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

THE UNIVERSITY OF NEW BRUNSWICK

Soon...!

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

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 theis 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

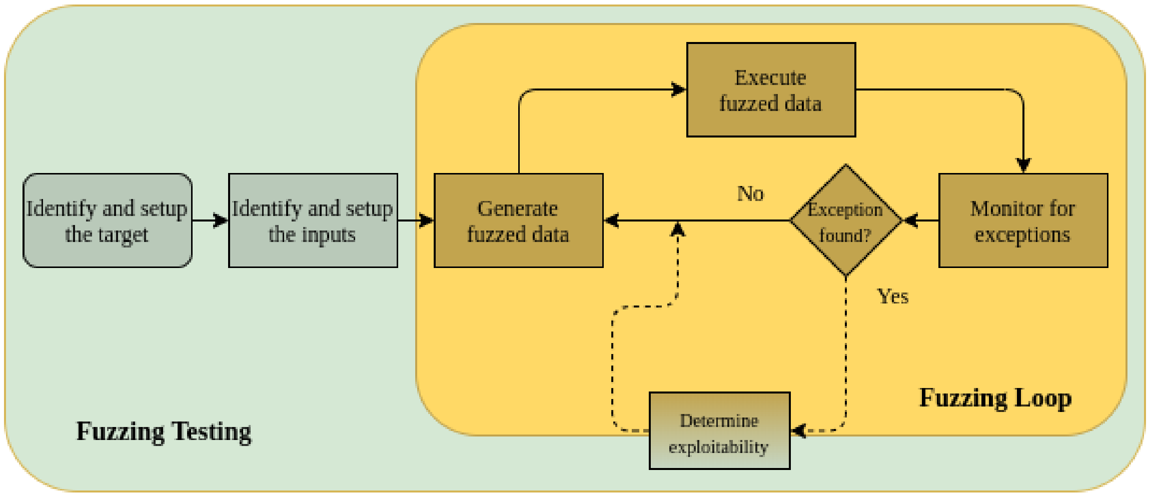


Figure 2.1: Fuzzing phases

exception is the primary purpose of this stage.

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]

```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 2.1: 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
```

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. Instrumeting 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 - WAAAFL) 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 structure of the LLVM project is shown in Figure 2.2:

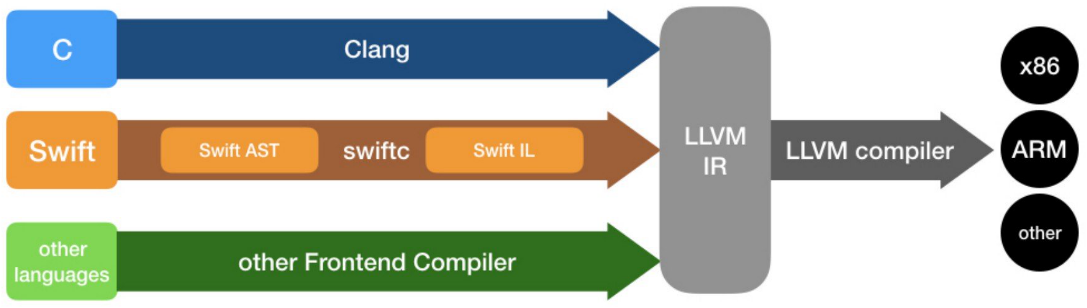


Figure 2.2: LLVM architecture: A front-end compiler generates the LLVM IR, and then it is converted into machine code [1]

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 2.2:

```
// 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:

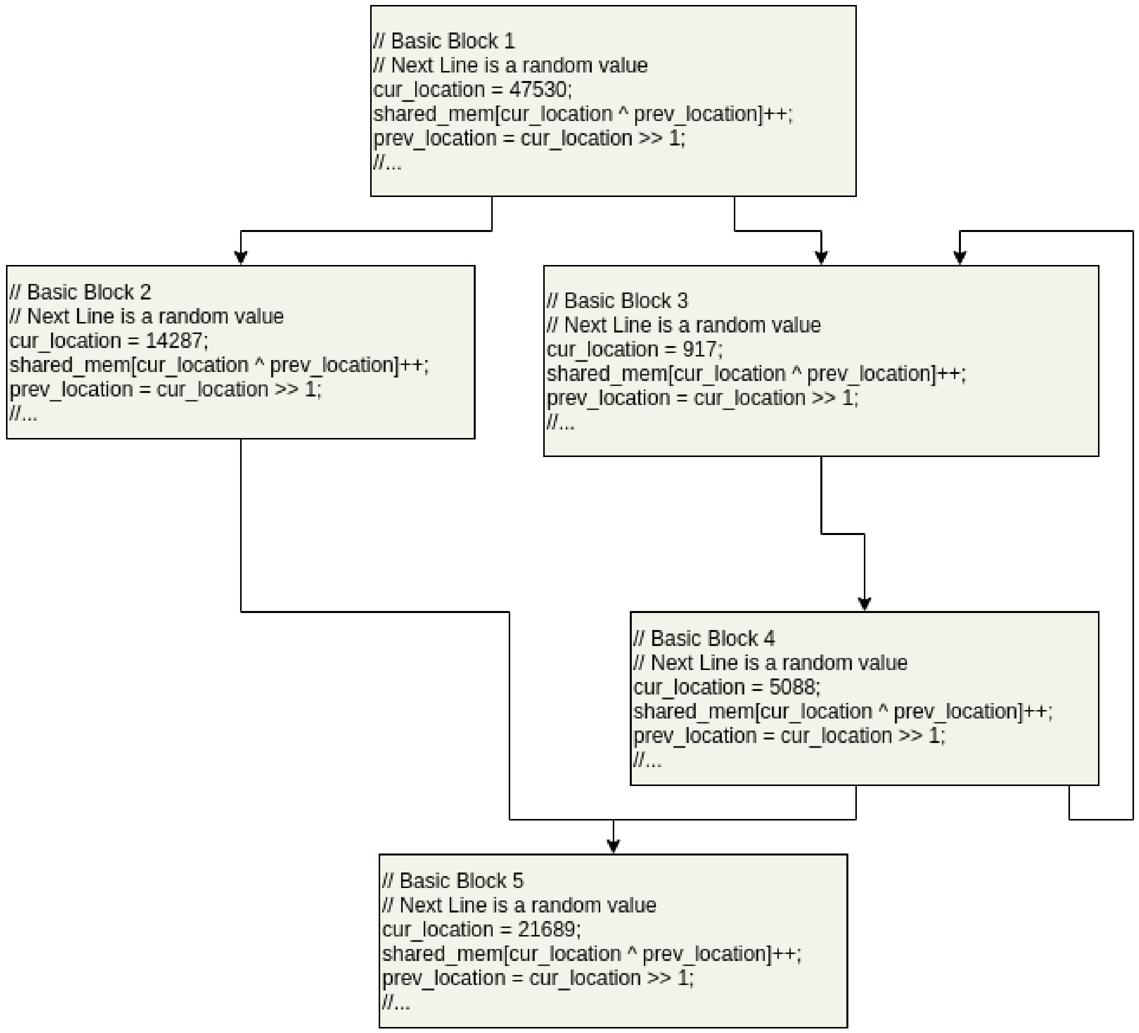
```
  cur_location = <COMPILE_TIME_RANDOM>;
  shared_mem[cur_location ^ prev_location]++;
  prev_location = cur_location >> 1;
```

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 2.3, 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.



```
// 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 2.3: Example for instrumented basic blocks

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]"

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

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

Listing 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.

2.4 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 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 2.5: 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 2.6: Execute AFL

An execution of AFL fuzzing occurs after validating the binary for fuzz testing. The Algorithm 1 illustrates 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     $fuzz\_one(Entry)$;
```

In the fuzzing loop, AFL collects the queue entries and updates the bitmaps for the new entries. Then the first element of the queue is selected for further fuzzing. We demonstrate the pseudocode for fuzzing an entry in Algorithm 2:

Algorithm 2: $fuzz\_one$: Fuzz one Entry

```
Input: $queueEntry$
1 $test\_case \leftarrow Entry.test\_case\ calibrate(test\_case)$;
2 $bitflip(test\_case)$;
3 $save\_if\_interesting(test\_case)$;
4 $random\_havoc(test\_case)$;
```

It is noticeable that any newly generated input is executed once before any evaluations.

Generate fuzzed data

AFL generates new inputs after selecting the **favored inputs**, which are more likely to be selected for more fuzzing.

- **Favored inputs**: After an input is executed, its favor factor is calculated, and later, if it is still interesting for AFL, it is marked as Favored. 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." [32]

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

  The preference of AFL for the favored inputs increases the performance of AFL in finding new paths. A higher $favored\_factor$ represents a faster execution on smaller inputs.

- **Mutation strategies**: To generate new inputs, AFL takes a queue of inputs and tries mutating and running each one of them. The mutation strategies are in a queue of different strategies that are run on an input to generate more inputs eventually. These strategies include bit-fliping, byte-fliping, simple arithmetic, known integers, stack tweaks, and test case splicing. [33]

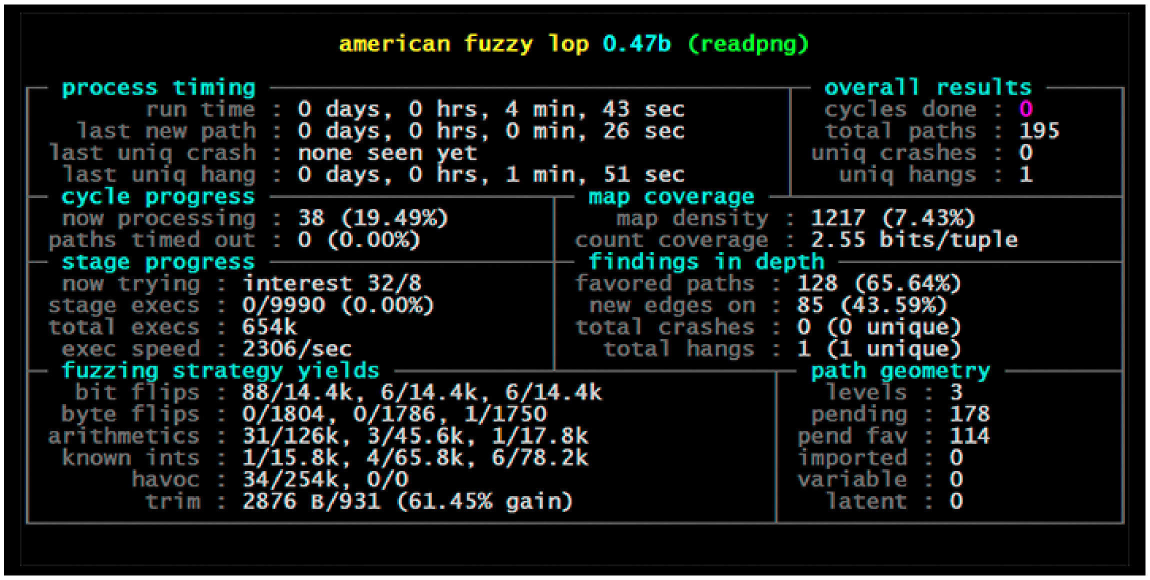
Status screen

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.



```
                  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 2.4: AFL status screen

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.

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.

2.5 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 *WAAAFL*!

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 [34] 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):

```c
// 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**:

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

This way, we can count the number of memory copies in an execution. The `CountAll-Visitor` structure keeps track of the instructions that LLVM considers them as memory copies.

Now we can insert our instructions to the basicblocks:

```
 1 // snippet of wafl-llvm-pass.so.cc
 2 #include <math.h>
 3 // ..
 4
 5 bool WAFLCoverage::runOnModule(Module &M) {
 6   // ...
 7   GlobalVariable *WAFLMapPtr =
 8       new GlobalVariable(M, PointerType::get(Int8Ty, 0), false,
 9       GlobalValue::ExternalLinkage, 0, "__wafl_area_ptr");
10   // ..
11
12   for (auto &F : M) {
13     for (auto &BB : F) {
14
15       // ...
16
17       LoadInst *IcntPtr = IRB.CreateLoad(WAFLMapPtr);
18       MapPtr->setMetadata(M.getMDKindID("nosanitize"),
19         MDNode::get(C, None));
20
21       Value* EdgeId = IRB.CreateXor(PrevLocCasted, CurLoc);
22       Value *IcntPtrIdx =
23           IRB.CreateGEP(IcntPtr, EdgeId);
24
25       /* Count the instructions */
26       CountAllVisitor CAV;
27       CAV.visit(BB);
28
29       /* Setup the counter for storage */
30       u8 log_count = (u8) log2(CAV.Count);
31       Value *CNT = IRB.getInt8(log_count);
32
33       LoadInst *IcntLoad = IRB.CreateLoad(IcntPtrIdx);
34       Value *IcntIncr = IRB.CreateAdd(IcntLoad, CNT);
35
36       IRB.CreateStore(IcntIncr, IcntPtrIdx)
37         ->setMetadata(M.getMDKindID("nosanitize"),
38         MDNode::get(C, None));
39
40       inst_blocks++;
41     }
42   }
43 }
```

Listing 3.3: LLVM-mode instrumentation pass

In Line 7, we locate the shared bitmap. Lines 16 loads the pointer to the bitmap and configures the meta data for storage [35].

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 to collect coverage information, and at the same time, maximise the number of instructions in executions. The lines 21 to 23 loads the pointer to the appropriate location on the bitmap.

In each basicblock, the visitors look for the **copies**. In different executions, we noticed that the number of instructions increases so fast, and to control this number, we calculate its log (Lines 30-31):

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

Now that we have calculated `CNT`, we load the pointer on the bitmap and add `CNT` to the content of the pointer. Waffle stores the result of the addition in the same pointer. (Lines 36-39)

Applying the instrumentation

3.4 Concluding remarks

Chapter 4

Simulation

This chapter presents the details of our implementation of Waffle.

We implement and evaluate the performance of Waffle on an Ubuntu 18.4.5 LTS operating system. The computer consists of 16GB of RAM and Intel® Core™ i7-3770 CPU.

In the following sections, we will explain the implementation and the execution of Waffle in more detail.

4.1 Implementation

We have implemented Waffle on Memlock, which is based on AFL fuzzer. To develop Waffle, we added and modified approximately 400 lines of code in the C programming language. We also implemented the instrumentations under the LLVM framework.

4.2 Instrumentation

To inject code into the binary of the target program, we modify the two files *waffle-llvm-rt.o.c* and *waffle-llvm-pass.so.cc*. The first file is responsible for the initial setup for the SHM and fork server. The later file injects the basic-block level instrumentation, which contains our feature collection procedure.

In the file *waffle-llvm-rt.o.c* first, we specify an array of 64KB, in addition to the shared memory implemented in Memlock. This array collects the instruction counters and monitors the execution of the program through each basic-block.

```
    u32   __wafl_icnt_initial[ICNT_SIZE];
    u32*  __wafl_icnt_ptr = __wafl_icnt_initial;
```

Listing 4.1: waffle-llvm-rt.o.c

Here $ICNT\_SIZE$ is equal to 2^{14} words, which makes the size of 64KB.

In the file *waffle-llvm-pass.so.cc* we implement the LLVM pass, which helps us with injecting basic-block level instructions. For our purposes, we are using the instruction visitor, which we explained in Chapter 2.

After the above modifications on Memlock, we can *make* the LLVM project within Waffle. Our improvements will be applied in the *waffle-clang*, which we can use for compiling the target program.

To compile a source code to an executable, we specify the path to *waffle-clang* as the compiler for building executable target programs. For instance, if the program contains a single source file, we can use the following command:

```
./waffle-clang -i <sourcecode-path> -o <executable-path>
```

Listing 4.2: Compile a single file using waffle-clang

Choosing the right compiler here is necessary as *waffle-clang/waffle-clang++* injects the proposed instrumentation. Generally, we can define the path to the compiler by exporting it to the environment variables:

```
export CC=waffle-clang
export CXX=waffle-clang++
```

Listing 4.3: Compile using waffle-clang

Waffle uses the resulting executables for its fuzzing procedure.

4.3 Start fuzzing

Waffle takes the current entry from the queue, fuzzes it for a while, and when finished, turns back to the queue for another entry.

To use the instrumentation features, Waffle participates in sharing memory with the target program. The instantiated shared memory will be passed to the target program every time the fuzzer executes the program.

After the execution of the target program, collected data are stored in three arrays, $trace\_bits$, $perf\_bits$, and $icnt\_bits$. These arrays contain the information for coverage, stack memory consumption, and the counters for each basic block's instructions. The total size of the shared memory is 128KB.

Memlock calculates a *stallness* for measuring the performance of the fuzzer. If the behavior of execution of input is too stall, Memlock skips fuzzing the current input entry and continues with the next entry of the queue. In Waffle, we do not intend to measure stallness, and we switch between fuzzing approaches, as mentioned before.

AFL trims the test-cases to increase the performance of coverage-finding techniques. As trimming the results may affect the target program's resource consumption, Memlock and Waffle disable this method in the fuzzing stage.

Next, Waffle assesses its interest in considering the input as a favorable input.

```
total_counts = 0;
for (i = 0; i < ICNT_SIZE; i++) {
  if (icnt_bits[i]) {
    total_counts += icnt_bits[i];
    if (top_rated[i]) {
      if (icnt_bits[i] < max_counts[i]) continue;
    }
    /* Insert ourselves as the new winner. */
    top_rated[i] = q;

    /* if we get here, we know that icnt_bits[i]==max_counts[i] */
    score_changed = 1;
  }
}
if(total_counts >= max_total_counts){
  top_rated[i] = q;
  score_changed = 1;
  }
```

Listing 4.4: Update bitmap scores

Here $ICNT\_SIZE$ is equal to the size of the bitmap for collecting the instruction counters. In case we find a new max for the number of instructions in a basic-block, we select the current input as the winner - favorable. [listing 4.4]

We are also selecting inputs with a total number of instructions more than any other input that was executed before.

```
if (top_rated[i]) {
  /* if top rated for any i, will be favored */
  u8 was_favored_already = top_rated[i]->favored;

  top_rated[i]->favored = 1;

  /* increments counts only if not also favored for another i */
  if (!was_favored_already){
    queued_favored++;
    if (!top_rated[i]->was_fuzzed) pending_favored++;
  }
}
```

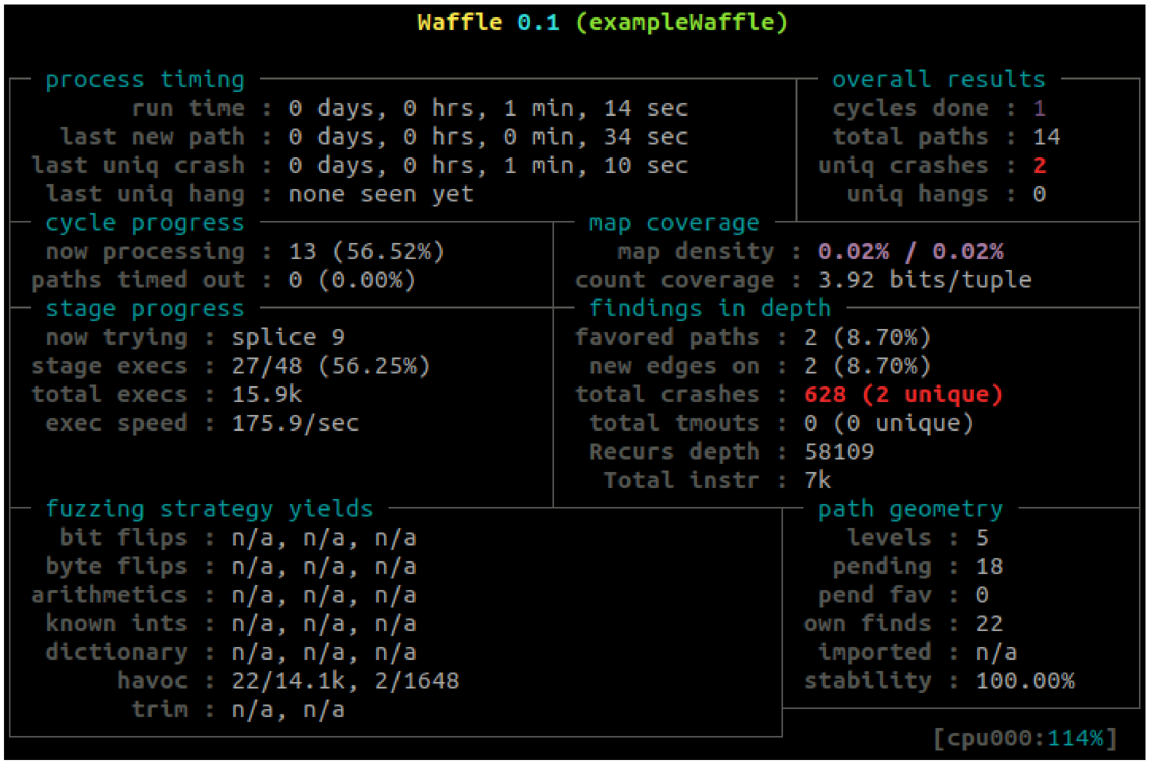
Listing 4.5: Cull queue

After collecting the execution features for an input, Waffle continues culling the queue and selects unique favorite inputs for the next generations. [listing 4.5]

Notice that AFL and Memlock do the same method for selecting favorite inputs, except that they do not consider an overall execution feature, which Waffle knows as max_total_counts.

4.4 Monitoring fuzzing procedure

To measure the performance of AFL, we can use the live status screen.



```
                         Waffle 0.1 (exampleWaffle)

┌─ process timing ─────────────────────────────────────┬─ overall results ──────┐
│        run time : 0 days, 0 hrs, 1 min, 14 sec       │  cycles done : 1       │
│   last new path : 0 days, 0 hrs, 0 min, 34 sec       │  total paths : 14      │
│ last uniq crash : 0 days, 0 hrs, 1 min, 10 sec       │ uniq crashes : 2       │
│  last uniq hang : none seen yet                      │   uniq hangs : 0       │
├─ cycle progress ───────────────────┬─ map coverage ──┴────────────────────────┤
│  now processing : 13 (56.52%)      │    map density : 0.02% / 0.02%           │
│ paths timed out : 0 (0.00%)        │ count coverage : 3.92 bits/tuple         │
├─ stage progress ───────────────────┼─ findings in depth ──────────────────────┤
│  now trying : splice 9             │ favored paths : 2 (8.70%)                │
│ stage execs : 27/48 (56.25%)       │  new edges on : 2 (8.70%)                │
│ total execs : 15.9k                │ total crashes : 628 (2 unique)           │
│  exec speed : 175.9/sec            │  total tmouts : 0 (0 unique)             │
│                                    │  Recurs depth : 58109                    │
│                                    │   Total instr : 7k                       │
├─ fuzzing strategy yields ──────────┴─────────────┬─ path geometry ────────────┤
│   bit flips : n/a, n/a, n/a                      │    levels : 5              │
│  byte flips : n/a, n/a, n/a                      │   pending : 18             │
│ arithmetics : n/a, n/a, n/a                      │  pend fav : 0              │
│  known ints : n/a, n/a, n/a                      │ own finds : 22             │
│  dictionary : n/a, n/a, n/a                      │  imported : n/a            │
│       havoc : 22/14.1k, 2/1648                   │ stability : 100.00%        │
│        trim : n/a, n/a                           ├────────────────────────────┘
└──────────────────────────────────────────────────┘             [cpu000:114%]
```

Figure 4.1: Status screen

In the above screen, the total number of instructions executed and the maximum depths for the stack is presented. Eventually, as the

4.5 Summary

In this chapter, we covered the following topics:

- The implementation of the instrumentation
- The modifications in the fuzzing procedure
- We also improved the status screen to illustrate the details of the fuzzing procedure and for future assessments

Chapter 5

Results and Evaluation

Chapter 6

Conclusions and Future Work

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 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.

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

[28] 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.

[29] LLVM. Clang: a c language family frontend for llvm. `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] american fuzzy lop - a security-oriented fuzzer. `https://github.com/google/AFL`, 2020.

[33] Michal Zalewski. Binary fuzzing strategies: what works, what doesn't. `https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html`, 2014.

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

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

Glossary (if any)

start writing here.

Vita

Candidate's full name:
University attended (with dates and degrees obtained):
Publications:
Conference Presentations: