Waffle

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, Faculty of Computer Science
Examining Board:	N/A
External Examiner:	N/A

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

.

Abstract

Start writing from here. (not more than 350 words for the doctoral degree and not more than 150 words for the masters degree).

Dedication

Dedicated to knowledge.

Acknowledgements (if any)

Start writing here. This is optional.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

Start writing here. This is optional.

$\sum$ \sum
$\bigcap$ \bigcap

Chapter 1

Introduction

1.1 Introduction

1.2 Summary of contributions

The fundamental goal of this thesis is to suggest a technique developed on AFL to identify the vulnerabilities related to excessive resource usages. The summary of our contributions follows:

- For **instrumentations**, we have proposed a technique of instrumenting a program to log the usage of resources in runtime. To collect the information, we leverage the *visitor* functions that LLVM project provides. We have empirically proved that the new instrumentation does not bring a noticeable overhead.
- We have changed the fuzzing procedure to consider the new instrumentations and enhance the generations of inputs with a higher number of executions of the specified visited instructions.

- We integerate the instrumentations and fuzzing procedure on top of AFL. Our experiments have shown an improvement in the code coverage of AFL. As the current version of our fuzz testing has introduced new bottlenecks, we may improve the code coverage more significantly. The source code is available on github [4].

1.3 Thesis Organization

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 [5]. 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 the following publishers: *ACM digital library*, *Elsevier ScienceDirect*, *IEEEXplore digital library*, *Springer online library*, *Wiley InterScience*, *USENIX*, and *Semantic scholar*. The queries for the literature reviews are "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

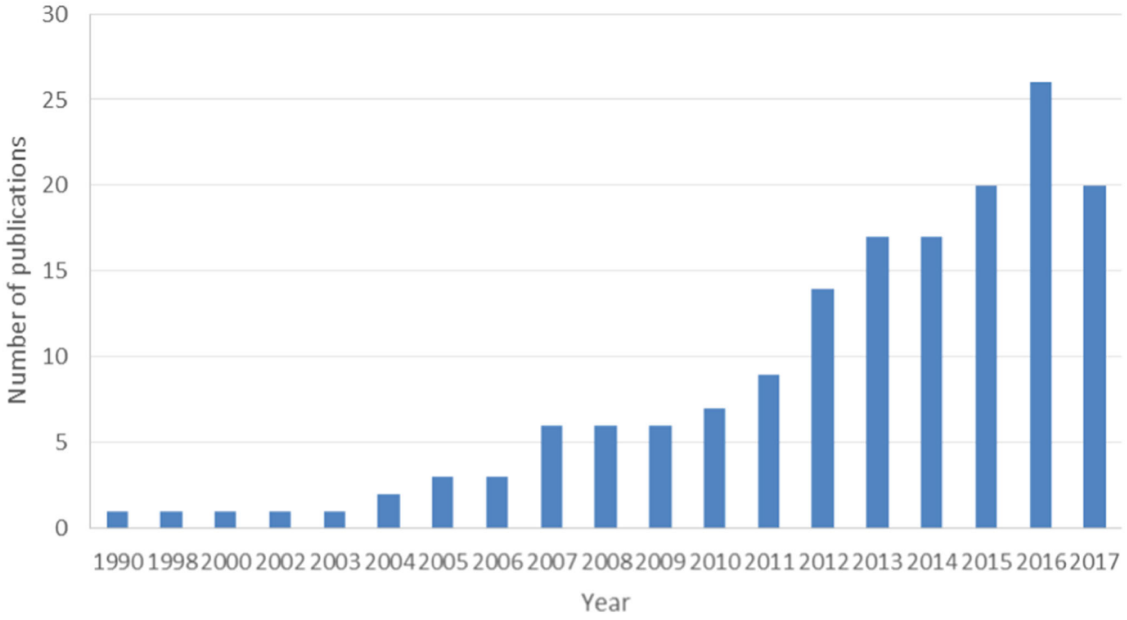


Figure 2.1: Fuzzing papers published between January 1st, 1990 and June 30 2017. [1]

run. An exception is a signal that is thrown, indicating an unexpected behavior. If 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 executions 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 [6, 7]. 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 [8]. 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" [9]. 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 [10].

American Fuzzy Lopper (AFL) [11] 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* [12] and *QEMU* [13]. "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." 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 [14]. *DynamoRIO* [15], *Frida* [16], and *PIN* [17] 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 **??**. 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." [18] 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 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. [19]. 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 [20]. The static analysis evaluates the source code or binary to expose the vulnerabilities without executing the program.

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*.

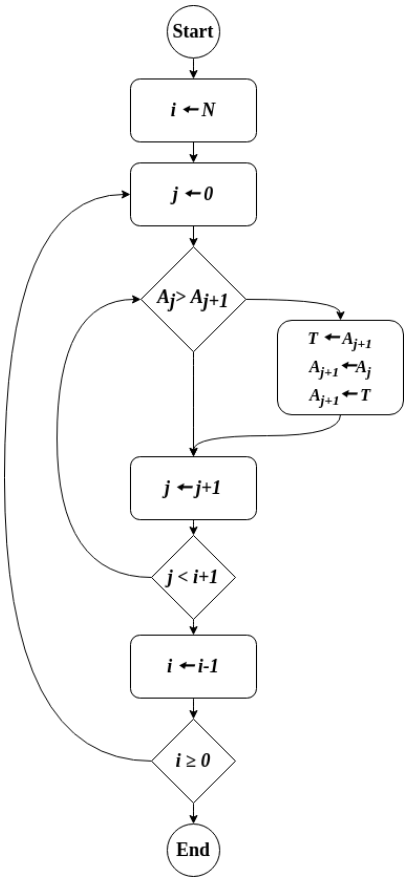
An execution processes a path (sequence) of instructions from an entry to any

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

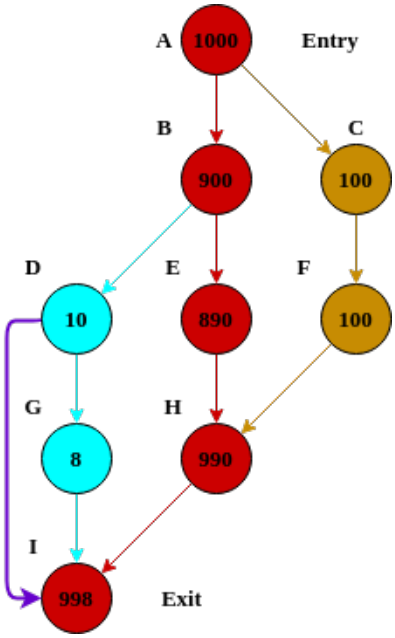
```
   Input: A, N
 1 i ← N;
 2 do
 3 |   j ← 0;
 4 |   do
 5 |   |   if A_j > A_{j+1} then
 6 |   |   |   SWAP(A_j, A_{j+1});
 7 |   |   j ← j + 1;
 8 |   while j < i + 1;
 9 |   i ← i − 1;
10 while i >= 0;
```

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 seperable basic blocks, but it does not imply that there is no other basic blocks or edges revealed after more trials. Code coverage measures number of basic blocks which could be reached in an experiment of trials.

Denial of service (DoS) is a category of vulnerabilities through network that prevents services from correctly responding back 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" [21]. In software domain, the vulnerability is either due to an early termination through a crash, or the program terminates with a timeout.



(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 vulnerabilities arise after target program executes with a triggering input. Miller introduced fuzz testing to examine the vulnerabilities of a collection of Unix utilities [22]. 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 [23]. 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* [24]. After the initial 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

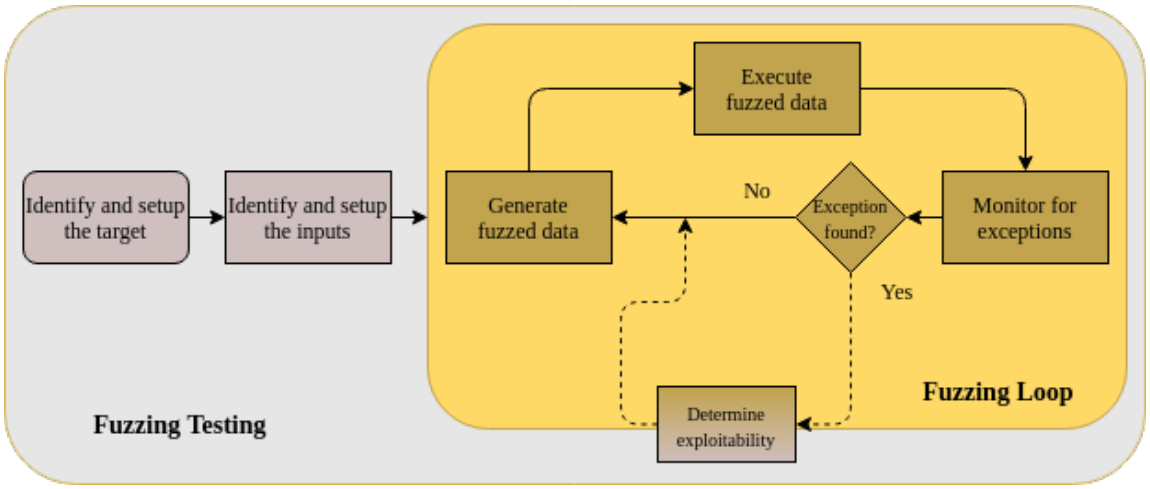


Figure 2.3: Fuzzing phases. Inspired by the definition of Sutton et al. [2]

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.

2.2.2 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.

2.2.2.1 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 API's, and the errors are expected to occur in the procedure of trying

various inputs. Blackbox fuzzing is an effective technique though its simplicity [25].

The introduced fuzzer by Miller [22] 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 [26] and [27] 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 blackboxed program as well, so as [28] and [29] 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 it's 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 on 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 [30] 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 a recent research by Mansur et al. [31], 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.

2.2.2.2 Whitebox fuzzing

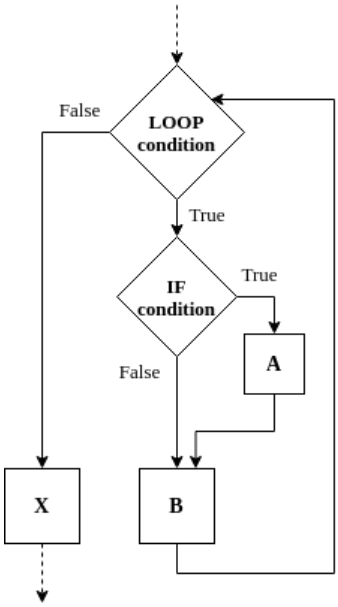
Whitebox fuzzing works with the source code of the target. The source code contains the logic of the program and can anticipate the executions' behavior without executing the program (concretely). Symbolic execution [32] is a reliable whitebox fuzzing strategy that analyzes the source code. This analysis 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 [33]. 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 [34], a whitebox fuzzer, was developed as an alternative to blackbox fuzzing to cover the lack of blackbox fuzzers [35]. It can also use dynamic and *concolic*

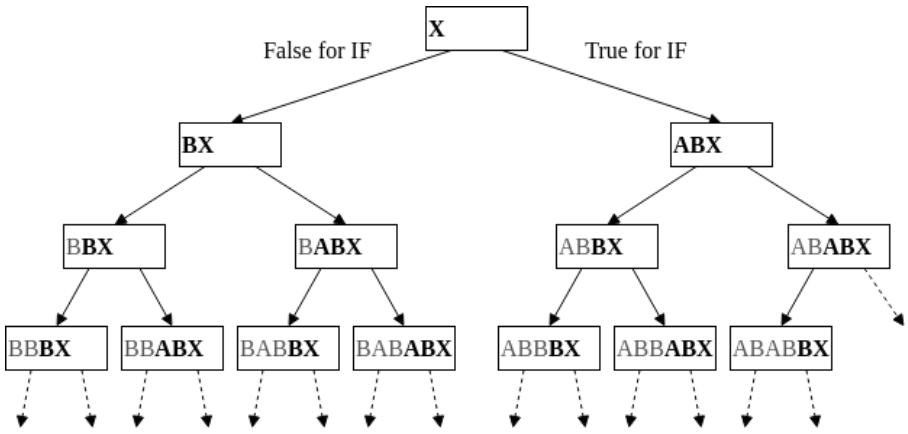
execution [36] and use taint analysis to locate the regions of seed files influencing values used by the program [37]. Concolic execution is an effective combination of symbolic execution and concrete (dynamic) executions; in a dynamic execution the fuzzer executes the program and analyzes the run. Godefroid et al. [38] have also introduced a whitebox fuzzer that investigates the grammar for parsing the input files without any prior knowledge.

2.2.2.3 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



(a) Path explosion



(b) Path explosion tree

Figure 2.4: Path explosion example

program's logic. A concrete execution of a program represents a reproducible procedure 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, 39].

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 [40] 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 [41] 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.

2.2.3 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 behavior of an execution, and does not analyze the instructions, 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*.

2.2.3.1 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 [42]. Bohme et al. [8] 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 a path for discovery of new paths**. Later, the inputs with higher energy add more fuzzed inputs to the queue.

Steelix [43] 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 light-weight 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.

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

Table 2.1: Program awareness for fuzzing

The code coverage is measured by considering a light-weight instrumentations. This method helps fuzzing to monitor the program without changing the program's resource usage (time, memory, etc.) by a noticeable amount. Hence, due to unaccessibility 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 [14].

2.2.3.2 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 [44] 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

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

Table 2.2: Input generation techniques for fuzzing

performance-guided fuzzer, PerfFuzz [9] aims to generate inputs for executions with higher **execution time** by counting the number of times each edge (jump out of 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 [45] 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 greybox 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 help seed other, more labor- or resource-intensive testing regimes down the road." [46] AFL is designed to perform **fast** and **reliable**, and at the same time, benefits from the **simplicity** and **chainability** features [47]:

- **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 which 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 to execute the un-instrumented binary under a dynamic instrumentation,

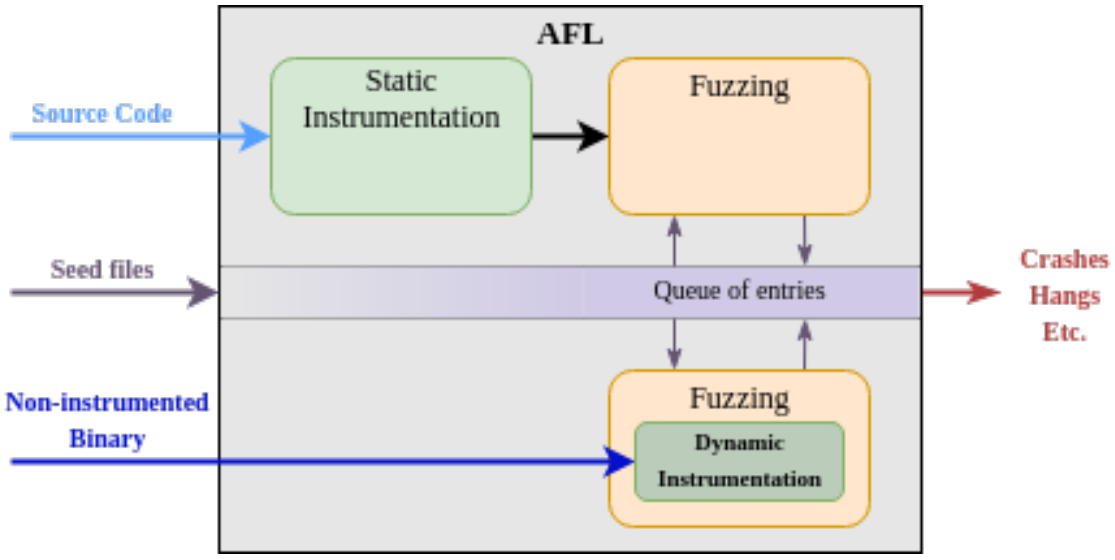


Figure 2.5: AFL's procedure: simplified

which executes the instrumenting instructions wrapped around the basic blocks of the program.

2.3.1 Instrumentation

The collection of coverage information is generated during the execution; when the execution steps into a basicblock, the procedure 3.1 stores the visited edge (pair of two consecutive basicblocks) in CFG. The hashing only stores the information from the previous basicblock and the current basicblock which we are already in. For instance, suppose we have CFG of a program as shown in 2.6; by giving the persmission to 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 basicblock 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

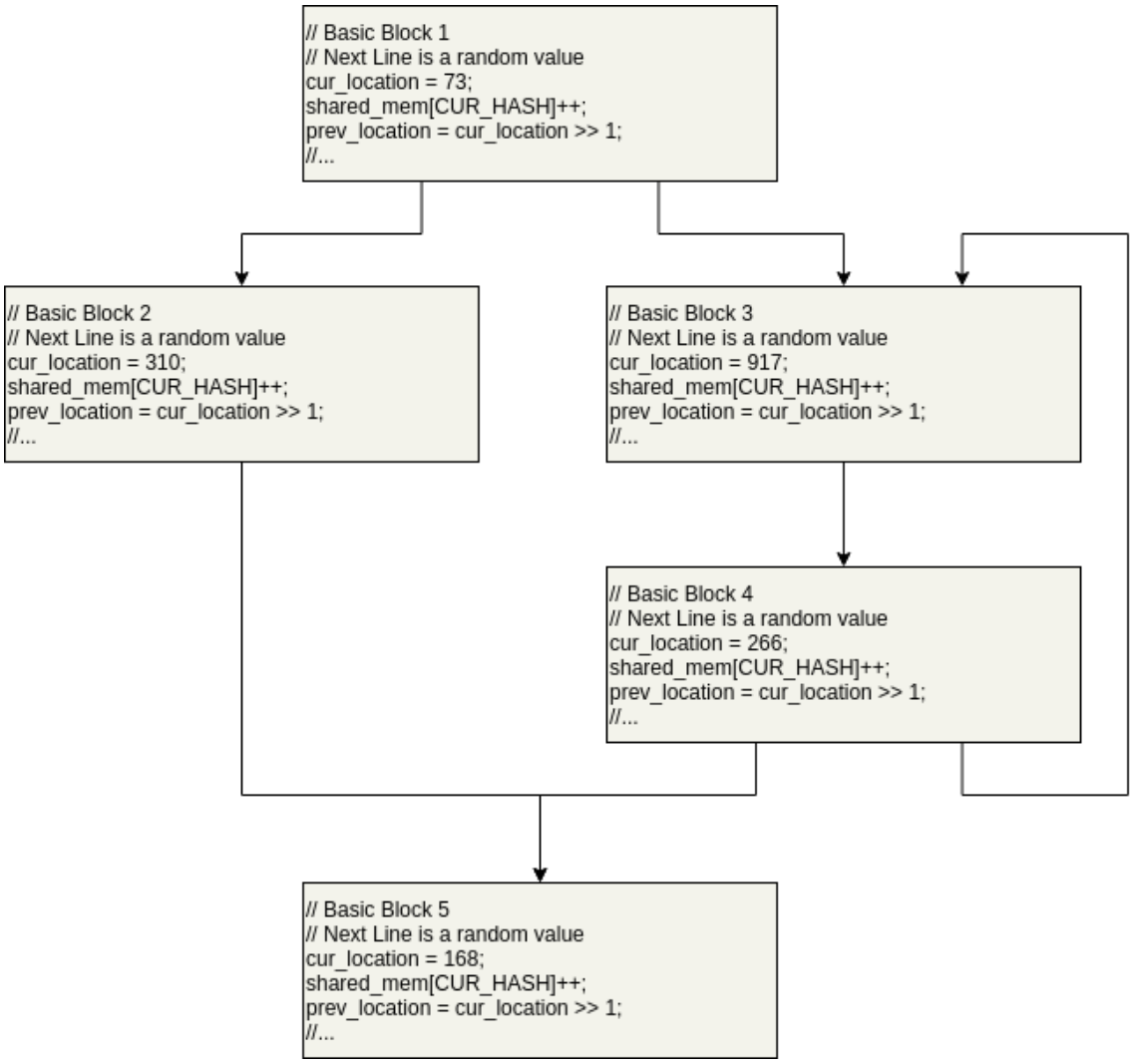


Figure 2.6: Example for instrumented basic blocks

instructions anywhere in the program. Dynamic instrumentation is another method which let let the fuzzer to use the instrumentation instructions while executing the program. By default, AFL uses `QEMU` for dynamic instrumentation [14]. The emulator wraps 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 the purpose of this article, we need to dig more into the procedure of instrumentation in llvm-mode [48].

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." [12] 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 development of a custom compiler in every step (*pass*) through the conversion of source code to executable binary file.
- **Clang** [49] provides a front-end for compiling C language family (C, C++,

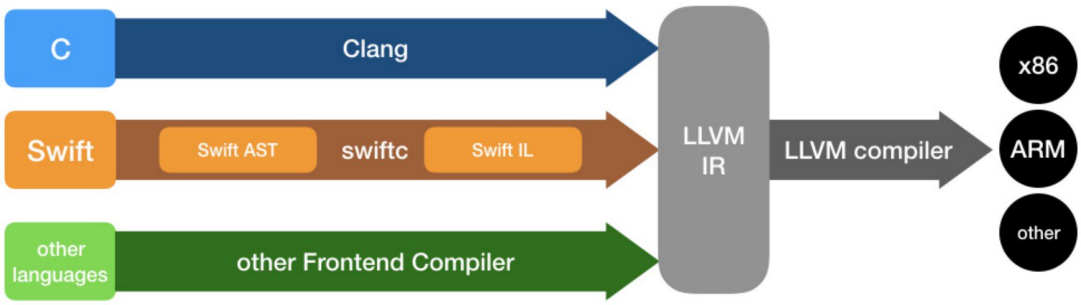


Figure 2.7: LLVM architecture: A front-end compiler generates the LLVM IR, and then it is converted into machine code [3]

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 [50]. The IR is then translated into an executable binary for the machine's CPU (Figure 2.7).

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 on 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**, and 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 basicblock 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 it's preperations; 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. [47]
  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 are 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, and 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 it's execution speed; the more locations visited stored in *coverage bitmap*, and the lower the execution speed are prefereable, 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 summerizes 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

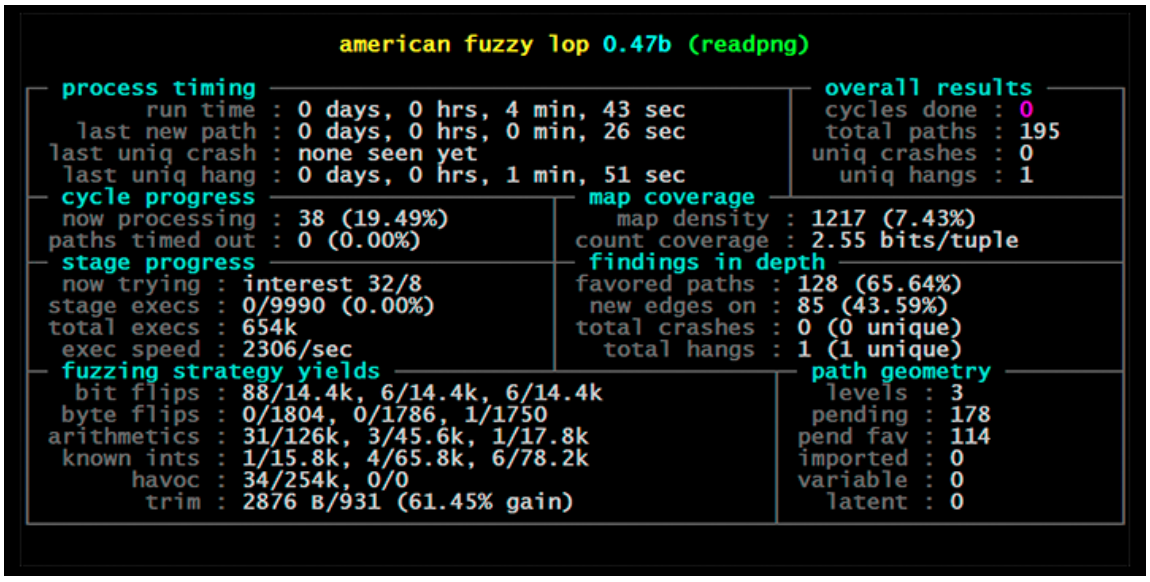


Figure 2.8: AFL status screen

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 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 an input referes to which generation the input belongs to. Considering the first input seeds as depth 0, the generated population from these inputs increase the depth by one. This shows how far the fuzzing has progressed.

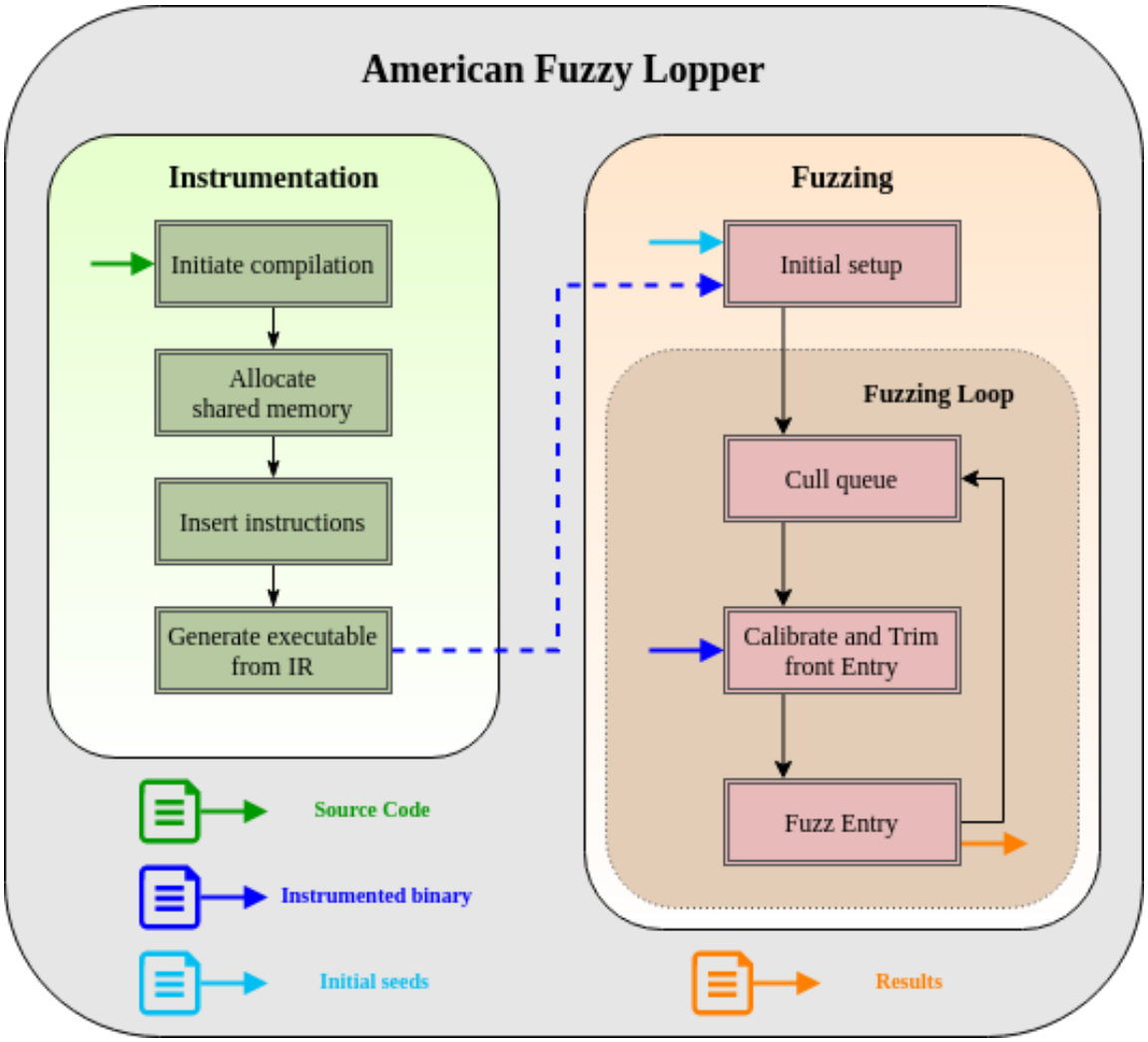


Figure 2.9: An overview of the whole fuzzing procedure of AFL

2.3.4 AFL fuzzing chain

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

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

Listing 2.3: Instrument *sample_vul*.c

The result file, `sample_inst`, is an executable which 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 recieving 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 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 server for responding the requests. As a result, finding such executions may help with stopping the server from responding appropriately.

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 for new code coverages, and eventually, the corpus evolves with coverage-guided inputs. In a performance-guided fuzzing, the inputs gradually produce fuzzed data which use more resources. Concurrent fuzzing suggests that if the corpora of parallel fuzzers is shared with rest of the fuzzers, the outcome is a corpus which tracks 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 [51].

2. 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 maximizes the occurrences of the targeted instructions. The coverage information derived from AFL remains the exploration methology of AFL, and the performance-guidance exploits the discovered regions by finding excessive resource usages.

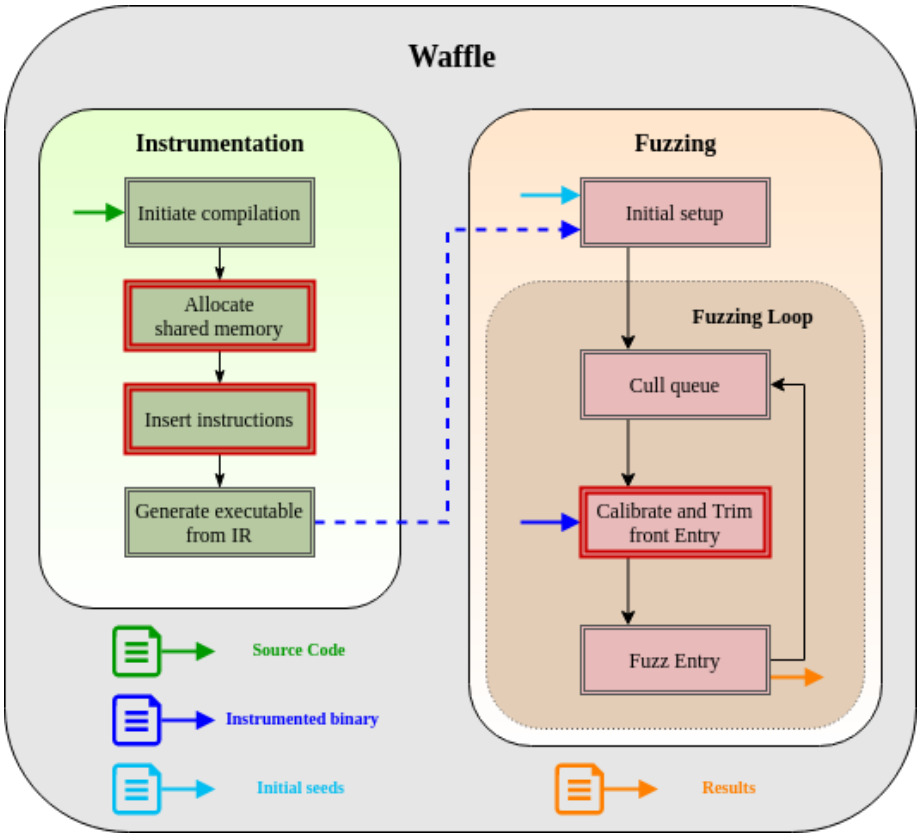


Figure 3.1: Fuzzing phases of Waffle. The red rectangles specify the changed components.

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 on 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 fasion. Waffle exploits the coverage-guided findings to discover regions of code, and tries to increase resource usages through the 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 coverage and resource usage of the executions, 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 introduce Waffle.

3.2.1 Resource complexity of execution

To address resource usage of an execution, we estimate the engaged resource usage of executions. Waffle does not analyse 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 presume a set of instructions and monitors the occurrences. A trace of the envolvement 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, 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 it's performance array (As a result, the hitmaps 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, a 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 execution's procedure. Waffle includes 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, instrumented program leaves a trace of the ERUs (collected by the *visitor* functions) of the basic blocks on 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 psudocode 3.1) for instrumentation during the compilation of any given source code.

```
1  counter = lg(count_instructions())
2  cur_location = <COMPILE_TIME_RANDOM>;
3  edge = cur_location ^ prev_location;
4
5  cov_shared_mem[edge]++;
6  per_shared_mem[edge]+=counter;
7
8  prev_location = cur_location >> 1;
```

Listing 3.1: Select element and update in shared_mem

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

#define ORC_SIZE (1 << 16)

u32  __wafl_ORC_initial[ORC_SIZE];
u32* __wafl_area_ptr = __wafl_ORC_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_ORC_ptr);
  }
}
```

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

3.2.2.1 Visitors

The implementation of *LLVM's instruction visitor functions* [52] helps Waffle in searching and counting 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 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 according basic block.

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

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

```

```
7   // Any visited instruction is counted in a specified range
8   void visitInstruction(Instruction &I) {
9     ++Count;
10  }
11 };
```

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.

```
1  // snippet of wafl-llvm-pass.so.cc
2  #include <math.h>
3  // ...
4  bool WAFLCoverage::runOnModule(Module &M) {
5    // ...
6    GlobalVariable *WAFLMapPtr =
7      new GlobalVariable(M, PointerType::get(Int32Ty, 0), false,
8      GlobalValue::ExternalLinkage, 0, "__wafl_area_ptr");
9
10   // ...
11   for (auto &F : M) {
12     for (auto &BB : F) {
13       /* Count the instructions */
14       CountAllVisitor CAV;
15       CAV.visit(BB);
16
17       // ...
18       LoadInst *ERUPtr = IRB.CreateLoad(WAFLMapPtr);
19       MapPtr->setMetadata(M.getMDKindID("nosanitize"),
20         MDNode::get(C, None));
21
22       Value* EdgeId = IRB.CreateXor(PrevLocCasted, CurLoc);
23       Value *ERUPtrIdx =
24           IRB.CreateGEP(ERUPtr, EdgeId);
25
26       /* Setup the counter for storage */
27       u32 log_count = (u32) log2(CAV.Count+1);
28       Value *CNT = IRB.getInt32(log_count);
29
30       LoadInst *ERULoad = IRB.CreateLoad(ERUPtrIdx);
31       Value *ERUIncr = IRB.CreateAdd(ERULoad, CNT);
32
33       IRB.CreateStore(ERUIncr, ERUPtrIdx)
34         ->setMetadata(M.getMDKindID("nosanitize"),
35         MDNode::get(C, None));
36
```

```
37        inst_blocks++;
38      }
39    }
40  }
```

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 an 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.

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

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

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
```

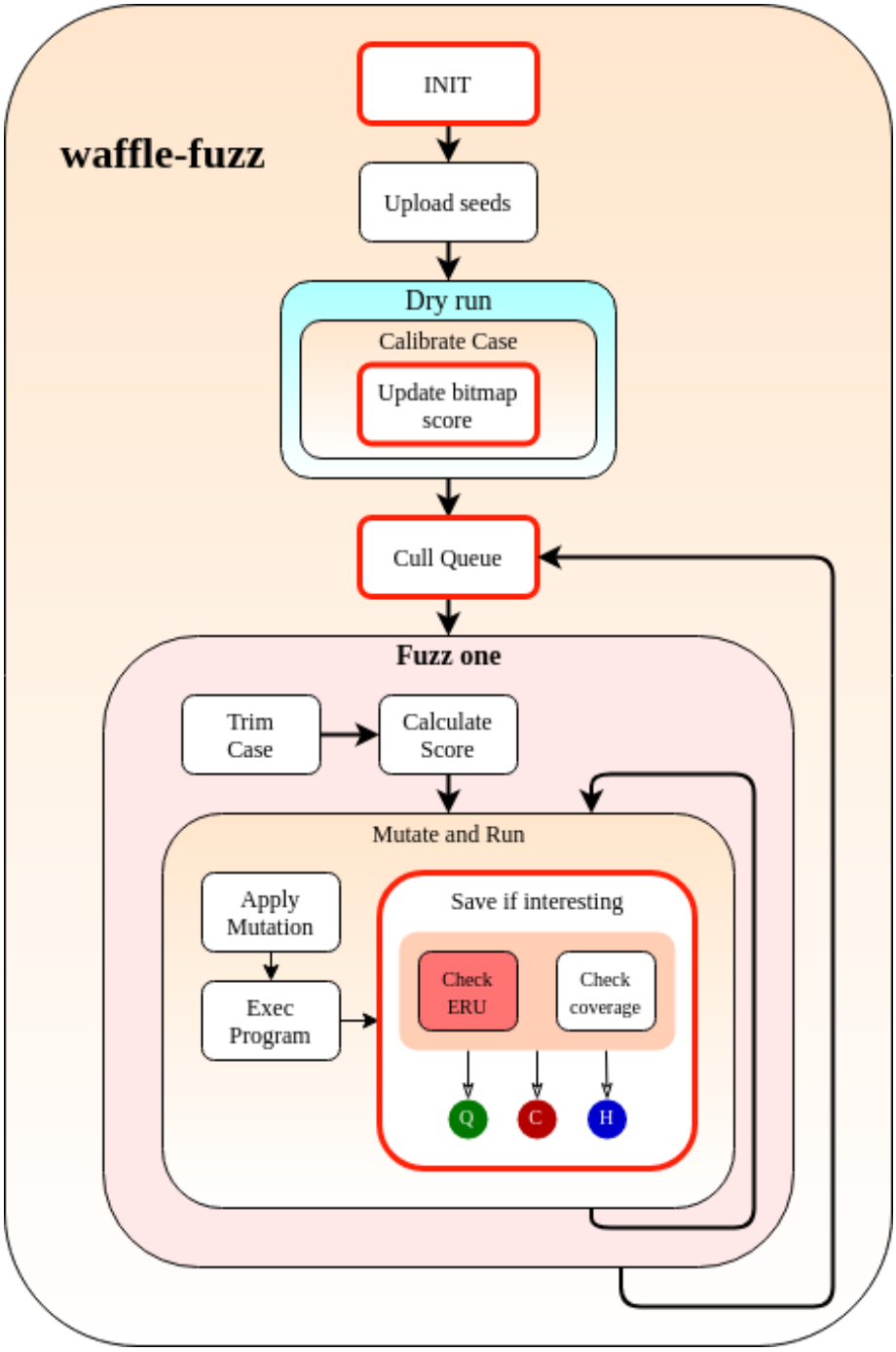


Figure 3.2: Waffle fuzzer: The red rectangles specify the new or changed components.

3.2.3 Fuzzing

Waffle follows the same fuzzing procedure with modifications in processing the inputs and selecting 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. After uploading the seeds, the fuzzing loop starts by culling the queue, modifying the priorities of the entries based on their execution features, i.e. the coverage and performance features, and finally starts fuzzing the next entry of queue. While calibrating the case, Waffle changes the bitmap scores (tracked by `top_rated`s), and let's the fuzzer to considers the winners of the coverage test and performance tests. Before entering the mutation phase, Waffle calculates the score for havoc stage and trims the input as much as it can. The mutation methods are the same as it was in AFL. After each mutation, Waffle executes the program with the mutated input and collects the execution's info. These data are then processed and the features are extracted for the guidance. Waffle first checks if there is a new coverage, and then examines the info for new ERU's in execution. The mutation continues in deterministic and non-deterministic stages, and then the fuzzer continues with the next entry after culling the queue. As an overview, the rectangles with red boundries are the modified modules, and the module for checking ERU is added to the system.

Besides the initial configurations for starting a fuzz testing, Waffle adds a new shared memory to the system. `trace_ERU` is an array in which each cell represents the ERU of an edge - or more than one edge, in case of collisions. The shared memory is then passed to the program for execution, and before each execution, it is cleared of any prior information. The structure of `queue_entry`'s is changed by adding a new variable (i.e. `TERU`) for tracking the total ERU after execution of entry.

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

In addition, the `top_rated` array for tracking the winners of each edge is extended to maintain both the coverage and the performance winners (Figure 3.3). To track the performance measurements, Waffle first checks if an edge is in the coverage map, and then processes the performance info, and if a more resource exhaustive value is retrieved of execution, the second cell of the according index points to the winner `queue_entry`.

Before starting the fuzzing loop, Waffle performs a dry run on the uploaded seeds to define an initial overview of the program; each execution fills the according `top_rated[]`'s indices with the entries with a highest score. This score is based on two main factors: i) the `fav_factor` of the execution as it was in AFL, and ii) maximizing the TERU of the corresponding execution. To investigate these execution's info, Waffle understands the paths and ERUs through the *caliberation* of each test case. Waffle utilizes `update_bitmap_score` function with the power to analyze the new performance feature (i.e. ERUs) to update the information about the current state of queue entries 3.5. The sparsness of the shared memories (`trace_bits[]`

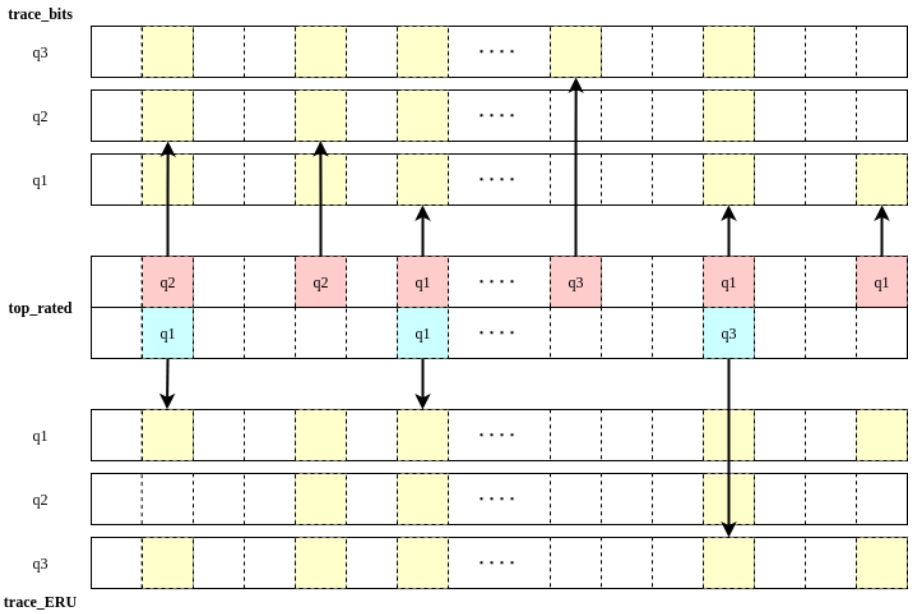


Figure 3.3: `top_rated` array: Waffle keeps track of the relevant coverage and performance features

and `ERUs[]`), and an expectation based on the fact that the repitition of same edges increases the TERU of execution, it is suitable to check the `var_bytes[]` array, an array specifying the edges with variable number of visitation in an execution (Line 13). Briefly speaking, Waffle stores the pointer to the current entry of queue (i.e. i_{th} index) in `top_rated`, if it shows a higher TERU, the second index of the `top_rated[i]` is assigned to this entry, else, it updates the first index for explaining a faster (and a smaller input size) execution 3.3. When the score is evaluated and the fitness of the entries are evaluated based on the information stored in `top_rated`, the calibration phase continues, and Waffle stores the results for future use - that is when it culls the queue in the next step.

The generated queue of entries is checked to assign the priorities of the entries. The order of the entries in queue does not change, but Waffle (as AFL does) assignes the favorness of entries by comparing them based on the current state of the fuzzing. In another words, each element of the queue is analyzed, and if it *looks* like a favorite entry, it is tagged as favorite, and else, it is marked as redundant and is ignored on it's turn to process. While processing the `top_rated` entries, if an entry shows new coverage, it is tagged as favorite, otherwise, if it has is showing a *different* ERU, the entry is tagged as the one selected for its *class*. The classes of ERUs is based on integer value of the logarithm of its value, that is:

$$ERU\_class = \lfloor \lg(eru) \rfloor \tag{3.3}$$

which indicates the position of the leftmost bit in the integeral representation of ERU. The classes are used to remove the edges of the same class, and as a result, each edge is processes maximum of 32 times (for 32bit ERU integers). The outcome of the for loop on `top_rated` holds the values of 0, for being not favored, 1 for being

```
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

favored, and 2 to specify an entry representing its class. Yet, as Waffle is trying to increase the resource exhaustion, by iterating over the list of entries, we select the class and its representative entry with the highest ERU and change its favored value to 1. The rest of the entries with $favored == 2$ are set to redundant ($favored \leftarrow 0$). An snippet of the code is shown in Listing 3.6.

```
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(var_bytes[j] && 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;
  }
}
```

Listing 3.6: Waffle `cull_queue`

The first favored entry of the queue is selected for fuzzing. First, Waffle keeps the *trimming* procedure same as AFL's, and does not consider the changes in the `ERUs[]`. This is because of the fact that the trimming procedure maintains the values of the `trace_bits[]` to remain the same, which their changes also modifies the cells of `ERUs[]`. Before the mutation loop, Waffle determines the effort it wants to take

for the current case, and calculates `perf_score` for this purpose.

In both deterministic and havoc stages of mutation, 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 is occured, or iii) a hang is occured. The results in (ii) and (iii) does not need any modification. Waffle generates inputs and examines them to verify if they show a *new coverage*, or they are exhaustive. The main goals of Waffle are to guide the code coverage to grow, as well as generating resource exhaustive executions. Listing 6 presents a pseudocode of *save_if_interesting*. The function `is_exhaustive()` aggregates the values of the `ERUs[]` and the new *interesting* feature arises when the `Total ERU` of the execution is in the same class of ERUs (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 u8 is_exhaustive() {
2    u32 teru = 0;
3    u64* current = (u64*)trace_bits;
4    u32* curreru = ERUs;
5    u32 i = (MAP_SIZE >> 3);
6
7    while(i--) {
8      if (unlikely(*current)) {
9        for(u32 j=i<<3; j<i<<3+8; j++) {
10         teru += *(curreru+j);
11       }
12     }
13     current ++;
14     curreru += 8;
15   }
16
17   return teru >= (max_TERU>>1);
```

```
18 }
```

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 [53]. Waffle generates a C-compiler program (`./waffle-clang`) based on `clang` which maintains the indicated instrumentation. We investigate the instrumentated 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 seleced basic block are specified as well. The orange sections, locate and COV, are inherited from AFL's implementation, and Waffle leverages on 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 on 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 assigned pointer.

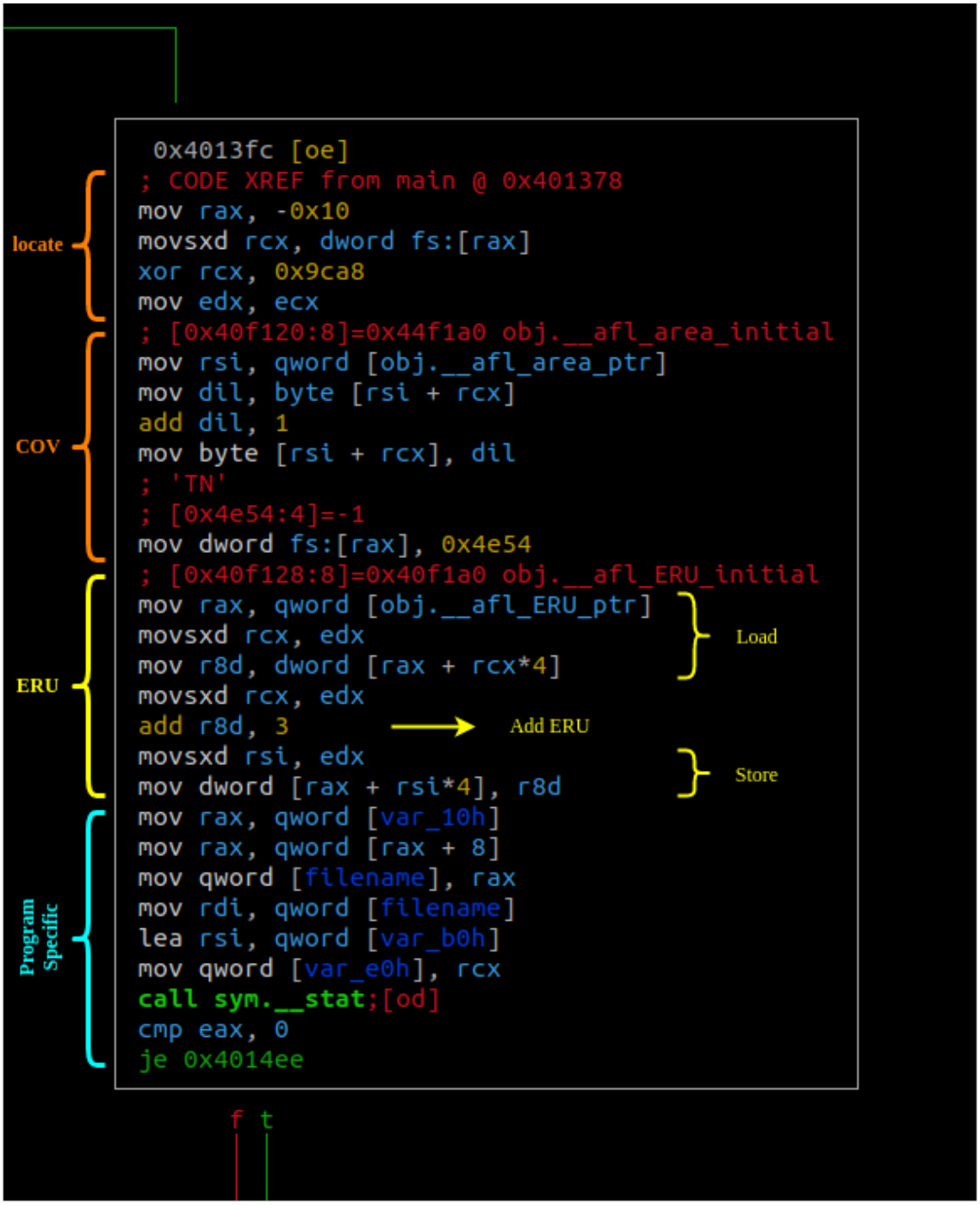


Figure 3.4: Instrumentation illustrated in basic blocks. This basic block contains the coverage-based and ERU-based instructions

3.3.2 Fuzzing

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

This chapter introduced the development of Waffle. Waffle extends AFL in two parts:

1. **Instrumentation**: **llvm_mode** directory is responsible for the instrumentation in Waffle. As we explained in this chapter, `llvm_mode/waffle-llvm-rt.o.c` contains the recipe for instrumenting the program in the compilation.

   AFL only keeps a coverage-bitmap, but in addition to the coverage-finding methodology, Waffle leverages the **visitor functions** in LLVM for assessing the more resource-exhaustive executions. Visitor functions count the targeted instructions, and Waffle saves the result in an extra *shared_memory*.

   Waffle counts the instructions in each basic block, and reduces the counted values for saving into the memory. The size of the *shared_memory* has increased

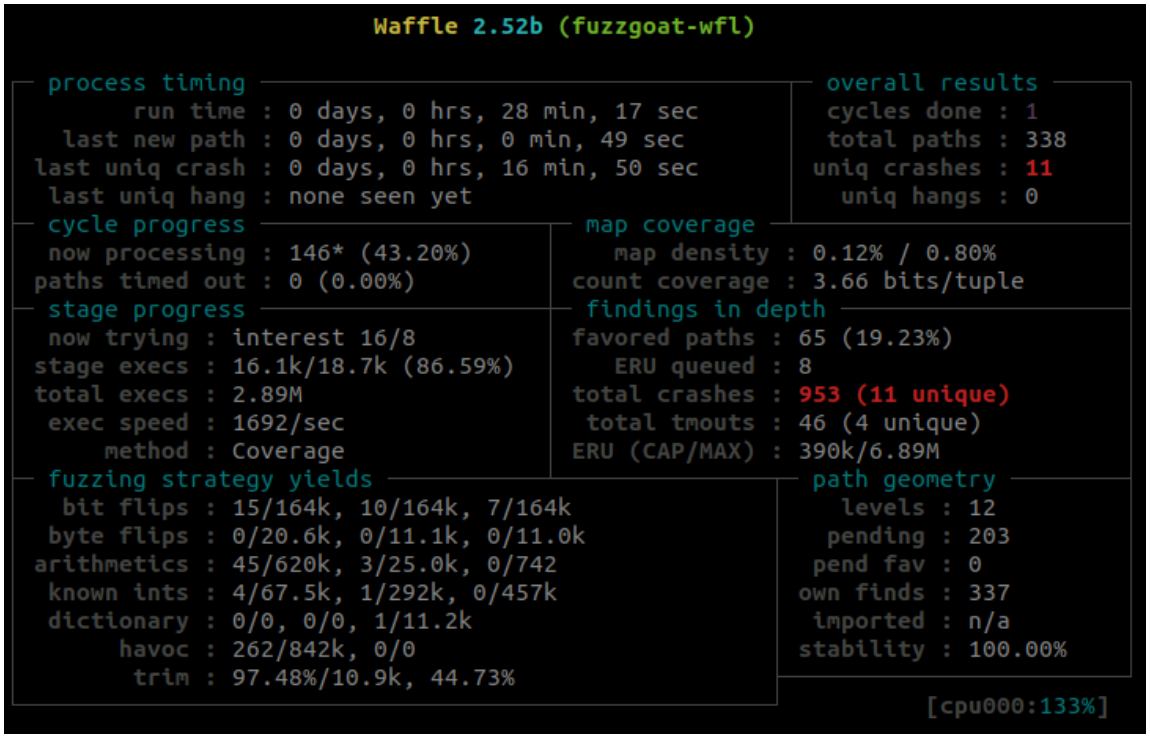


Figure 3.5: Waffle's status screen.

4x in Waffle.

2. **Fuzzing**: Waffle uses the coverage bitmap and the instruction-counter bitmap for emphasizing the more beneficial fuzzing entries. The main changes are in the procedures of the functions `calibrate_case` and `calc_score()`. Generally speaking, an interesting test-case runs faster, has more code coverage, and executes more instructions from a specified set of instructions.

We also reviewed some of the modifications in the source code to Waffle's project. The project is located on github, in a public repository [4].

Chapter 4

Simulation

4.1 Introduction

We developed Waffle to analyze the executables, with experimental instrumentation, to enhance the performance of AFL in finding test-cases exposing vulnerabilities due to resource-exhaustion. In this chapter, we compare the performance of Waffle with state-of-the-art coverage-based fuzzers, **AFL** [11], and **libfuzzer** [54]. To evaluate the performance of Waffle, we use **fuzzbench** [55] comparing the fuzzers on different programs.

In the next section, we briefly describe the **fuzzbench** project, and explain how we utilize this service in this thesis. Next, we evaluate our work, and we continue with a discussion on the performance of Waffle.

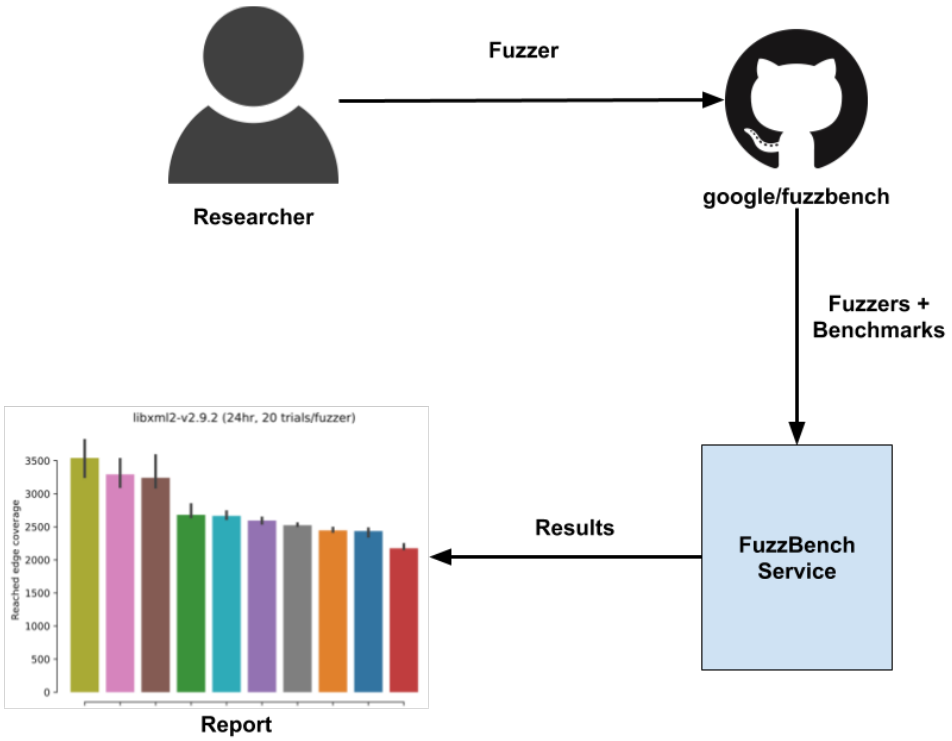


Figure 4.1: Fuzzbench overview

4.2 FuzzBench

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." [55]

Of the main features of Fuzzbench, we use the following features to test Waffle:

- **Fuzzer API:** An API for integrating fuzzers.
- **Custom/Standard benchmarks:** FuzzBench provides a set of real-world

benchmarks. In addition, we can add our custom benchmarks or use any **OSS-Fuzz** project as a benchmark.

- **Reporting:** A reporting library is provided for generating graphs and statistical tests. These reports illustrate the performance of each fuzzer on different aspects. The evaluations are based on the stats such as the performance of fuzzers in finding unique vulnerabilities, or the growth and statistics of the code coverage during the experiments.

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 that FuzzBench could communicate with. FuzzBench requires three files to provide the instructions for introducing a fuzzer:

- `builder.Dockerfile:` This file builds the fuzzer in a docker container. The recipe can be found in Appendix 6.B.1.
- `runner.Dockerfile:` This file defines the image that will be used to run benchmarks with Waffle.

```
1 FROM gcr.io/fuzzbench/base-image
```

- `fuzzer.py:` This file explains how to build and fuzz benchmarks using Waffle [Appendix 6.B.2].

According to the steps suggested for running an experiment, we add Waffle to FuzzBench, and we can assess if the fuzzer is working properly in the FuzzBench's environment.

```
export FUZZER_NAME=waffle
export BENCHMARK_NAME=libpng-1.2.56

# Building the fuzzer and benchmark in the fuzzer's environment
make build-$FUZZER_NAME-$BENCHMARK_NAME

# Required for evaluating the experiment
make format
make presubmit
```

Listing 4.1: Final steps for adding Waffle to FuzzBench

We faced some problems while adding Waffle, as the FuzzBench project **must** be built before the fuzzer is inserted into the `fuzzers` directory [56]. Otherwise, **local experiments** would fail due to existing bugs.

Start an experiment

There are three methods for initiating an experiment in FuzzBench:

- **Request in FuzzBench** "The FuzzBench service automatically runs experiments that are requested by users twice a day at 6:00 AM PT (13:00 UTC) and 6:00 PM PT (01:00 UTC). [57]"
- **Setting up a Google Cloud Project:** We can run our experiment on **GCP** [58].
- **Run a local experiment:** We can start the experiment on a non-cloud platform as well. Running a experiment on a local computer may lack enough resources for running different tests in parrallel.

We use a **local experiment** for evaluations. The local computer runs on Ubuntu 18.04 64bits, with Intel® Core™ i7-3770 CPU @ 3.40GHz × 8, and 16

GBs of RAM. The measurements for the performance of HDD, show 30MB/s for reading from the disk, and can write for 25MB/s.

4.3 FuzzBench Reports

FuzzBench starts reporting after building the project in a local experiment. Building AFL and Waffle on a benchmark such as *libpng-1.2.56* took 30 minutes on the local computer, and FuzzBench doesn't generate any reports before that.

We tested Waffle in 6-hours experiments. The test computer could not run an experiment with more than **two fuzzers** and **one benchmark**, and the higher setups fail to finish. Each experiment contains **3** trials for each pair of $< FuzzerX, Benchmark >$, where FuzzerX is either Waffle, AFL, or LibFuzzer(LF).

None of the trials could find any crashes/hangs during 6-hours trials.

Our experiments are:

1. Waffle vs AFL on libpng-1.2.56
2. Waffle vs LibFuzzer on libpng-1.2.56
3. Waffle vs AFL on openthread-2019-12-23
4. Waffle vs LibFuzzer on openthread-2019-12-23
5. Waffle vs AFL on php_php-fuzz-execute
6. Waffle vs LibFuzzer on php_php-fuzz-execute
7. Waffle vs AFL on curl_curl_fuzzer_http
8. Waffle vs AFL on sqlite3_ossfuzz

We evaluate the results according to the generated reports. The graphs [Figures 4.2, **??**, and **??**] illustrate the covered regions of the code in each experiment, the pairwise unique coverage, and the growth of the code coverage over time. Green color represents Waffle in all figures.

For the benchmarks *libpng-1.2.56*, *php_php-fuzz-execute*, and *openthread*, we analyzed the performance of Waffle with each on of the other fuzzers. In our setups, we continued testing *sqlite3_ossfuzz* and *curl_curl_fuzzer_http* with Waffle and AFL for further investigations.

Performance measurements

FuzzBench compares the performance of the fuzzers based on the discovered code coverage of each experiment. Meaning that, for example, FuzzBench does not check how many executions a fuzzer performs; instead, it considers the variety of the generated inputs, according to their execution paths.

Table 4.1 shows the summary of our results. The table shows the coverage stats for the participating fuzzers; the columns are $mean coverage$, $standard deviation$, and the number of $unique findings$ of each fuzzer in each experiment.

Figure 4.2 shows the growth of the code coverage that each fuzzer had explored. In testing *libpng* [4.2a], *openthread* [4.2c], and *curl* [4.2g], we can see that both Waffle and AFL eventually converge, and it suggests that, here, Waffle performs indifferently in the discovery of new regions of code (higher code coverage). Waffle also shows latency in generating the first generations of the inputs and continues a close competition with AFL.

For *php_php-fuzz-execute*, AFL shows 15% better performance; however, for the

Benchmark	Fuzzer	Waffle Mean Coverage	Fuzzer Mean Coverage	Waffle STD	Fuzzer STD	Unique (Waffle/Fuzzer)
libpng-1.2.56	AFL	1509	1510	0.57	0.57	0/1
libpng-1.2.56	LibFuzzer	1509	1951	0.57	6.65	0/425
php_php-fuzz-execute	AFL	147945	169447	5451.77	2069.34	1270/18902
php_php-fuzz-execute	LibFuzzer	145816	138666	2580.20	234.08	10179/335
openthread-2019-12-23	AFL	5226	5216	12.58	24.13	0/2
openthread-2019-12-23	LibFuzzer	5242	5852	3.21	24.24	6/384
sqlite3_ossfuzz	AFL	33510	32245	2631.01	872.98	1435/1088
curl_curl_fuzzer_http	AFL	17142	17290	215.89	192.12	55/154

Table 4.1: Statistics of the experiments.

other benchmark, *sqlite3_ossfuzz*, Waffle has enhanced the performance of AFL by 4%.

LibFuzzer is showing better performance than Waffle as in Figures 4.2b and 4.2d. LibFuzzer found 30% and 12% more code coverage for fuzzing *libpng* and *openthread*, respectively. On the other hand, fuzzing *php* has 5% more code coverage under Waffle.

Comparing the graphs on the left column (AFL) and the right column (LF) for the benchmarks, *libpng*, *openthread*, and *php*, shows the following rankings:

1. **libpng:** 1: LibFuzzer. 2: Waffle/AFL.
2. **openthread:** 1: LibFuzzer. 2: Waffle/AFL.
3. **php:** 1: AFL. 2: Waffle. 3: LibFuzzer.

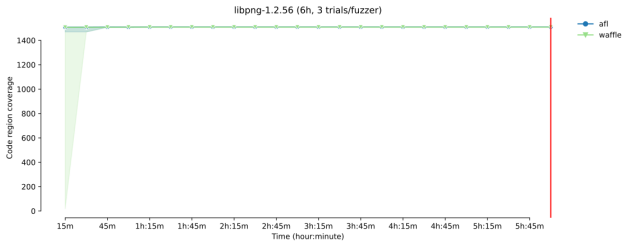
In the third scenario, we see that in the *6-hours* trials, Waffle succeeds LibFuzzer for fuzzing *php*.

In Figure **??**, we can analyze the distribution of the code-coverages of each trial pair. The code coverage of AFL shows lower deviations compared to Waffle.

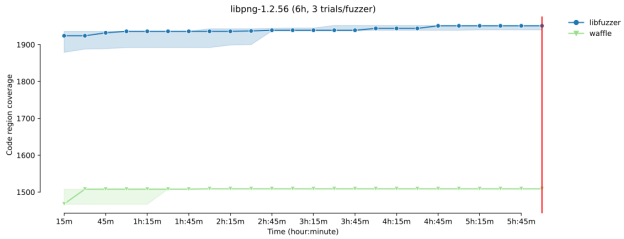
In the end, the number of unique findings of each fuzzer is shown in the last column. Overally speaking, Waffle outperforms AFL in fuzzing *sqlite3*

4.4 Performance Bottlenecks of Waffle

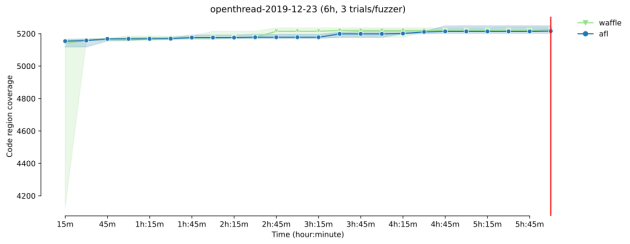
As we explained in Chapter 3, after each execution of the target program, Waffle runs the function `has_new_icnt()` after `has_new_bits()`, for detecting the changes



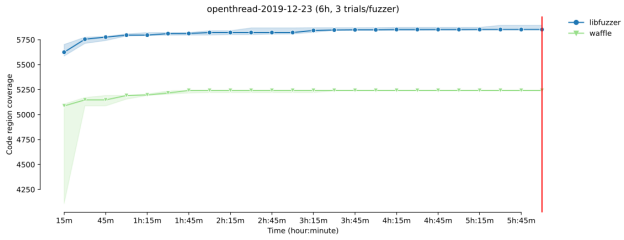
(a) Waffle-AFL libpng-1.2.56



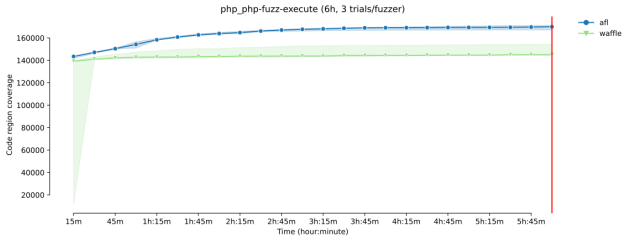
(b) Waffle-LF libpng-1.2.56



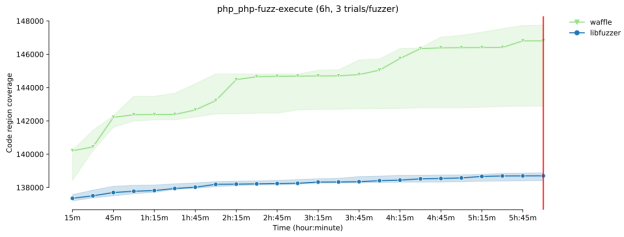
(c) Waffle-AFL openthread-2019-12-23



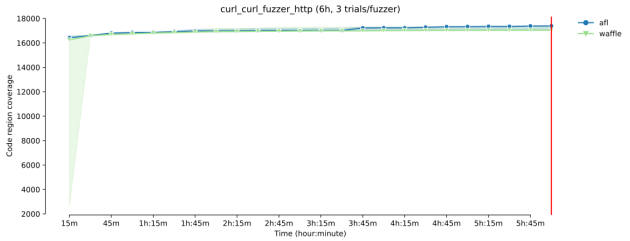
(d) Waffle-LF openthread-2019-12-23



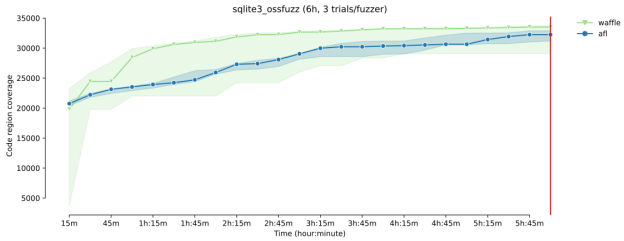
(e) Waffle-AFL php_php-fuzz-execute



(f) Waffle-LF php_php-fuzz-execute



(g) Waffle-AFL curl_curl_fuzzer_http

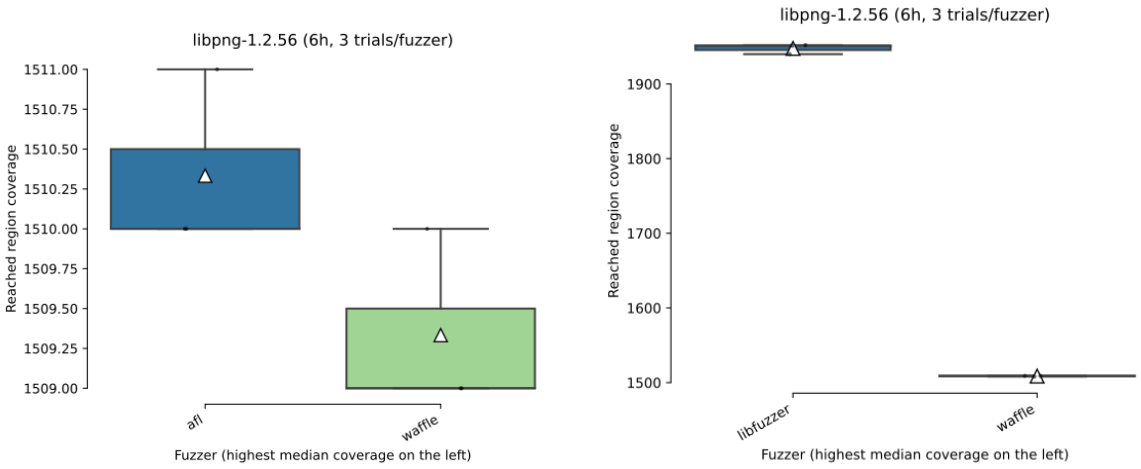


(h) Waffle-AFL sqlite3_ossfuzz

Figure 4.2: Mean code coverage growth over time

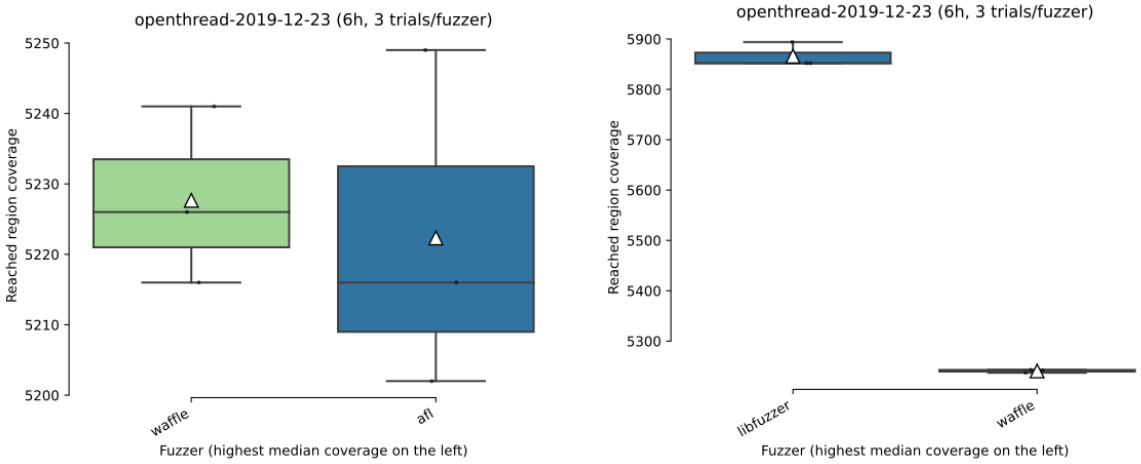
of the instruction counters on each edge. `has_new_icnt()` helps Waffle prioritize the edges that their instruction counters increase faster.

To investigate the performance changes for the insertion of `has_new_icnt()`, we ran this function on a sparse array of 32bit integers. On the other hand, we ran the



(a) Waffle-AFL libpng-1.2.56

(b) Waffle-LF libpng-1.2.56

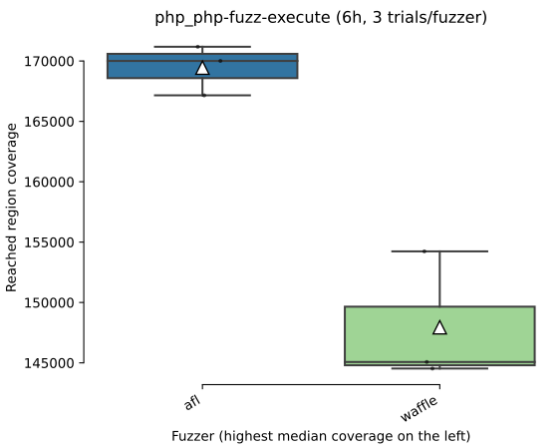


(c) Waffle-AFL openthread-2019-12-23

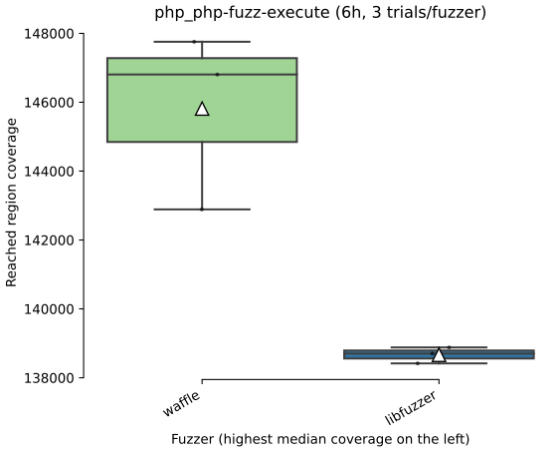
(d) Waffle-LF openthread-2019-12-23

Figure 4.3: Reached code coverage distribution

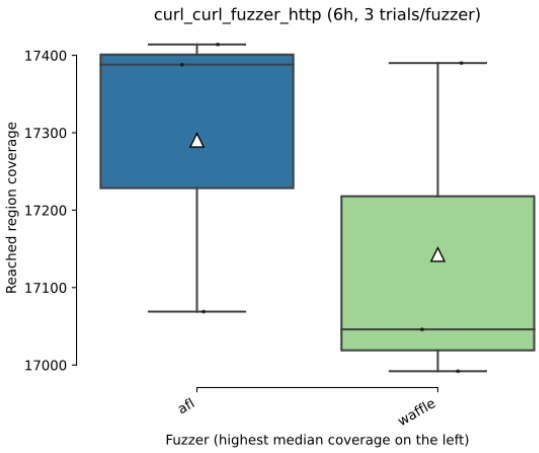
function `has_new_bits()` on a sparse array of 8bit integers, for the same number of iterations. The results showed that the trial of `has_new_icnt()` takes 15000 milliseconds, while the mentioned execution of `has_new_bits()` takes 1500 milliseconds, on the local computer. As a result, the execution of both of the functions consecutively, takes **11x** longer.



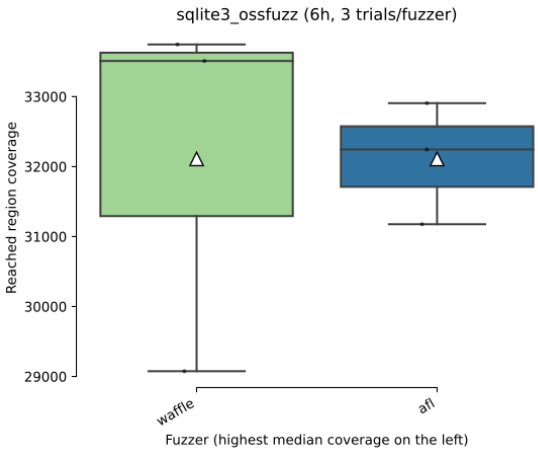
(e) Waffle-AFL php_php-fuzz-execute



(f) Waffle-LF php_php-fuzz-execute



(g) Waffle-AFL curl_curl_fuzzer_http



(h) Waffle-AFL sqlite3_ossfuzz

Figure 4.3: Reached code coverage distribution (cont.)

Chapter 5

Future Works and Conclusions

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] Waffle project. `https://github.com/behnamarbab/memlock-waffle/tree/waffle`, 2021.

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

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

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

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

[9] 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.

[10] 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.

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

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

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

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

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

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

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

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

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

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

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

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

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

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

[50] 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.

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

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

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

[54] K Serebryany. libfuzzer a library for coverage-guided fuzz testing. *LLVM project*, 2015.

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

[56] Local experiment issue - erro[0612]. `https://github.com/google/fuzzbench/issues/946`, 2021.

[57] Requesting an experiment. `https://google.github.io/fuzzbench/getting-started/adding-a-new-fuzzer/#requesting-an-experiment`, 2021.

[58] Setting up gcp. `https://google.github.io/fuzzbench/running-a-cloud-experiment/setting-up-a-google-cloud-project`, 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.

```
1  havoc_stage:
2    /* The havoc stage mutation code is also invoked when splicing
       files; if the splice_cycle variable is set, generate different
       descriptions and such. */
3
4    if (!splice_cycle) {
5      stage_max   = (doing_det ? HAVOC_CYCLES_INIT : HAVOC_CYCLES) *
6                    perf_score / havoc_div / 100;
7    }
8    else {
9      stage_max   = SPLICE_HAVOC * perf_score / havoc_div / 100;
10   }
11
12   /* We essentially just do several thousand runs (depending on
       perf_score) where we take the input file and make random stacked
       tweaks. */
13   for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {
```

```
14      u32 use_stacking = 1 << (1 + UR(HAVOC_STACK_POW2));
15      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  && \
27   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.

```
1  """Integration code for Waffle fuzzer."""
2
3  import json
4  import os
5  import shutil
6  import subprocess
7
8  from fuzzers import utils
9
10
11 def prepare_build_environment():
12     """Set environment variables used to build targets for AFL-based
13     fuzzers."""
14     cflags = ['-fsanitize-coverage=trace-pc-guard']
15     utils.append_flags('CFLAGS', cflags)
16     utils.append_flags('CXXFLAGS', cflags)
17
18     os.environ['CC'] = 'clang'
19     os.environ['CXX'] = 'clang++'
20     os.environ['FUZZER_LIB'] = '/libAFL.a'
21
22
23 def build():
24     """Build benchmark."""
25     prepare_build_environment()
26
27     utils.build_benchmark()
28
29     print('[post_build] Copying waffle-fuzz to $OUT directory')
30     # Copy out the waffle-fuzz binary as a build artifact.
31     shutil.copy('/source_files/waffle/waffle-fuzz', os.environ['OUT'
       ])
32
33
```

```
34 def get_stats(output_corpus, fuzzer_log):  # pylint: disable=unused-
   argument
35     """Gets fuzzer stats for Waffle."""
36     # Get a dictionary containing the stats Waffle reports.
37     stats_file = os.path.join(output_corpus, 'fuzzer_stats')
38     with open(stats_file) as file_handle:
39         stats_file_lines = file_handle.read().splitlines()
40     stats_file_dict = {}
41     for stats_line in stats_file_lines:
42         key, value = stats_line.split(': ')
43         stats_file_dict[key.strip()] = value.strip()
44 
45     # Report to FuzzBench the stats it accepts.
46     stats = {'execs_per_sec': float(stats_file_dict['execs_per_sec'
   ])}
47     return json.dumps(stats)
48 
49 
50 def prepare_fuzz_environment(input_corpus):
51     """Prepare to fuzz with AFL or another AFL-based fuzzer."""
52     # Tell AFL to not use its terminal UI so we get usable logs.
53     os.environ['AFL_NO_UI'] = '1'
54     # Skip AFL's CPU frequency check (fails on Docker).
55     os.environ['AFL_SKIP_CPUFREQ'] = '1'
56     # No need to bind affinity to one core, Docker enforces 1 core
   usage.
57     os.environ['AFL_NO_AFFINITY'] = '1'
58     # AFL will abort on startup if the core pattern sends
   notifications to
59     # external programs. We don't care about this.
60     os.environ['AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES'] = '1'
61     # Don't exit when crashes are found. This can happen when corpus
       from
62     # OSS-Fuzz is used.
63     os.environ['AFL_SKIP_CRASHES'] = '1'
64 
65     # AFL needs at least one non-empty seed to start.
66     utils.create_seed_file_for_empty_corpus(input_corpus)
67 
68 
69 def run_waffle_fuzz(input_corpus,
70                     output_corpus,
71                     target_binary,
72                     additional_flags=None,
73                     hide_output=False):
74     """Run the fuzzer"""
75     # Spawn the waffle fuzzing process.
76     print('[run_waffle_fuzz] Running target with waffle-fuzz')
77     command = ['./waffle-fuzz', '-i', input_corpus, '-o',
   output_corpus, '-d', '-m', 'none', '-t', '1000']
78     if additional_flags:
79         command.extend(additional_flags)
80     dictionary_path = utils.get_dictionary_path(target_binary)
81     if dictionary_path:
```

```
82          command.extend(['-x', dictionary_path])
83      command += [
84          '--',
85          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