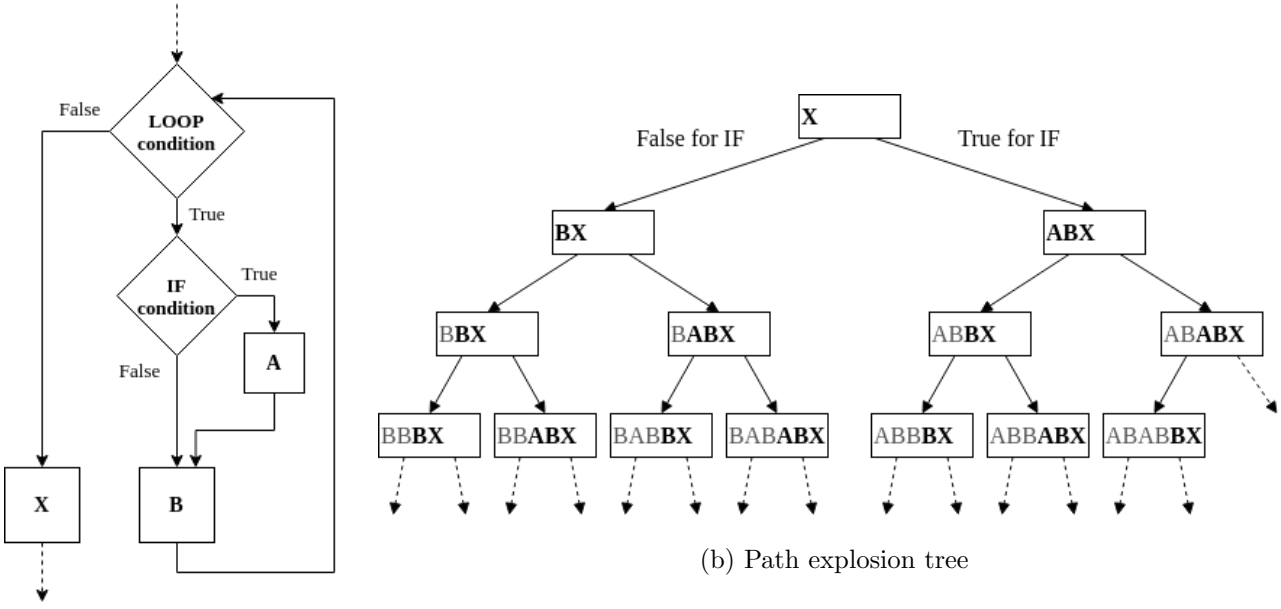
Whitebox fuzzing works with the source code of the target. The source code contains the logic of the program and can anticipate the executions' behavior without executing the program (concretely). Symbolic execution [31] is a reliable whitebox fuzzing strategy that analyzes the source code. This analysis replaces the variables with symbols that consider the constraints for each data. This technique helps its fuzzer discover inputs that increase code coverage by discovering new branches after conditional instructions were satisfied. This method detects *hidden* bugs faster due to the powerful constraint solvers [32]. Although the symbolic execution can solve the conditional branching theoretically, this technique suffers from *path explosion* problem. As an example, in Figure 2.4a a sample section of a program containing a `loop` and an `if` statement within the loop. Figure 2.4b shows the tree of the actions taken until the program reaches the basic block $X$. Solving the current loop requires an exponentially growing number of paths that the fuzzer needs to visit. Whitebox fuzzing is not very practical in the industry as it is expensive (time-consuming / resource-consuming) and requires the source code, which may not be available for testers.

SAGE [33], a whitebox fuzzer, was developed as an alternative to blackbox fuzzing to cover the lack of blackbox fuzzers [34]. It can also use dynamic and *concolic*

*execution* [35] and use taint analysis to locate the regions of seed files influencing values used by the program [36]. Concolic execution is an effective combination of symbolic execution and concrete (dynamic) executions; in a dynamic execution, the fuzzer executes the program and analyzes the run. Godefroid et al. [37] has also introduced a whitebox fuzzer that investigates the grammar for parsing the input files without any prior knowledge.

#### 2.2.3.3 Greybox fuzzing

Greybox fuzzing resides between whitebox and blackbox fuzzing, as it has partial knowledge (awareness) about the internals of the target application. The source code is not analyzed, but the executions of the binary files are the main data source for discovering the vulnerabilities in action; the actual application's logic is not considered for the analysis, but the instructions illustrate an overview of the compiled program's logic. A concrete execution of a program represents a reproducible pro-




(a) Path explosion

(b) Path explosion tree

Figure 2.4: Path explosion example

cedure that is executed and can be monitored for its behavior detection. Greybox fuzzer obtains the runtime information from the code instrumentation, and by using other techniques such as taint analysis, concolic executions (obtaining the logic from the binary), and methods for acquiring more information after the **partial knowledge** of the program [1, 38].

The code coverage is a viable feature for detecting the new paths that the fuzzer never executed. Registering new paths for testing helps fuzzers in finding different regions of the code for potential vulnerabilities. Later the fuzzer tests the input that caused the new path to check if fuzzing the input can reveal a new vulnerability after constructing tweaked inputs out of the current input. This loop of testing the different inputs continues until specific termination signals show up. AFLGo [39] is a greybox fuzzer that tries digging into the deeper application's basic blocks so that it can reach a specific region. AFLGo focuses on fuzzing more of the inputs that guide the execution as it gets closer to the specified region. This greybox fuzzer shows effective performance in testing a program for detecting errors in new patches of an application and helps with crash reproduction. Another greybox fuzzer, VUzzer [40] enhances the instrumentation to collect control- and data-flow features, which leads to guiding the agnostic fuzzer to find more *intersting* inputs, with less effort (fewer trials of the fuzzed inputs). The core feature in a greybox fuzzer is the *application agnosic* characteristic that targets any executable and observable program in the fuzzer's environment.

### 2.2.4 Input generation

Greybox fuzz testing requires features to distinguish between various inputs and pick the test cases that help the fuzzer find bugs. Code coverage is a distinguishing feature for preference over the inputs. As described before 2.2.1, code coverage

summarizes the execution's behavior and does not analyze the instruction. Instead, the graph of the basic blocks (CFG) is analyzed. For **coverage-guided** fuzzing, the corpus of inputs extends as the fuzzer finds new *execution paths*. On the other hand, a **performance-guided** fuzzer seeks *resource-exhaustive procedures*.

#### 2.2.4.1 Coverage-guided fuzzing

Coverage-based fuzzing is a technique for fuzz testing that instruments the target without analyzing the logic of the program. In a greybox and whitebox coverage-based fuzzing, the instrumentation detects the different paths of the executions [1]. The applied instrumentation collects runtime information such as data coverage, statement coverage, block coverage, decision coverage, and path coverage [41]. Bohme et al. [7] introduced a coverage-based greybox fuzzer that benefits from the Markov Chain model. The fuzzer calculates the *energy* of the inputs based on the **potency of path for the discovery of new paths**. Later, the inputs with higher energy add more fuzzed inputs to the queue.

Steelix [42] is a coverage-guided greybox fuzzer. It implements a coverage-based fuzz testing that is boosted with a *program-state based* instrumentation for collecting the *comparison progress* of the program. The comparison progress keeps the information about *interesting comparisons*. The heavy-weight fuzzing process of Steelix contains an initial lightweight static analysis of the binary. The static analysis returns the basic block and comparison information. Later, the concrete execution of the fuzzed test cases determines the state of the program based on the triggered comparison jumps. In addition to the coverage increasing generation of inputs, Steelix knows how to solve the *magic* comparisons, and looks for new states of the program based on the resolved comparisons.

The code coverage is measured by considering lightweight instrumentations.

| Types | Benefits | Limitations | Example Fuzzers |
|---|---|---|---|
| Whitebox | Deep/Hidden bug finding | - Path Explosion<br>- Accessibility to source code | SAGE<br>BuzzFuzz |
| Blackbox | - Fast test case generation<br>- General applicability | - Shallow bug finding<br>- Blind | LigRE<br>Storm |
| Greybox | - Deep bug finding<br>- General applicability | - Access to binary<br>- Requires instrumentation | AFL<br>LibFuzzer |

Table 2.1: Program awareness for fuzzing

This method helps the fuzzing to monitor the program without changing the program's resource usage (time, memory, etc.) by a noticeable amount. Hence, due to inaccessibility to the source code, dynamic instrumentation is used as a reliable technique for collecting the runtime information. The trade-off for using DI is the increasing performance cost; for instance, QEMU costs 2-5x slower executions [13].

#### 2.2.4.2 Performance-guided fuzzing

To enhance the capability of a coverage-based fuzzer, a performance-guided fuzzing technique collects resource-usage information and leverages code coverage techniques for exploring the CFG. SlowFuzz [43] is a performance-guided coverage-based greybox fuzzer, which measures the length of the executed instructions in a total (complete) execution. SlowFuzz has an interest in evolving the corpus of test cases to discover new paths or generate more resource exhaustive executions. Another performance-guided fuzzer, PerfFuzz [8] aims to generate inputs for executions with higher **execution time** by counting the number of times each edge (jump out

| Types | Features | Drawbacks | Examples |
|---|---|---|---|
| Coverage-guided | Light-weight instrumentation | Low guidance measurements | AFL<br>LibFuzzer<br>Honggfuzz |
| Performance-guided | - More features for guidance<br>- Find resource-exhaustion | Heavy weight instrumentation | SlowFuzz<br>PerfFuzz<br>MemLock |

Table 2.2: Input generation techniques for fuzzing

of the basic block) of the CFG is visited. Next, inputs with higher edge counts take more effect on the corpora's evolution. These two performance fuzzers can detect pathological inputs related to CPU usage. Memlock [44] guides the performance of the fuzzing to produce memory-exhaustive inputs. It investigates memory usage by calculating the maximum runtime memory required during executions. MemLock uses static performance instrumentations for profiling memory usage.

## 2.3 American Fuzzy Lopper (AFL)

Michal Zalewski developed American Fuzzy Lopper as a coverage-guided grey-box fuzzer. He introduces this open-source project as "a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test inputs that trigger new internal states of the targeted binary. This substantially improves the functional coverage for the fuzzed code. The compact synthesized corpus produced by the tool helps seed other, more labor- or resource-intensive testing regimes down the road." [45] AFL is designed to perform **fast** and **reliable**, and at the same time, benefits from the **simplicity** and **chainability** features [46]:

- **Speed:** Avoiding the time-consuming operations and increasing the number of executions over time.
- **Reliability:** AFL takes strategies that are program-agnostic, leveraging only the coverage metrics for more discoveries. This feature helps the fuzzer to perform consistently in finding the vulnerabilities in different programs.
- **Simplicity:** AFL provides different options, helping the users enhance the fuzz testing in a straightforward and meaningful way.

- **Chainability:** AFL can test any binary which is executable and is not constrained by the target software. A driver for the target program can connect the binary to the fuzzer.

AFL tests the program by running the program and monitoring the execution path for each run. To extract information from a run, AFL offers multiple **instrumentation** techniques for constructing hashes of the explored paths while following the executing path of the actual program. AFL requires instrumented binaries that provide the execution information when they are run by AFL. AFL is a whitebox/-greybox fuzzer when it can effectively insert the instrumentations into the program. Figure 2.5 shows a simplified illustration of the procedure of AFL. In whitebox fuzzing, AFL takes the source code of the program and executes static instrumentation during the compilation of the program, and passes the generated binary to the fuzzing module. On the other hand, the greybox feature of AFL lets the fuzzer execute the un-instrumented binary under dynamic instrumentation, which executes the instrumenting instructions wrapped around the basic blocks of the program.

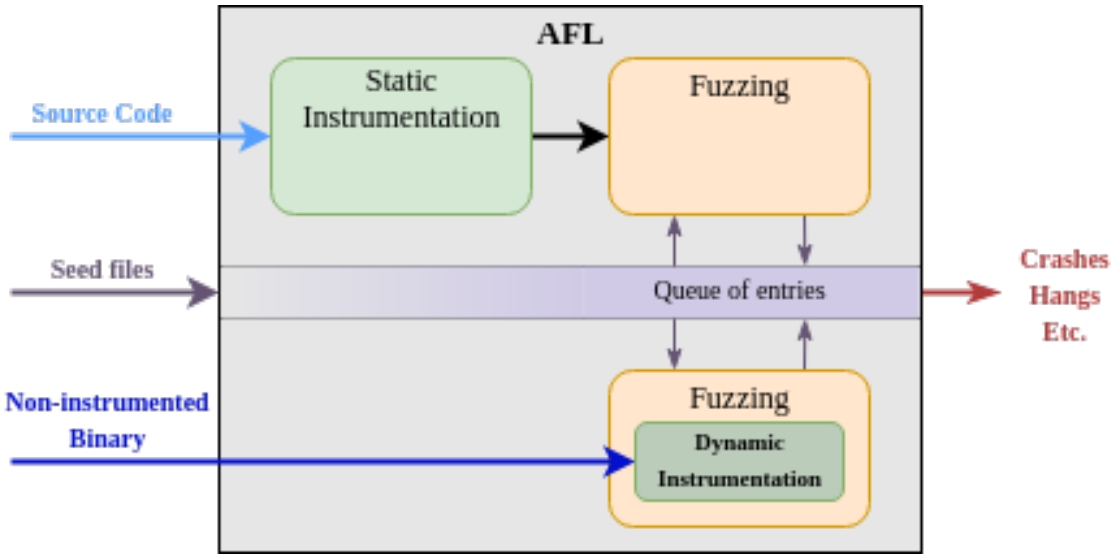



Figure 2.5: AFL's procedure: simplified

### 2.3.1 Instrumentation

The collection of coverage information is generated during the execution; when the execution steps into a basic block, the procedure 3.1 stores the visited edge (pair of two consecutive basic blocks) in CFG. The hashing only stores the information from the previous basic block and the current basic block which we are already in. For instance, suppose we have CFG of a program as shown in 2.6; by permitting the program to modify a memory region shared with AFL - which AFL also has access to it - the coverage instructions generate a summary of the executed path. For example, suppose we have an instrumented program with the random values which are set in compile time (for simplicity, suppose that initially random value $cur\_location = 1010$). Running the first basic block assigns $CUR\_HASH$ to $73 \oplus 1010 = 955$; the content of index 955 of shared memory is then increased by one, and the execution continues to the next basic block. If the execution is jumped into basic block 2, the content of $shared\_mem[310 \oplus (73 >> 1) = 274]$ is incremented by one, and so on. This procedure continues along with the main execution and in the end, the content of the shared memory contains a hashing of the traveled path. In this scenario, taking the path $1 \rightarrow 2 \rightarrow 5$ results in an array of zeros except for $\{51 : 1, 274 : 1, 955 : 1\}$ as the hashed path.

```
1    cur_location = <COMPILE_TIME_RANDOM>;
2    shared_mem[cur_location ^ prev_location]++;
3    prev_location = cur_location >> 1;
```

Listing 2.2: Select element and update in shared_mem

AFL uses both dynamic and static instrumentation for profiling executions. Static instrumentation is applied using **LLVM** modules which can analyze and insert instructions anywhere in the program. Dynamic instrumentation is another method that lets the fuzzer use the instrumentation instructions while executing the program. By default, AFL uses `QEMU` for dynamic instrumentation [13]. The emulator wraps

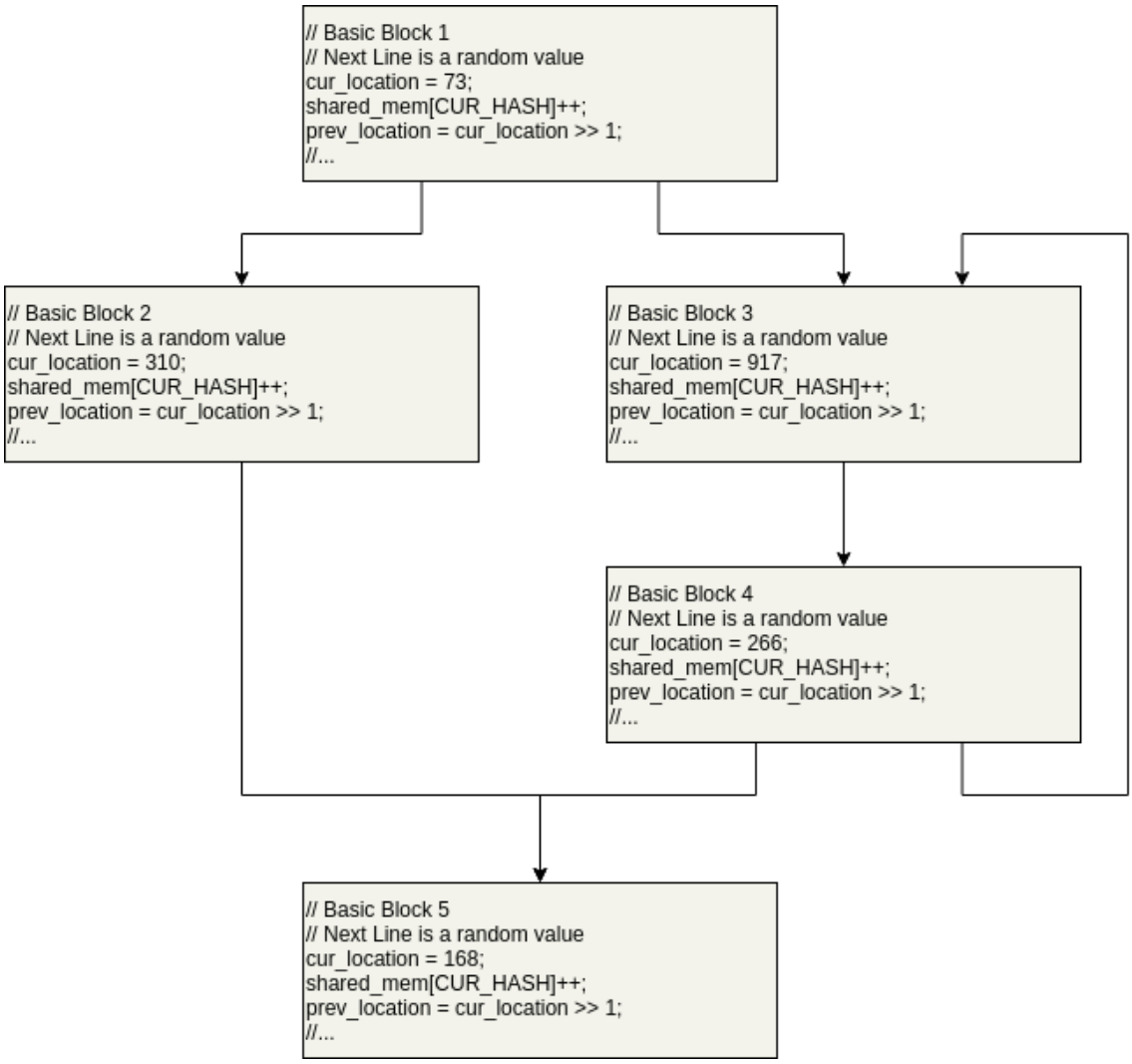



Figure 2.6: Example for instrumented basic blocks

the genuine instructions into analyzable modules and constructs the execution path while running. This technique, **qemu-mode**, causes a slow down of 10-100% for each execution compared to the **llvm-mode**. For this article, we need to dig more into the procedure of instrumentation in llvm-mode [47].

#### 2.3.1.1 LLVM

"The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name **LLVM** itself is not an acronym; it is the full name of the project." [11] Two of the relevant projects used by AFL are:

- The **LLVM Core** libraries contain source/target-independent optimizers as well as code generators for popular CPUs. These well-documented modules assist the development of a custom compiler in every step (*pass*) through the conversion of source code to an executable binary file.
- **Clang** [48] provides a front-end for compiling C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) for the LLVM Project. Clang uses the LLVM Core libraries to generate an *Intermediate Representation* (**IR**) of the source code [49]. The IR is then translated into an executable binary for the machine's CPU (Figure 2.7).

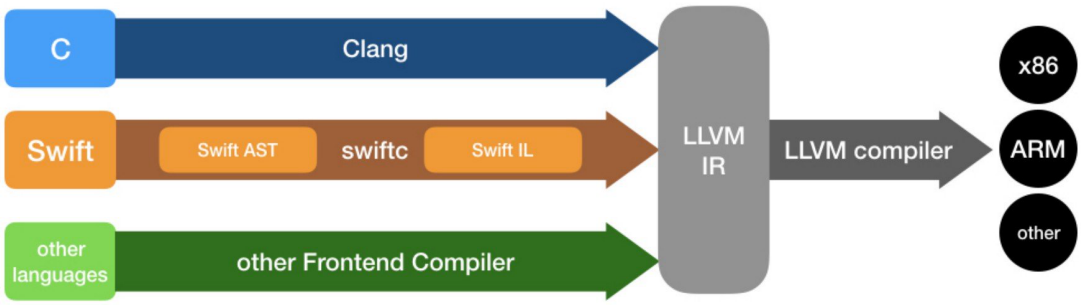



Figure 2.7: LLVM architecture: A front-end compiler generates the LLVM IR, and then it is converted into machine code [3]

AFL utilizes the compilation *passes* of Clang with a custom recipe for *module pass*. Passes are the modules performing the transformations and optimizations of a compilation. Each pass is applied to a specified section of the code. For instance, the **ModulePass** class (and any classes derived from this class) performs the analysis of the code and the insertion of new instructions. Other classes such as **FunctionPass**, **LoopPass**, etc., perform their instructions using different parts of the code, and they contain less information about the rest of the program. AFL uses only the ModulePass and iterates over every basic block existing in the program.

### 2.3.2 AFL Fuzz

The automatic fashion of testing a targeted software using AFL requires a modified execution of the software with coverage-based instructions. The coverage-based instructions are used during the fuzz testing to provide a summary of the execution under AFL's supervision. AFL passes the next test case from the queue of entries to the program, and *caliberates* the test case so that it is ready to be fuzzed. After the calibration of the case, AFL tries to trim the case such that the execution's profile remains the same.

AFL fuzzes the case after its preparations; different mutation techniques are applied to the test case, and the newly generated test cases are analyzed to check if they are producing any new execution behavior based on their code coverage and execution's $speed \times file\_size$. The cases which pass the prior check are considered as *interesting* cases and are added to the queue of entries. We will investigate the details of this functionality later in this article.

AFL uses various mutation techniques for fuzzing a case. There are two main stages for mutations:

- **Deterministic** stage: This early stage contains a sequence of operations which contain:

    1. Sequential bit flips with varying lengths and stepovers
    2. Sequential additions and subtractions of small integers
    3. Sequential insertion of know interesting integers - such as *0*, *1*, *MAX_INT* and etc. [46]
    4. Sequential replacement of content of the case by dictionary tokens provided: AFL accepts a dictionary of known values - such as magic values used in the header of a file. For instance, a dictionary for HTML tags which is also provided in the AFL's repository, contains known HTML patterns such as `tag_header="<header>"` , which AFL would use to replace some bytes of the content of the case with string `<header>`.

- **Non-deterministic** stage: This stage contains two main operations:

    1. Random HAVOC: A sequence of random mutations is applied on the input in this stage. The operations include actions on file such as insertion and deletion of random bytes, cloning subsequences of input, etc. The repetition of this stage depends on the **performance score** of the current case under investigation. AFL calculates this score based on the code coverage of the test case and its execution speed; the more locations visited stored in *coverage bitmap*, and the lower the execution speed is preferable, and as a result, AFL increases the score based on that.
    2. Splicing: If non of the previous stages result in any new findings, AFL tries selecting a random case from the queue, and copies a subsequence of that case into the current case, and relies on the HAVOC stage for mutating the results.

As we discussed, the input files generated for the target program after the fuzzing phase are processed for any *interesting* feature. This procedure goes through two functions called as `save_if_interesting()` and *update_bitmap_score()*:

- `save_if_interesting()`: This function checks if there are any new bits in the shared trace bits (which stores the coverage information) and if a finding is showing new previously unset bits, it tags the case as an interesting one. This function also summarizes the trace bits to reduce the processing effort in later checks - to compare new findings with this generated case.
- `update_bitmap_score()`: AFL maintains a list of `top_rated[]` entries for every byte in the bitmap. Each element of the `top_rated[]` entries tracks the fastest `queue_entry` which visits an edge.

AFL analyzes the cases using the above-provided functions, and the best queue entries are updated eventually in the fuzzing procedure. To select the next element of the queue for fuzzing, AFL uses this information for tagging the *favorite* and *redundant* entries. These operations are done under the function *cull_queue()*. Each element of the list `top_rated[]` contains a pointer to the `queue_entries[]`, and marks the pointed entry as a `favored` entry. After a complete walk over the list, the remaining unfavored entries are marked as `redundant`. In every fuzzing cycle over the queue entries, `cull_queue()` prepares the queue and then AFL takes the next front entry which is also *favored*.

### 2.3.3 Status screen

The **status screen** is a UI for the status of the fuzzing procedure. As it is shown in Figure 2.8, there are various stats provided in real-time updates:

1. **Process timing**: This section tells about how long the fuzzing process is running.

2. **Overall results**: A simplified information about the progress of AFL in finding paths, hangs, and crashes.

3. **Cycle progress**: As mentioned before, AFL takes one input and repeats mutating it for a while. This section shows the information about the current cycle that the fuzzer is working with.

4. **Map coverage**: The AFL's documentation explains the information in this section as "The section provides some trivia about the coverage observed by the instrumentation embedded in the target binary. The first line in the box tells you how many branches we have already hit, in proportion to how much the bitmap can hold. The number on the left describes the current input; the one on the right is the entire input corpus's value. The other line deals with the variability in tuple hit counts seen in the binary. In essence, if every taken branch is always taken a fixed number of times for all the inputs we have tried, this will read "1.00". As we manage to trigger other hit counts for every

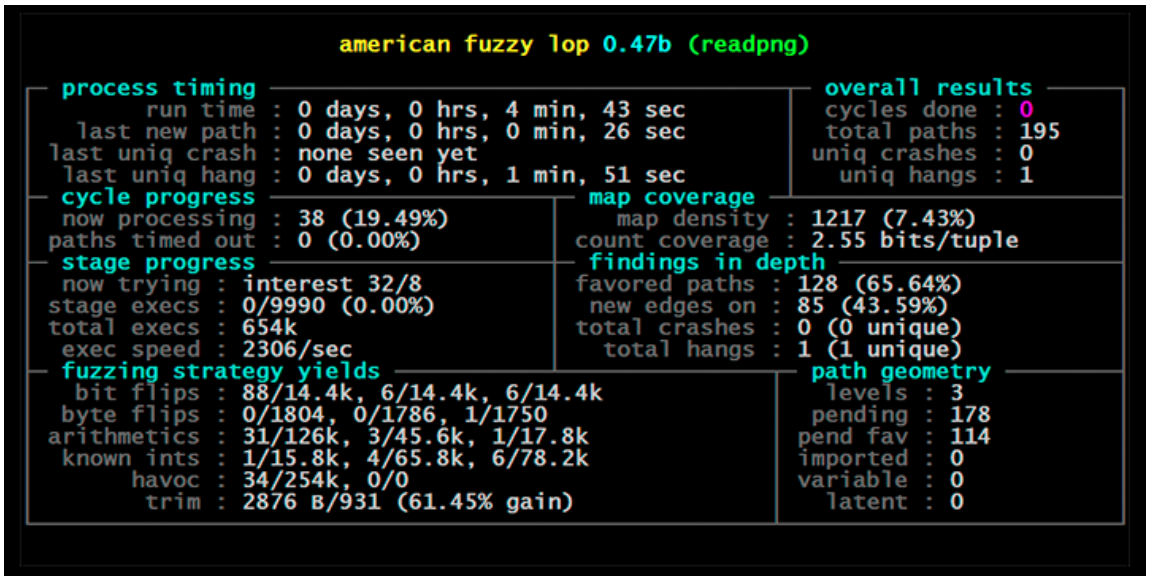



Figure 2.8: AFL status screen

branch, the needle will start to move toward "8.00" (every bit in the 8-bit map hit) but will probably never reach that extreme.

Together, the values can help compare the coverage of several different fuzzing jobs that rely on the same instrumented binary."

5. **Stage progress**: The information about the current mutation stage is briefly provided here. This regards to `fuzz_one()` function in which the new fuzzed inputs are being generated through mutation stages.

6. **Findings in depth**: The number of crashes and hangs and any other findings are presented in this section.

7. **Fuzzing strategy yields**: To illustrate more stats about the strategies used since the beginning of fuzzing. For the comparison of those strategies, AFL keeps track of how many paths it has explored, in proportion to the number of executions attempted.

8. **Path geometry**: The information about the inputs and their depths, which says how many generations of different paths were produced in the process. The depth of input refers to which generation the input belongs to. Considering the first input seeds as depth 0, the generated population from these inputs increases the depth by one. This shows how far the fuzzing has progressed.

### 2.3.4 AFL fuzzing chain

Figure 2.9 illustrates the procedure of fuzzing, from static instrumentation to fuzz testing the target and outputting results. To start the procedure, AFL instruments the program and configures the target binary for the fuzz testing stage. To perform the instrumentation, AFL calls its compiler, which applies the instrumentation when its process is finished [Listing 2.3]:

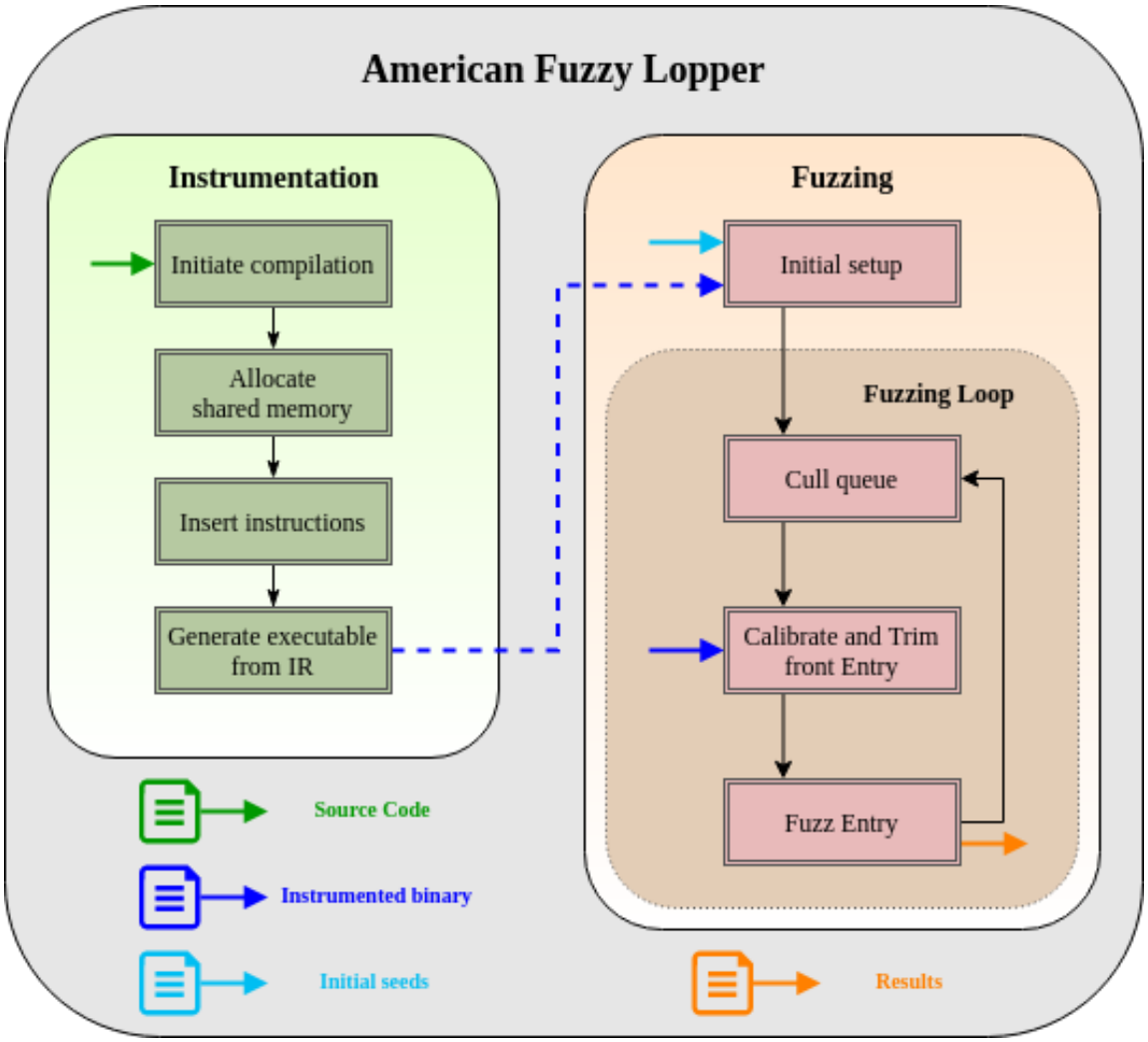



Figure 2.9: An overview of the whole fuzzing procedure of AFL

```
afl-clang sample.c -o sample_inst
```

Listing 2.3: Instrument *sample_vul*.c

The resulting file, `sample_inst`, is an executable that contains shared memory for later analysis in fuzzing. Now AFL can start testing the program for probable vulnerabilities. The test requires the input and output directories, as well as the command for executing the program [Listing 2.4]. The fuzzing continues until receiving a halt signal (For instance, by pressing *Ctrl+C*).

```
# afl-fuzz -i <in_dir> -o <out_dir> [options] -- /path/to/fuzzed/app [params]
```

```
afl-fuzz -i in_dir -o out_dir -- ./sample_inst
```

Listing 2.4: Execute AFL

## 2.4 Concluding remarks

In this chapter, we reviewed the previous works that inspired us for the development of Waffle. We covered these topics:

- A brief description of the previous fuzzers.
- The recognition of whitebox, blackbox, and greybox fuzzers.
- Code coverage technique and its applications in fuzz testing were explained.
- We briefly explained the instrumentation with LLVM and visitor functions.
- We dug into the state-of-the-art fuzzer, AFL, and researched its fuzzing procedure.

In the next chapter we will explore more into the modifications we applied on AFL to achieve Waffle.

# Chapter 3

# Proposed Fuzzer

## 3.1 Problem Statement

A time-consuming and recourse-consuming execution can be used for attacking a server. For instance, suppose that a server is using an API, and it also prevents an excessive number of requests to defend against attacks such as DoS or DDoS. In such a scenario, to attack the server, one may request the server to call heavy executions and put pressure on the server to respond to the requests. As a result, finding such executions may help with stopping the server from responding appropriately.

The performance-guided fuzzing technique utilizes the fuzzer to discover the worst-case complexities of a program. Memory-guided fuzzing targets data structures and memory usage in execution, revealing vulnerable memory consumptions. On the other hand, fuzzers such as SlowFuzz and PerfFuzz monitor the performance-related instructions to detect excessive CPU usages. Hence, the set of resources under investigation in each fuzzer does not include both CPU and Memory usages together. To investigate the resources which can get compromised for revealing hidden resource-usage vulnerabilities, we suggest two solutions:

1. Parallel fuzzing: The corpus of the queued test cases contains the latest findings of a fuzzer. A coverage-based fuzzer seeks new code coverages, and eventually, the corpus evolves with coverage-guided inputs. In a performance-guided fuzzing, the inputs gradually produce fuzzed data which uses more resources. Concurrent fuzzing suggests that if the corpora of parallel fuzzers are shared with the rest of the fuzzers, the outcome is a corpus that tracks the guiding features of all fuzzers. This approach generates vulnerabilities guided by considering both coverage and performance measurements. The synchronization of the fuzzers is simplified in various state-of-the-art fuzzers; for instance, AFL-based and LibFuzzer-based fuzzers can communicate with each other using the provided APIs [50].

2. Waffle: The fuzzer integrates the coverage-based and performance-based fuzzing and generates inputs for exploiting these features.

## 3.2 Waffle

The exponential search space for different inputs with a specific behavior, such as a crash or hang, is approached by evolutionary algorithms of AFL to investigate the possible inputs for producing such events. AFL leverages code coverage, file size, and execution time to guide the genetic algorithm to maintain the cases which discover more regions of the code, in a practically fast fashion. Waffle exploits the coverage-guided findings to discover regions of code and tries to increase resource usages through evolution.

As illustrated in Figure 3.1, Waffle has major modifications on AFL in instrumentation and fuzzing phases. The shared memory is designed to be capable of storing the *resource complexity* of an execution. Waffle also extends AFL's Coverage Pass to collect performance features. In the fuzzing phase, the fuzzer evaluates code

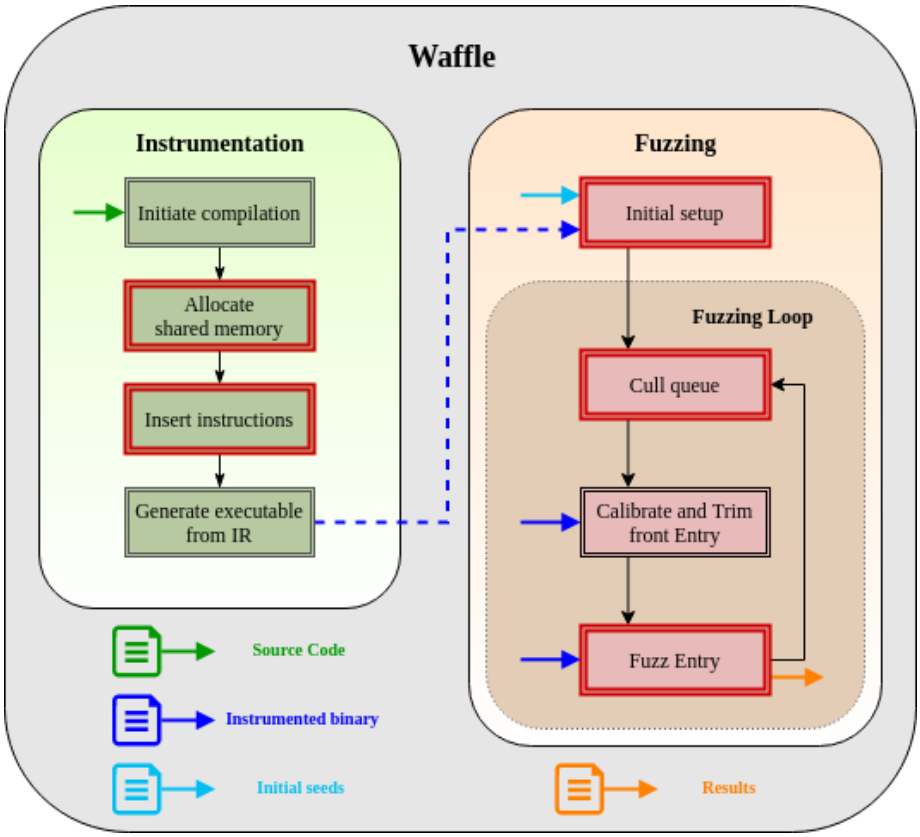



Figure 3.1: Fuzzing phases of Waffle. The red rectangles specify the changed components.

coverage and resource usage of the executions (`cull_queue`) and considers a fitness for each of the generated entries. Next, the front entry in the queue of entries is passed to the fuzzing loop for testing, and the loop continues before a termination signal. In the following sections, we inspect the changes made in AFL to implement Waffle.

### 3.2.1 Resource complexity of execution

To address resource usage of execution, we estimate the engaged resource usage of executions. Waffle does not analyze the source code for finding the resource complexities, such as time and space complexity; however, it records the resource-consuming instructions. For instance, an instruction such as `memcpy` takes CPU usage (and time) for its execution and may access the program's available memory.

To bring in the involved instructions in a complete run of a program, Waffle presumes a set of instructions and monitors the occurrences. A trace of the involvement of the instructions is stored in the shared memory for the need of fuzzing.

We define the **Estimated Resource Usage** (shortened to ERU) of an application as *an estimation of the resources required to execute a program.* This definition follows the program's performance based on the effort for completing an execution. To calculate the ERU of a program in a concrete execution, each of the instructions using the *engaging resources* is monitored. (e.g. `memcpy` instructions)

Waffle replicates AFL's coverage-discovery techniques with modifications to calculate and leverage ERU for performance guidance. While fuzzing the inputs, the ERU of the executions is stored in an array of the same size as AFL's coverage map, and for each hit on the coverage map, Waffle updates the same index of its performance array (As a result, the hit maps of both of the arrays are the same, and they both can represent the code-coverage). The target index is calculated by tracing the taken edge on the program's CFG. In AFL, a favored entry points to an input which suggests something *interesting*, that is, better code coverage or a faster (and with a smaller input size) execution. Furthermore, Waffle finds the *favor* in any entry which is either coverage-guided (better code coverage) or performance-guided (more exhaustive execution).

### 3.2.2 Instrumentation

The `llvm-mode` subproject under Waffle's directory contains the recipe for building a clang-based compiler with the aforementioned instrumentations. The resulting compiler, i.e. `./waffle-clang`, has the capability for inserting the guidance instructions into the program. Waffle initiates the compilation by locating the shared memory to pass the measurements out of the execution's procedure. Waffle includes

```
// snippet of wafl-llvm-rt.o.c

#define ERU_SIZE (1 << 16)

u32  __wafl_ERU_initial[ERU_SIZE];
u32* __wafl_area_ptr = __wafl_ERU_initial;

static void __afl_map_shm(void) {
  u8 *id_str = getenv(SHM_ENV_VAR);

  if (id_str) {
    u32 shm_id = atoi(id_str);
    __wafl_area_ptr = shmat(shm_id, NULL, 0);

    if (__wafl_area_ptr == (void *)-1) _exit(1);

    memset(__wafl_area_ptr, 0, sizeof __wafl_ERU_ptr);
  }
}
```

Listing 3.2: LLVM instrumentation initialization - `__wafl_area_ptr` is the region that is allocated for instruction counters

an (extra to AFL) array of 4-bytes integers for tracking the ERU of each basic block. As a result, Waffle requires $5 \times 2^{16} = 320KBs$ of memory space in addition to the genuine program's memory consumption (Listing 3.2). Correspondingly, the instrumented program leaves a trace of the ERUs (collected by the *visitor* functions) of the basic blocks on an array of ERUs. After the initial configurations for instrumentation, the compilation recipe is applied to the compilation procedure and the generated compiler follows the instructions (as seen in pseudocode 3.1) for instrumentation during the compilation of any given source code.

```
counter = lg(count_instructions())
cur_location = <COMPILE_TIME_RANDOM>;
edge = cur_location ^ prev_location;

cov_shared_mem[edge]++;
per_shared_mem[edge]+=counter;

prev_location = cur_location >> 1;
```

Listing 3.1: Select element and update in shared_mem

#### 3.2.2.1 Visitors

The implementation of *LLVM's instruction visitor functions* [51] helps Waffle to search and count the instances of the instructions used in an execution. LLVM provides functions for `getting` and `setting` the instructions in a range of the code section of the program. Listing 3.3 shows an example of how Waffle uses the visitor counters; the exampled member function `visitInstruction(Instruction &I)` checks if an instruction is of **any** type - which can be set to detect other specific types, such as `memcpy` - and Waffle uses an instance of the class `CountAllVisitor` containing the occurrences of (any) instructions in a specific range of the code section. As mentioned before, this code section is selected from the first instruction of a basic block to the last instruction before leaving the corresponding basic block.

```
#include "llvm/IR/InstVisitor.h"

struct CountAllVisitor : public InstVisitor<CountAllVisitor> {
  unsigned Count;
  CountAllVisitor() : Count(0) {}

  // Any visited instruction is counted in a specified range
  void visitInstruction(Instruction &I) {
    ++Count;
  }
};
```

Listing 3.3: Visitors example

Listing 3.4 shows a snippet of Waffle's Module Pass procedure. In line 6, a pointer to the shared array is introduced to the scope of the program. After this initial setup, Waffle digs into the basic blocks and creates an instance of the `CountAllVisitor` struct. By passing the current basic block to the visitor module, the result of the counting is returned and is stored in `CAV` variable.

```
// snippet of wafl-llvm-pass.so.cc
#include <math.h>
// ...
bool WAFLCoverage::runOnModule(Module &M) {
  // ...
  GlobalVariable *WAFLMapPtr =
    new GlobalVariable(M, PointerType::get(Int32Ty, 0), false,
```

```
    GlobalValue::ExternalLinkage, 0, "__wafl_area_ptr");

  // ...
  for (auto &F : M) {
    for (auto &BB : F) {
      /* Count the instructions */
      CountAllVisitor CAV;
      CAV.visit(BB);

      // ...
      LoadInst *ERUPtr = IRB.CreateLoad(WAFLMapPtr);
      MapPtr->setMetadata(M.getMDKindID("nosanitize"),
        MDNode::get(C, None));

      Value* EdgeId = IRB.CreateXor(PrevLocCasted, CurLoc);
      Value *ERUPtrIdx =
          IRB.CreateGEP(ERUPtr, EdgeId);

      /* Setup the counter for storage */
      u32 log_count = (u32) log2(CAV.Count+1);
      Value *CNT = IRB.getInt32(log_count);

      LoadInst *ERULoad = IRB.CreateLoad(ERUPtrIdx);
      Value *ERUIncr = IRB.CreateAdd(ERULoad, CNT);

      IRB.CreateStore(ERUIncr, ERUPtrIdx)
        ->setMetadata(M.getMDKindID("nosanitize"),
        MDNode::get(C, None));

      inst_blocks++;
    }
  }
}
```

Listing 3.4: LLVM-mode instrumentation pass

The linear increment in `CAV` has a relatively large variance for the counters in each basic block. For instance, if Waffle chooses to count all instructions in execution, the basic blocks contributed an average of 18 instructions for the ERU [1]. To prevent overflows of the ERU cells in long executions, Waffle maintains the logarithm of each counter (Equation 3.1). This modified counter is then set as a constant value for increasing the corresponding edge-index of the ERU array (Formula 3.2). It is noticeable that the content of each cell may be increased by more than one edge; this is caused by the conflicts in the hashing procedure of edges.

[1] Tested C++ implementations of QuickSort, MergeSort, and DFS

$$CNT = \log_2^{CAV+1} \tag{3.1}$$

$$ERU[edge] += \sum_{visit} \log_2^{CAV_{visit}+1} \approx \sum_{visit} \log_2^{CAV_{visit}} = visits \times \log_2^{CAV_{visit}} \tag{3.2}$$

After applying the above recipe for instrumentation, the obtained `waffle-clang` compiler is then used for generating an executable with the mentioned features. To use this compiler, the following command creates the binary file we are looking for:

```
./waffle-clang target.c -o instr_target.bin
```

### 3.2.3 Fuzzing

Waffle follows the same fuzzing procedure as AFL's, with some modifications in processing the inputs, and selecting the next inputs to fuzz. As illustrated in Figure 3.2, Waffle initializes the procedure by introducing new variables to the entries of the queue, as well as the changes to the `top_rated` array of entries. Next, after uploading the seeds, a dry run is performed on the provided seeds for fuzz testing, and the `top_rated` entries are updated to establish a measurement for selecting a favored input for testing. Followingly, the fuzzing loop starts by an initial culling of the queue, modifying the priorities of the entries by comparing the execution features tracked in `top_rated`, i.e. the coverage and performance features, and finally starts fuzzing the selected entry of queue. `fuzz_one` function (**Fuzz one** box in Figure 3.2), accepts a queue entry and tests the program with mutations of the corresponding input file. The next major modification in our fuzzer is submitted in the procedure of saving an interesting input (`save_if_interesting`). This function now considers the performance values, and input with either an interesting coverage-based or ERU-based performance is saved for future fuzzing. By running the function `save_if_interesting` on each of the input files returned from mutation, either a new queue entry (**Q**), crash (**C**), or hang (**H**) is produced, and the corresponding input is

stored in an associated directory. This function, `save_if_interesting`, also updates the bitmap scores (3.5) and reevaluates the `top_rated` entries.

Waffle initiates the tracing arrays for both coverage (`trace_bits`) and ERU

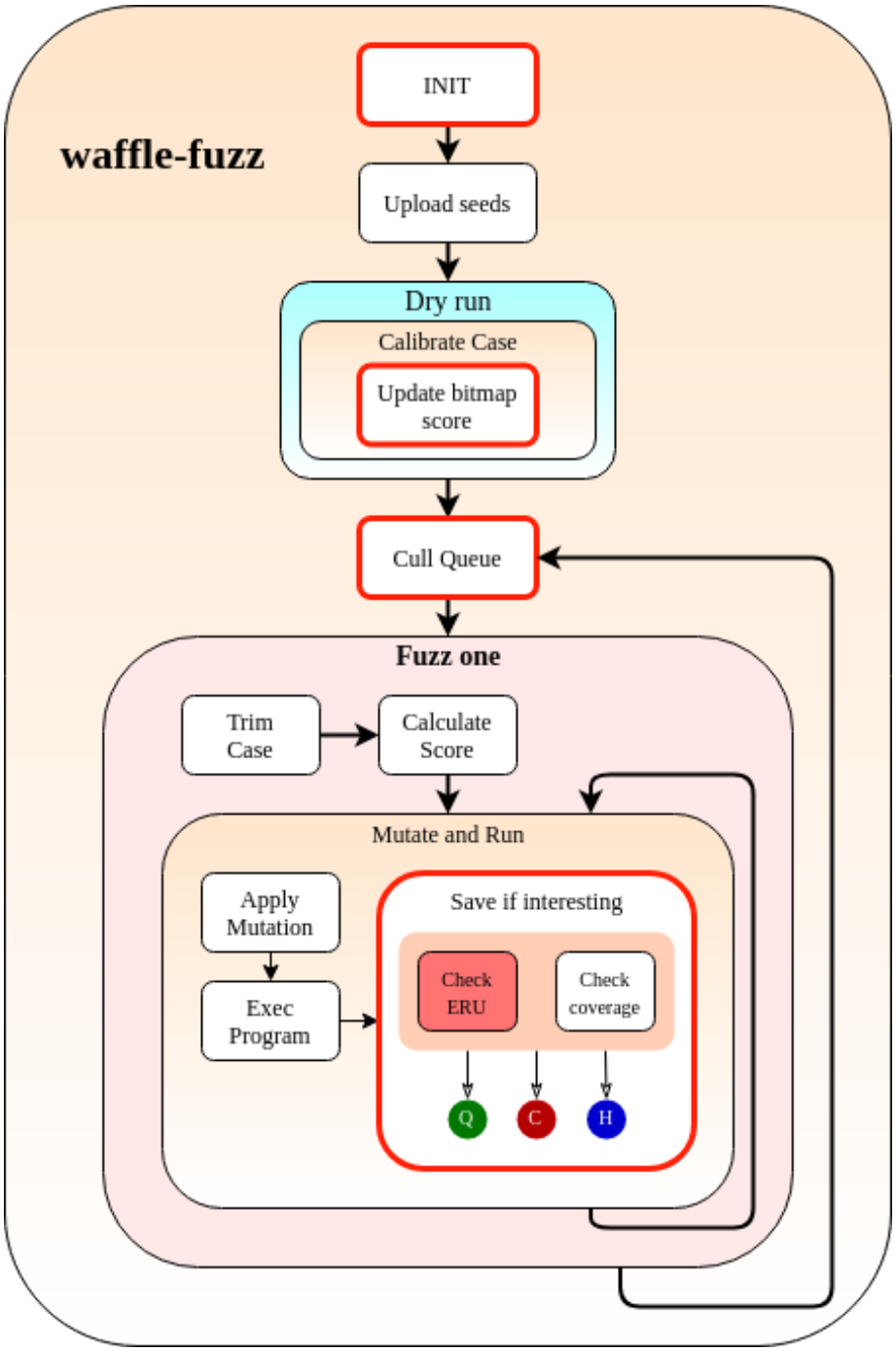



Figure 3.2: Waffle fuzzer: The red rectangles specify the new or changed components.

```
static void update_bitmap_score(struct queue_entry* q) {
  u32 i;
  u64 fav_factor = q->exec_us * q->len;

  /* For every byte set in ERU_list[], see if there is a previous
   winner,
     and how it compares to us. */
  for (i = 0; i < MAP_SIZE; i++){
    if(trace_bits[i][0]){
      if (top_rated[i][0]) {
        if (fav_factor > top_rated[i][0]->exec_us * top_rated[i
   ][0]->len) {
          if(top_rated[i][1]) {
            // Ignore the fav_factor
            if(!var_bytes[i] || top_rated[i][1]->TERU >= q->TERU) {
              continue;
            }
          }
          top_rated[i][1] = q;
          score_changed = 2;
          continue;
        }
        // ...
        top_rated[i][0] = q;
        score_changed = 1;
      }
    }
  }
  return;
}
```

Listing 3.5: `update_bitmap_score`: Waffle ignores the fav_factor in case of a long lasting execution

((trace_ERU)). These arrays have the same structure as mentioned in the previous subsection (3.2.2). The structure of `queue_entry`'s is changed by adding two new variables, `TERU`, for tracking the summation of the ERUs after executing a test entry, and a normalized version of TERU as $nTERU = \lfloor \log_2 TERU \rfloor$. On the other hand, the `top_rated` array which tracks the winners of each edge, is extended to maintain both the coverage and the performance winners (Figure 3.3). To better understand this approach, we explain the functionality of `cull_queue()` as shown in Listing 3.6.

If an entry shows an interesting coverage behavior, it updates the `top_rated` array, controlled by the cells under the entry's coverage map; if a cell on `trace_bits`

is showing a better coverage, the first pointer of `top_rated` is changed to the current entry (which is already stored in the queue of entries). If there are no interests in the code-coverage, Waffle assesses `TERU` of the `queue_entry`, and updates the lower (second) index of `top_rated` array by comparing the current `TERU` with `TERU`s of the pointed entries stored in `top_rated`.

Waffle adds any entry with a coverage-related or ERU-related benefit, to the queue of entries. Later, when it is time to select an entry for fuzzing, Waffle culls the queue as shown in Listing 3.6. Each element of `top_rated` list is either empty or is pointing to a queue's entry. Waffle iterates on the `top_rated` entries, and as shown in the code, it first checks if it can detect a new code coverage in a cell index $i$ (lines 18 to 25), otherwise, the fuzzer tries the ERU for a discovery. In lines 26 to 34, for the same index $i$, if `top_rated[i][1]` is available and it is not *visited* before, then every other `top_rated[j][1]` with the same approximated `nTERU`, for $j$ in the range of indices, is set as *visited* and is skipped in this iteration. This approach helps the fuzzer to set at least one entry with the same `nTERU`, if it was not tagged as

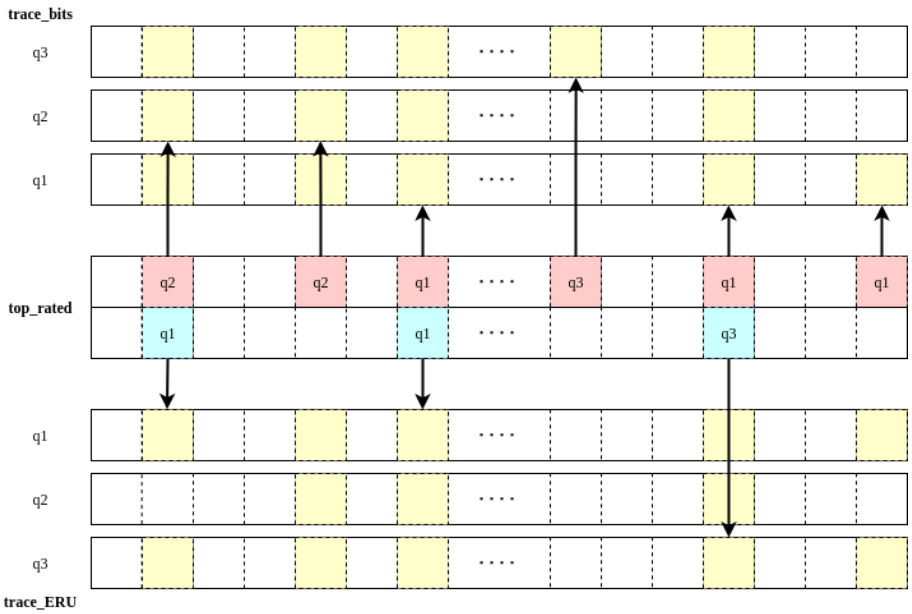



Figure 3.3: `top_rated` array: Waffle keeps track of the relevant coverage and performance features

a coverage-guiding test case, and is introducing a distinct execution with a specific `nTERU`. This technique helps the fuzzer to find the most beneficial test cases in the queue, and ignore the obsolete ones.

```
static void cull_queue(void) {

  struct queue_entry* q, * tmpq;
  static u8 temp_v[MAP_SIZE >> 3];
  /* The visited var_bytes */
  static u8 temp_vv[MAP_SIZE];
  u32 i;
  memset(temp_v, 255, MAP_SIZE >> 3);
  memcpy(temp_vv, var_bytes, MAP_SIZE);
  q = queue;

  while (q) {
    q->favored = 0;
    q = q->next;
  }

  for (i = 0; i < MAP_SIZE; i++)
    if (top_rated[i][0] && (temp_v[i >> 3] & (1 << (i & 7)))) {
      u32 j = MAP_SIZE >> 3;
      /* Remove all bits belonging to the current entry from temp_v.
    */
      while (j--)
        if (top_rated[i][0]->trace_mini[j])
          temp_v[j] &= ~top_rated[i][0]->trace_mini[j];
      top_rated[i][0]->favored = 1;
    }
    else if(top_rated[i][1] && temp_vv[i]) {
      u32 j = MAP_SIZE;
      /* Remove all bits belonging to the current entry from temp_vv
   . */
      while (j--)
        if(top_rated[j][1])
          if(top_rated[i][1]->nTERU == top_rated[j][1]->nTERU)
            temp_vv[j] = 0;
      top_rated[i][1]->favored = 2;
    }

  q = queue;
  tmpq = q;

  while (q) {
    if(q->favored==2 && q->nTERU >= tmpq->nTERU) {
      tmpq->favored = 0;
      q->favored = 1;
      tmpq = q;
    }
    mark_as_redundant(q, !q->favored);
    q = q->next;
  }
```

```
48 }
```

Listing 3.6: Waffle `cull_queue`

After the queue was culled, the first favored entry of the queue is selected for fuzzing. First, Waffle keeps the *trimming* procedure the same as AFL's and does not consider the changes in the `ERUs[]` while trimming the case. This is because the trimming procedure maintains the values of the `trace_bits[]` to remain the same, and as a result, the `ERUs[]` remain the same, and there is no need to check this array. Before getting into the mutation loop, Waffle determines the effort it wants to take for fuzzing the current case and calculates `perf_score` for this purpose (same as AFL's).

In both deterministic and havoc stages of mutation, the fuzzer executes the program on new files and stores them if they expose interesting information. An input is interesting if i) it generates an input based on the heuristics for guidance, to generate corrupted executions, ii) a crash has occurred, or iii) a hang has occurred. The results in (ii) and (iii) do not need any modification in our work. Waffle generates inputs and examines them to verify if they show a *new coverage*, or they expose an exhaustive execution. Waffle aims to guide the code coverage to explore, as well as to generate resource-exhaustive executions. Listing 6 presents pseudocode of *save_if_interesting*. The function `is_exhaustive()` aggregates the values of the `ERUs[]` and if the execution increases the current maximum ERU, the test case for this performance is picked as *interesting* (Listing 3.7).

**Input:** *in – memorytestcase*

1 **if** *is_exhaustive() OR has_new_bits()* **then**
2 ⌊ SAVE the case into queue directory
3 **if** *hanged* **then**
4 ⌊ SAVE the case into hangs directory
5 **if** *crashed* **then**
6 ⌊ SAVE the case into crashes directory

```
1  static inline u32 is_exhaustive() {
2    curr_teru = 0;
3    u64* current = (u64*)trace_bits;
4    u32* curreru = ERUs;
5
6    u32 i = (MAP_SIZE >> 3);
7    while(i--) {
8      if (unlikely(*current)) {
9        for(u32 j=0; j<8; j++) {
10         curr_teru += curreru[j];
11       }
12     }
13     current ++;
14     curreru += 8;
15   }
16
17   if(curr_teru > max_TERU) {
18     max_TERU = curr_teru;
19     return 1;
20   }
21
22   return 0;
23 }
```

Listing 3.7: Find the interest based on the execution's resource usage

## 3.3 Application: fuzzgoat

In this section, we assess the performance of Waffle by fuzzing *fuzzgoat*. Fuzzgoat is an open-source C project with predesigned vulnerabilities inside it [52]. Waffle generates a C-compiler program (`./waffle-clang`) based on `clang` which maintains the indicated instrumentation. We investigate the instrumented binary (call it `./fuzzgoat-wfl`) using *radare2*. Next, Waffle gets `./fuzzgoat-wfl` as the

target program, and the procedure continues fuzzing until we stop it.

### 3.3.1 Instrumentation

*./waffle-clang* compiles the source code to the program. By specifying this compiler for *making* the fuzzgoat project, the resulting binary is generated 3.3.1.

```
cd ./fuzzgoat/
export CC=~/Waffle/waffle-clang
make
```

This command results in `./fuzzgoat-wfl` binary file. We pass this file to *radare2* to confirm the instrumentation. Figure 3.4 illustrates one of the basic blocks collected from the program, and different sections of the selected basic block are specified as well. The orange sections, locate and COV, are inherited from AFL's implementation and Waffle leverages these parts to find the next edge index, and update the trace_bits of the program. Next, the instructions for measuring the program's ERU come after. As stated in the figure, the first three instructions try to load the content of the edge-index of the ERU array. The loaded value is then increased by the precalculated value of the estimated resource usage of the current basic block. Then, the result of the addition is passed back into the array and updates the content of the assigned pointer.

### 3.3.2 Fuzzing

The steps for running Waffle are similar to AFL; `./waffle-fuzz` gets the input and output directories, in addition to the instrumented program:

```
cd fuzzgoat/
waffle-fuzz -i in_dir -o out_dir -- ./fuzzgoat @@
```

Compared to AFL's status screen, Waffle informs about the state of the ERU's through fuzzing. As an example, Figure 3.5 shows a fuzzing procedure after 28

minutes of testing fuzzgoat. In `stage progress` section, `method` explains the causal method which added the selected entry (`current_entry` under testing) to the queue; either it is discovered as a `Coverage` finding, or it is caused by a resource `Exhaustion`.

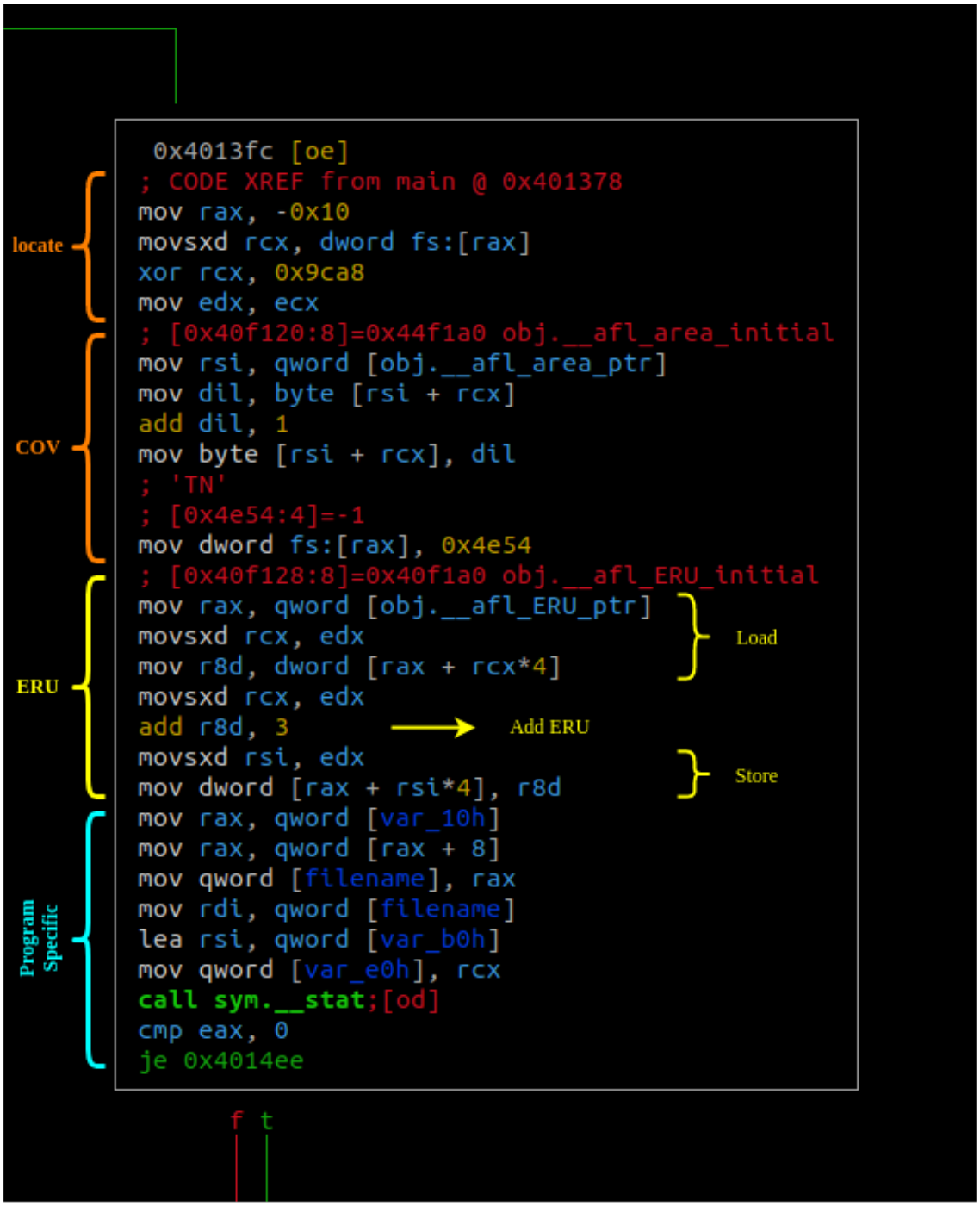



Figure 3.4: Instrumentation illustrated in basic blocks. This basic block contains the coverage-based and ERU-based instructions

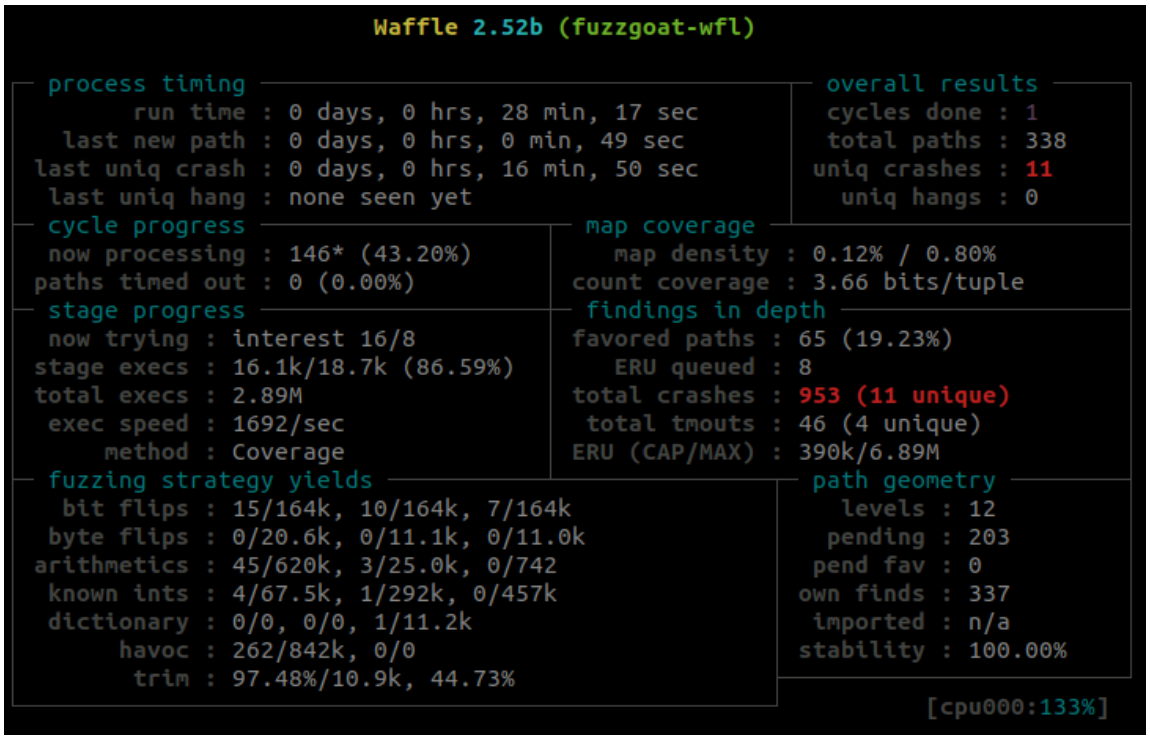



Figure 3.5: Waffle's status screen.

Next to this section, the section `findings in depth` contains two new fields, `ERU queued` and `ERU (CAP/MAX)`. `ERU queued` specifies the queue entries which were added with an **exhaustion** feature. The other field, `ERU (CAP/MAX)` shows the average of the highest 100 ERU's which were acquired through fuzz testing, and the `MAX` value indicates the highest value for all found ERUs.

## 3.4 Concluding remarks

The guidance in coverage-guided fuzzers such as AFL generates variable inputs with different execution paths. Hence, the input generation would escape from creating resource exhaustive procedures. Our solution to this problem is implemented in Waffle. Waffle measures the number of instructions that are under our investigation and utilizes the trace of the measurements to search for exhaustive vulnerabilities. Our approach contains the following steps:

1. To effectively count the instructions, Waffle enhances the instrumentation phase with the capability of exporting the instructions' information to the fuzzer. Waffle changes AFL's Module Pass for these considerations. With the benefit of Visitor functions, the instructions in basic blocks of a program are counted and defined as static values to the program. Every time a basic block is executed, the Estimated Resource Usage of the basic block is stored in an edge corresponding to the explored execution path.

2. The prior step configures a clang compiler that can get a C program and outputs a binary with the proper instrumentations.

3. The instrumented program is passed to the fuzzer, and the testing begins. Every time the program is executed with an input, the coverage and ERU information are stored in separate shared memories. Waffle pursues the coverage information to explore the code and exploits the ERU data to eventually increase the resource usages of the executions.

4. The performance of the analysis of each execution required a careful implementation of both instrumentation and fuzzing phases. We will investigate the performance of the executions in the next section.

In the last section of this chapter, we also tested a predesigned vulnerable program to confirm the initial performance of Waffle. In the next chapter, we compare the performance of Waffle with state-of-the-art fuzzers such as AFL, AFLFast, and Libfuzzer.

# Chapter 4

# Simulation and Experimental Evaluation

## 4.1 Introduction

In this chapter, we explore the benchmarking and evaluation of our developed fuzzer. `FuzzBench` [53], is an automated benchmarking tool that selects a set of fuzzers and predefined benchmarks for fuzzing, and evaluates the performance of each fuzzer on each of the benchmarks in separate trials. In each trial, a fuzzer-specific binary with the according instrumentations is generated, and the fuzzer performs its testing until its time limit is ended. We can modify the amount of testing time in advance, and in this experiment, we have chosen 3 trials for each pair of (fuzzer, benchmark), which distinctively perform 12 of hours testing. In our experiments, we evaluated the performance of Waffle compared to `AFL`, `AFLPlusPlus`, and `AFLFast`. Each fuzzer performs it's testing on `freetype2-2017`, `libjpeg-turbo-07-2017`, `libpng-1.2.56`, and `libxml2-v2.9.2` (details in [54]). Fuzzbench generates pair-wise reports for each fuzzer and targeted benchmarks.

In the next sections, we first explain the configurations for comparing the perfor-

mance of Waffle with other selected fuzzers. Next, an overview of the benchmarking is illustrated, and we analyze the results after. The results of the trials are reported by fuzzbench itself, but as the reports do not contain execution times, we analyze the generated queue entries of each of the fuzzers, with the appropriate binaries. In the end, we contribute our resolutions of the tests.

## 4.2 FuzzBench

### 4.2.1 Fuzzer Benchmarking As a Service

"FuzzBench is a free service that evaluates fuzzers on a wide variety of real-world benchmarks, at Google scale. The goal of FuzzBench is to make it painless to rigorously evaluate fuzzing research and make fuzzing research easier for the community to adopt." [53]

Fuzzbench provides different modules for customized fuzzers. As illustrated in 4.1, we first introduce Waffle to fuzzbench by adding a directory of `Dockerfile`s, with the recipes for building and running the fuzz testing. Next, by passing the name of the fuzzers and the benchmarks, the evaluations begin to run as prescribed (3 trials, 12 hours). On the termination of all experiments, fuzzbench aggregates the performance of the fuzzers and benchmarks, and generates a comparative report of the whole benchmarking, and ranks the fuzzers based on their performance. Hence, the execution times are processed separately for the execution time feature. In order to understand the whole procedure of testing, we explain how we have added Waffle to the system, and continue with the steps taken to get the results.

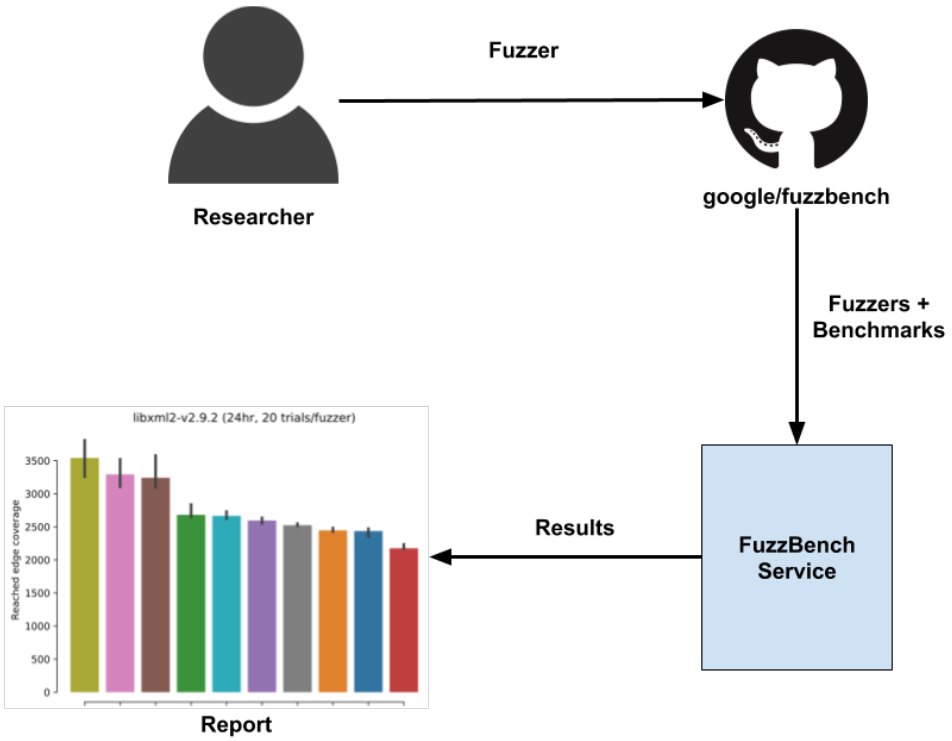



Figure 4.1: Fuzzbench overview

#### 4.2.1.1 Add Waffle to FuzzBench

The current version of FuzzBench contains more than 30 different fuzzers available for testing. To evaluate Waffle, we need to first add Waffle to the list of known fuzzers which Fuzzbench communicates with. One of the options for adding a new fuzzer is by building a docker image, which builds Waffle project and passes `waffle-clang` compiler to fuzzbench for generating the binaries of the benchmarks. To build such docker images, we include the following files into `<fuzzbench-root>/fuzzers/waffle`:

- `builder.Dockerfile`: This file builds the fuzzer in a docker container. Fuzzbench clones Waffle's project into the container, builds the project, and specifies a driver to provide the logs of the executions for fuzzbench. The recipe

can be found in Appendix 6.B.1.

- `runner.Dockerfile`: This file introduces compilation procedure for generating an instrumented binary of the target program. As Waffle is extending AFL, fuzzbench follows the same recipe, and generates the target file with the introduced (Waffle's) compiler.

- `fuzzer.py`: When fuzzbench finishes its setup for running a test, the content of `fuzzer.py` executes fuzz testing without showing the status screen, and creates running instances of the trials. [Appendix 6.B.2].

#### 4.2.1.2 Experiment setups

We ran our experiments on a virtual machine with an Intel® Xeon(R) CPU E5-2695 v4 @ 2.10GHz × 8 processor, supported by 64 GiBs of RAM. We chose Ubuntu 20.04.3 LTS as the operating system for executing and analyzing Unix-specific programs. To evaluate Waffle, we added a docker recipe for building the project, and an AFL-based driver software is passed to the container to communicate with fuzzbench [55]. In addition, a configuration file that includes the trial repetitions and time-to-live for our experiments.

## 4.3 Evaluation metrics

Fuzzbench implements a Clang-based fuzzer-independent code coverage measurer, which calculates the code coverage of the benchmarks after completion of each trial. Same as mentioned in AFL's code coverage measurement, the taken edges exhibit the covered regions in an execution. Table 4.1 shows the benchmarks used in our tests (within fuzzbench); the last column of the table shows the total number of edges for each of the benchmarks. Other columns indicate if the provided benchmark

| Benchmark | Dictionary | Format | Seeds | Edges |
|---|---|---|---|---|
| freetype2-2017 | ✗ | TTF, OTF, WOFF | 2 | 19056 |
| q libjpeg-turbo-07-2017 | ✗ | JPEG | 1 | 9586 |
| libpng-1.2.56 | ✓ | PNG | 1 | 2991 |
| libxml2-v2.9.2 | ✓ | XML | 0 | 50461 |

Table 4.1: List of benchmarks used in evaluation

is supported by a dictionary for the syntax of the inputs, the format of the inputs, and the number of provided seeds for fuzzing the programs.

The main metric for fuzzbench's comparisons is the code coverage, but as we intended to evaluate the execution times as well, we developed a post-processing module for checking the execution times of the generated entries of the trials of fuzzbench. Benchmarks are built within fuzzbench Docker containers, and instead of rebuilding the benchmarks separately, we investigate the execution times within the container with an un-instrumented binary of the target. The measurement of the execution times is implemented in Python 3. The execution times are measured by running the target program with each of the queue entries 10 times, and we assigned the average of the values as the execution time on the provided input.

## 4.4 Evaluation results

The report generated by fuzzbench collecting the coverage information is placed in `report-data` directory, besides the experiment files located in `xp-data`. The reports are generated in figures and an HTML file for illustrating the results. We first analyze the code coverages, and then we examine the execution times.

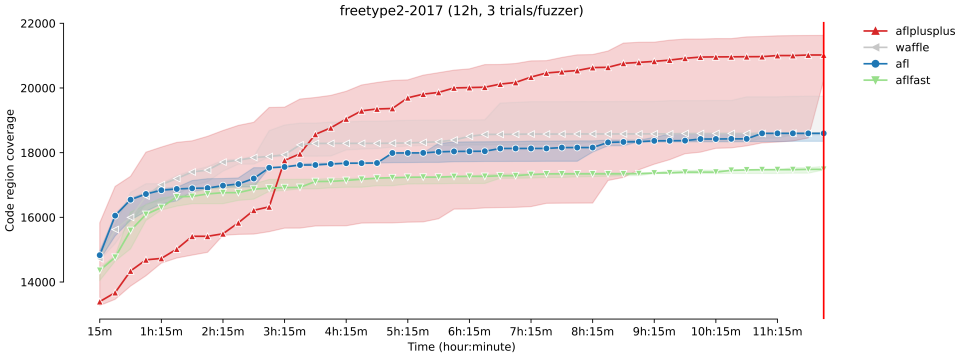
### 4.4.1 Code coverage

#### 4.4.1.1 Code coverage growth

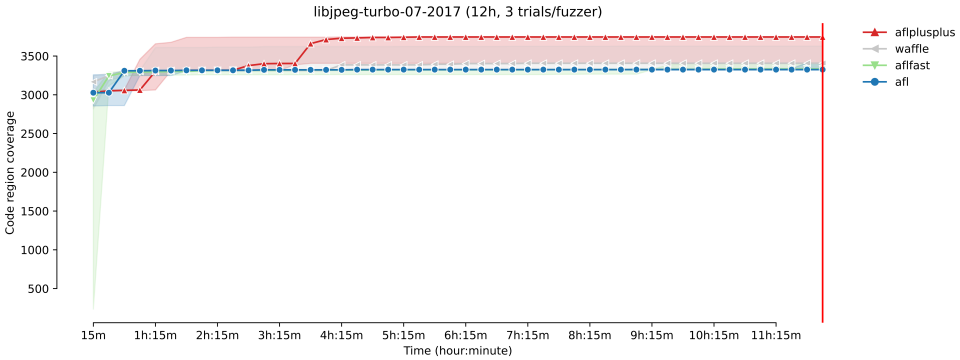
Figure 4.2 shows the code coverage growth of the fuzzers while fuzzing each benchmark. There are three trials for each fuzzer, and the illustration shows the highest code coverage, the minimum code coverage, and the median of the trials' results. AFL++ shows a significantly better performance than other benchmarks. AFL++ has the highest performance in coverage among all other benchmarks which are provided in the fuzzbench project. Waffle, as expected, is seemingly performing the same as AFL. This is followed by the fact that Waffle performs the same code coverage approach, but there are new different features for producing queue entries. In 4.2a, the trials of Waffle show more variance in performance. Hence, AFL's trials grow close to each other. In the paper-based on fuzzbench results [56], the `deterministic stages` of AFL's fuzzing is behind such performance. It is also noticeable that Waffle discovers significantly more regions in at least one of the trials. Testing libjpeg-turbo-07-2017 (Figure 4.2b) and libpng-1.2.56 (Figure 4.2c) show a close performance for AFL, AFLFast, and Waffle 4.2b. Another interesting result collected from fuzzing libxml2-v2.9.2 suggests a difficulty for Waffle in exploring the code coverage, but eventually, Waffle gets the pace closer to other fuzzers. In addition, AFL is the winner of the 4th experiment.

#### 4.4.1.2 Unique code coverage findings

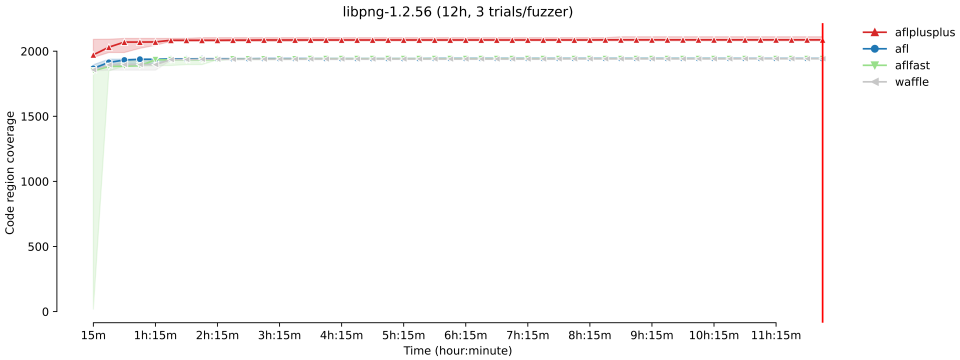
Based on the results collected after each trial, fuzzbench also measures the uniqueness of the fuzzer's findings. Figure 4.3 shows the number of code regions that were uniquely covered in a pairwise fashion. The coloring specifies the significance of the differences; the darker the color is, the more findings are uniquely found by the fuzzer on the bottom.



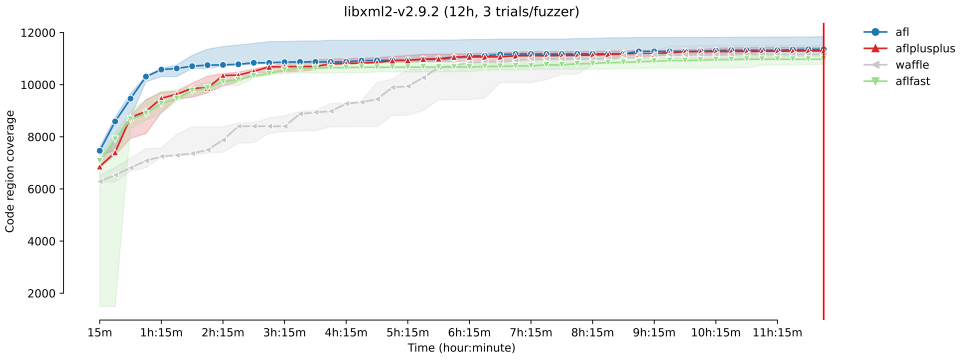

(a) freetype2-2017




(b) libjpeg-turbo-07-2017

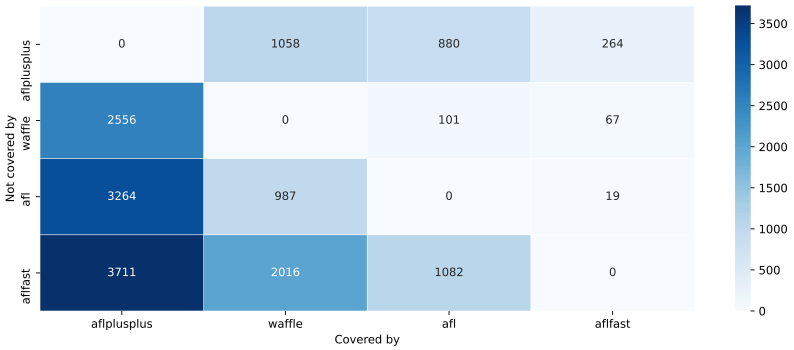



(c) libpng-1.2.56

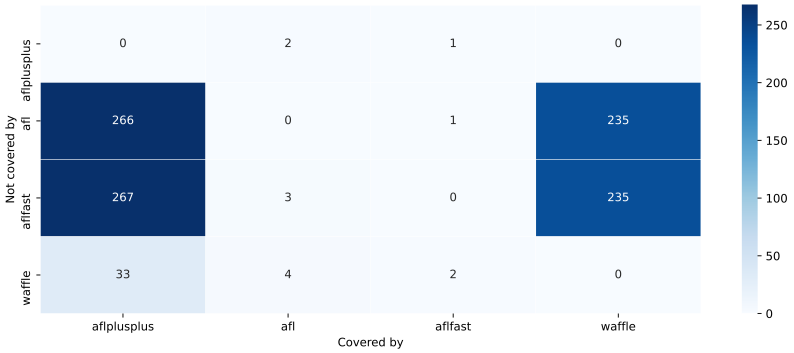



(d) libxml2-v2.9.2

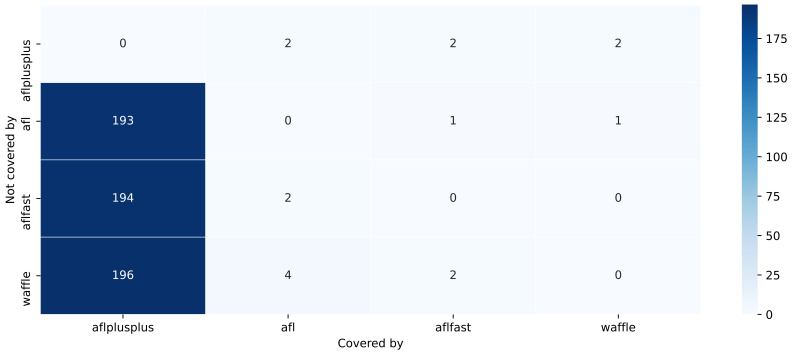
Figure 4.2: Coverage growth during the trials



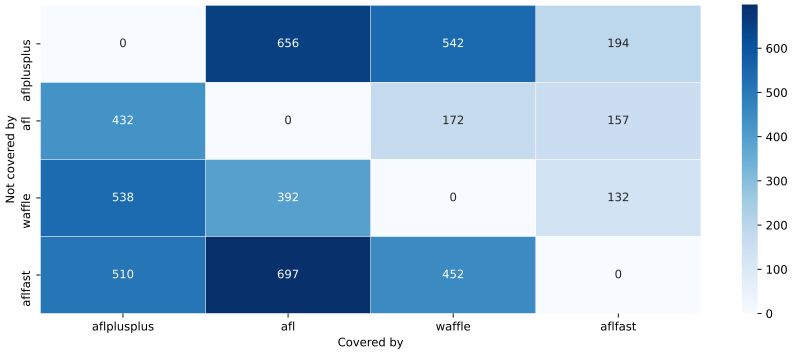

(a) freetype2-2017




(b) libjpeg-turbo-07-2017




(c) libpng-1.2.56




(d) libxml2-v2.9.2

Figure 4.3: Unique coverage findings

| Fuzzer | Benchmarks | Coverage STD | Mean Coverage | Max Coverage |
|---|---|---|---|---|
| freetype | Waffle | 673.115 | 18968.3 | 19745.0 |
| | AFL | 146.550 | 18520.0 | 18612.0 |
| | AFLFast | 59.732 | 17508.0 | 17576.0 |
| | AFL++ | 694.859 | 20961.3 | 21625.0 |
| libjpeg | Waffle | 155.094 | 3454.6 | 3628.0 |
| | AFL | 51.156 | 3352.0 | 3411.0 |
| | AFLFast | 41.761 | 3354.0 | 3402.0 |
| | AFL++ | 195.438 | 3634.6 | 3749.0 |
| libpng | Waffle | 0.577 | 1941.6 | 1942.0 |
| | AFL | 2.645 | 1943.0 | 1945.0 |
| | AFLFast | 0.577 | 1942.3 | 1943.0 |
| | AFL++ | 16.258 | 2092.6 | 2111.0 |
| libxml | Waffle | 225.646 | 11246.3 | 11504.0 |
| | AFL | 355.017 | 11456.3 | 11851.0 |
| | AFLFast | 152.346 | 10953.6 | 11093.0 |
| | AFL++ | 84.571 | 11334.3 | 11431.0 |

Table 4.2: Coverage stats

Table 4.2 contains the coverage stats of the experiments. These stats describe an average coverage performance (on three trials) for each fuzzer on benchmarks.

### 4.4.2 Execution time

The performance of the fuzzers, based on execution times, is shown in histograms of instances of the tests (Figures 4.4 to 4.7). In Figures 4.4, 4.6 and 4.7, although the total number of findings in Waffle is less than AFL's, the Waffle's findings are spread more toward higher execution times.

Table 4.3 has the performance of **one** trial of each test, based on the execution time.

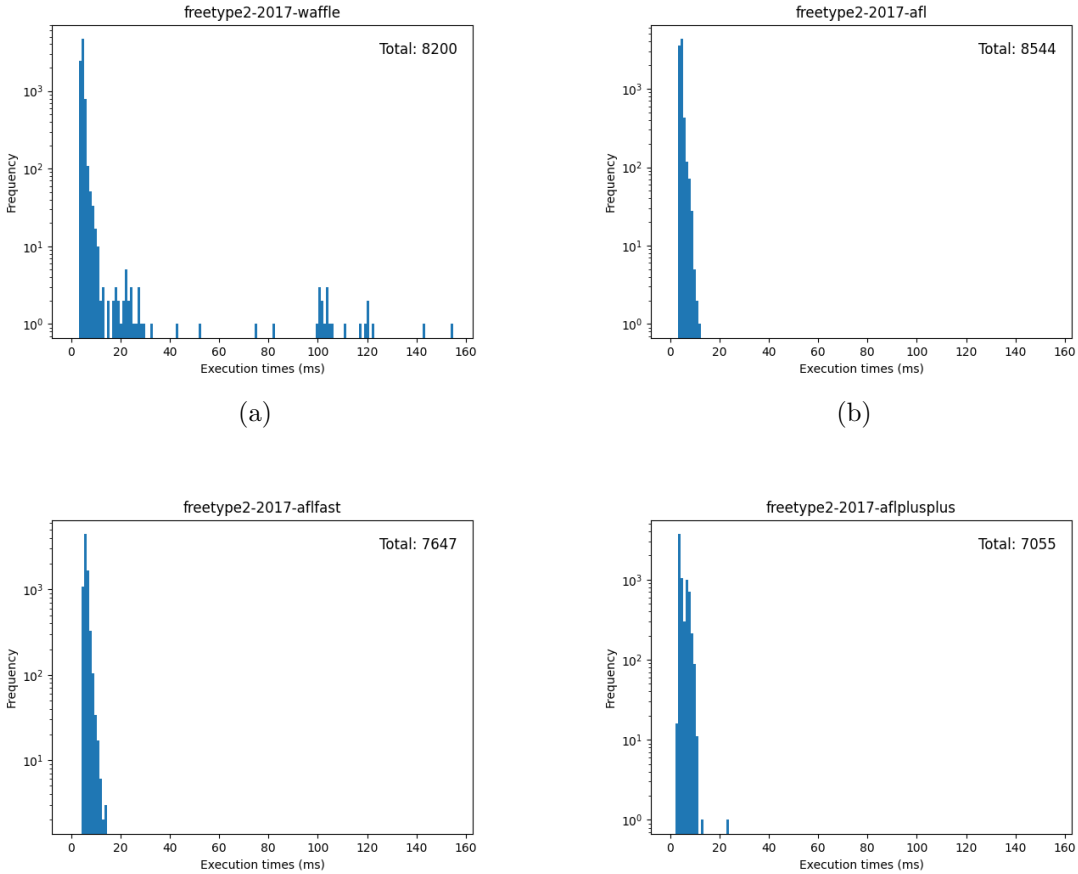



Figure 4.4: Histogram of execution times: freetype2-2017

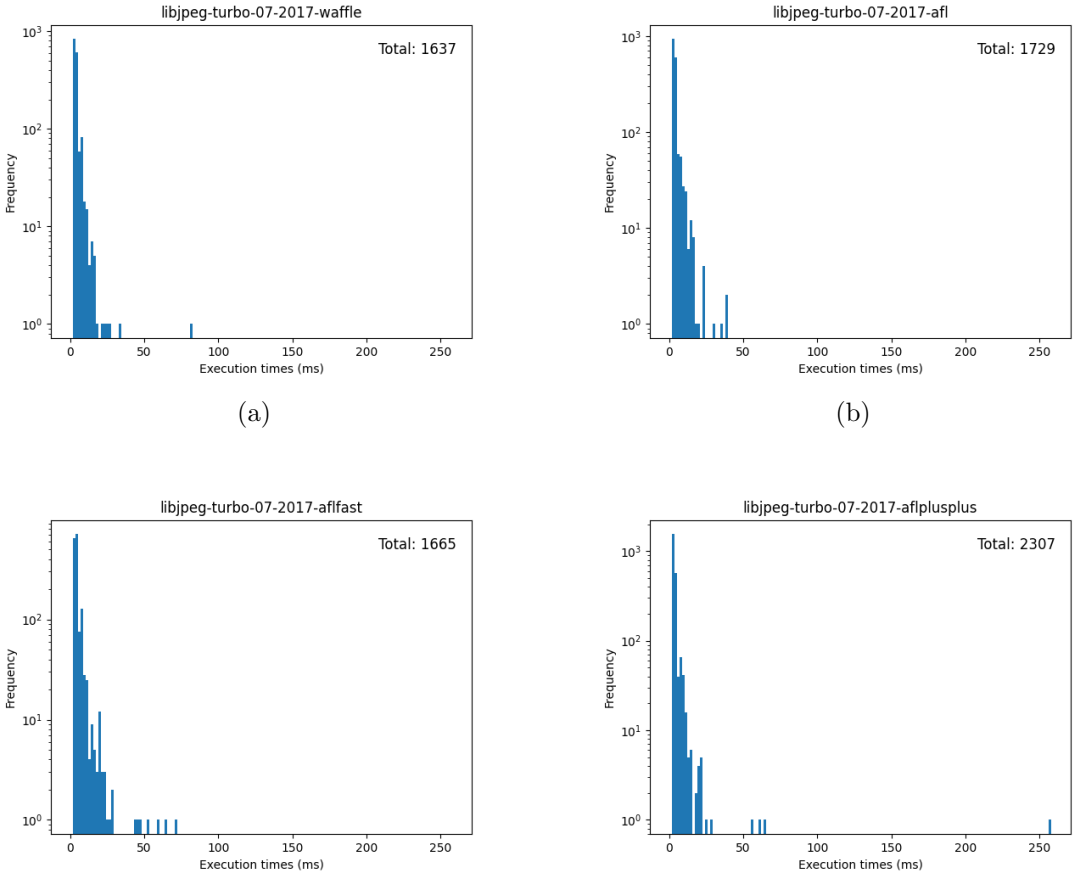



Figure 4.5: Histogram of execution times: libjpeg-turbo-07-2017

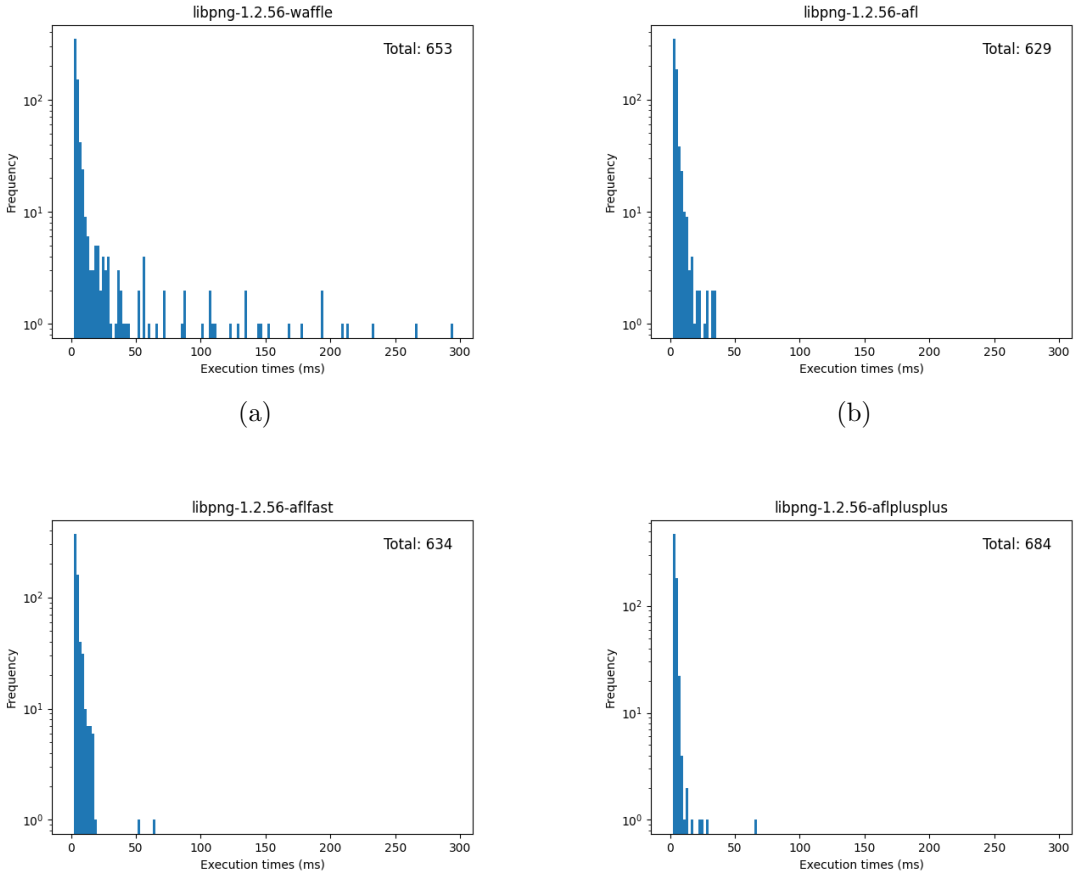
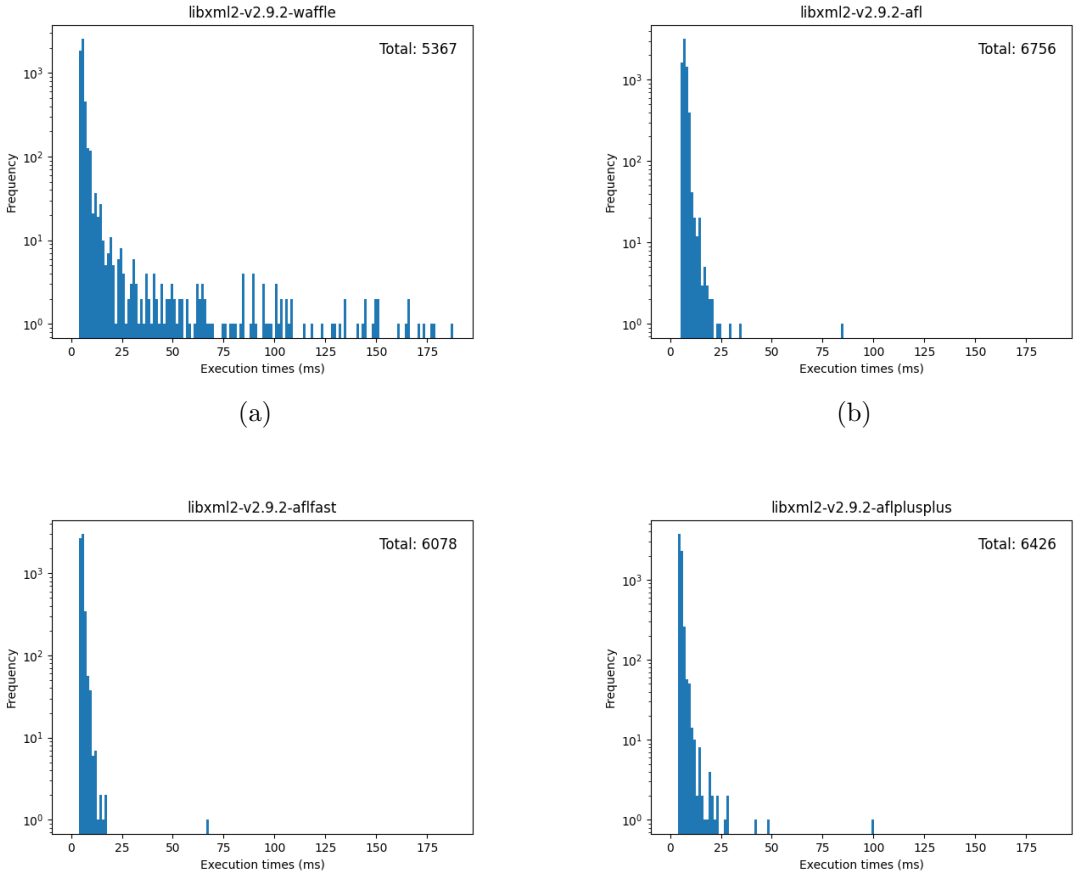



Figure 4.6: Histogram of execution times: libpng-1.2.56




(a) (b)

(c) (d)

Figure 4.7: Histogram of execution times: libxml2-v2.9.2

| *Benchmarks* | *Fuzzer* | $Min_{(ms)}$ | $Max_{(ms)}$ | $Mean_{(ms)}$ | $Median_{(ms)}$ | *STD* | *Total* |
|---|---|---|---|---|---|---|---|
| freetype | Waffle | 4.2 | 155.0 | 5.7 | 5.2 | 5.647 | 8201 |
| | AFL | 4.3 | 12.7 | 5.2 | 5.0 | 0.668 | 8544 |
| | AFLFast | 5.4 | 14.3 | 6.7 | 6.5 | 0.831 | 7647 |
| | AFL++ | 3.9 | 23.1 | 5.7 | 4.9 | 1.682 | 7056 |
| libjpeg | Waffle | 3.1 | 81.3 | 4.6 | 3.9 | 2.884 | 1637 |
| | AFL | 3.3 | 38.2 | 4.7 | 3.9 | 2.674 | 1729 |
| | AFLFast | 3.3 | 71.6 | 5.3 | 4.1 | 4.374 | 1665 |
| | AFL++ | 3.0 | 258.8 | 4.4 | 3.7 | 6.025 | 2307 |
| libpng | Waffle | 2.8 | 295.1 | 12.1 | 3.9 | 31.245 | 643 |
| | AFL | 2.7 | 34.8 | 5.0 | 3.8 | 3.882 | 629 |
| | AFLFast | 2.9 | 64.9 | 4.9 | 3.8 | 3.967 | 635 |
| | AFL++ | 2.8 | 65.1 | 4.1 | 3.5 | 3.003 | 684 |
| libxml | Waffle | 5.2 | 188.0 | 8.3 | 6.2 | 13.071 | 5369 |
| | AFL | 6.0 | 85.0 | 7.7 | 7.5 | 1.551 | 6757 |
| | AFLFast | 5.2 | 67.5 | 6.2 | 6.0 | 1.064 | 6078 |
| | AFL++ | 5.0 | 100.8 | 6.1 | 5.9 | 1.781 | 6426 |

Table 4.3: Execution times stats

## 4.5 Conclusion

In this chapter, we evaluated our fuzzer with two metrics: code coverage, and execution time. Fuzzbench generates a code coverage report in different plots and tables, and we illustrated the gradual performance of the findings in Figure 4.2. In another perspective, Figure 4.3, the findings suggest the capability of Waffle in finding unique findings, which are expected due to the allowance of ERU-based guidance.

To measure the execution times, histograms of the findings were compared as in Figures 4.4 to 4.7. Based on the distribution of the figures, we conclude that Waffle generates more time-consuming executions, which was the goal of this thesis.

# Chapter 5

# Conclusions and Future works

## 5.1 Conclusions

In this thesis, we tried to propose a solution for coverage-based fuzz testing, which has the capability of generating resource-consuming inputs. Our implementations consider time as the main feature for resource-consumptions. The challenges in our work contained modifications in instrumentation, fuzzing, and evaluating the results. The instrumentation collects the coverage information, and we insert instructions for tracking an Estimation of Resource Usages during the preparation of the executives. After passing the resulting programs to our fuzzer, Waffle takes a greedy technique to increase the resource (time) usages by storing the recent most time-consuming inputs during fuzzing. This leads to a performance-guidance besides the available coverage-guidance, so that various regions of code are covered, and the regions are exploited for higher resource-usage.

To mitigate the problem of comparing the performance of different fuzzers, we used FuzzBench for benchmarking Waffle. We added Waffle to the Fuzzbench's project, and let our fuzzer to start benchmarking with three other AFL-based fuzzers (AFL, AFLFast, AFL++). The results of the Fuzzbench's reports showed that al-

though Waffle has a slight slower input's generation, it discovered more code coverages of some of the benchmarks. On the other hand, we post-processed the resulting inputs of each fuzzer to collect execution times. The results showed that Waffle could generate inputs with significant higher execution time. We provided statistics that indicates the capability of Waffle in processing time-consuming inputs.

## 5.2 Future works

We explain some possible future works in three domains:

1. Instrumentation:

   The usage of a performance array slows down the execution of the program and the analysis of executions in the fuzzing phase. Possible suggested enhancements for mitigating this problem are enriching the instrumentation with more pre-compilation analysis of the binary, and reducing the size of the shared memory for saving the performance data.

2. Fuzzing:

   The processing of the inputs is impacted by evaluating the shared memory. In AFL, the performance of the executions' analysis is enhanced by decreasing the comparisons made while checking the shared array, yet, Waffle compares all 4-bytes cells of the shared array to find new fitnesses. In addition, as Waffle is looking for resource (time) consuming executions, the growth of the queue entries, and the execution of those entries takes longer duration. Refering to this problem, an efficient less number of executions for testing resource-consuming executions would help the fuzzer.

3. Benchmarking:

The suggested testing configurations for fuzzbenching the fuzzers is a minimum of 20 trials which takes 24 hours of testing for each fuzzer. The setup for such configuration was not applicable on the local machines, but Google accepts free cloud testings after submitting changes to the project. By adding a stable version of Waffle to the project, we can investigate the performance of our fuzzer among other provided fuzzers. Another work, which would enhance the benchmarking procedure is to add the time/resource-consumption measurements while processing the benchmarks, and generate the reports in a human-readable document.

# Bibliography


[1] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[2] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[3] OmniSci. What is llvm. `https://www.omnisci.com/technical-glossary/llvm`, 2020. [Online]; accessed in 2020.

[4] Afl-cve. `https://github.com/mrash/afl-cve`, 2019.

[5] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann Publishers San Francisco, 1998.

[6] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[8] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265. ACM, 2018.

[9] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[10] american fuzzy lop - a security-oriented fuzzer. `https://github.com/google/AFL`, 2021.

[11] LLVM. Llvm project. `http://llvm.org/`, 2020. [Online]; accessed in 2020.

[12] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. California, USA, 2005.

[13] High-performance binary-only instrumentation for afl-fuzz. `https://github.com/mirrorer/afl/blob/master/qemu_mode/README.qemu`, 2020.

[14] Dynamorio. `https://dynamorio.org/`, 2021.

[15] Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. `https://frida.re/`, 2021.

[16] Pin - a dynamic binary instrumentation tool. `https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html`, 2021.

[17] Iso 27005, information technology — security techniques — information security risk management (third edition). `https://www.iso27001security.com/html/27005.html`, 2018.

[18] Yung-Yu Chang, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. Trend analysis of the cve for software vulnerability management. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, pages 1290–1293. IEEE, 2011.

[19] Yunfei Su, Mengjun Li, Chaojing Tang, and Rongjun Shen. An overview of software vulnerability detection. *International Journal of Computer Science And Technology*, 7(3):72–76, 2016.

[20] Denial of service software attack - owasp foundation. `https://owasp.org/www-community/attacks/Denial_of_Service`, 2021.

[21] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[22] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[23] Michal Zalewski. Pulling jpegs out of thin air. `https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html`, 2014.

[24] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522, 2013.

[25] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: toward a stateful network protocol fuzzer. In *International Conference on Information Security*, pages 343–358. Springer, 2006.

[26] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*, pages 330–347. Springer, 2015.

[27] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 523–538, 2012.

[28] Fabien Duchene, Roland Groz, Sanjay Rawat, and Jean-Luc Richier. Xss vulnerability detection using model inference assisted evolutionary fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 815–817. IEEE, 2012.

[29] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[30] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in smt solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 701–712, 2020.

[31] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[32] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[33] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[34] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1066–1071. IEEE, 2011.

[35] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[36] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 474–484. IEEE Computer Society, 2009.

[37] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[38] Young-Hyun Choi, Min-Woo Park, Jung-Ho Eom, and Tai-Myoung Chung. Dynamic binary analyzer for scanning vulnerabilities with taint analysis. *Multimedia Tools and Applications*, 74(7):2301–2320, 2015.

[39] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017.

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[41] Qian Yang, J Jenny Li, and David M Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009.

[42] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

[43] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2155–2168. ACM, 2017.

[44] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 765–777, 2020.

[45] Michal Zalewski. American fuzzy lop.(2014). `http://lcamtuf.coredump.cx/afl`, 2019.

[46] More about afl. `https://afl-1.readthedocs.io/en/latest/about_afl.html`, 2019.

[47] Fast llvm-based instrumentation for afl-fuzz. `https://github.com/google/AFL/blob/master/llvm_mode/README.llvm`, 2019.

[48] Clang: a c language family frontend for llvm. `https://clang.llvm.org/`, 2020. [Online]; accessed in 2021.

[49] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[50] Tips for parallel fuzzing. https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt, 2016.

[51] Base class for instruction visitors. https://llvm.org/doxygen/InstVisitor_8h_source.html, 2021.

[52] Fuzzgoat: Vulnerable c program. https://github.com/fuzzstati0n/fuzzgoat.git, 2017.

[53] László Szekeres Jonathan Metzman, Abhishek Arya, and L Szekeres. Fuzzbench: Fuzzer benchmarking as a service. *Google Security Blog*, 2020.

[54] Default benchmarking projects used in fuzzbench. https://github.com/google/fuzzbench/tree/master/benchmarks, 2021.

[55] Afl driver for controlling the fuzzing procedure. https://raw.githubusercontent.com/llvm/llvm-project/5feb80e748924606531ba28c97fe65145c65372e/compiler-rt/lib/fuzzer/afl/afl_driver.cpp, 2018.

[56] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1393–1403, 2021.

# Chapter 6

# Appendix

## 6.A Waffle

### 6.A.1 random_havoc

An abstract implementation of the `havoc_stage` used in AFL and Waffle. Most of the commands are removed, and the remaining comments describe the operations of this stage.

```
havoc_stage:
  /* The havoc stage mutation code is also invoked when splicing
   files; if the splice_cycle variable is set, generate different
   descriptions and such. */

  if (!splice_cycle) {
    stage_max   = (doing_det ? HAVOC_CYCLES_INIT : HAVOC_CYCLES) *
                  perf_score / havoc_div / 100;
  }
  else {
    stage_max   = SPLICE_HAVOC * perf_score / havoc_div / 100;
  }

  /* We essentially just do several thousand runs (depending on
   perf_score) where we take the input file and make random stacked
    tweaks. */
  for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {
    u32 use_stacking = 1 << (1 + UR(HAVOC_STACK_POW2));
    for (i = 0; i < use_stacking; i++) {
```

```
16      switch (UR(15 + ((extras_cnt + a_extras_cnt) ? 2 : 0))) {
17        case 0:
18          /* Flip a single bit somewhere. Spooky! */
19        case 1:
20          /* Set byte to interesting value. */
21        case 2:
22          /* Set word to interesting value, randomly choosing endian
    . */
23        case 3:
24          /* Set dword to interesting value, randomly choosing
    endian. */
25        case 4:
26          /* Randomly subtract from byte. */
27        case 5:
28          /* Randomly add to byte. */
29        case 6:
30          /* Randomly subtract from word, random endian. */
31        case 7:
32          /* Randomly add to word, random endian. */
33        case 8:
34          /* Randomly subtract from dword, random endian. */
35        case 9:
36          /* Randomly add to dword, random endian. */
37        case 10:
38          /* Just set a random byte to a random value. Because, why
    not. We use XOR with 1-255 to eliminate the possibility of a no-
    op. */
39        case 11 ... 12:
40          /* Delete bytes. We're making this a bit more likely than
    insertion (the next option) in hopes of keeping files reasonably
     small. */
41        case 13:
42          /* Clone bytes (75%) or insert a block of constant bytes
    (25%). */
43        case 14:
44          /* Overwrite bytes with a randomly selected chunk (75%) or
     fixed bytes (25%). */
45
46          /* Values 15 and 16 can be selected only if there are any
    extras present in the dictionaries. */
47        case 15:
48          /* Overwrite bytes with an extra. */
49        case 16:
50          /* Insert an extra. Do the same dice-rolling stuff as for
    the previous case. */
51        }
52      }
53      /* Write a modified test case, run program, process results.
    Handle error conditions, returning 1 if it's time to bail out.
    This is a helper function for fuzz_one(). */
54    }
```

Listing 6.1: Random havoc stage

## 6.B FuzzBench

We have reviewed the recipe for adding Waffle to FuzzBench in this section.

### 6.B.1 builder.Dockerfile

Builds Waffle for the usage in FuzzBench.

The `parent_image` is an image instance with premitive configurations, and is on `ubuntu:xenial` OS. Building Python, installing python requirements, and installing the `google-cloud-sdk`, are the operations applied on the `parent_image`.

```
ARG parent_image
FROM $parent_image

RUN apt-get clean
RUN apt-get update --fix-missing
RUN apt-get -y install wget git build-essential software-properties-common apt-transport-https --fix-missing

# The llvm we are looking for
RUN wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key | apt-key add -
RUN add-apt-repository ppa:ubuntu-toolchain-r/test && echo $(gcc -v)
RUN apt-add-repository "deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-6.0 main" -y
RUN apt-get update
RUN apt-get install -y gcc-7 g++-7 clang llvm

# Clone and build the repository for Waffle
RUN git clone -b waffle --single-branch https://github.com/behnamarbab/memlock-waffle.git /source_files
RUN cd /source_files/waffle && \
    unset CFLAGS CXXFLAGS && \
    AFL_NO_X86=1 make && \
    cd llvm_mode && make

#  Install the driver for communicating with FuzzBench
RUN wget https://raw.githubusercontent.com/llvm/llvm-project/5feb80e748924606531ba28c97fe65145c65372e/compiler-rt/lib/fuzzer/afl/afl_driver.cpp -O /source_files/afl_driver.cpp  && \
    cd /source_files/waffle && unset CFLAGS CXXFLAGS && \
    clang -Wno-pointer-sign -c /source_files/waffle/llvm_mode/afl-llvm-rt.o.c -I/source_files/waffle/ && \
    clang++ -stdlib=libc++ -std=c++11 -O1 -c /source_files/afl_driver.cpp  && \
```

```
    ar r /libAFL.a *.o
```

Listing 6.2: Recipe for building Waffle FuzzBench

### 6.B.2 fuzzer.py

This python program specifies the sequence of the actions Waffle takes, in order to start fuzzing and use the benchmark as the target program. This program modifies the file located in {FUZZBENCH_DIR}/fuzzers/AFL/fuzzer.py. As Waffle is based on AFL, this program can start the process same as the AFL's fuzzer.py.

```
"""Integration code for Waffle fuzzer."""

import json
import os
import shutil
import subprocess

from fuzzers import utils


def prepare_build_environment():
    """Set environment variables used to build targets for AFL-based
    fuzzers."""
    cflags = ['-fsanitize-coverage=trace-pc-guard']
    utils.append_flags('CFLAGS', cflags)
    utils.append_flags('CXXFLAGS', cflags)

    os.environ['CC'] = 'clang'
    os.environ['CXX'] = 'clang++'
    os.environ['FUZZER_LIB'] = '/libAFL.a'


def build():
    """Build benchmark."""
    prepare_build_environment()

    utils.build_benchmark()

    print('[post_build] Copying waffle-fuzz to $OUT directory')
    # Copy out the waffle-fuzz binary as a build artifact.
    shutil.copy('/source_files/waffle/waffle-fuzz', os.environ['OUT'
    ])


def get_stats(output_corpus, fuzzer_log):  # pylint: disable=unused-
    argument
    """Gets fuzzer stats for Waffle."""
    # Get a dictionary containing the stats Waffle reports.
```

```
    stats_file = os.path.join(output_corpus, 'fuzzer_stats')
    with open(stats_file) as file_handle:
        stats_file_lines = file_handle.read().splitlines()
    stats_file_dict = {}
    for stats_line in stats_file_lines:
        key, value = stats_line.split(': ')
        stats_file_dict[key.strip()] = value.strip()

    # Report to FuzzBench the stats it accepts.
    stats = {'execs_per_sec': float(stats_file_dict['execs_per_sec'])}
    return json.dumps(stats)


def prepare_fuzz_environment(input_corpus):
    """Prepare to fuzz with AFL or another AFL-based fuzzer."""
    # Tell AFL to not use its terminal UI so we get usable logs.
    os.environ['AFL_NO_UI'] = '1'
    # Skip AFL's CPU frequency check (fails on Docker).
    os.environ['AFL_SKIP_CPUFREQ'] = '1'
    # No need to bind affinity to one core, Docker enforces 1 core usage.
    os.environ['AFL_NO_AFFINITY'] = '1'
    # AFL will abort on startup if the core pattern sends notifications to
    # external programs. We don't care about this.
    os.environ['AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES'] = '1'
    # Don't exit when crashes are found. This can happen when corpus from
    # OSS-Fuzz is used.
    os.environ['AFL_SKIP_CRASHES'] = '1'

    # AFL needs at least one non-empty seed to start.
    utils.create_seed_file_for_empty_corpus(input_corpus)


def run_waffle_fuzz(input_corpus,
                    output_corpus,
                    target_binary,
                    additional_flags=None,
                    hide_output=False):
    """Run the fuzzer"""
    # Spawn the waffle fuzzing process.
    print('[run_waffle_fuzz] Running target with waffle-fuzz')
    command = ['./waffle-fuzz', '-i', input_corpus, '-o', output_corpus, '-d', '-m', 'none', '-t', '1000']
    if additional_flags:
        command.extend(additional_flags)
    dictionary_path = utils.get_dictionary_path(target_binary)
    if dictionary_path:
        command.extend(['-x', dictionary_path])
    command += [
        '--',
        target_binary,
```

```
86          # Pass INT_MAX to afl the maximize the number of persistent
    loops it
87          # performs.
88          '2147483647'
89      ]
90      print('[run_waffle_fuzz] Running command: ' + ' '.join(command))
91      output_stream = subprocess.DEVNULL if hide_output else None
92      subprocess.check_call(command, stdout=output_stream, stderr=
    output_stream)
93
94
95  def fuzz(input_corpus, output_corpus, target_binary):
96      """Run waffle-fuzz on target."""
97      prepare_fuzz_environment(input_corpus)
98
99      run_waffle_fuzz(input_corpus, output_corpus, target_binary)
```

Listing 6.3: Recipe for running fuzzing with Waffle on a target