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

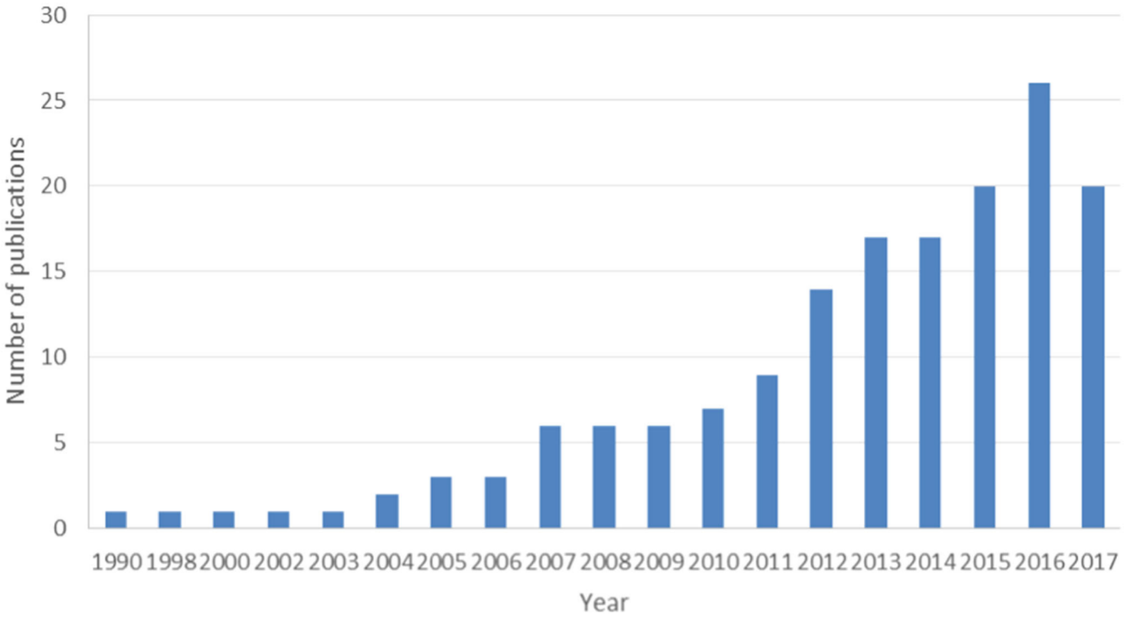


Figure 2.1: Fuzzing papers published between January 1st, 1990 and June 30 2017. [1]

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 exception 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* to the program. Instrumentation is the process of injecting logging instructions into the program, and static instrumentation 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*. In dynamic instrumentation, 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 dynamic instrumentation tools.

In this chapter, we begin with reviewing the previous works that lead to this thesis 2.3. Next, we describe the implementation of AFL and its llvm-mode in 2.4. 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*

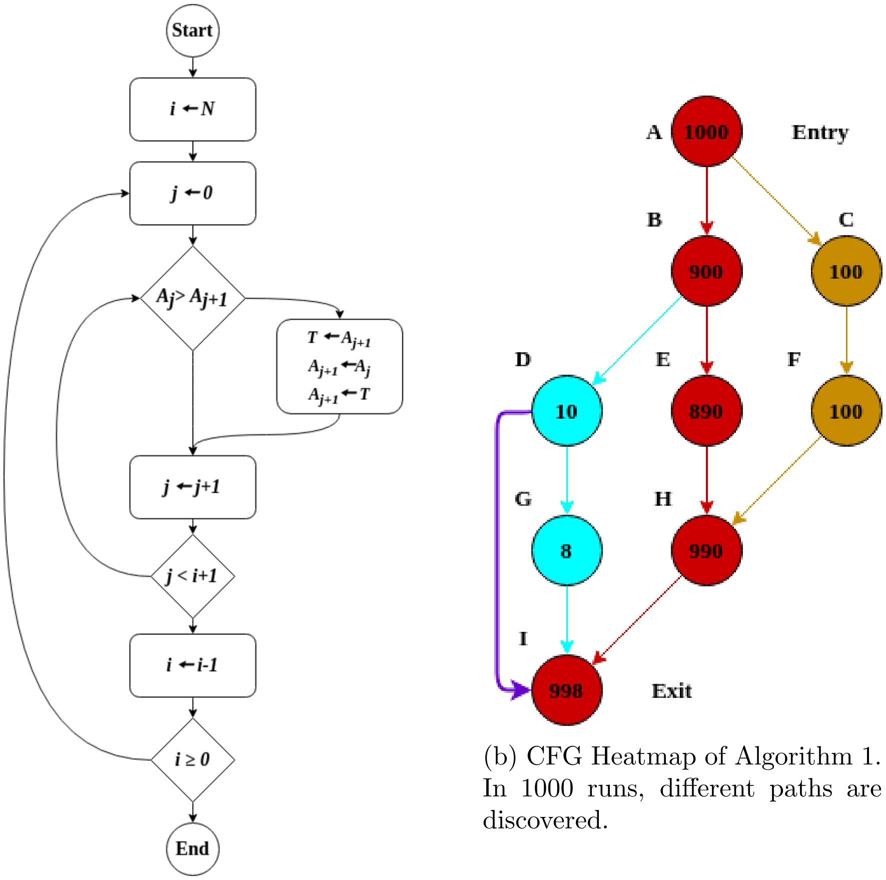
Algorithm 1: Pseudocode of bubblesort on array A of size N

```
Input: $A, N$
$i \leftarrow N$;
do
    $j \leftarrow 0$;
    do
        if $A_j > A_{j+1}$ then
            $SWAP(A_j, A_{j+1})$;
        $j \leftarrow j + 1$;
    while $j < i + 1$;
    $i \leftarrow i - 1$;
while $i >= 0$;
```

and a relevant *jump instruction*.

An execution processes a path (sequence) of instructions from an entry to any exit location of the program. For instance, consider Figure 2.2b as a CFG illustrating the executed paths of 1000 trials of the program's execution. The numbers in the basic blocks indicate the number of times each basic block is visited. A path such as $A \rightarrow B \rightarrow E \rightarrow H \rightarrow I$ has been explored more than other execution paths. Basic block D is visited occasionally and the edge $D \rightarrow I$ directly goes to the Exit, representing bugs in basic block D. These 1000 trials have discovered 9 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



(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

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.

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

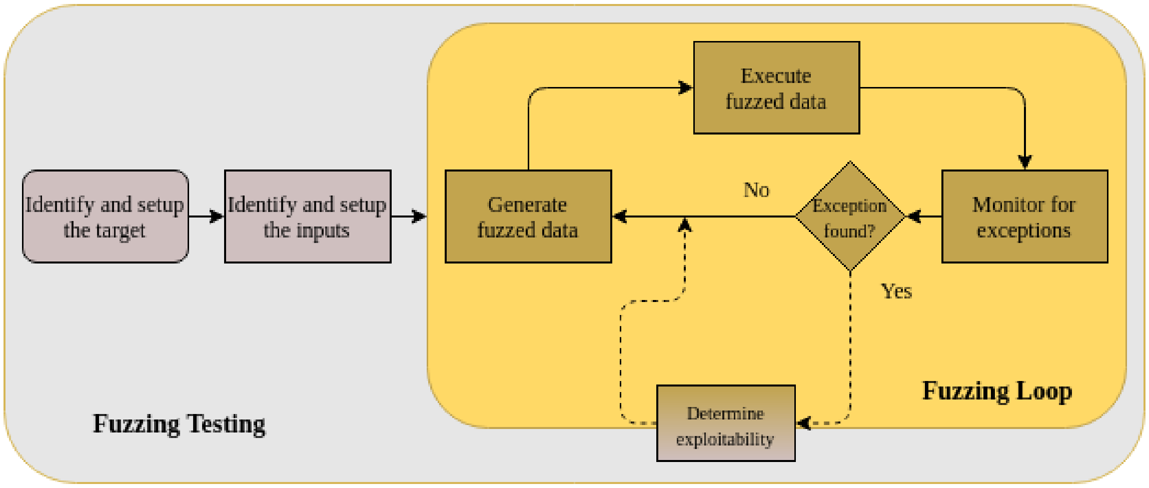


Figure 2.3: Fuzzing phases. Inspired by the definition of Sutton et al. [2]

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 looks for any exceptions returning 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.

```
string magic_string = random_string();
if(magic_string == "M4G!C")
    return succeed();
else
    return failed();
```

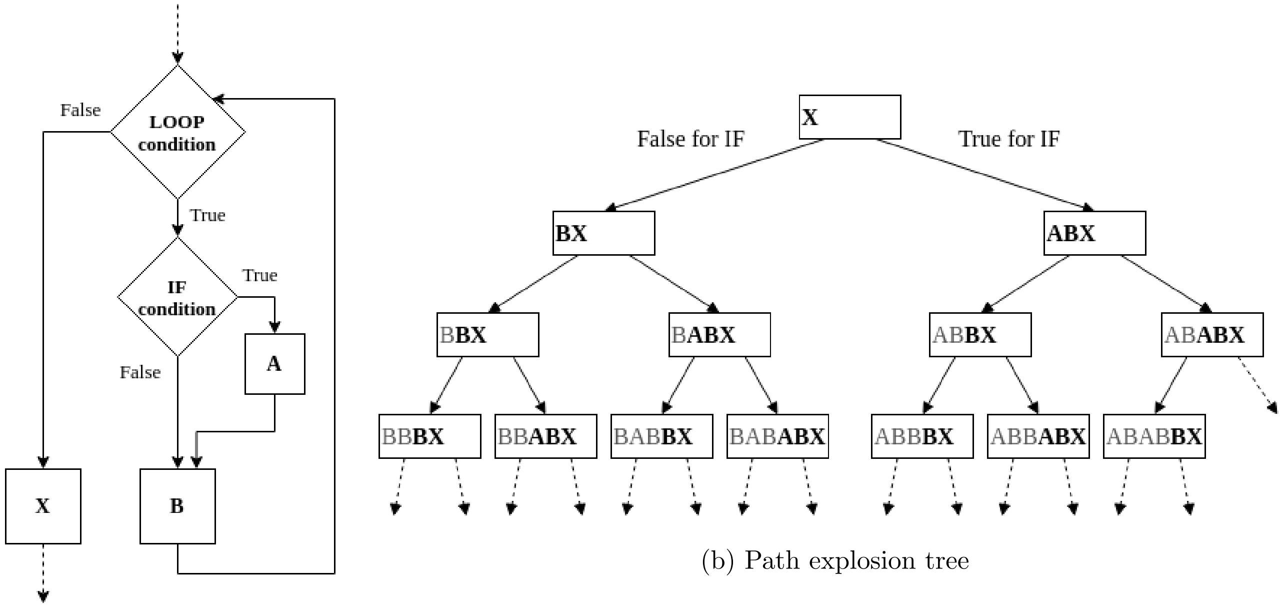
Listing 2.1: Magic Value: `M4G!C` is a magic value

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.



(b) Path explosion tree

(a) Path explosion

Figure 2.4: Path explosion example

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 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

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 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.

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 dynamic instrumentation 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 total execution path. SlowFuzz has

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

an interest in evolving the corpus of test cases to discover new paths or generate more resource exhaustive executions. Another 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 AFL

Michal Zalewski initially developed American Fuzzy Lopper. 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, benefit from the **simplicity** and **chainability** of the fuzzer [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 consistent 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 “allows you to build a standalone feature that leverages the QEMU ”user emulation” mode and allows callers to obtain instrumentation output for black-box, closed-source binaries”, working as a greybox fuzzer [14]. The instrumentation using **QEMU** on a binary has an average performance cost of 2-5x, which is better than other tools such as **DynamoRIO** and **PIN**.

2.3.1 LLVM

“The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. The LLVM project has multiple components. The core of the project is itself called “LLVM.” This project contains all of the tools, libraries, and header files needed to process intermediate representations and converts them into object files.

Tools include an assembler, disassembler, bitcode analyzer, and bitcode optimizer." [12, 48]

LLVM can be used as a compiler framework, separated into "front-end" and "back-end." The front-end contains the lexers and parsers, and it accepts the source code to a program and returns the **intermediate representation (IR)** of the program. The back-end converts the IR into machine language.

For instrumentation, we insert the logging instructions into each basic block of the program in the front-end. **Clang** is part of the LLVM toolchain for compiling C/C++ source code. By definition, "**clang** is a C, C++, and Objective-C compiler that encompasses preprocessing, parsing, optimization, code generation, assembly, and linking."[49] We extend the phases of compilation so that we are injecting the instructions in compilation.

LLVM converts an **IR** of a program into machine language instructions. The structure of the LLVM project is shown in Figure 2.5:

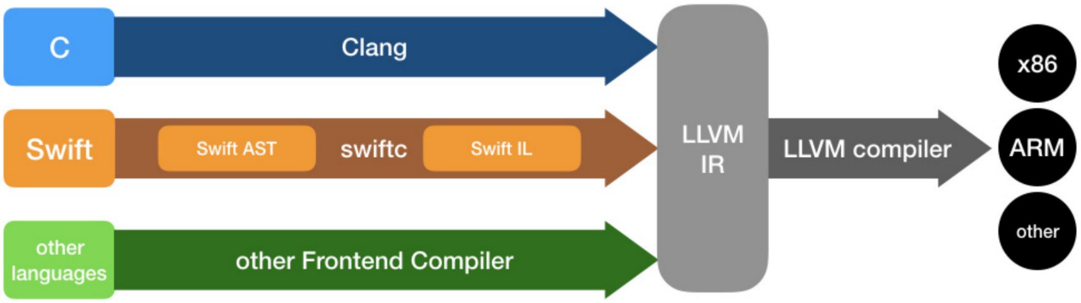


Figure 2.5: LLVM architecture: A front-end compiler generates the LLVM IR, and then it is converted into machine code [3]

The instrumentation is applied before the IR generation, and the LLVM IR is fed into the LLVM compiler to generate the machine-specific instructions. As our instrumentation does not affect the LLVM IR compilation, we will not investigate the generated IR.

2.3.1.1 Instrumentation and coverage measurements

Waffle is based on AFL and we are extending the AFL's instrumentation in our work. The goal of using instrumentation for AFL is to differentiate code coverages. There are two techniques for instrumentation in AFL:

1. *llvm_mode*: AFL takes the source code and an instrumentation recipe and generates the instrumented binary of the target program.
2. *qemu_mode*: AFL leverages the QEMU mode to obtain instrumentation output for closed-source binaries. We don't use this mode in this thesis.

In the LLVM recipe, we instantiate the bitmap and assign it to the shared memory for modifications. The remaining instructions for the recipe will be applied on the basic blocks in **AFLCoverage** module. This module takes effect in compilation of the program before the generation of IR. We can see some of the implementation of this **pass** in Listing 2.2:

```
// LLVM-mode instrumentation pass
bool AFLCoverage::runOnModule(Module &M) {

  /* Instrument all the things! */
  for (auto &F : M)
    for (auto &BB : F) {
      BasicBlock::iterator IP = BB.getFirstInsertionPt();
      IRBuilder<> IRB(&(*IP));

      if (AFL_R(100) >= inst_ratio) continue;

      /* Make up cur_loc */
      unsigned int cur_loc = AFL_R(MAP_SIZE);
      ConstantInt *CurLoc = ConstantInt::get(Int32Ty, cur_loc);

      /* Load prev_loc */
      LoadInst *PrevLoc = IRB.CreateLoad(AFLPrevLoc);
      PrevLoc->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(
   C, None));
      Value *PrevLocCasted = IRB.CreateZExt(PrevLoc, IRB.getInt32Ty
   ());

      /* Load SHM pointer */
      LoadInst *MapPtr = IRB.CreateLoad(AFLMapPtr);
      MapPtr->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C
   , None));
      Value *MapPtrIdx =
          IRB.CreateGEP(MapPtr, IRB.CreateXor(PrevLocCasted, CurLoc)
   );

      /* Update bitmap */
      LoadInst *Counter = IRB.CreateLoad(MapPtrIdx);
      Counter->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(
   C, None));
      Value *Incr = IRB.CreateAdd(Counter, ConstantInt::get(Int8Ty,
   1));
      IRB.CreateStore(Incr, MapPtrIdx)
          ->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C,
   None));

      /* Set prev_loc to cur_loc >> 1 */
      StoreInst *Store =
          IRB.CreateStore(ConstantInt::get(Int32Ty, cur_loc >> 1),
   AFLPrevLoc);
      Store->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C,
    None));

      inst_blocks++;
    }

  return true;
}
```

Listing 2.2: AFLCoverage module

The recipe for instrumentation fills out the coverage bitmap with the hash values of the paths executed. The instructions are as followed:

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

Listing 2.3: Select element and update in shared_mem

AFL instruments by adding these instructions into basic blocks. First, a random value is assigned to *curr_location*. Next, it is XORed with the previous location's value, *prev_location*, and the resulting value is the location on *shared_mem*, the *coverage bitmap*, which is incremented by one. The third and final instruction is reseting the *prev_location* to a new value.

When AFL runs the instrumented program, every time an instrumented basic block is executed, a dedicated location of *shared_mem* in the bitmap is incremented. This algorithm recognizes the different paths that AFL runs through. For instance, in figure 2.6, suppose that we have an instrumented program with the random values which is set in compile time. An execution that walks over basic blocks $1 \rightarrow 2 \rightarrow 5$ will increase the value of the corresponding locations by 1; for instance, an increment on $\mathit{shared\_mem}[14287 \oplus 23765]$ is applied for the transition of $1 \rightarrow 2$ and $\mathit{shared\_mem}[7143 \oplus 21689]$ for $2 \rightarrow 5$. We can see that the paths $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ and $1 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 4 \rightarrow 5$ (which contains a loop), set different locations on bitmap.

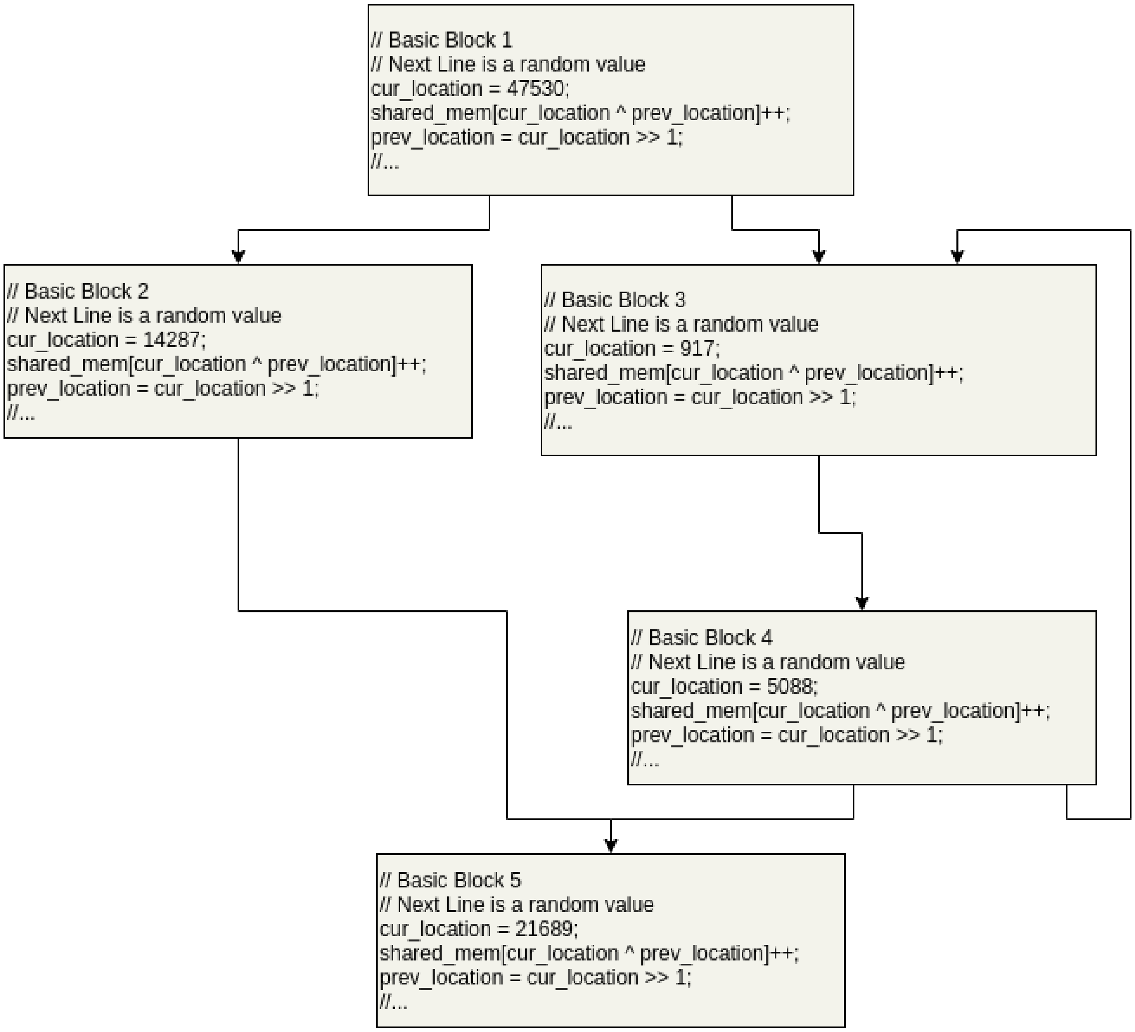
AFL uses this coverage feature for discovering new inputs with new code coverages.

Visitor functions

"Instruction visitors are used when you want to perform different actions for different kinds of instructions without having to use lots of casts and a big switch statement (in your code, that is). [50]"

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

struct CountAllVisitor : public InstVisitor<CountAllVisitor> {
  unsigned Count;
  CountAllVisitor() : Count(0) {}
  // Any visited instruction is counted in a specified range
  void visitInstruction(Instruction &I) {
```



```
// Basic Block 1
// Next Line is a random value
cur_location = 47530;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
//...
```

```
// Basic Block 2
// Next Line is a random value
cur_location = 14287;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
//...
```

```
// Basic Block 3
// Next Line is a random value
cur_location = 917;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
//...
```

```
// Basic Block 4
// Next Line is a random value
cur_location = 5088;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
//...
```

```
// Basic Block 5
// Next Line is a random value
cur_location = 21689;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
//...
```

Figure 2.6: Example for instrumented basic blocks

```
    ++Count;
  }
};
```

Listing 2.4: Visitors example

The specified range can be any two iterators, which can be a Module, Function, BasicBlock, Instruction or any other range between two instruction addresses.

2.3.2 AFL Fuzz

afl-fuzz.c has the instructions for fuzzing the target instrumented-program. The Algorithm 2 illustrates a brief pseudocode of the execution of *afl-fuzz*:

```
Algorithm 2: afl-fuzz
Input: $in\_dir$, $out\_dir$, $instrumented\ Target$
initialize fuzzer;
while fuzzing is not terminated do
    cull queue and update bitmaps;
    $Entry \leftarrow q.first\_entry()$;
    $fuzz\_one(Entry)$;
```

After the environmental initializations, the fuzzing loop continues until receiving a termination signal. In every iteration of the loop, AFL first culls the corpus of the generated entries. This method assigns a **favor-factor** (Eq 2.1) to each queue_entry and marks the **favorite entries**, as they execute faster and the size of the files are smaller than the rest of the corpus. AFL finds a favorable path for “having a minimal set of paths that trigger all the bits seen in the bitmap so far, and focus on fuzzing them at the expense of the rest.” [11]

$$fav\_factor = e.exec\_time \times e.length \tag{2.1}$$

An entry is selected after culling the queue. AFL evolves the input corpus by generating new entries out of the selected input entry - `fuzz_one()`.

2.3.2.1 fuzz_one()

Algorithm 3: *$fuzz\_one$*: Fuzz one Entry

```
Input: $queueEntry$
$T \leftarrow Entry.test\_case$;
$calibrate(T)$;
$trim(T)$;
$perf\_score \leftarrow calculate\_score(T)$;
if $deterministic\_mode$ then
    $deterministic\_stages(T)$;
$MX \leftarrow calc\_stages(perf\_score, \mathcal{C})$;
foreach $element$ in $(0, MX)$ do
    $random\_havoc(T)$;
```

Fuzzing a single entry requires the calibration of the test-cases; calibration helps AFL in evaluating the stats of the current entry. This evaluation is stored in `perf_score`, and the number of trials for generating new inputs from the **random_havoc** stage is calculated using the `perf_score`. AFL, as a coverage-based fuzzer, assigns higher performance score for the entries that are executed faster and have bigger bitmaps.

The evolutionary algorithms for generating new entries are applied in two stages: **deterministic** and **random_havoc** stages. As shown in Algorithm 3, AFL initially tries the basic, deterministic algorithms. These algorithms are executed for a specific number of times, in the same order, and once for each fuzzing trial. . Bit-flipping, byte-flipping, simple arithmetic operations, using known integers and values from dictionaries, are a sequence of mutations that AFL applies on an entry in **deterministic** stage. Each one of the above operations tweak a small portion of the fuzzed input, and does not modify the file in large portions - up to 32 bits changes in each tweak.

The **havoc** stage is a cycle of stacked random tweaks. AFL assesses the current entry to insert into the queue. Each mutation is selected randomly and with a higher `perf_score`, this stage continues more fuzzing over the current entry. The `random_havoc` stage consists techniques such as bit flips, overwriting with random and interesting integers, block deletion, block duplication, and (if supplied) assorted dictionary-related operations [51]. An abstract implementation of the `havoc_stage` can be found in Appendix 6.A.1.

2.3.2.2 calculate_score()

This function calculates how much AFL desires to iterate in havoc stage, for the current entry. By default, AFL is interested in fuzzing an input with less execution-time, and simultaneously, showing more coverage, and it's generation depth is higher. The depth of a child entry is one more than the depth of it's parent. For more information, check the following abstraction of the function [Listing 2.5]:

```
/* Calculate case desirability score to adjust the length of havoc
   fuzzing. A helper function for fuzz_one(). Maybe some of these
   constants should go into config.h. */

static u32 calculate_score(struct queue_entry* q) {
  u32 perf_score = 100;

  /* Adjust score based on execution speed of this path, compared to
     the global average. Multiplier ranges from 0.1x to 3x. Fast
     inputs are less expensive to fuzz, so we're giving them more air
     time. */

  if (q->exec_us * 0.1 > avg_exec_us) perf_score = 10;
  else if (q->exec_us * 4 < avg_exec_us) perf_score = 300;
  // Check other conditions in between

  /* Adjust score based on bitmap size. The working theory is that
     better coverage translates to better targets. Multiplier from
     0.25x to 3x. */
  if (q->bitmap_size * 0.3 > avg_bitmap_size) perf_score *= 3;
  else if (q->bitmap_size * 3 < avg_bitmap_size) perf_score *= 0.25;
  // Check other bitmap_sizes in between

  /* Adjust score based on handicap. Handicap is proportional to how
     late in the game we learned about this path. Latecomers are
     allowed to run for a bit longer until they catch up with the
     rest. */

  /* Final adjustment based on input depth, under the assumption
     that fuzzing deeper test cases is more likely to reveal stuff
     that can't be discovered with traditional fuzzers. */

  switch (q->depth) {
    case 0 ... 3:   break;
    case 14 ... 25: perf_score *= 4; break;
    // Check other cases in between
    default:        perf_score *= 5;
  }

  /* Make sure that we don't go over limit. */
  if (perf_score > HAVOC_MAX_MULT * 100) perf_score = HAVOC_MAX_MULT
    * 100;

  return perf_score;
}
```

Listing 2.5: An abstract implementation of calculate_score()

2.3.2.3 common_fuzz_stuff()

The newly fuzzed (generated) inputs must pass `common_fuzz_stuff()` for validating and instantiating a `queue_entry`. The validation checks the length of the fuzzed file, executes the program and keeps the exit-value of the execution. In the end, AFL calls `save_if_interesting()` to insert the entry into the queue, if it is interesting. The function `has_new_bits()` considers how interesting an entry is.

```
/* 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(). */

EXP_ST u8 common_fuzz_stuff(char** argv, u8* out_buf, u32 len) {
  // Validate the file
  // ...
  // Validate the execution
  fault = run_target(argv, exec_tmout);
  // If the file is "interesting", add it into queue
  queued_discovered += save_if_interesting(argv, out_buf, len, fault
     );
  // ...
  return 0;
}

```

```c
/* Check if the result of an execve() during routine fuzzing is interesting, save or queue the input test case for further analysis if so. Returns 1 if entry is saved, 0 otherwise. */

static u8 save_if_interesting(char** argv, void* mem, u32 len, u8 fault) {
  if (fault == crash_mode) {
      hnb = has_new_bits(virgin_bits);
  }
  if(!hnb) return 0;

  queue_top->exec_cksum = hash32(trace_bits, MAP_SIZE, HASH_CONST);

  /* Try to calibrate inline; this also calls update_bitmap_score() when successful. */
  res = calibrate_case(argv, queue_top, mem, queue_cycle - 1, 0);

  // Save the file and return
}
```

Listing 2.6: An abstract implementation of common_fuzz_stuff()

2.3.2.4 has_new_bits()

AFL calls this function after each execution of the program, and the optimization of this code participates an important role.

```c
/* Check if the current execution path brings anything new to the table. Update virgin bits to reflect the finds. Returns 1 if the only change is the hit-count for a particular tuple; 2 if there are new tuples seen. Updates the map, so subsequent calls will always return 0.

   This function is called after every exec() on a fairly large buffer, so it needs to be fast. We do this in 32-bit and 64-bit flavors. */

static inline u8 has_new_bits(u8* virgin_map) {
  u64* current = (u64*)trace_bits;
  u64* virgin  = (u64*)virgin_map;
  u32  i = (MAP_SIZE >> 3);
  while (i--) {

  /* Optimize for (*current & *virgin) == 0 - i.e., no bits in current bitmap that have not been already cleared from the virgin map - since this will almost always be the case. */

    if (unlikely(*current) && unlikely(*current & *virgin)) {
      if (likely(ret < 2)) {
        u8* cur = (u8*)current;
        u8* vir = (u8*)virgin;

        /* Looks like we have not found any new bytes yet; see if any non-zero bytes in current[] are pristine in virgin[]. */
        if ((cur[0] && vir[0] == 0xff) || (cur[1] && vir[1] == 0xff) ||
            (cur[2] && vir[2] == 0xff) || (cur[3] && vir[3] == 0xff) ||
            (cur[4] && vir[4] == 0xff) || (cur[5] && vir[5] == 0xff) ||
            (cur[6] && vir[6] == 0xff) || (cur[7] && vir[7] == 0xff)) ret = 2;
        else ret = 1;
      }
      *virgin &= ~*current;
    }
    current++; virgin++;
  }

  if (ret && virgin_map == virgin_bits) bitmap_changed = 1;
  return ret;
}
```

Listing 2.7: The implementation of has_new_bits()

2.3.3 Status screen

The **status screen** is a UI for the status of the fuzzing procedure. As it is shown in Figure 2.7, there are multiple stats provided in real-time updates:

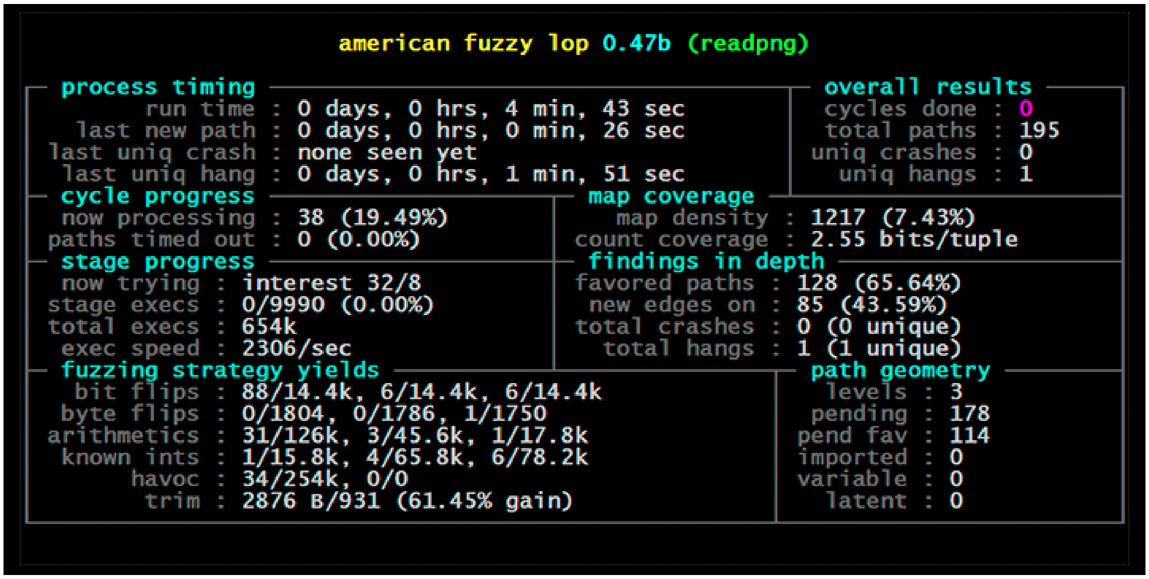
1. **Process timing**: This section tells about how long the fuzzing process is running.
2. **Overall results**: A simplified information about the progress of AFL in finding paths, hangs, and crashes.
3. **Cycle progress**: As mentioned before, AFL takes one input and repeats mutating it for a while. This section shows the information about the current cycle that the fuzzer is working on.

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

6. **Findings in depth**: The crashes and hangs and any other findings (here we have the other information about the coverage) are presented in this section.



```
                      american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec       │  cycles done : 0  │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec       │  total paths : 195│
│ last uniq crash : none seen yet                      │ uniq crashes : 0  │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec       │   uniq hangs : 1  │
├─ cycle progress ──────────────────┬─ map coverage ─┴───────────────────┤
│  now processing : 38 (19.49%)     │    map density : 1217 (7.43%)      │
│ paths timed out : 0 (0.00%)       │ count coverage : 2.55 bits/tuple   │
├─ stage progress ──────────────────┼─ findings in depth ────────────────┤
│  now trying : interest 32/8       │ favored paths : 128 (65.64%)       │
│ stage execs : 0/9990 (0.00%)      │  new edges on : 85 (43.59%)        │
│ total execs : 654k                │ total crashes : 0 (0 unique)       │
│  exec speed : 2306/sec            │   total hangs : 1 (1 unique)       │
├─ fuzzing strategy yields ─────────┴─────────────┬─ path geometry ──────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k        │    levels : 3        │
│  byte flips : 0/1804, 0/1786, 1/1750            │   pending : 178      │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k         │  pend fav : 114      │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k         │  imported : 0        │
│       havoc : 34/254k, 0/0                      │  variable : 0        │
│        trim : 2876 B/931 (61.45% gain)          │    latent : 0        │
└─────────────────────────────────────────────────┴──────────────────────┘
```

Figure 2.7: AFL status screen

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

8. **Path geometry**: The information about the inputs and their depths, which says how many generations of different paths were produced in the process. For instance, we call the seeds we provided for fuzzing the ”level 1” inputs. Next, a new set of inputs is generated as ”level 2”, the inputs derived from ”level 2” are ”level 3,” and so on.

2.3.4 Start Fuzz Testing

AFL requires the instrumented binary for execution. To start the instrumentation, AFL uses *afl-clang*, which is built with the coverage recipe included. The following command instruments the sample program 2.10:

```
  afl-clang sample_vul.c -o sample_vul_i
```

Listing 2.8: Instrument *sample_vul*.c

Now AFL can run this program in *afl-fuzz* with the coverage instrumentations.

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

Listing 2.9: Execute AFL

The fuzzing continues until the fuzz testing is stopped using a termination signal. Pressing *Ctrl+C* is a common command for this purpose. All of the recorded information are accessed through the output directory *out_dir*.

2.4 Performance of AFL

To evaluate the performance of a fuzzer and assess the execution of it, the sample **C** code is implemented. (Listing 2.10) [52]

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void vul_function(char * msg) {
    char *ptr = NULL;
    ptr = malloc(strlen(msg));
    strcpy(ptr, msg);
    printf("%s\n", ptr);
}

int main(int argc, char *argv[])
{
    if(argc<2) {
        printf("At least one input is needed");
        return 1;
    }

    vul_function(argv[1]);

    return 0;
}
```

Listing 2.10: Sample vulnerable program

The sample program has a **Heap buffer overflow vulnerability**. If we compile this code with GCC and debugging flags, the vulnerability stays hidden and the program executes without any errors:

```
$ gcc sample_vul.c -o sample_vul -Wall -Werror -g
$ ./sample_vul hello_world
hello_world
```

Listing 2.11: Compile the sample program

One way to detect the prior vulnerability, is to add AddressSanitizer flag for the compilation [53]. ASan is a fast memory error detector. This tool uses memory poisoning for the detection of a heap buffer overflow. (You can find more features of this tool in the reference) [54]

After we provide the ASan flag for the compilation, we face an error with the same input as the previous example (Listing 2.4):

```
$ gcc sample_vul.c -o sample_vul_asan -Wall -Werror -g -fsanitize=address
$ ./sample_vul_asan hello_world
=================================================================
==304989==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000001b at
     pc 0x7f04ce49215d bp 0x7fff3c893100 sp 0x7fff3c8928a8
WRITE of size 12 at 0x60200000001b thread T0
  ...
  ...
SUMMARY: AddressSanitizer: heap-buffer-overflow (/lib/x86_64-linux-gnu/libasan.so
    .5+0x9c15c)
  ...
  ...
==304989==ABORTING
```

The detection of the vulnerabilities or any other exceptions, signals the operating system for a misbehaviour from the program. A fuzzer would need this signal to evaluate the execution of the target program.

2.5 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 Introduction

Coverage-based fuzzers investigate the inputs with different code coverages. A coverage-based fuzzer such as AFL spreads the domain of the test cases and saves the inputs with new code coverage, so that vulnerabilities would appear after fuzzing the interesting inputs. On the other hand, AFL scans the queue of test cases and prioritizes the inputs with less execution time and shorter length 2.1. This strategy leads to faster generation of the test cases and chooses a faster test case over a slower one. This makes AFL show interest in generating crashes over hangs.

Performance fuzzing with tools such as SlowFuzz [44], MemLock [45], and PerfFuzz [9] have showed great enhancement in finding the pathological inputs. SlowFuzz is a LibFuzzer-based performance fuzzer that counts the total number of executed instructions of a specific family type - CMP. Adding a new type of instruction to the list of monitored instructions is cumbersome, and there are no tools for collecting the instructions info from the current version of SlowFuzz. PerfFuzz is based on AFL and counts the total number of all instructions in an execution. In memlock, the fuzzer collects the heap/stack-related memory instructions. PerfFuzz cannot count the memory instructions separately, and MemLock is not capable of considering other types of instructions.

To untangle the above issues, we propose **Waffle** (**W**hat **A**n **A**mazing **AFL** - WAAAFL). Waffle is a coverage-guided performance fuzzer, that is developed on top of AFL. In addition to the coverage strategies of AFL, Waffle uses an instrumentation method to **count the instructions**. Waffle can target **any custom type of instructions** that is defined using appropriate LLVM libraries. Compared to AFL, Waffle exercises test cases with higher number of instructions, and as a result, generated test cases are run slower. The executions reaching a predefined timeout are revealed as hangs and the corresponding input file is stored as an evidence for a vulnerability.

In this chapter, we introduce Waffle in more details. In the next section, we explain the implementation of Waffle, as well as the techniques we used for the purpose of this research. Section **REF-REQUIRED** concentrates on the applications of Waffle, and we conclude in the last chapter.

3.2 Motivating example

The number of effective instructions can affect on the time complexity of a program. To exemplify our problem, we pick a program that has a variety of different execution-times, based on the inputs we provide for the program.

Quicksort [55] is a well-known fast algorithm for sorting a list of numbers. This divide-and-conquer algorithm selects a pivot and finds the position of the pivot on the list. After the selection, the other numbers of the list are swapped, until all the numbers that are less than the pivot are on one side, and the rest are on the other side of the pivot. Then A quicksort is called on each side, and we continue until there is no more unknown position for the numbers in the final sorted list.

This algorithm has a best-case scenario with $\mathcal{O}(n \log n)$ for the time complexity, and the worst-case occurs happens in $\mathcal{O}(n^2)$. The worst scenario is when we select the pivot and all other numbers are not swapped; as a result, we have to try the remaining elements of the list before the selection of the next pivot. The best-case scenario occurs when the pivot splits the list into two partitions that the difference between the length of the partitions is less than or equal to one. The average time complexity is $\mathcal{O}(n \log n)$.

The following code is an implementation of **quicksort** in C language:

```c
#include<stdio.h>

void swap(int* a, int* b) {
  int t = *a; *a = *b; *b = t;
}

int partition (int arr[], int low, int high) {
  int pivot = arr[high];
  int i = (low - 1); // Index of smaller element
  for (int j = low; j <= high- 1; j++)
    if (arr[j] <= pivot) {
      i++; // increment index of smaller element
      swap(&arr[i], &arr[j]);
    }

  swap(&arr[i + 1], &arr[high]);
  return (i + 1);
}

void quickSort(int arr[], int low, int high){
  if (low < high) {
    int pi = partition(arr, low, high);

    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}

void printArray(int arr[], int size) {
  int i;
  for (i=0; i < size; i++)
    printf("%d ", arr[i]);
  printf("\n");
}

// Driver program to test above functions
int main() {
  int arr[] = {10, 7, 8, 9, 1, 5};
  int n = sizeof(arr)/sizeof(arr[0]);
  quickSort(arr, 0, n-1);
  printf("Sorted array: \n");
  // printArray(arr, n);
  return 0;
}
```

Listing 3.1: Quicksort

We will consider testing the above code with Waffle in the next chapter.

3.3 Instrumentation

waffle-llvm-rt.o.c

We initialize the instrumentation with setting up the shared memory for Waffle (Listing 3.2):

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

u8  __wafl_icnt_initial[ICNT_SIZE];
u8* __wafl_area_ptr = __wafl_icnt_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_icnt_ptr);
  }
}
```

Listing 3.2: LLVM instrumentation bootstrap

__wafl_area_ptr is the region that is allocated for counting the instructions, and is later shared when the instrumented program is running in fuzz testing.

The size of the bitmap __wafl_icnt_ptr is equal to $ICNT\_SIZE = 2^{16}$; the size of the bitmaps are equal in both AFL and Waffle.

wafl-llvm-pass.so.cc

Next, we inject our instrumentation into the program. As mentioned in the previous chapter, this stage requires the LLVM modules to analyze and insert the instructions. First, we define the **visitors**:

```
struct CountAllVisitor : public InstVisitor<CountAllVisitor> {
  unsigned Count;
  CountAllVisitor() : Count(0) {}
  void visitMemCpyInst(MemCpyInst &I) { ++Count;}
};
```

We count the number of memory copies in an execution. The CountAllVisitor structure keeps track of the instructions that LLVM considers them as memory copies.

The highest number of instructions is a goal that Waffle follows, so that the new generations of the inputs are executed with more instructions. The hit-counts in the shared array starting from __wafl_area_ptr, are the number of times an edge is visited. Each time we visit an edge, we add the counted instructions to the appropriate index. The content of the array tracks the impact of edges in increasing the total number of instructions, and Waffle leverages these values to measures the influence of the changes in each index. [next section: Fuzzing]

The length of the shared memories are constant, but each time Waffle fuzzes a test-case, it needs to aggregate the content of the array and find the total number of instructions. Instead of calculating this total number in the fuzzing procedure, Waffle stores the total number of instructions in another shared memory region, `sys_data` [Listing 3.3]. Whenever the function `instr_AddInsts()` is called, the `MaxInstCount` is increased by the provided value. Later, when the program is finishing its execution, the shared memory containing the `MaxInstCount` is updated.

```c
struct sys_data {
  int MaxInstCount;
};

static int MaxInstCount = 0;

void __attribute__((destructor)) traceEnd(void) {
  unsigned char *mem_str = getenv(MEM_ENV_VAR);

  if (mem_str) {
    unsigned int shm_mem_id = atoi(mem_str);
    struct sys_data *da;

    da = shmat(shm_mem_id, NULL, 0);
    if (da == (void *)-1) _exit(1);

    da->MaxInstCount = MaxInstCount;
  }
}

void instr_AddInsts(int cnt) {
  MaxInstCount += cnt;
}
```

Listing 3.3: sys_data in instrumentation

Now we can insert our instructions to the basicblocks:

```cpp
// snippet of wafl-llvm-pass.so.cc
#include <math.h>
// ...
bool WAFLCoverage::runOnModule(Module &M) {
  // ...
  GlobalVariable *WAFLMapPtr =
    new GlobalVariable(M, PointerType::get(Int8Ty, 0), false,
    GlobalValue::ExternalLinkage, 0, "__wafl_area_ptr");

  llvm::FunctionType *icntIncrement =
    llvm::FunctionType::get(builder.getVoidTy(), icnt_args, false);
  llvm::Function *icnt_Increment =
    llvm::Function::Create(icntIncrement,
      llvm::Function::ExternalLinkage, "instr_AddInsts", &M);
  // ...
  for (auto &F : M) {
```

```
17     for (auto &BB : F) {
18       // ...
19       LoadInst *IcntPtr = IRB.CreateLoad(WAFLMapPtr);
20       MapPtr->setMetadata(M.getMDKindID("nosanitize"),
21         MDNode::get(C, None));
22 
23       Value* EdgeId = IRB.CreateXor(PrevLocCasted, CurLoc);
24       Value *IcntPtrIdx =
25           IRB.CreateGEP(IcntPtr, EdgeId);
26 
27       /* Count the instructions */
28       CountAllVisitor CAV;
29       CAV.visit(BB);
30 
31       /* Setup the counter for storage */
32       u8 log_count = (u8) log2(CAV.Count);
33       Value *CNT = IRB.getInt8(log_count);
34 
35       LoadInst *IcntLoad = IRB.CreateLoad(IcntPtrIdx);
36       Value *IcntIncr = IRB.CreateAdd(IcntLoad, CNT);
37 
38       IRB.CreateStore(IcntIncr, IcntPtrIdx)
39         ->setMetadata(M.getMDKindID("nosanitize"),
40         MDNode::get(C, None));
41 
42       IRB.CreateCall(icnt_Increment, ArrayRef<Value*>({ CNT }));
43 
44       inst_blocks++;
45     }
46   }
47 }
```

Listing 3.4: LLVM-mode instrumentation pass

In Line 6, we locate the shared bitmap. Line 19 loads the pointer to the bitmap and configures the meta-data for storage [56].

Same as AFL, Waffle stores the counters in the **hashed value** of the path we explored (Listing 2.3). The usage of the coverage-guided hashed values, helps Waffle in collecting the coverage information, and at the same time, maximizing the number of instructions in executions. Instructions in lines 23 to 25, load the pointer to the appropriate location on the bitmap.

In each basicblock, the visitors look for the **memory copies**. In different executions, we could see that the number of instructions increases rapidly, and to control this number, we calculate its log value (Lines 32-33):

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

Now that we have calculated `CNT`, we load the pointer on the bitmap and add `CNT` to the content of the pointer and replaces it with the new summation. (Lines 36-39)

The last step in instrumenting a basic-block, is to call the `instr_AddInsts`. The command in line 42 calls the function with the previously set pointer, and sends `CNT` as the argument to this function.

Waffle applies these instrumentations in the compile-time, and when the program is being run, the performance and coverage information are set.

Run the instrumentation

To compile the target program with the adjusted instrumentation, we replace the C/C++ compilers (clang/clang++) with `waffle-clang`. “This program is a drop-in replacement for clang, similar in most respects to ../afl-gcc. It tries to figure out compilation mode, adds a bunch of flags, and then calls the real compiler.” [57] Except refactoring the filenames and the name of the variables, we did not apply any other modifications on `waffle-clang.c`.

Same as 2.11, we can start instrumentation by the command below:

```
./waffle-clang sample_vul.c -o sample_vul_waffle
```

Notice that we can insert compilation flags, such as `-fsanitize=address`, to enhance the performance of Waffle.

3.4 Fuzzing

Waffle extends the instrumentation and the fuzzing procedure of AFL. The instrumented binary from the previous stage is given to the *waffle-fuzz*, and Waffle develops the new features in *waffle-fuzz.c*.

To analyze the implementation of Waffle, we merge the algorithm 2 and algorithm 3:

Algorithm 4: $waffle - fuzz$

```
Input: $queueEntry$
initialize fuzzer;
while fuzzing is not terminated do
    cull_queue();
    $T \leftarrow q.first\_entry()$;
    $calibrate(T)$;
    $trim(T)$;
    $perf\_score \leftarrow calculate\_score(T)$;
    if deterministic_mode then
        $deterministic\_stages(T)$;
    $MX \leftarrow calc\_stages(perf\_score, \mathcal{C})$;
    foreach element in $(0, MX)$ do
        $random\_havoc(T)$;
```

The lines in red are the instructions that were modified in Waffle.

Calibration

Each entry of the queue is calibrated in order to collect the execution stats. Besides the execution-time and the bitmap-coverage information, Waffle runs the target program and collects the added **instruction counters**. Each instruction counter is monitored in the shared_memory, `icnt_bits[]`, and `top_rated_icnt[]` tracks the changes on `icnt_bits[]` in an `struct` of type `queue_entries`, .

Waffle collects the instrumentation's data after running the binary. The values of `icnt_bits` and `trace_bits` track the performance and the coverage-instrumentation. As described in Section Instrumentation, `sys_data` stores the sum of the instruction counters.

AFL checks the uniqueness of an execution by keeping a hashed value of `trace_bits[]`, representing a unique ID for a path-coverage. Within the discovery of a new coverage, Waffle analyzes both the *coverage* array and *instruction counters* to process the new hit-counts:

- `has_new_bits()`: AFL keeps the hit-counts of `trace_bits[]`, for identifying the new edges appeared in the execution of the current test-case. `trace_bits[]` is a sparse array, and AFL tracks the modified indices for better performance.

- `has_new_icnt()`: In addition to the previous method, Waffle searches for the highest values in `icnt_bits[]`. Each cell in `icnt_bits` contains 4 bytes of information. We saw in Listing 3.4 that in an index, such as `i`, `icnt_bits[i]` is added with `CNT`, while the increments in `trace_bits[]` are always by one. An analysis on `./sample_vul_instr` showed that the average value for `CNT`s is more than 3. As a result, the variance for the values in `icnt_bits[]` is higher than `trace_bits[]`'s. To decrease this variance and enhance the performance of Waffle, it leverages a constant float number, `MAX_CNT_MULT`:

```
static inline u8 has_new_icnt() {
  // #define MAX_CNT_MULT 1.05
  int ret = 0;
  for (int i = 0; i < ICNT_SIZE; i++) {
    if (unlikely(icnt_bits[i]) && unlikely(icnt_bits[i] >
   MAX_CNT_MULT*max_icnts[i])) {
      if(likely(ret<2)) {
        ret = 2;
        max_icnts[i] = icnt_bits[i];
      }
      else ret = 1;
    }
  }
  return ret;
}
```

Listing 3.5: has_new_icnt()

By default, Waffle sets $MAX\_CNT\_MULT = 1.05$, which means it expects at least 5% increase for `icnt_bits[i]` before updating the content of index i.

The length of both `icnt_bits[]` and `trace_bits[]` is the same, but `icnt_bits` consumes 4x more of the memory:

```
#define MAP_SIZE_POW2     16
#define MAP_SIZE          (1 << MAP_SIZE_POW2)
#define ICNT_SIZE         MAP_SIZE

EXP_ST u8* trace_bits;    /* SHM with coverage bitmap    */
EXP_ST u32* icnt_bits;    /* SHM with performance bitmap */
```

Listing 3.6: Configurations of the bitmaps

After analyzing the execution information, the calibration stage updates the undefined or old values of the current `queue_entry`.

Calculate performance score

The most favorable entries for Waffle, are the inputs with the highest `total_icnt`. In addition to the calculations of AFL, Waffle also compares the `q->total_icnt` with the average `total_icnt` of all previous entries, `avg_total_icnt`.

```
static u32 calculate_score(struct queue_entry* q) {
  u32 perf_score = 100;
  u64 avg_total_icnt = total_icnt / total_bitmap_entries;
  // ...
  if (q->total_icnt > avg_total_icnt * 3) perf_score *= 1.4;
  else if(q->total_icnt > avg_total_icnt * 2) perf_score *= 1.2;
  else if(q->total_icnt > avg_total_icnt) perf_score *= 0.85;
  else if(q->total_icnt > avg_total_icnt * 0.75) perf_score *= 0.7;
  else perf_score *= 0.5;
  // ...
}
```

Listing 3.7: The modifications of Waffle in `calc_score()`

With all these modifications, Waffle can start it's fuzzing. The only difference between the execution of Waffle and AFL, is the name of the fuzzer.

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

Listing 3.8: Execute AFL

3.5 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 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** [58]. To evaluate the performance of Waffle, we use **fuzzbench** [59] 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.

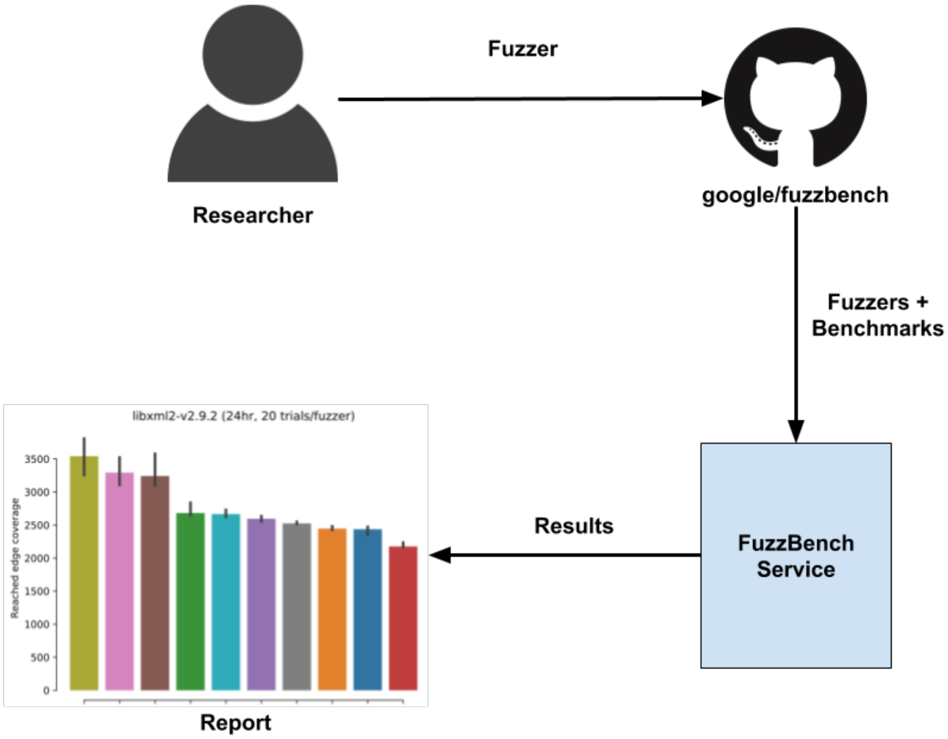


Figure 4.1: Fuzzbench overview

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.” [59]

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.

```
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 [60]. 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). [61]"
- **Setting up a Google Cloud Project:** We can run our experiment on **GCP** [62].
- **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, 4.3, 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 $meancoverage$, $standarddeviation$, and the number of $uniquefindings$ 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**	**Fuzzer**	**Waffle**	**Fuzzer**	**Unique**
		Mean Coverage	**Mean Coverage**	**STD**	**STD**	**(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.

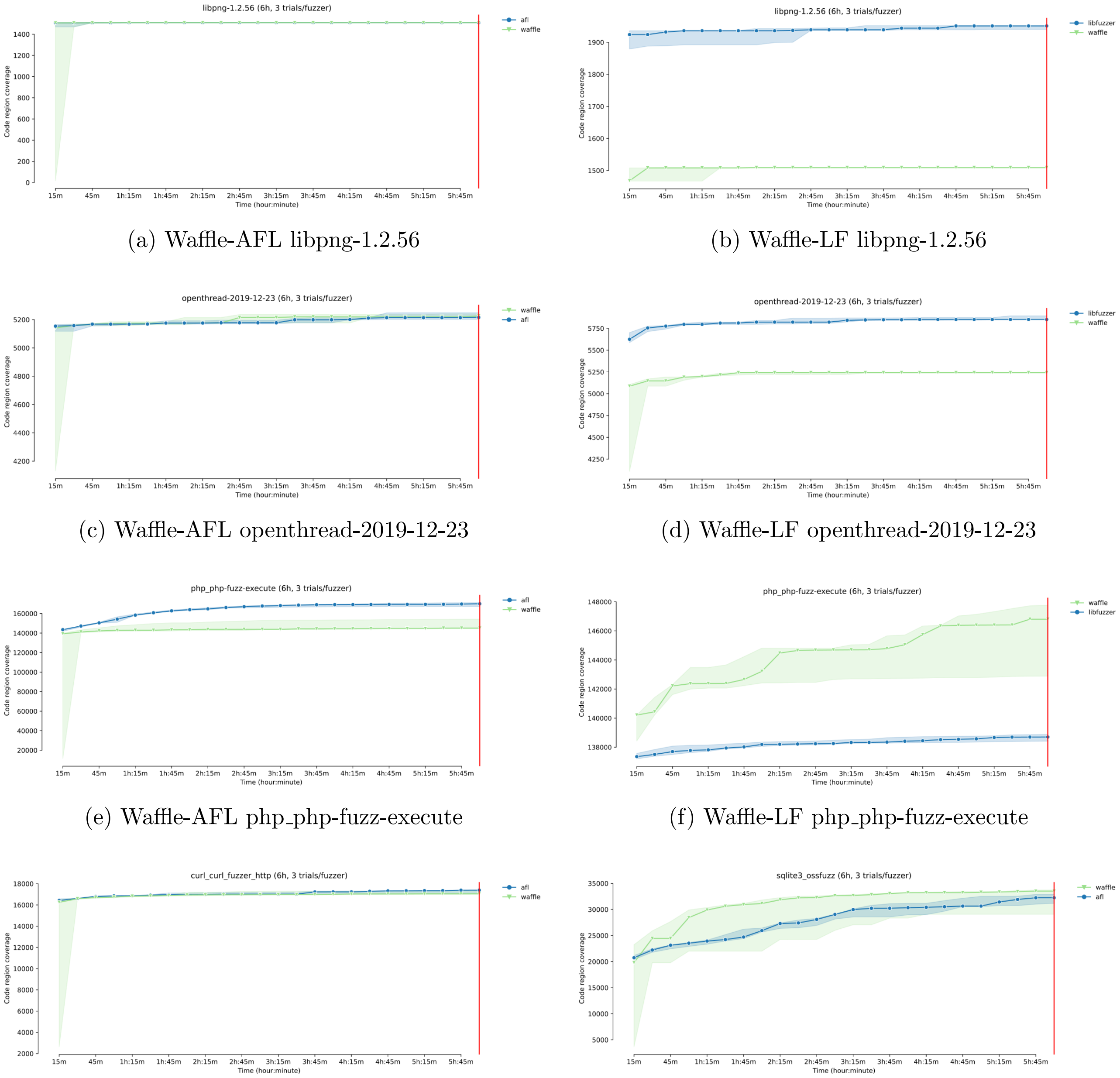
In the third scenario, we see that in the *6-hours* trials, Waffle succeeds LibFuzzer for fuzzing *php*.

In Figure 4.3, 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

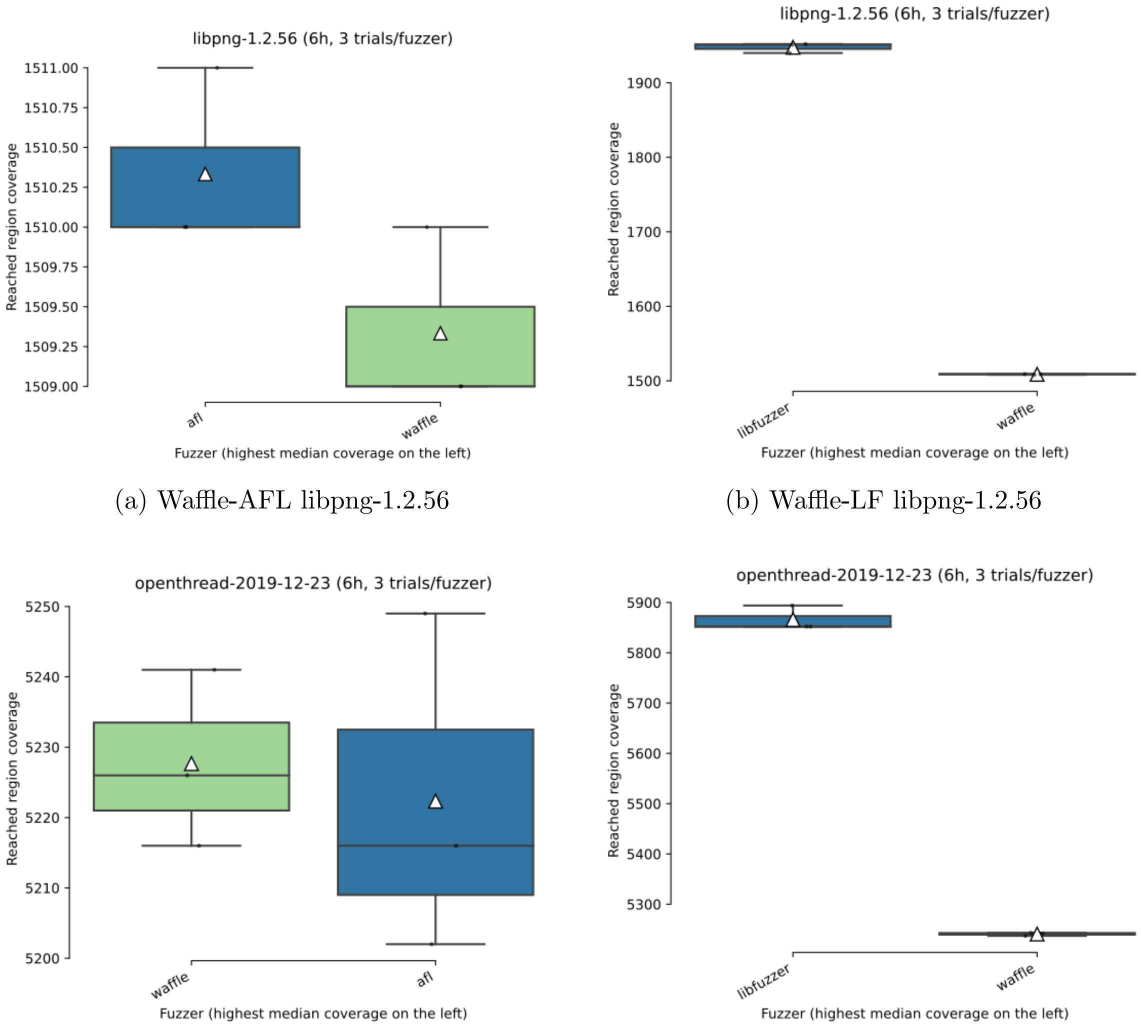
(g) Waffle-AFL curl_curl_fuzzer_http

(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

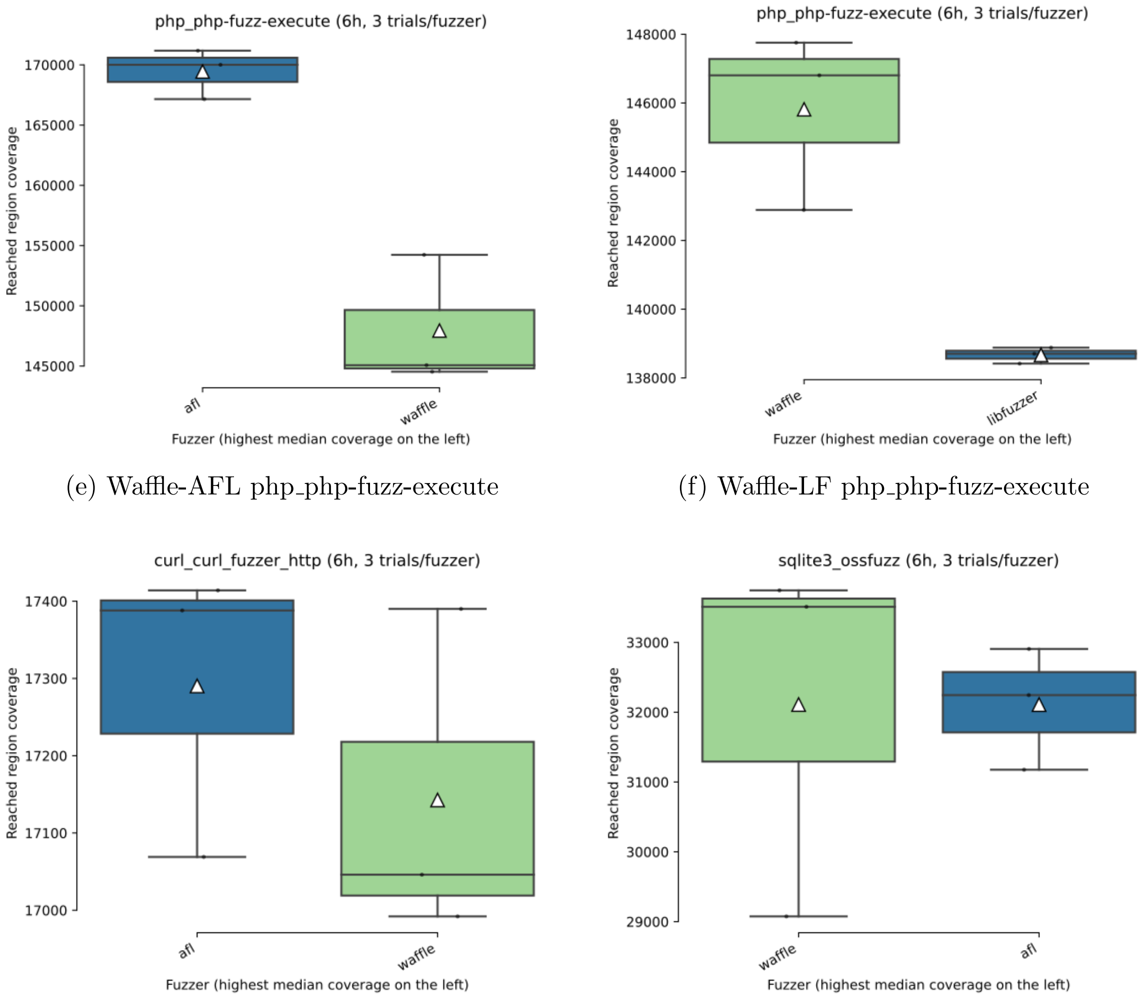
(b) Waffle-LF libpng-1.2.56

(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] 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.

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

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

[51] Afl user guide. https://afl-1.readthedocs.io/en/latest/user_guide.html, 2019.

[52] Finding memory bugs with addresssanitizer. https://embeddedbits.org/finding-memory-bugs-with-addresssanitizer/, 2020.

[53] Google/addresssanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer, 2019.

[54] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 309–318, 2012.

[55] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[56] add nosanitize metadata to more coverage instrumentation instructions. https://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20150302/263798.html, 2015.

[57] source of afl-clang-fast.c. https://github.com/google/AFL/blob/master/llvm_mode/afl-clang-fast.c, 2021.

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

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

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

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

[62] 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.

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

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

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

```c
  u32 use_stacking = 1 << (1 + UR(HAVOC_STACK_POW2));
  for (i = 0; i < use_stacking; i++) {
    switch (UR(15 + ((extras_cnt + a_extras_cnt) ? 2 : 0))) {
      case 0:
        /* Flip a single bit somewhere. Spooky! */
      case 1:
        /* Set byte to interesting value. */
      case 2:
        /* Set word to interesting value, randomly choosing endian. */
      case 3:
        /* Set dword to interesting value, randomly choosing endian. */
      case 4:
        /* Randomly subtract from byte. */
      case 5:
        /* Randomly add to byte. */
      case 6:
        /* Randomly subtract from word, random endian. */
      case 7:
        /* Randomly add to word, random endian. */
      case 8:
        /* Randomly subtract from dword, random endian. */
      case 9:
        /* Randomly add to dword, random endian. */
      case 10:
        /* 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. */
      case 11 ... 12:
        /* Delete bytes. We're making this a bit more likely than insertion (the next option) in hopes of keeping files reasonably small. */
      case 13:
        /* Clone bytes (75%) or insert a block of constant bytes (25%). */
      case 14:
        /* Overwrite bytes with a randomly selected chunk (75%) or fixed bytes (25%). */

        /* Values 15 and 16 can be selected only if there are any extras present in the dictionaries. */
      case 15:
        /* Overwrite bytes with an extra. */
      case 16:
        /* Insert an extra. Do the same dice-rolling stuff as for the previous case. */
    }
  }
  /* 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(). */
}
```

Listing 6.1: Random havoc stage

6.B FuzzBench

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

6.B.1 builder.Dockerfile

Builds Waffle for the usage in FuzzBench.

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

```
ARG parent_image
FROM $parent_image

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

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

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

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

Listing 6.2: Recipe for building Waffle FuzzBench

6.B.2 fuzzer.py

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

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

import json
import os
import shutil
import subprocess

from fuzzers import utils


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

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


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

    utils.build_benchmark()

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


```

```python
def get_stats(output_corpus, fuzzer_log):  # pylint: disable=unused-argument
    """Gets fuzzer stats for Waffle."""
    # Get a dictionary containing the stats Waffle reports.
    stats_file = os.path.join(output_corpus, 'fuzzer_stats')
    with open(stats_file) as file_handle:
        stats_file_lines = file_handle.read().splitlines()
    stats_file_dict = {}
    for stats_line in stats_file_lines:
        key, value = stats_line.split(': ')
        stats_file_dict[key.strip()] = value.strip()

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


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

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


def run_waffle_fuzz(input_corpus,
                    output_corpus,
                    target_binary,
                    additional_flags=None,
                    hide_output=False):
    """Run the fuzzer"""
    # Spawn the waffle fuzzing process.
    print('[run_waffle_fuzz] Running target with waffle-fuzz')
    command = ['./waffle-fuzz', '-i', input_corpus, '-o', output_corpus, '-d', '-m', 'none', '-t', '1000']
    if additional_flags:
        command.extend(additional_flags)
    dictionary_path = utils.get_dictionary_path(target_binary)
    if dictionary_path:
        command.extend(['-x', dictionary_path])
    command += [
        '--',
        target_binary,
        # Pass INT_MAX to afl the maximize the number of persistent loops it
        # performs.
        '2147483647'
    ]
    print('[run_waffle_fuzz] Running command: ' + ' '.join(command))
    output_stream = subprocess.DEVNULL if hide_output else None
    subprocess.check_call(command, stdout=output_stream, stderr=output_stream)


def fuzz(input_corpus, output_corpus, target_binary):
    """Run waffle-fuzz on target."""
    prepare_fuzz_environment(input_corpus)

    run_waffle_fuzz(input_corpus, output_corpus, target_binary)
```

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