

Behnam Fuzzer

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

Soon...!

© Behnam Bojnordi Arbab, 2020

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

Abstract	ii
Dedication	iii
Acknowledgments	iv
Table of Contents	v
List of Tables	vii
List of Figures	viii
Abbreviations	ix
1 Introduction	1
2 Background	2
2.1 Introduction	2
2.2 Literature Review	2
2.3 LLVM	9
2.3.1 Instrumentation and coverage measurements	10
2.4 AFL	14
2.5 Concluding remarks	22

3	Proposed Fuzzer	23
3.1	Introduction	23
3.2	Motivating example	24
3.3	Instrumentation	26
3.4	Fuzzing	30
3.5	Concluding remarks	34
4	Simulation	35
4.1	Introduction	35
4.2	FuzzBench	35
4.3	FuzzBench Reports	38
	Bibliography	44
5	Appendix	49
5.A	Waffle	49
5.A.1	random_havoc	49
5.B	FuzzBench	51
5.B.1	builder.Dockerfile	51
5.B.2	fuzzer.py	52

List of Tables

4.1 Statistics of the experiments. 40

List of Figures

2.1	Fuzzing phases	3
2.2	LLVM architecture: A front-end compiler generates the LLVM IR, and then it is converted into machine code [1]	10
2.3	Example for instrumented basic blocks	13
2.4	AFL status screen	20
4.1	Fuzzbench overview	36
4.2	The above figures displays the reached coverage in the end of the experiments. The green boxes represent Waffle’s code-coverage. . . .	41
4.3	The above figures displays the reached coverage in the end of the experiments. The green boxes represent Waffle’s code-coverage. . . .	42
4.4	The above figures displays the reached coverage in the end of the experiments. The green boxes represent Waffle’s code-coverage. . . .	43

List of Symbols, Nomenclature or Abbreviations

Start writing here. This is optional.

\sum \sum
 \bigcap \bigcap

Chapter 1

Introduction

Chapter 2

Background

2.1 Introduction

The term **fuzzing** or **fuzz testing** was first introduced in late 80's. OWAPS defines it as a tool for “finding implementation bugs using malformed/semi-malformed data injection in an automated fashion”. [2]

There are different implementations of fuzzers for different purposes. Our fuzzer is a coverage-based whitebox fuzzer, and we will explain their definition in this chapter. In this chapter, we begin by reviewing the previous works that lead to this thesis 2.2. We bring up the features we use from LLVM in section 2.3. In section 2.4 we review the implementation of AFL, and we wrap up this chapter with conclusions.

2.2 Literature Review

Sutton et al. [3] defined the procedure of fuzz testing, as shown in Figure 2.1:

1. Generally, the first stage is **to identify the target**. A target may be a software or a combination of software and hardware [4]. The targeted software is any program that is executed on a machine. For the rest of this article, a target is software.

2. To execute the target program, we need to specify how the program parses the inputs. Inputs are a set of environmental variables, file formats, and any other parameters that affect the program's execution.

A fuzzer needs a set of seeds for initialization. This set could be empty, and a fuzzer without any sample inputs may find valid fuzzed files out of thin air [5] that the target program considers them as valid.

3. The fuzzing loop starts with generating fuzzed inputs. The fuzzer provides these files for the program to execute and process them.
4. In this stage, the inputs are processed and executed. Depending on the resources needed for the target program's execution, this stage can be the bottleneck for the fuzzing process. The executions' interesting behavior and information are collected for the evolution of the inputs in future iterations.
5. If an exceptional event happens during the program's execution and the program exits unsuccessfully, we say an exception is occurred. These exceptions are the vulnerabilities that may be exploitable and cause security problems. Handling an exception properly and pinpointing the inputs responsible for the exception is the primary purpose of this stage.

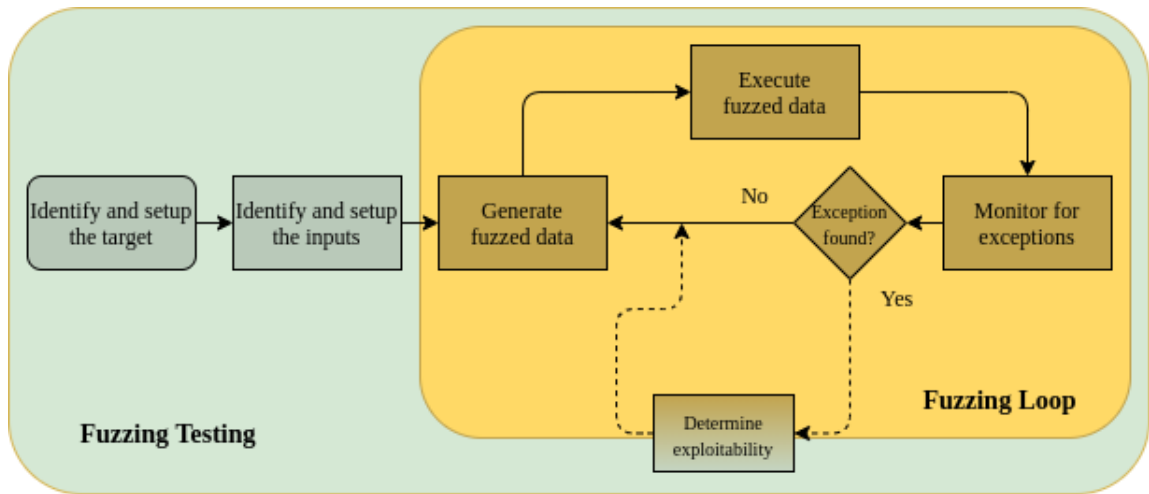


Figure 2.1: Fuzzing phases

6. The last stage of the fuzzing is evaluating the reported vulnerabilities. This evaluation explains whether the vulnerabilities are exploitable or not.

A common feature in most of the fuzzers is the fuzzing loop which is looking for more valuable inputs. The stages may vary depending on the problem and goals of a fuzzer. We will walk over the stages of AFL as our fuzzer is based on AFL, and we will be back on the stages of fuzzing in the following sections.

Sample program

To evaluate the performance of a fuzzer and assess the execution of it, the sample C code is implemented. (Listing 2.1) [6]

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 void vul_function(char * msg) {
6     char *ptr = NULL;
7     ptr = malloc(strlen(msg));
8     strcpy(ptr, msg);
9     printf("%s\n", ptr);
10 }
11
12 int main(int argc, char *argv[])
13 {
14     if(argc<2) {
15         printf("At least one input is needed");
16         return 1;
17     }
18
19     vul_function(argv[1]);
20
21     return 0;
22 }
```

Listing 2.1: 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 2.2: Compile the sample program

One way to detect the prior vulnerability, is to add AddressSanitizer flag for the compilation [7]. 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) [8]

After we provide the ASan flag for the compilation, we face an error with the same input as the previous example (Listing 2.2):

```
$ 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.

Black/Grey/White-box fuzzing

The *colorful* representation of fuzzers depends on how much information we can collect from any execution of the target. In a blackbox fuzzing, we do not gather any information from the execution, and the fuzzing tries new inputs expecting an error to occur.

```
$ ./blackbox-fuzzer ./sample_vul_asan
[Blackbox fuzzing: sample_vul_asan]
```

The introduced fuzzer by Miller [9] was of the very first naive blackbox fuzzers targeting a collection of Unix utilities on different Unix versions. It runs the fuzzing for different lengths of inputs for each utility (of the total 88 utilities) and expects a **crash**, **hang**, or a **succeed** after the execution of the program. The generation of inputs is by the mutation of the inputs' content, and the target program is a blackbox for the fuzzer. As a result, this fuzzer is a mutation-based blackbox fuzzer. One of the **downsides** of the blackbox fuzzing is that the program may face some branches with **magic values**, constraining the variable to a specific set of values; the fuzzer has to apply the exact magic value, which may have a very low probability - it is almost impossible to generate a specific 1 KB string of bytes randomly.. In [10] and [11] 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 [12] and [13] have targeted web applications and found ways to attack some the websites, looking for different vulnerabilities, such as XSS.

Whitebox fuzzing works with the source code of the target. In this technique, an **instrumentation** is applied before the compilation of the program. Instrumenting a program with debugging instructions, is the procedure of injecting new instructions into the resulting binary, without affecting the logic of the program. We will analyze the instrumentations more in the LLVM section.

The following snippet of code shows an example of such execution 2.2. *whitebox-instr-c* compiles the program with the required instrumentations and *whitebox-fuzzer* fuzzes the program appropriately.

```
$ ./whitebox-instr-c sample_vul.c --output sample_wht_inst
$ ./whitebox-fuzzer sample_wht_inst
[Whitebox fuzzing: sample_wht_inst]
```

SAGE, a whitebox fuzzer, was developed as an alternative to blackbox fuzzing to cover the lacks of blackbox fuzzers [14]. With the benefit of having the source code and internal knowledge for fuzzing, a whitebox fuzzer can leverage symbolic constraints for symbolic analysis to solve the constraints (such as magic values) in the program [15]. It can also use dynamic and concolic execution [16] and use taint analysis to locate the regions of seed files influencing values used by program [17]. In addition, a whitebox fuzzer can find the grammar for parsing the input files without any prior knowledge [18]. It is a noticeable performance enhancement as we have the source code.

Greybox fuzzing resides between whitebox and blackbox fuzzing, as we only have partial knowledge about the internals of the target program. We do not have the source code, but we have some knowledge about the program (for instance, we have the binary file), and as a result, we have the instructions of the program (2.2):

```
$ ./greybox-instr-c ./sample_vul_asan --output sample_gry_inst
$ ./greybox-fuzzer sample_gry_inst
[Greybox fuzzing: sample_gry_inst]
```

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 [19]. 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**.

Coverage-based fuzzing

Coverage-based fuzzing is 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 [20].

AFL instruments the program with only the coverage information (section AFL).

The instrumentation can collect execution’s data such as data coverage, statement coverage, block coverage, decision coverage, and path coverage [21]. Bohme et al. [22] introduced a coverage-based greybox fuzzer that extends AFL and benefits from the Markov Chain model. The fuzzer calculates the **energy** of the inputs based on the potency of a path for the discovery of new paths.

In another article by Bohme et al. [23], they introduce their directed fuzzing by the idea of checking the code-coverage for providing inputs that guide the program execution toward some targeted locations. Some of the applications of such a fuzzing approach are patch testing and crash reproduction, which has different use cases compared to a non-directed coverage-based fuzzers.

Performance fuzzing

The **types** of vulnerabilities that a fuzzer is involved with may be different from other fuzzers. For example, AFL looks for crashes or hangs by selecting and mutating the inputs, and at the same time, it considers the code coverage, size of the inputs, and each execution time of the target program. PerfFuzz [24] is a greybox fuzzer based on AFL, which aims to generate inputs for executions with higher **execution time** while using most of the features of AFL in code exploration. PerfFuzz counts how many times each of the edges of the control flow graph (CFG) is executed. Using SlowFuzz [25], we can consider any type of resource as a feature to detect the worst-case scenarios (inputs) for a given program. In another project based on AFL, Memlock [26] investigates memory exhaustion by calculating the maximum runtime memory required during executions. A disadvantage in previous works in performance is that the development of the fuzzer for considering different instructions is cumbersome.

Waffle (What An Amazing AFL - WAAFL) is a coverage-based whitebox

fuzzer that is based on AFL’s base code. Waffle leverages **visitors** to collect the stats of different instructions during the execution. To learn more about Waffle, we study the features we benefit from the LLVM, as well as the current features of AFL that help us in reaching the goals of this thesis.

2.3 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.” [27, 28]

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.” [29] 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 struc-

ture of the LLVM project is shown in Figure 2.2:

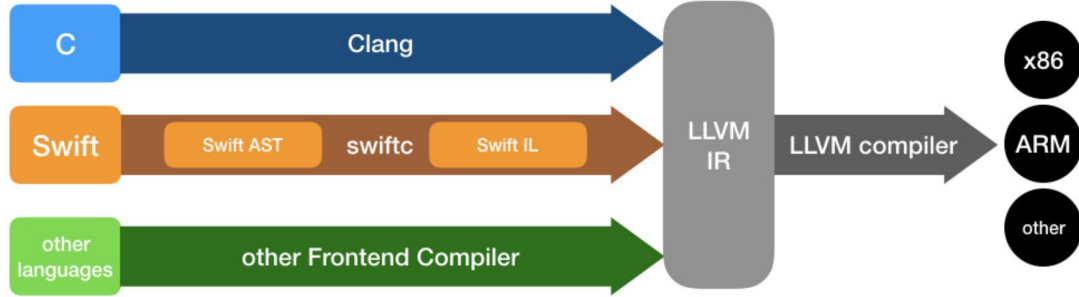


Figure 2.2: LLVM architecture: A front-end compiler generates the LLVM IR, and then it is converted into machine code [1]

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 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 2.3:

```

1 // LLVM-mode instrumentation pass
2 bool AFLCoverage::runOnModule(Module &M) {
3
4     /* Instrument all the things! */
5     for (auto &F : M)
6         for (auto &BB : F) {
7             BasicBlock::iterator IP = BB.getFirstInsertionPt();
8             IRBuilder<> IRB(&(*IP));
9
10            if (AFL_R(100) >= inst_ratio) continue;
11
12            /* Make up cur_loc */
13            unsigned int cur_loc = AFL_R(MAP_SIZE);
14            ConstantInt *CurLoc = ConstantInt::get(Int32Ty, cur_loc);
15
16            /* Load prev_loc */
17            LoadInst *PrevLoc = IRB.CreateLoad(AFLPrevLoc);
18            PrevLoc->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(
19            C, None));
20            Value *PrevLocCasted = IRB.CreateZExt(PrevLoc, IRB.getInt32Ty
21            ());
22
23            /* Load SHM pointer */
24            LoadInst *MapPtr = IRB.CreateLoad(AFLMapPtr);
25            MapPtr->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C
26            , None));
27            Value *MapPtrIdx =
28                IRB.CreateGEP(MapPtr, IRB.CreateXor(PrevLocCasted, CurLoc)
29            );
30
31            /* Update bitmap */
32            LoadInst *Counter = IRB.CreateLoad(MapPtrIdx);
33            Counter->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(
34            C, None));
35            Value *Incr = IRB.CreateAdd(Counter, ConstantInt::get(Int8Ty,
36            1));
37            IRB.CreateStore(Incr, MapPtrIdx)
38                ->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C,
39            None));
40
41            /* Set prev_loc to cur_loc >> 1 */
42            StoreInst *Store =
43                IRB.CreateStore(ConstantInt::get(Int32Ty, cur_loc >> 1),
44            AFLPrevLoc);
45            Store->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C,
46            None));
47
48            inst_blocks++;
49        }
50    }
51    return true;
52 }

```

Listing 2.3: AFLCoverage module

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

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

Listing 2.4: 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 resetting 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 2.3, 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 *shared_mem*[$14287 \oplus 23765$] is applied for the transition of $1 \rightarrow 2$ and *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). [30]”

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

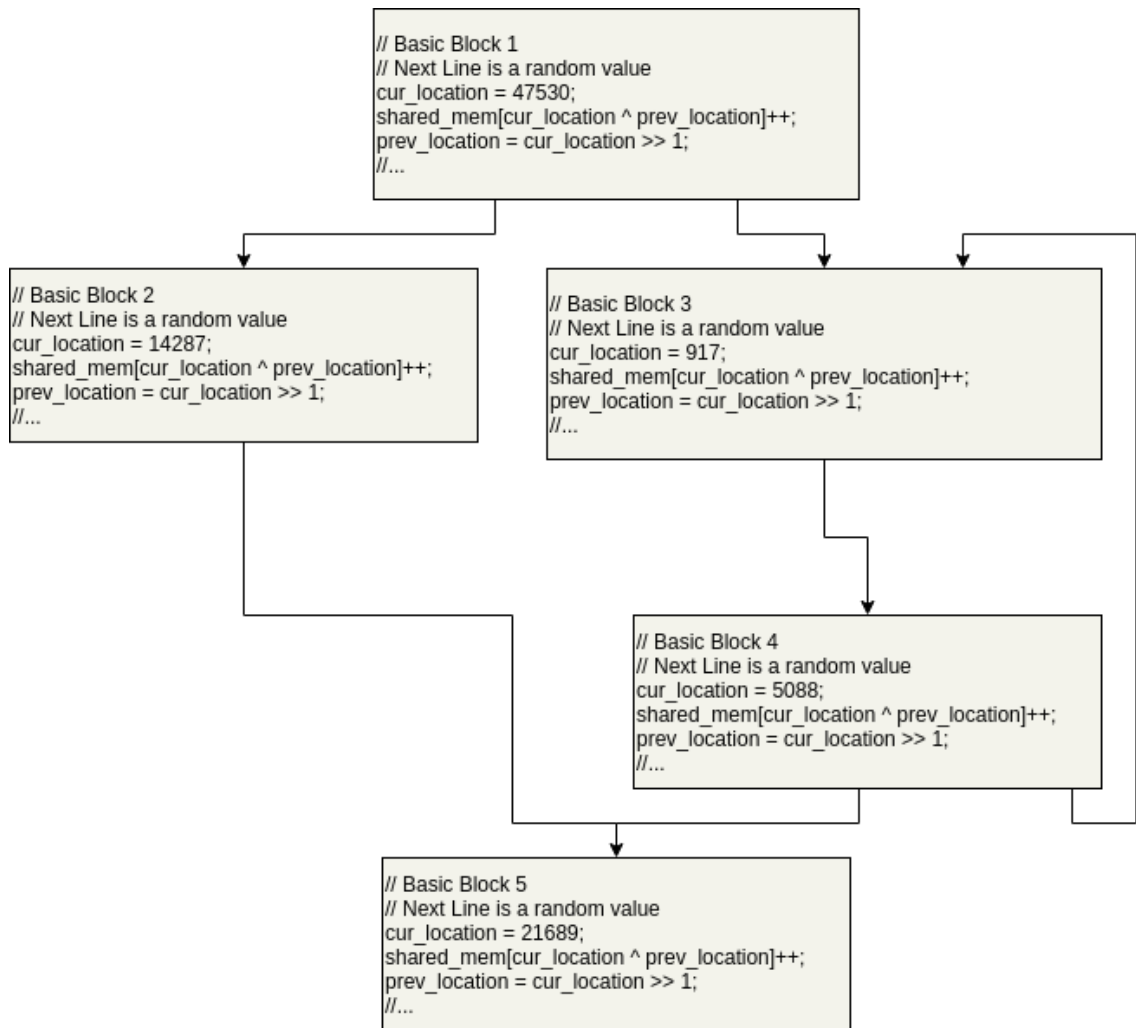


Figure 2.3: Example for instrumented basic blocks

```

2
3 struct CountAllVisitor : public InstVisitor<CountAllVisitor> {
4     unsigned Count;
5     CountAllVisitor() : Count(0) {}
6     // Any visited instruction is counted in a specified range
7     void visitInstruction(Instruction &I) {
8         ++Count;
9     }
10 };

```

Listing 2.5: 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.

2.4 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 corpora produced by the tool help seed other, more labor- or resource-intensive testing regimes down the road.” [31]

AFL is designed to perform **fast** and **reliable**, and at the same time, benefit from the **simplicity** and **chainability** of the fuzzer [32]:

- **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-fuzz.c has the instructions for fuzzing the target instrumented-program. The Algorithm 1 illustrates a brief pseudocode of the execution of *afl-fuzz*:

Algorithm 1: afl-fuzz

Input: *in_dir, out_dir, instrumented Target*

```

1 initialize fuzzer;
2 while fuzzing is not terminated do
3   cull queue and update bitmaps;
4    $Entry \leftarrow q.first\_entry()$ ;
5   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.” [33]

$$fav_factor = e.exec_time \times e.length \quad (2.1)$$

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()**.

fuzz_one()

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**,

Algorithm 2: *fuzz_one*: Fuzz one Entry

Input: *queueEntry*
1 $T \leftarrow \text{Entry.test_case};$
2 $\text{calibrate}(T);$
3 $\text{trim}(T);$
4 $\text{perf_score} \leftarrow \text{calculate_score}(T);$
5 **if** *deterministic_mode* **then**
6 $\text{deterministic_stages}(T);$
7 $MX \leftarrow \text{calc_stages}(\text{perf_score}, \mathcal{C});$
8 **foreach** *element* in $(0, MX)$ **do**
9 $\text{random_havoc}(T);$

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 2, 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 [34]. An abstract implementation of the **havoc_stage** can be found in Appendix 5.A.1.

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 2.6]:

```
1 /* 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. */
2
3 static u32 calculate_score(struct queue_entry* q) {
4     u32 perf_score = 100;
5
6     /* 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. */
7
8     if (q->exec_us * 0.1 > avg_exec_us) perf_score = 10;
9     else if (q->exec_us * 4 < avg_exec_us) perf_score = 300;
10    // Check other conditions in between
11
12    /* Adjust score based on bitmap size. The working theory is that
       better coverage translates to better targets. Multiplier from
       0.25x to 3x. */
13    if (q->bitmap_size * 0.3 > avg_bitmap_size) perf_score *= 3;
14    else if (q->bitmap_size * 3 < avg_bitmap_size) perf_score *= 0.25;
15    // Check other bitmap_sizes in between
16
17    /* 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. */
18
19    /* 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. */
20
21    switch (q->depth) {
22        case 0 ... 3:    break;
23        case 14 ... 25: perf_score *= 4; break;
24        // Check other cases in between
25        default:        perf_score *= 5;
26    }
27
28    /* Make sure that we don't go over limit. */
29    if (perf_score > HAVOC_MAX_MULT * 100) perf_score = HAVOC_MAX_MULT
        * 100;
```

```

30
31     return perf_score;
32 }

```

Listing 2.6: An abstract implementation of `calculate_score()`

`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.

```

1  /* 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(). */
2
3  EXP_ST u8 common_fuzz_stuff(char** argv, u8* out_buf, u32 len) {
4      // Validate the file
5      // ...
6      // Validate the execution
7      fault = run_target(argv, exec_tmout);
8      // If the file is "interesting", add it into queue
9      queued_discovered += save_if_interesting(argv, out_buf, len, fault
10     );
11     // ...
12     return 0;
13 }
14
15 /* 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. */
16
17 static u8 save_if_interesting(char** argv, void* mem, u32 len, u8
   fault) {
18     if (fault == crash_mode) {
19         hnb = has_new_bits(virgin_bits);
20     }
21     if(!hnb) return 0;
22
23     queue_top->exec_cksum = hash32(trace_bits, MAP_SIZE, HASH_CONST);
24
25     /* Try to calibrate inline; this also calls update_bitmap_score()
   when successful. */
26     res = calibrate_case(argv, queue_top, mem, queue_cycle - 1, 0);
27
28     // Save the file and return

```

28 }

Listing 2.7: An abstract implementation of common_fuzz_stuff()

has_new_bits()

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

```

1  /* 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.
2
3  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. */
4
5  static inline u8 has_new_bits(u8* virgin_map) {
6      u64* current = (u64*)trace_bits;
7      u64* virgin = (u64*)virgin_map;
8      u32 i = (MAP_SIZE >> 3);
9      while (i--> 0) {
10
11         /* 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. */
12
13         if (unlikely(*current) && unlikely(*current & *virgin)) {
14             if (likely(ret < 2)) {
15                 u8* cur = (u8*)current;
16                 u8* vir = (u8*)virgin;
17
18                 /* Looks like we have not found any new bytes yet; see if
                    * any non-zero bytes in current[] are pristine in virgin[]. */
19                 if ((cur[0] && vir[0] == 0xff) || (cur[1] && vir[1] == 0xff)
20                     || (cur[2] && vir[2] == 0xff) || (cur[3] && vir[3] == 0xff)
21                     || (cur[4] && vir[4] == 0xff) || (cur[5] && vir[5] == 0xff)
22                     || (cur[6] && vir[6] == 0xff) || (cur[7] && vir[7] == 0xff)
23                 ) ret = 2;
24                 else ret = 1;
25             }
26             *virgin &= ~*current;
27             current++; virgin++;
28         }
29
30         if (ret && virgin_map == virgin_bits) bitmap_changed = 1;

```

```

31     return ret;
32 }

```

Listing 2.8: The implementation of has_new_bits()

Status screen

The **status screen** is a UI for the status of the fuzzing procedure. As it is shown in Figure 2.4, 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

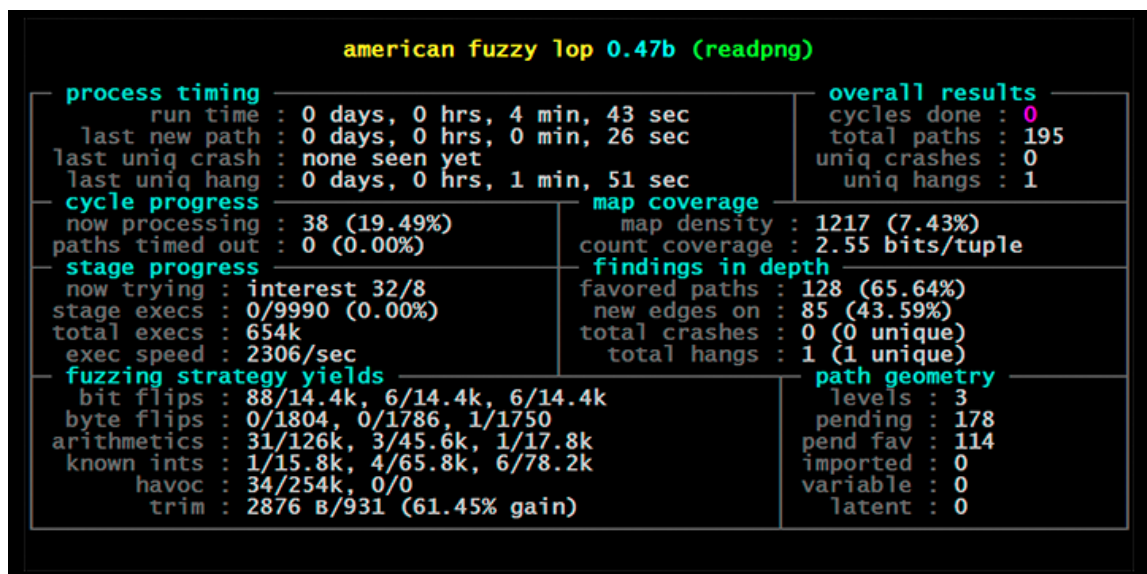


Figure 2.4: AFL status screen

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.
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.

Start Fuzzing

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 2.1:

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

Listing 2.9: 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 2.10: 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*.

2.5 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

In this chapter, we introduce **Waffle** in more details. Waffle is a tool capable of finding the vulnerabilities related to (theoretically) any resource exhaustion. The first section explains a motivating example leading to our proposed fuzzer. This fuzzer is based on AFL and extends its implementation. For monitoring the resources, we use compile-time instrumentation of the target program using LLVM's APIs; we take advantage of **visiting** APIs that let us keep track of any instructions defined for LLVM. As a result, the instructions related to any resource are counted, and this information is later used in the fuzzing stage.

AFL is the state-of-the-art in finding vulnerabilities and as it is amazing to be developed, the name of our fuzzer comes after *WAAAFLL*!

In this chapter, we are contributing the following topics:

- An implementation of a fuzzer for finding the worst-case scenario in an algebraic problem.
- We use the **visitor** functions, which are not used in previous works, as we are

aware of.

- A new instrumentation for collecting runtime information about resource usages. In Waffle, we focus on maximising the number of instructions.
- A new fuzz testing approach for collectively considering the former features of AFL, as well as the features we introduce in Waffle.

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 [35] 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:

```
1 #include<stdio.h>
2
3 void swap(int* a, int* b) {
```

```

4   int t = *a; *a = *b; *b = t;
5   }
6
7   int partition (int arr[], int low, int high) {
8       int pivot = arr[high];
9       int i = (low - 1); // Index of smaller element
10      for (int j = low; j <= high- 1; j++)
11          if (arr[j] <= pivot) {
12              i++; // increment index of smaller element
13              swap(&arr[i], &arr[j]);
14          }
15
16      swap(&arr[i + 1], &arr[high]);
17      return (i + 1);
18  }
19
20  void quickSort(int arr[], int low, int high){
21      if (low < high) {
22          int pi = partition(arr, low, high);
23
24          quickSort(arr, low, pi - 1);
25          quickSort(arr, pi + 1, high);
26      }
27  }
28
29  void printArray(int arr[], int size) {
30      int i;
31      for (i=0; i < size; i++)
32          printf("%d ", arr[i]);
33      printf("\n");
34  }
35
36  // Driver program to test above functions
37  int main() {
38      int arr[] = {10, 7, 8, 9, 1, 5};
39      int n = sizeof(arr)/sizeof(arr[0]);
40      quickSort(arr, 0, n-1);
41      printf("Sorted array: \n");
42      // printArray(arr, n);
43      return 0;
44  }

```

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):

```
1 // snippet of wafll-llvm-rt.o.c
2
3 u8  __wafll_icnt_initial[ICNT_SIZE];
4 u8* __wafll_area_ptr = __wafll_icnt_initial;
5
6 static void __afl_map_shm(void) {
7
8     u8 *id_str = getenv(SHM_ENV_VAR);
9
10    if (id_str) {
11        u32 shm_id = atoi(id_str);
12        __wafll_area_ptr = shmat(shm_id, NULL, 0);
13
14        if (__wafll_area_ptr == (void *)-1) _exit(1);
15
16        memset(__wafll_area_ptr, 0, sizeof __wafll_icnt_ptr);
17    }
18 }
```

Listing 3.2: LLVM instrumentation bootstrap

`__wafll_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 `__wafll_icnt_ptr` is equal to $ICNT_SIZE = 2^{16}$; the size of the bitmaps are equal in both AFL and Waffle.

`wafll-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**:

```
1 struct CountAllVisitor : public InstVisitor<CountAllVisitor> {
2     unsigned Count;
3     CountAllVisitor() : Count(0) {}
4     void visitMemCpyInst(MemCpyInst &I) { ++Count;}
5 };
```

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 `__wafll_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.

```
1 struct sys_data {
2     int MaxInstCount;
3 };
4
5 static int MaxInstCount = 0;
6
7 void __attribute__((destructor)) traceEnd(void) {
8     unsigned char *mem_str = getenv(MEM_ENV_VAR);
9
10    if (mem_str) {
11        unsigned int shm_mem_id = atoi(mem_str);
12        struct sys_data *da;
13
14        da = shmat(shm_mem_id, NULL, 0);
15        if (da == (void *)-1) _exit(1);
16
17        da->MaxInstCount = MaxInstCount;
18    }
19 }
20
21 void instr_AddInsts(int cnt) {
```

```

22   MaxInstCount += cnt;
23 }

```

Listing 3.3: sys_data in instrumentation

Now we can insert our instructions to the basicblocks:

```

1  // snippet of wafl-llvm-pass.so.cc
2  #include <math.h>
3  // ...
4  bool WAFLCoverage::runOnModule(Module &M) {
5      // ...
6      GlobalVariable *WAFLMapPtr =
7          new GlobalVariable(M, PointerType::get(Int8Ty, 0), false,
8                          GlobalValue::ExternalLinkage, 0, "__wafl_area_ptr");
9
10     llvm::FunctionType *icntIncrement =
11         llvm::FunctionType::get(builder.getVoidTy(), icnt_args, false);
12     llvm::Function *icnt_Increment =
13         llvm::Function::Create(icntIncrement,
14                             llvm::Function::ExternalLinkage, "instr_AddInsts", &M);
15     // ...
16     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 [36].

Same as AFL, Waffle stores the counters in the **hashed value** of the path we explored (Listing 2.4). 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):

$$CNT = \log_2^{CAV} \quad (3.1)$$

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.” [37] Except refactoring the filenames and the name of the variables, we did not apply any other modifications on **waffle-clang.c**.

Same as 2.2, 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 1 and algorithm 2:

Algorithm 3: *waffle – fuzz*

Input: *queueEntry*

```

1 initialize fuzzer;
2 while fuzzing is not terminated do
3   cull_queue();
4    $T \leftarrow q.first\_entry()$ ;
5   calibrate( $T$ );
6   trim( $T$ );
7    $perf\_score \leftarrow calculate\_score(T)$ ;
8   if deterministic_mode then
9     deterministic_stages( $T$ );
10   $MX \leftarrow calc\_stages(perf\_score, \mathcal{C})$ ;
11  foreach element in  $(0, MX)$  do
12    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 pro-

gram and collects the added **instruction counters**. Each instruction counter is monitored in the shared memory, `icnt_bits[]`, and `topRated_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 CNTs 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`:

```
1 static inline u8 has_new_max() {
2     // #define MAX_CNT_MULT 1.05
3     int ret = 0;
4     for (int i = 0; i < ICNT_SIZE; i++) {
5         if (unlikely(icnt_bits[i]) && unlikely(icnt_bits[i] >
6             MAX_CNT_MULT*max_icnts[i])) {
7             if(likely(ret<2)) {
8                 ret = 2;
9                 max_icnts[i] = icnt_bits[i];
```



```

9     }
10    else ret = 1;
11    }
12    }
13    return ret;
14 }

```

Listing 3.5: has_new_max()

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:

```

1 #define MAP_SIZE_POW2      16
2 #define MAP_SIZE           (1 << MAP_SIZE_POW2)
3 #define ICNT_SIZE          MAP_SIZE
4
5 EXP_ST u8* trace_bits;      /* SHM with coverage bitmap */
6 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`.

```

1 static u32 calculate_score(struct queue_entry* q) {
2     u32 perf_score = 100;
3     u64 avg_total_icnt = total_icnt / total_bitmap_entries;
4     // ...
5     if (q->total_icnt > avg_total_icnt * 3) perf_score *= 1.4;
6     else if (q->total_icnt > avg_total_icnt * 2) perf_score *= 1.2;
7     else if (q->total_icnt > avg_total_icnt) perf_score *= 0.85;
8     else if (q->total_icnt > avg_total_icnt * 0.75) perf_score *= 0.7;
9     else perf_score *= 0.5;
10    // ...
11 }

```

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

With all these modifications, Waffle can start its 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 [38].

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** [33], and **libfuzzer** [39]. To evaluate the performance of Waffle, we use **fuzzbench** [40] 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.

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 rigor-

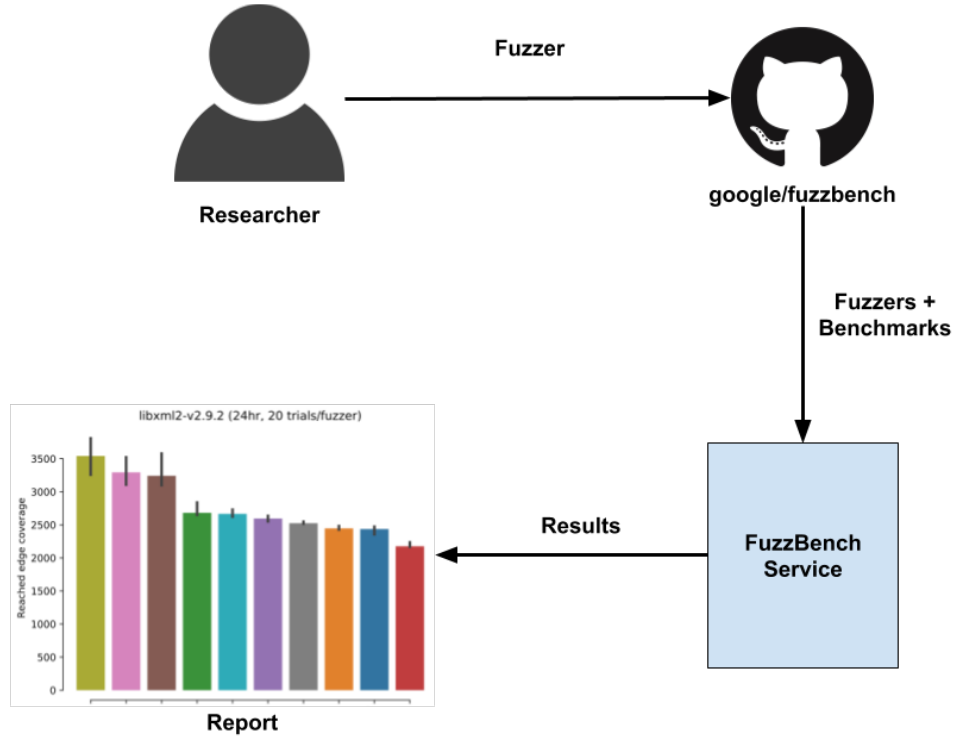


Figure 4.1: Fuzzbench overview

ously evaluate fuzzing research and make fuzzing research easier for the community to adopt.” [40]

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 5.B.1.
- `runner.Dockerfile`: This file defines the image that will be used to run benchmarks with Waffle.

```
1 FROM gcr.io/fuzzbench/base-image
```
- `fuzzer.py`: This file explains how to build and fuzz benchmarks using Waffle [Appendix 5.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 [41]. 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). [42]”
- **Setting up a Google Cloud Project:** We can run our experiment on **GCP** [43].
- **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 \times 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 automatically.

We tested Waffle in 6-hours experiments. Unfortunately, the local computer could not run an experiment with more than **two Fuzzers** and **one benchmark**. A more robust computer and using cloud platforms can enhance the performance of FuzzBench. Each experiment contains **3** trials for each fuzzer.

Our experiments are:

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

We evaluate the results according to the generated reports. The graphs [Figures 4.2, 4.3, 4.4] illustrate the covered regions of the code in each experiment, the pairwise unique coverage, and the growth of the coverage over time. None of the experiments could find any vulnerabilities within 6 hours time limit.

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

Table 4.1: Statistics of the experiments.

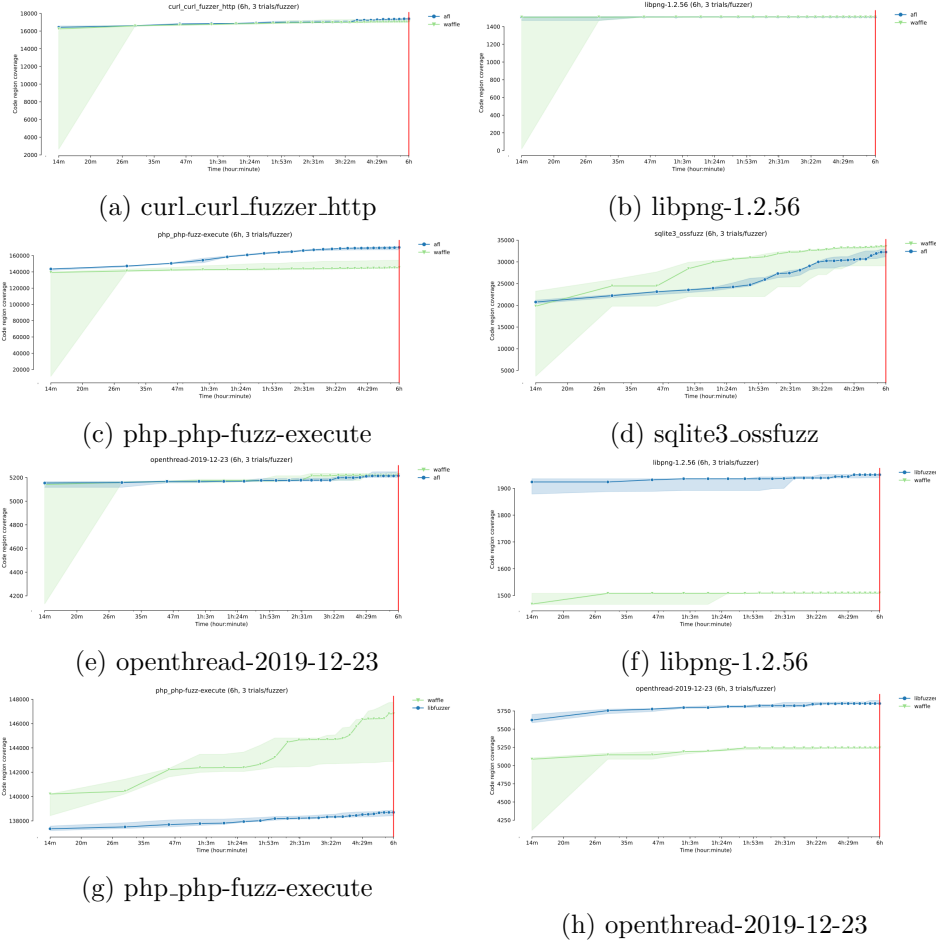


Figure 4.2: The above figures displays the reached coverage in the end of the experiments. The green boxes represent Waffle’s code-coverage.

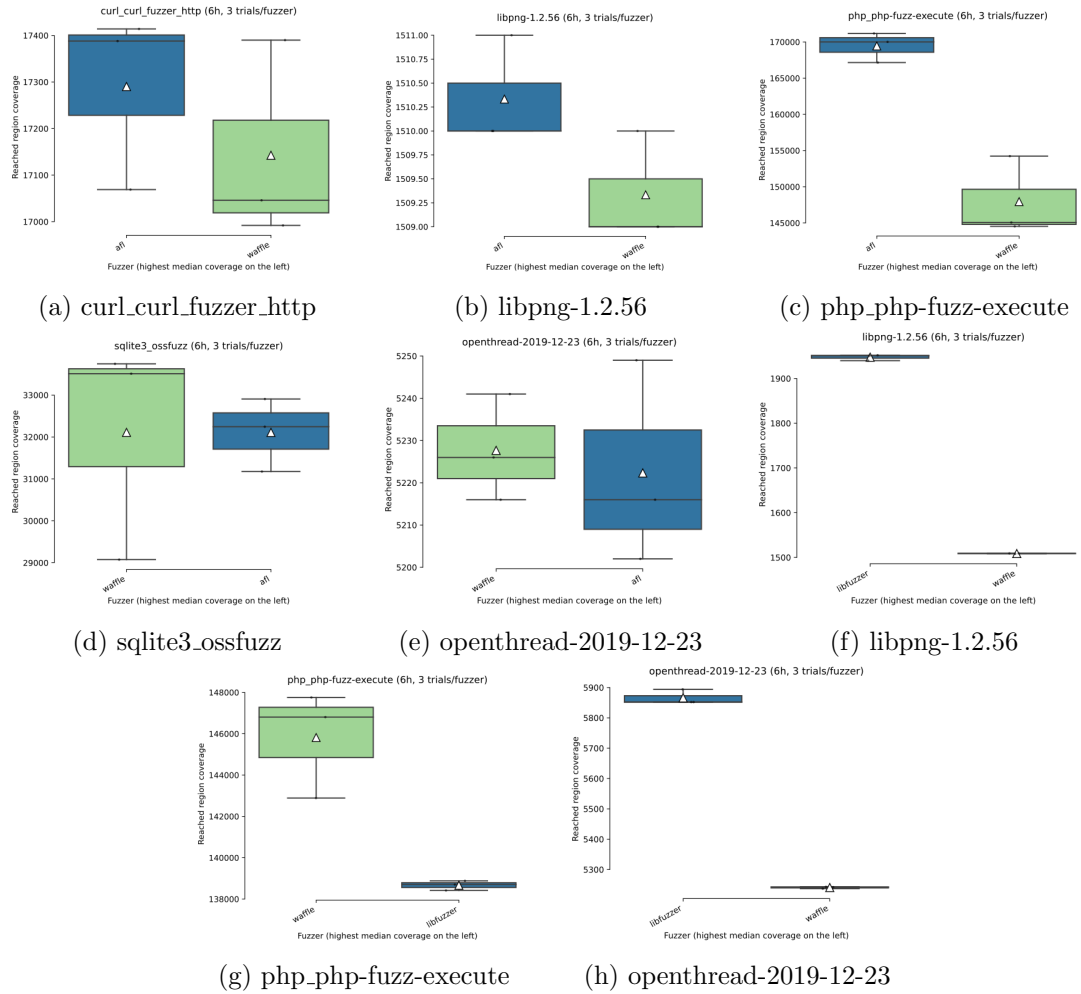
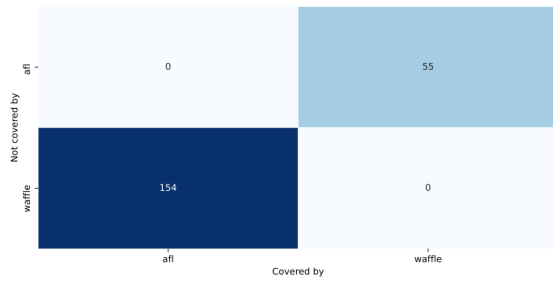
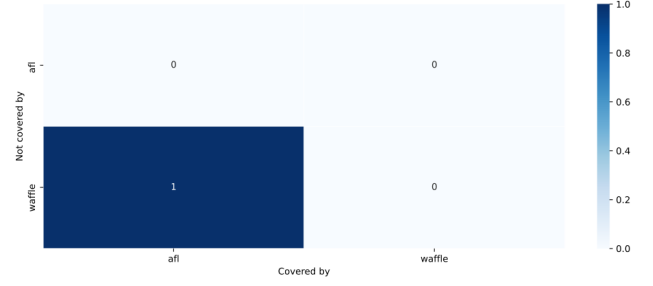


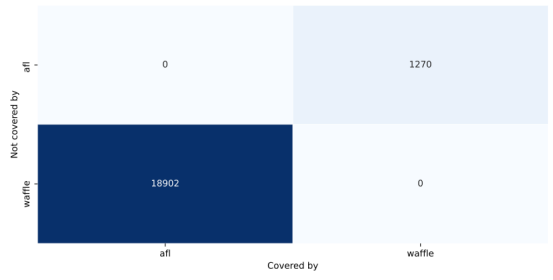
Figure 4.3: The above figures displays the reached coverage in the end of the experiments. The green boxes represent Waffle’s code-coverage.



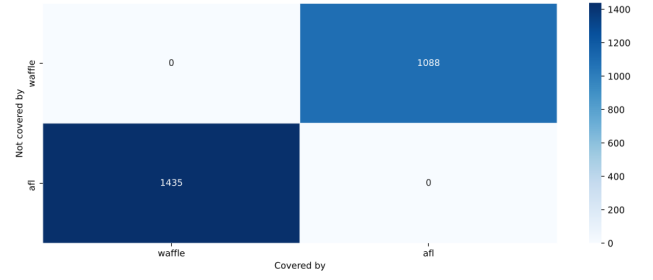
(a) curl.curl_fuzzer_http



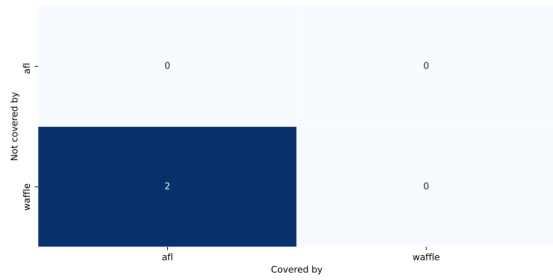
(b) libpng-1.2.56



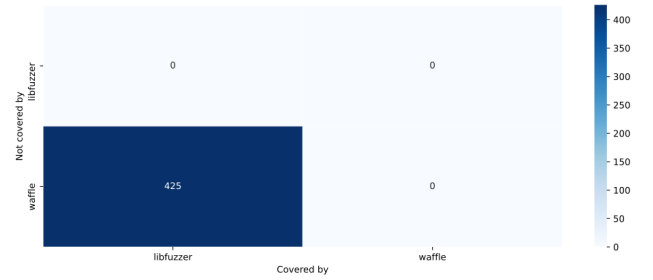
(c) php_php-fuzz-execute



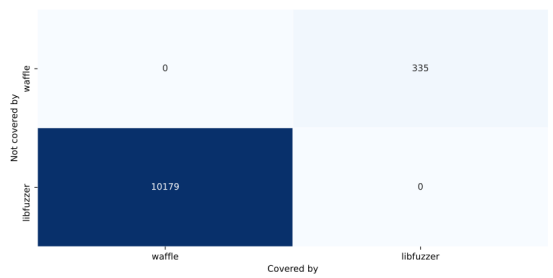
(d) sqlite3_ossfuzz



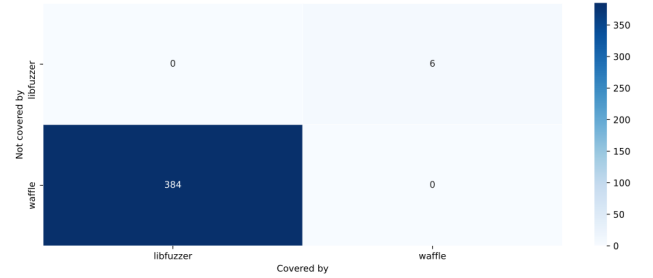
(e) openthread-2019-12-23



(f) libpng-1.2.56



(g) php_php-fuzz-execute



(h) openthread-2019-12-23

Figure 4.4: The above figures displays the reached coverage in the end of the experiments. The green boxes represent Waffle's code-coverage.

Bibliography

- [1] OmniSci. What is llvm. <https://www.omnisci.com/technical-glossary/llvm>, 2020. [Online]; accessed in 2020.
- [2] The Open Web Application Security Project. <https://www.owasp.org>, 2019.
- [3] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.
- [4] 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.
- [5] Michal Zalewski. Pulling jpegs out of thin air. <https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html>, 2014.
- [6] Finding memory bugs with addresssanitizer. <https://embeddedbits.org/finding-memory-bugs-with-addresssanitizer/>, 2020.
- [7] Google/addresssanitizer. <https://github.com/google/sanitizers/wiki/AddressSanitizer>, 2019.
- [8] 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.

- [9] 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.
- [10] Greg Banks, Marco Cova, Viktoria Felmetzger, 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.
- [15] 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.

- [16] 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.
- [17] 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.
- [18] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based white-box fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.
- [19] High-performance binary-only instrumentation for afl-fuzz. https://github.com/mirrorer/afl/blob/master/qemu_mode/README.qemu, 2020.
- [20] 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.
- [21] Qian Yang, J Jenny Li, and David M Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009.
- [22] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] 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. ICSE, 2020.
- [27] LLVM. Llm project. <http://llvm.org/>, 2020. [Online]; accessed in 2020.
- [28] Chris Lattner and Vikram Adve. Llm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [29] LLVM. Clang: a c language family frontend for llm. <https://clang.llvm.org/>, 2020. [Online]; accessed in 2020.
- [30] Base class for instruction visitors. https://llvm.org/doxygen/InstVisitor_8h_source.html, 2021.
- [31] Michal Zalewski. American fuzzy lop.(2014). <http://lcamtuf.coredump.cx/afl>, 2019.
- [32] More about afl. https://afl-1.readthedocs.io/en/latest/about_afl.html, 2019.
- [33] american fuzzy lop - a security-oriented fuzzer. <https://github.com/google/AFL>, 2020.

- [34] Afl user guide. https://afl-1.readthedocs.io/en/latest/user_guide.html, 2019.
- [35] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.
- [36] add nosanitize metadata to more coverage instrumentation instructions. <https://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20150302/263798.html>, 2015.
- [37] source of afl-clang-fast.c. https://github.com/google/AFL/blob/master/llvm_mode/afl-clang-fast.c, 2021.
- [38] Waffle project. <https://github.com/behnamarbab/memlock-waffle/tree/waffle>, 2021.
- [39] K Serebryany. libfuzzer a library for coverage-guided fuzz testing. *LLVM project*, 2015.
- [40] László Szekeres Jonathan Metzman, Abhishek Arya, and L Szekeres. Fuzzbench: Fuzzer benchmarking as a service. *Google Security Blog*, 2020.
- [41] Local experiment issue - erro[0612]. <https://github.com/google/fuzzbench/issues/946>, 2021.
- [42] Requesting an experiment. <https://google.github.io/fuzzbench/getting-started/adding-a-new-fuzzer/#requesting-an-experiment>, 2021.
- [43] Setting up gcp. <https://google.github.io/fuzzbench/running-a-cloud-experiment/setting-up-a-google-cloud-project>, 2021.

Chapter 5

Appendix

5.A Waffle

5.A.1 random_havoc

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

```
1 havoc_stage:
2  /* The havoc stage mutation code is also invoked when splicing
   files; if the splice_cycle variable is set, generate different
   descriptions and such. */
3
4  if (!splice_cycle) {
5      stage_max = (doing_det ? HAVOC_CYCLES_INIT : HAVOC_CYCLES) *
6                  perf_score / havoc_div / 100;
7  }
8  else {
9      stage_max = SPLICE_HAVOC * perf_score / havoc_div / 100;
10 }
11
12 /* We essentially just do several thousand runs (depending on
   perf_score) where we take the input file and make random stacked
   tweaks. */
13 for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {
14     u32 use_stacking = 1 << (1 + UR(HAVOC_STACK_POW2));
15     for (i = 0; i < use_stacking; i++) {
16         switch (UR(15 + ((extras_cnt + a_extras_cnt) ? 2 : 0))) {
17             case 0:
18                 /* Flip a single bit somewhere. Spooky! */
```

```

19         case 1:
20             /* Set byte to interesting value. */
21         case 2:
22             /* Set word to interesting value, randomly choosing endian
23             . */
24         case 3:
25             /* Set dword to interesting value, randomly choosing
26             endian. */
27         case 4:
28             /* Randomly subtract from byte. */
29         case 5:
30             /* Randomly add to byte. */
31         case 6:
32             /* Randomly subtract from word, random endian. */
33         case 7:
34             /* Randomly add to word, random endian. */
35         case 8:
36             /* Randomly subtract from dword, random endian. */
37         case 9:
38             /* Randomly add to dword, random endian. */
39         case 10:
40             /* Just set a random byte to a random value. Because, why
41             not. We use XOR with 1-255 to eliminate the possibility of a no-
42             op. */
43         case 11 ... 12:
44             /* Delete bytes. We're making this a bit more likely than
45             insertion (the next option) in hopes of keeping files reasonably
46             small. */
47         case 13:
48             /* Clone bytes (75%) or insert a block of constant bytes
49             (25%). */
50         case 14:
51             /* Overwrite bytes with a randomly selected chunk (75%) or
52             fixed bytes (25%). */
53
54             /* Values 15 and 16 can be selected only if there are any
55             extras present in the dictionaries. */
56         case 15:
57             /* Overwrite bytes with an extra. */
58         case 16:
59             /* Insert an extra. Do the same dice-rolling stuff as for
60             the previous case. */
61     }
62 }
63
64 /* Write a modified test case, run program, process results.
65 Handle error conditions, returning 1 if it's time to bail out.
66 This is a helper function for fuzz_one(). */
67 }

```

Listing 5.1: Random havoc stage

5.B FuzzBench

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

5.B.1 builder.Dockerfile

Builds Waffle for the usage in FuzzBench.

The `parent_image` is an image instance with primitive 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`.

```
1 ARG parent_image
2 FROM $parent_image
3
4 RUN apt-get clean
5 RUN apt-get update --fix-missing
6 RUN apt-get -y install wget git build-essential software-properties-
   common apt-transport-https --fix-missing
7
8 # The llvm we are looking for
9 RUN wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key | apt-key
   add -
10 RUN add-apt-repository ppa:ubuntu-toolchain-r/test && echo $(gcc -v)
11 RUN apt-add-repository "deb http://apt.llvm.org/xenial/ llvm-
   toolchain-xenial-6.0 main" -y
12 RUN apt-get update
13 RUN apt-get install -y gcc-7 g++-7 clang llvm
14
15 # Clone and build the repository for Waffle
16 RUN git clone -b waffle --single-branch https://github.com/
   behnamarbab/memlock-waffle.git /source_files
17 RUN cd /source_files/waffle && \
18     unset CFLAGS CXXFLAGS && \
19     AFL_NO_X86=1 make && \
20     cd llvm_mode && make
21
22 # Install the driver for communicating with FuzzBench
23 RUN wget https://raw.githubusercontent.com/llvm/llvm-project/5
   feb80e748924606531ba28c97fe65145c65372e/compiler-rt/lib/fuzzer/
   afl/afl_driver.cpp -O /source_files/afl_driver.cpp && \
24     cd /source_files/waffle && unset CFLAGS CXXFLAGS && \
25     clang -Wno-pointer-sign -c /source_files/waffle/llvm_mode/afl-
   llvm-rt.o.c -I/source_files/waffle/ && \
26     clang++ -stdlib=libc++ -std=c++11 -O1 -c /source_files/
   afl_driver.cpp && \
27     ar r /libAFL.a *.o
```

Listing 5.2: Recipe for building Waffle FuzzBench

5.B.2 fuzzer.py

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

```
1  """Integration code for Waffle fuzzer."""
2
3  import json
4  import os
5  import shutil
6  import subprocess
7
8  from fuzzers import utils
9
10
11 def prepare_build_environment():
12     """Set environment variables used to build targets for AFL-based
13     fuzzers."""
14     cflags = ['-fsanitize-coverage=trace-pc-guard']
15     utils.append_flags('CFLAGS', cflags)
16     utils.append_flags('CXXFLAGS', cflags)
17
18     os.environ['CC'] = 'clang'
19     os.environ['CXX'] = 'clang++'
20     os.environ['FUZZER_LIB'] = '/libAFL.a'
21
22
23 def build():
24     """Build benchmark."""
25     prepare_build_environment()
26
27     utils.build_benchmark()
28
29     print('[post_build] Copying waffle-fuzz to $OUT directory')
30     # Copy out the waffle-fuzz binary as a build artifact.
31     shutil.copy('/source_files/waffle/waffle-fuzz', os.environ['OUT',
32 ])
33
34 def get_stats(output_corpus, fuzzer_log): # pylint: disable=unused-
35     argument
36     """Gets fuzzer stats for Waffle."""
37     # Get a dictionary containing the stats Waffle reports.
38     stats_file = os.path.join(output_corpus, 'fuzzer_stats')
39     with open(stats_file) as file_handle:
40         stats_file_lines = file_handle.read().splitlines()
41         stats_file_dict = {}
42         for stats_line in stats_file_lines:
43             key, value = stats_line.split(': ')
```

```

43         stats_file_dict[key.strip()] = value.strip()
44
45     # Report to FuzzBench the stats it accepts.
46     stats = {'execs_per_sec': float(stats_file_dict['execs_per_sec'
47 ])}
48     return json.dumps(stats)
49
50 def prepare_fuzz_environment(input_corpus):
51     """Prepare to fuzz with AFL or another AFL-based fuzzer."""
52     # Tell AFL to not use its terminal UI so we get usable logs.
53     os.environ['AFL_NO_UI'] = '1'
54     # Skip AFL's CPU frequency check (fails on Docker).
55     os.environ['AFL_SKIP_CPUFREQ'] = '1'
56     # No need to bind affinity to one core, Docker enforces 1 core
57     usage.
58     os.environ['AFL_NO_AFFINITY'] = '1'
59     # AFL will abort on startup if the core pattern sends
60     notifications to
61     # external programs. We don't care about this.
62     os.environ['AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES'] = '1'
63     # Don't exit when crashes are found. This can happen when corpus
64     from
65     # OSS-Fuzz is used.
66     os.environ['AFL_SKIP_CRASHES'] = '1'
67
68     # AFL needs at least one non-empty seed to start.
69     utils.create_seed_file_for_empty_corpus(input_corpus)
70
71 def run_waffle_fuzz(input_corpus,
72                     output_corpus,
73                     target_binary,
74                     additional_flags=None,
75                     hide_output=False):
76     """Run the fuzzer"""
77     # Spawn the waffle fuzzing process.
78     print('[run_waffle_fuzz] Running target with waffle-fuzz')
79     command = ['./waffle-fuzz', '-i', input_corpus, '-o',
80 output_corpus, '-d', '-m', 'none', '-t', '1000']
81     if additional_flags:
82         command.extend(additional_flags)
83     dictionary_path = utils.get_dictionary_path(target_binary)
84     if dictionary_path:
85         command.extend(['-x', dictionary_path])
86     command += [
87         '--',
88         target_binary,
89         # Pass INT_MAX to afl the maximize the number of persistent
90         loops it
91         # performs.
92         '2147483647'
93     ]
94     print('[run_waffle_fuzz] Running command: ' + ' '.join(command))

```

```

91     output_stream = subprocess.DEVNULL if hide_output else None
92     subprocess.check_call(command, stdout=output_stream, stderr=
output_stream)
93
94
95 def fuzz(input_corpus, output_corpus, target_binary):
96     """Run waffle-fuzz on target."""
97     prepare_fuzz_environment(input_corpus)
98
99     run_waffle_fuzz(input_corpus, output_corpus, target_binary)

```

Listing 5.3: Recipe for running fuzzing with Waffle on a target