

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 AFL	7
2.3.1 Stages of fuzzing with AFL	8
2.4 LLVM	13
2.4.1 Instrumentation and coverage measurements	14
2.5 Concluding remarks	16
3 Proposed Fuzzer	17

3.1	Introduction	17
3.2	Motivating example	18
3.3	Instrumentation	18
3.3.1	Features	18
3.3.1.1	Memory consumption	18
3.3.1.2	Instruction counters	19
3.4	Fuzzing the target	21
3.5	Concluding remarks	21
4	Simulation	23
4.1	Target program and instrumentation	23
4.2	Program inputs and setups	23
4.3	Generating fuzzed data	23
4.4	Program execution and monitoring	23
4.5	Report vulnerabilities	23
5	Results and Evaluation	24
5.1	FuzzBench	24
5.2	results	24
6	Conclusions and Future Work	25
	Bibliography	26
A	Handling stack operations in Memlock	30
	Glossary	33
	Vita	

List of Tables

List of Figures

2.1	Fuzzing phases	3
2.2	AFL status screen	11
2.3	LLVM architecture: A front-end compiler generates the LLVM IR, and then it is converted into machine code [1]	13
2.4	Example for instrumented basic blocks	15
3.1	Memlock’s approach	22

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]

Since the growth of computer programs conquered all the globe, the profit of compromising the programs grew as well.

2.2 Literature Review

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

1. Generally speaking, the first stage is to identify the target. A target may be a software or a combination of both software and hardware [4]. The targeted software may be the software itself, or any part of the software, for instance, a library or an API. In the rest of this article, our target is software.
2. To execute the target program, we need to specify how the program parses the inputs. For instance, the target program may take several command-line

arguments as the input, or the arguments are file locations for reading data. Environmental variables, file formats, and any other variables that may change the program's execution are all considered an input, and a fuzzer should choose them for the execution of the target program. In any case, the fuzzer needs to know the data that would cause a vulnerability.

A fuzzer may need a set of seeds for initialization. This set could be empty, and a fuzzer without any sample inputs may find some valid files out of thin air [5] that are acceptable by the target program.

3. The generation of fuzzed data is the beginning of the fuzz testing loop. The target program needs a queue of inputs for testing the execution. New inputs are generated either by a dumb mutation of the content of the input, or by generating new content based on some heuristics. The prior procedure is part of a mutation-based fuzzing, and a generation-based fuzz testing uses the later. Waffle is a mutation-based fuzz testing platform based on AFL. In chapter 3, we will discuss more about how a mutation-based dumb fuzzer works intelligently.
4. A fuzzer takes the processed inputs and executes the target program using the provided inputs. Depending on the resources needed for the execution of the

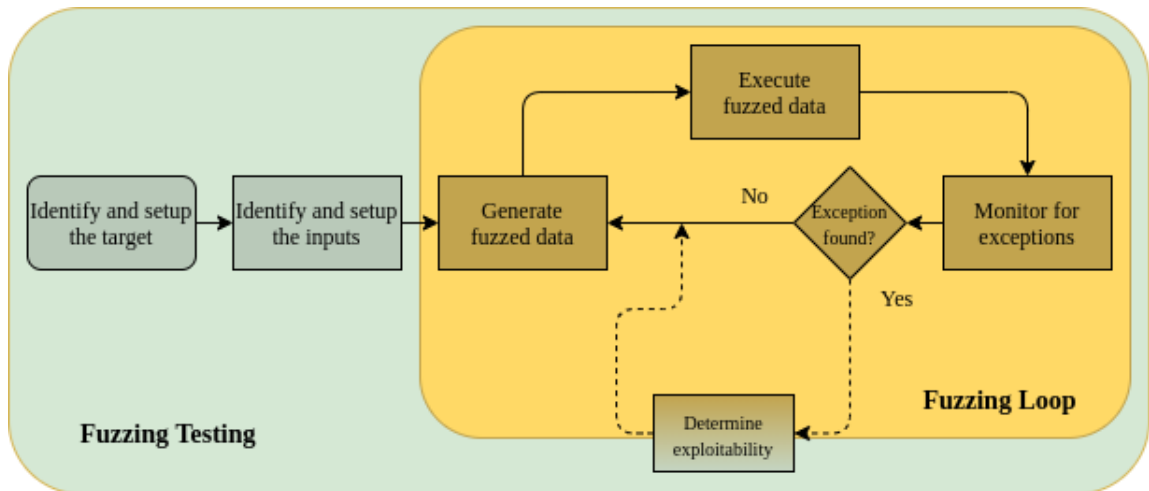


Figure 2.1: Fuzzing phases

target program, this stage can be the bottleneck for the procedure of fuzzing. Generally, the output of this stage is any detected misbehavior during the execution of inputs.

5. If an exceptional event happens during the execution of the program that prevents the program from exiting successfully, we say an exception has occurred. It is expected for a program to finish its execution by returning a success value, 0 for most machines. If any value other than the success value (0) is returned, These exceptions are the vulnerabilities that may be exploitable and may cause security problems. Handling an exception properly and pinpointing the input (or probably part of the content of the input) responsible for the exception is the primary purpose of this stage of fuzz testing.
6. A vulnerability may be harmless! Suppose we find an exception (vulnerability) in a module not handled within the program, and the operating system catches it. If another program uses this module, and the program is first validating the input for the module, then the domain of inputs for this module is constrained and filtered by the program, and the vulnerability may never occur. If we can find an exploitable vulnerability, the fuzzer must detect it and report it as an exploitable vulnerability. The generation phase may need this report for later usage.

The introduced fuzzer by Miller [6] was of the very first naive blackbox fuzzer 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 content of the inputs, and the target program is a blackbox for the fuzzer. As a result, this fuzzer is a mutation-based blackbox fuzzer.

A blackbox fuzzer randomly generates a different stream of inputs and does not decide how to choose those generations to enhance the performance of the fuzzer. Another downside of this fuzzer (and other blackbox fuzzers) is when the program faces some branches with magic values, constraining the variable to a specific set of values; the fuzzer has to apply the exact values for the constraining statement which may have a very low probability - it is almost impossible to generate a specific 1 KB string of bytes by chance. Nevertheless, when the target application is known, the performance of the blackbox fuzzing technique can be increased drastically. In [7] and [8] a set of network protocols are fuzzed as blackbox. Any application on the web may be considered as a blackbox program as well, so as [9] and [10] have targeted web applications and found ways to attack some the websites, looking for different vulnerabilities, such as XSS.

SAGE, a whitebox fuzzer was developed as an alternative to blackbox fuzzing, to cover the lacks of blackbox fuzzers [11]. A fuzzer is a whitebox fuzzer if we have the source code of the target program, and the fuzzer can use the information within the source code. 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 in the program [12]. It can also use dynamic, and concolic execution [13] and use taint analysis to locate the regions of seed files influencing values used by program [14]; whitebox fuzzing can also find the grammar for parsing the input files without any prior knowledge [15]. These techniques help in finding code coverage and a more intelligent search for vulnerabilities.

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 for the program, which can be helpful for the fuzzing. A handy feature for an execution is the code coverage of the run. A coverage-based greybox fuzzer keeps track of the different inputs with different coverages and may generate inputs for the next generations based on the new code coverages found in each iteration. To detect the coverage, we need an instrumentation [16] that we will explain more in section Instrumentation and coverage measurements.

Coverage-based fuzz testing benefits from instrumentation for discovering the parts of the (binary) code that are covered during the executions. Some criteria, such as data coverage, statement coverage, block coverage, decision coverage, and path coverage, can be used to investigate the overall code coverage. [17] Bohme et al. [18] introduced a coverage-based greybox fuzzer that extends AFL and benefits from the Markov Chain model. The fuzzer looks for the paths that were covered using seeds, and when a new path is seen, it is stored accordingly. Then it reports a probability that after fuzzing an input with path i , it generates an input with path j . The fuzzer chooses a new collection of seeds to fuzz based on the energies assigned to each seed; it generates more inputs from a seed with higher energy.

In another article by Bohme et al. [19], 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 fuzzing approach are patch testing and crash reproduction, which have different use cases compared to non-directed coverage-based fuzzers.

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 input, and the execution time of the target program. PerfFuzz [20] is another 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 exploring for more coverage. PerfFuzz counts how many times each of the edges of the control flow graph (CFG) is executed. Using SlowFuzz [21], 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 [22] investigates the memory exhaustion by calculating the maximum runtime memory required during executions.

Waffle, (What An AFL), is a coverage-based greybox fuzzer which is based on AFL's base code. To learn more about Waffle, we need to understand the current features of AFL that help us in reaching the goals of this thesis.

2.3 AFL

American Fuzzy Lop was originally developed by Michal Zalewski. 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 in the targeted binary. This substantially improves the functional coverage for the fuzzed code. The compact synthesized corpora produced by the tool are also useful for seeding other, more labor- or resource-intensive testing regimes down the road.” [23] AFL can dynamically test a target program and for this purpose, instrumentation is required for better performance of the fuzzer.

2.3.1 Stages of fuzzing with AFL

As our fuzzer is based on AFL, we will walk over the features of AFL that we need for our fuzzer. Based on 2.1, AFL has the following phases:

1. **Target identification and setup:** The target application for AFL is any executable program. The only limitation of AFL for the target is that the target executable must take a list of at least one input filename as the program's arguments.

Hence, it is not a necessity for the program to take only one argument in command line, instead, a driver for AFL can add different techniques of getting input for the program.

Next action before starting the fuzz testing, is instrumentation. If the instrumentation of the target program is not done yet, AFL doesn't have any technique for fuzzing except dumb fuzzing, which mutates the inputs completely randomly, without considering any information from the program's executions. We will discuss about the instrumentation more in this chapter.

2. **Identify and setup inputs:** The seeds for AFL must be placed in a directory, and AFL puts them in its queue for fuzzing later. Based on what our fuzzing goals are, preparing a corpora of different seeds with different code coverages, enhances the exploration of AFL to discover more execution paths. AFL can pick a subset of the provided seeds using [afl-cmin](#). This tool "tries to find the smallest subset of files in the input directory that still trigger the full range of instrumentation data points seen in the starting corpus". Instrumentation is mandatory for using [afl-cmin](#). AFL can also minimize test inputs using [afl-tmin](#). "The tool works with crashing and non-crashing test inputs alike. In the crash mode, it will happily accept instrumented and non-instrumented binaries. In the non-crashing mode, the minimizer relies on standard AFL

instrumentation to make the file simpler without altering the execution path.” [24]

Another option introduced by AFL is using dictionaries for investigating the possible grammars used in input parsing. As it is described in AFL’s documentation, “by default, afl-fuzz mutation engine is optimized for compact data formats - say, images, multimedia, compressed data, regular expression syntax, or shell scripts. It is somewhat less suited for languages with particularly verbose and redundant verbiage - notably including HTML, SQL, or JavaScript. To avoid the hassle of building syntax-aware tools, afl-fuzz provides a way to seed the fuzzing process with an optional dictionary of language keywords, magic headers, or other special tokens associated with the targeted data type - and use that to reconstruct the underlying grammar on the go”

After providing required or useful corpora of inputs, AFL can start its fuzz testing.

3. **Generate fuzzed data:** As discussed before, AFL is a **mutation-based** fuzzer, meaning that it generates new inputs by mutating (**favor**ed) inputs.

- **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 favorable path (collected using a **bitmap** after instrumentation) for the purpose of “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.” [24]

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

The preference of AFL for the favored inputs, increases the performance

of AFL in finding more crashes, but it is against the effort of finding an input with higher time consumption.

- **Mutation strategies:** In order 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-flipping, byte-flipping, simple arithmetics, known integers, stack tweaks, and test case splicing. [25]

4. **Execute fuzzed data:** To simplify, AFL takes an instrumented program, a directory for seeds and a directory for outputs:

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

Listing 2.1: Execute AFL

Before running the program using the seeds provided, AFL first looks for favored inputs, in the queue of seeds, then picks the favored one and runs the instrumented program using the input. To monitor the execution of the program, AFL creates a shared memory with the program, and during the execution of the program, this shared memory is filled with information from the instrumentation. The only shared information that AFL keeps is a bitmap of the program's basic blocks, that were visited in the execution. After the execution is finished, AFL processes the bitmap and decides on what the next stage would be. If the program is crashed or hanged, it is kept for exploitability purposes and if the error is unique, it is stored in the output directory. This loop is then repeated for other favored inputs.

5. **Monitor for exceptions:** Beside reporting the crashes and hangs, AFL also logs the information about it's fuzzing stages and stats. In general, AFL pro-

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 2.2: AFL status screen

vides a *status screen* which has eight different sections:

- Process timing:** This section tells about how long the process of fuzzing is running.
- Overall results:** A simplified information about the progress of AFL in finding paths, hangs and crashes.
- 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.
- 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 branch 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 value for the entire input corpus. 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 be useful for comparing the coverage of several different fuzzing jobs that rely on the same instrumented binary. "

- (e) **Stage progress:** The information about the current mutation stage is briefly provided here.
 - (f) **Findings in depth:** The crashes and hangs and any other findings (here we just have the other information about the coverage) are presented in this section.
 - (g) **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.
 - (h) **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 the "level 2", the inputs derived from "level 2" are "level 3" and so on.
6. **Exploitability:** Figuring out whether the crashes and hangs are exploitable or not, is an important stage for exposing and exploiting the vulnerabilities. AFL provides another automated tool for checking the inputs responsible for the faults and mutating those inputs in order to find a more problematic input. To use this tool, we have to provide **-C** option for afl-fuzz:

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

Listing 2.2: AFL Crash Triage

If we define the input directory as the directory of the crashes, then AFL starts it's crash exploration, by looking for other inputs with different paths, but the same state.

Another technique for assessing the exploitability manually, is to use a debugger and investigate the causes of the crashes/hangs.

2.4 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 contains all of the tools, libraries, and header files needed to process intermediate representations and converts it into object files. Tools include an assembler, disassembler, bitcode analyzer, and bitcode optimizer.” [26, 27] LLVM takes an **intermediate representation (IR)** of a program, and translates it to machine language.

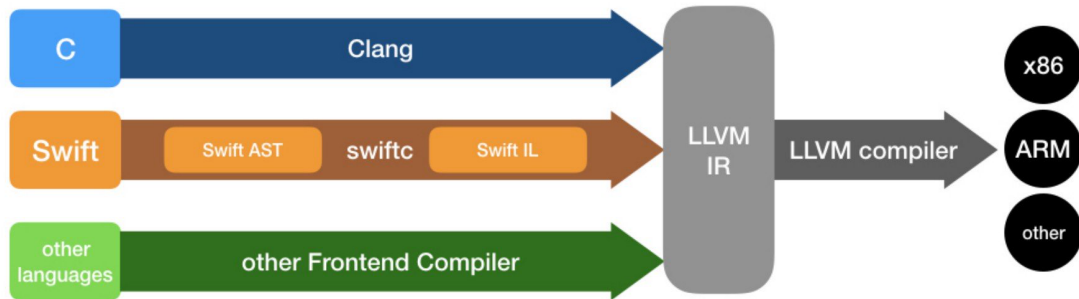


Figure 2.3: LLVM architecture: A front-end compiler generates the LLVM IR, and then it is converted into machine code [1]

“**clang** is a C, C++, and Objective-C compiler which encompasses preprocessing, parsing, optimization, code generation, assembly, and linking.”[28] It parses the source code, using the language-specific syntax, and converts it into a language-agnostic IR. We can use this feature for our instrumentation as it is followed.

2.4.1 Instrumentation and coverage measurements

The goal of using instrumentation for AFL, is to differentiate code coverages. AFL takes the source code and an instrumentation recipe, and generates the instrumented binary of the target program.

The recipe for instrumentation, fills out the coverage bitmap with the hash values of the path 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;
```

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, everytime an instrumented basic block is executed, a location of *shared_mem* in bitmap is increased. The reason for this algorithm is for differentiating different paths that go over the same basic blocks. For instance, in figure 2.4, suppose that we have an instrumented program, with the random values set in compile time. A simple execution would be walking over basic blocks $1 \rightarrow 2 \rightarrow 5$ will increase these locations by 1: *shared_mem*[2362223] for the transition of $1 \rightarrow 2$ and *shared_mem*[368221416] 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 values on bitmap.

AFL has the recipe for inserting the above instructions to the basic blocks using LLVM. It first compiles the program using this recipe for clang and after the execution of the program, we have a coverage bitmap of the locations we have visited during the execution. As a result, AFL uses this coverage feature for announcing new inputs with new coverages.

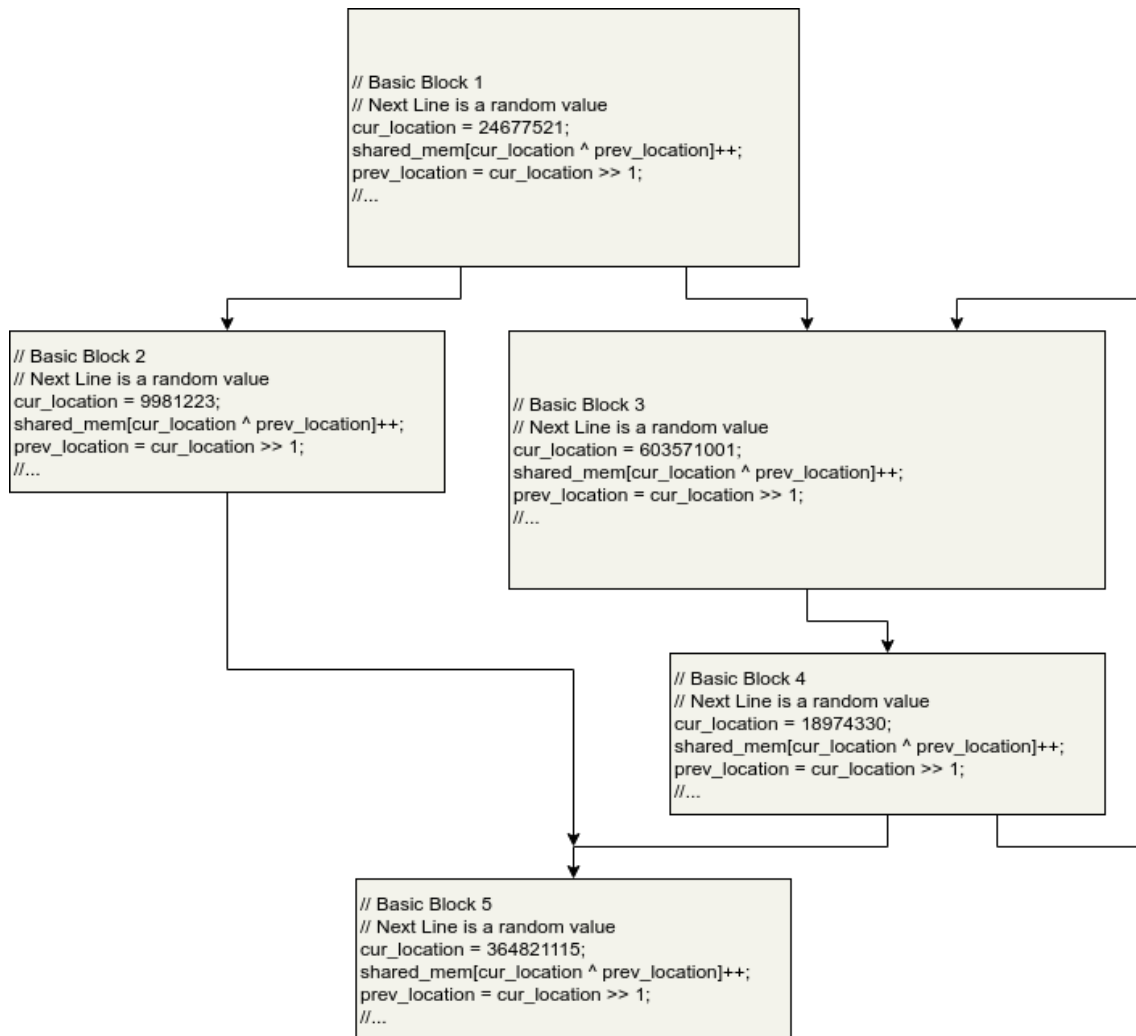


Figure 2.4: Example for instrumented basic blocks

2.5 Concluding remarks

In this chapter we described some basics of our work, that inspired us and lets us implement the current version of our thesis, Waffle. The topics covered in this chapter are:

- A brief description of the previous inspiring works for our fuzzer, as well as the stages of a fuzz testing procedure.
- We split different types of fuzzers into three categories, whitebox, blackbox, and greybox. These fuzzers have different access to the program resources and as a result, the fuzzing approaches were different.
- Code coverage and it's applications in fuzz testing is explained.
- We dig into the state-of-the-art fuzzer, AFL, and walked over it's stages that are important for our fuzzer.
- LLVM and it's applications in AFL, as well as the instrumentations done by AFL are described.

Altogether, we come up with a new approach for fuzzing, as we are aware of. These knowledge led us to introduce Waffle, which is explained in more details in the next chapters.

Chapter 3

Proposed Fuzzer

3.1 Introduction

In this chapter a new fuzzer is introduced, that is capable of finding the vulnerabilities related to (theoretically) any resource's exhaustion. The first section explains a motivating example leading to our proposed fuzzer. The fuzzer is based on AFL and uses the implementation of Memlock for memory usage assessments. 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 type of 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. The vulnerabilities found by our fuzzer are then tested for exploitability. The short-comings and performance expectations of our fuzzer are investigated before we conclude this chapter.

We will call our proposed fuzzer **Waffle**, which is derived from **What An Amazing AFL** - WAAAF! The summary of our contributions are as follows:

- A new instrumentation for collecting runtime information about resource usages, i.e. memory and time.

- A new fuzz testing algorithm for collectively considering the former features of AFL and Memlock, as well as the features we introduce in Waffle.

3.2 Motivating example

3.3 Instrumentation

As mentioned in Chapter 2, AFL uses the instrumentation for increasing the code coverage and Memlock uses the memory features, by calculating the maximum heap/stack size of the memory, used during the runtime. To add more features to our fuzzing, we first need to monitor and collect more runtime features, which are added to the target binary using our enhanced instrumentation.

3.3.1 Features

In addition to the features implemented and used in AFL, Waffle leverages 2 other features for guiding the fuzzing execution.

3.3.1.1 Memory consumption

Our memory consumption features are derived from the features used in Memlock [22]. To collect these information, Memlock monitors the heap or the stack’s usage during the runtime and depending on the allocation or deallocation instructions, the counter is increased or decreased accordingly. In appendix A we can see a section of the source code for Memlock that is responsible for injecting the new instructions. These instructions are added in compile-time and do not change the efficiency of the binary in execution speed. In addition, this job is a one time job that is done before the fuzzing is started.

Before AFL/Memlock starts fuzzing the program, it first shares memory with the

target program. When the fuzzer runs the program, the instrumented binary is capable of filling the **shared memory** according to any strategy we choose and LLVM supports.

Memlock has two arrays for collecting the runtime information. These arrays are `__afl_area_initial` which is implemented in AFL, and `__afl_perf_initial` for collecting the memory consuming features. Currently, the first array can keep 2^{16} elements, each one as a byte; the second array keeps 2^{14} elements of double words - 32bits.

As mentioned before, Memlock has two different fuzzing methods, one for collecting stack information and the other one for collecting heap information.

After a function is called, the injected instructions increase the counter corresponding to the current basic block by 1; every time a **return** instruction is called, this value is decreased by 1.

On the other hand, the fuzzer considering the heap information, must look for opcodes that affect the heap, e.g. the instructions inserted in **allocation** and **deallocation** functions, such as **malloc** or **free** functions.

After these instrumentations are applied on the target, Memlock can start fuzzing the generated binary.

3.3.1.2 Instruction counters

We are looking for features that can help the fuzzer find inputs causing a timeout, or memory exhaustion. We collect these information in runtime using our instrumentation.

The intuition behind this thesis comes from the fact that all the programming competitions announce two main resources that are limited for the execution of the submitted program. This means that a program must be run without any compile-time or run-time errors and generate the correct output; and the whole execution is

constrained with a specific time limit and memory limit. [29] In some competitions, the judge system lets the competitors read each others' submitted program. If they can find a vulnerability in the program that is exploitable and causes any expected result, except the correct answer, they receive the score for their successful **hacking** attempt. [30]

A fuzzer can automate the process of finding vulnerabilities; and targeting the algorithmic problems requires a resource-aware approach for investigating a larger range of problematic inputs.

Collecting the memory-related features are done by Memlock and Waffle needs to be aware of the time-consuming instructions that may lead to a timeout. To gather these information, Waffle **counts the number of instructions in a basic block and increases a counter by that value**. Waffle gathers the information with the help of **instruction visitors** defined by LLVM.

From the documentations of LLVM, "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)." For our purpose, we only take the visitor class `visitInstruction`, and only count how many instructions are located in the current basic block. This number is later added to the total value whenever the basic block is run in the execution.

The visitor is created and is called at the beginning of the basic block and it visits all the instructions inside the basic block. The visitor counts how many instructions are currently existing in the basic block, before we inject any other instruction. In 3.1 a section of the source code for Waffle is provided. In lines **1** to **8** the definition of the visitor class is shown. The only function that is overridden is `visitInstruction` function, which increments the `Count` variable by 1, for each instruction. In the AFL pass from line **10** to line **25**, the visitor is called within the instruction. After the visitor visited all the instructions (line **14**), the result is injected to collect the

number of instructions in the run-time. These values are calculated and injected in the compile time.

```

1 struct CountAllVisitor : public InstVisitor<CountAllVisitor> {
2     unsigned Count;
3     CountAllVisitor() : Count(0) {}
4
5     void visitInstruction(Instruction &I) {
6         ++Count;
7     }
8 };
9
10 bool AFLCoverage::runOnModule(Module &M) {
11     for (auto &F : M) {
12         for (auto &BB : F) {
13             CountAllVisitor CAV;
14             CAV.visit(BB);
15             LoadInst *IcntTotalCounter = IRB.CreateLoad(IcntPtr);
16             IcntTotalCounter->setMetadata(M.getMDKindID("nosanitize"),
MDNode::get(C, None));
17             ConstantInt *CNT = ConstantInt::get(Int32Ty, CAV.Count);
18             Value *IcntTotalIncr = IRB.CreateAdd(IcntTotalCounter, CNT);
19
20             Value *IcntTotalIncrCasted = IRB.CreateZExt(IcntTotalIncr, IRB
.getInt32Ty());
21             IRB.CreateStore(IcntTotalIncrCasted, IcntEPtr)
22                 ->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C,
None));
23         }
24     }
25 }

```

Listing 3.1: Waffle’s instruction counting

After the instrumentation is complemented and the target program is generated, Waffle can now start fuzzing the targeted binary.

3.4 Fuzzing the target

3.5 Concluding remarks

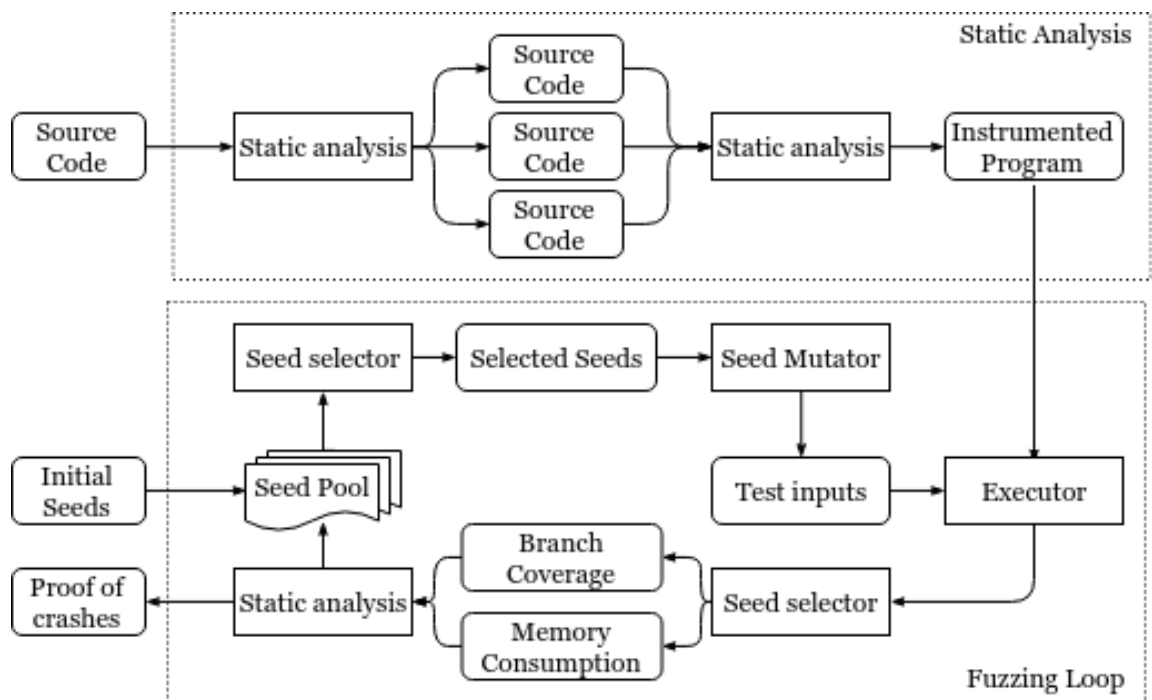


Figure 3.1: Memlock's approach

Chapter 4

Simulation

4.1 Target program and instrumentation

4.2 Program inputs and setups

4.3 Generating fuzzed data

4.4 Program execution and monitoring

4.5 Report vulnerabilities

Chapter 5

Results and Evaluation

5.1 FuzzBench

5.2 results

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] 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.
- [7] 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.

- [8] 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.
- [9] 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.
- [10] 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.
- [11] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.
- [12] 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.
- [13] 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.
- [14] 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.

- [15] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based white-box fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.
- [16] 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.
- [17] Qian Yang, J Jenny Li, and David M Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009.
- [18] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.
- [19] 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.
- [20] 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.
- [21] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slow-fuzz: 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.
- [22] 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.

- [23] Michal Zalewski. American fuzzy lop.(2014). <http://lcamtuf.coredump.cx/afl>, 2019.
- [24] american fuzzy lop - a security-oriented fuzzer. <https://github.com/google/AFL>, 2020.
- [25] Michal Zalewski. Binary fuzzing strategies: what works, what doesn't. <https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html>, 2014.
- [26] LLVM. Llmv project. <http://llvm.org/>, 2020. [Online]; accessed in 2020.
- [27] Chris Lattner and Vikram Adve. Llmv: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [28] LLVM. Clang: a c language family frontend for llvm. <https://clang.llvm.org/>, 2020. [Online]; accessed in 2020.
- [29] Shahriar Manzoor. Common mistakes in online and real-time contests. *XRDS: Crossroads, The ACM Magazine for Students*, 14(4):10–16, 2008.
- [30] Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. A survey on online judge systems and their applications. *ACM Computing Surveys (CSUR)*, 51(1):1–34, 2018.

Appendix A

Handling stack operations in Memlock

The following code A.1 is a shortend version of how memlock handles stack operations. In **line 6**, we can see that the `AFLCoverage` class is derived from a `ModulePass`, which runs a function pass over each function in the module. Using this pass, we can run over all instructions of IR. On **line 20** we define the pointer to the performance's array. In **lines 27** and **28**, we focus on each basic block; as a result, Memlock adds the instrumentations in each basic block. After loading the pointers that were defined earlier, on **line 53** Memlock defines `PerfTotalCounter`, for updating the corresponding pointer and in **line 54**, Memlock increments the counter of stack by one, if it is the first time we are visiting the current basic block. To figure out the type of instructions and find the *return call instruction*, Memlock iterates over all instructions of the current basic block and compares the opcode of the instruction, with the opcode defined for return instruction (**line 58**). If the return instruction is found, the stack counter will be decreased by one.

Note that the following code is not the exact code and the source code can be found on Memlock project [22]. Memlock handles the stack and heap operations in separate

files, doing different instrumentations for each of these two approaches.

```
1 // required packages are included
2
3 using namespace llvm;
4
5 namespace {
6     class AFLCoverage : public ModulePass {
7     public:
8         static char ID;
9         AFLCoverage() : ModulePass(ID) { }
10
11         bool runOnModule(Module &M) override;
12     }
13 };
14 }
15
16 bool AFLCoverage::runOnModule(Module &M) {
17     LLVMContext &C = M.getContext();
18     llvm::IRBuilder<> builder(context);
19
20     GlobalVariable *AFLPerfPtr =
21         new GlobalVariable(M, PointerType::get(Int32Ty, 0), false,
22                             GlobalValue::ExternalLinkage, 0, "
23         __afl_perf_ptr");
24
25     // To convert the range of EdgeID
26     ConstantInt* PerfMask = ConstantInt::get(Int32Ty, PERF_SIZE-1);
27
28     for (auto &F : M) {
29         for (auto &BB : F) {
30             BasicBlock::iterator IP = BB.getFirstInsertionPt();
31             IRBuilder<> IRB(&(*IP));
32
33             /* Load SHM pointer */
34             LoadInst *MapPtr = IRB.CreateLoad(AFLMapPtr);
35             MapPtr->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C, None));
36             Value *MapPtrIdx =
37                 IRB.CreateGEP(MapPtr, EdgeId);
38
39             /* Load the Performance Pointer */
40             LoadInst *PerfPtr = IRB.CreateLoad(AFLPerfPtr);
41             PerfPtr->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C, None));
42             Value *PerfBranchPtr =
43                 IRB.CreateGEP(PerfPtr, IRB.CreateAnd(EdgeId, PerfMask));
44
45             /* Increment performance counter for branch */
46             LoadInst *PerfBranchCounter = IRB.CreateLoad(PerfBranchPtr);
47             PerfBranchCounter->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C, None));
48             Value *PerfBranchIncr = IRB.CreateAdd(PerfBranchCounter,
```



```

48     ConstantInt::get(Int32Ty, 1));
49     IRB.CreateStore(PerfBranchIncr, PerfBranchPtr)
50     ->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C,
51     None));
52
53     /* Increment performance counter for total count */
54     LoadInst *PerfTotalCounter = IRB.CreateLoad(PerfPtr); // Index
55     0 of the perf map
56     PerfTotalCounter->setMetadata(M.getMDKindID("nosanitize"),
57     MDNode::get(C, None));
58     Value *PerfTotalIncr = IRB.CreateAdd(PerfTotalCounter,
59     ConstantInt::get(Int32Ty, 1));
60     IRB.CreateStore(PerfTotalIncr, PerfPtr)
61     ->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C,
62     None));
63
64     for(BasicBlock::iterator i = BB.begin(), i2 = BB.end(); i!=i2;
65     i++) {
66         IRBuilder<> MemFuzzBuilder(&(*i));
67
68         if(Instruction *inst = dyn_cast<Instruction>(i)) {
69             if(inst->getOpcode() == Instruction::Ret)
70             {
71                 /* Subtract the memory usage of stack after return */
72             }
73         }
74     }
75 }
76
77 return true;
78 }

```

Listing A.1: Memlock C++ code for stack operations

Glossary (if any)

start writing here.

Vita

Candidate's full name:

University attended (with dates and degrees obtained):

Publications:

Conference Presentations: