Università di Genova
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

A survey of fuzz-testing tools for vulnerability discovery

Andrea Straforini

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/

Università
di Genova

MSc Computer Science
Software Security and Engineering Curriculum

A survey of fuzz-testing tools for vulnerability discovery

Andrea Straforini

Advisor: Giovanni Lagorio Examiner: Matteo Dell'Amico

March, 2022

Table of Contents

Chapter 1

Introduction

When we think about the software development life cycle, we know that it should include tasks that belong to the field of *Application Security* (*AppSec*), whose purpose is to fix and prevent security issues. This field includes and acts on all stages, from analysis through implementation and maintenance.

There are many approaches to AppSec, each effective at discovering different categories of vulnerabilities and each with different costs and timelines.

Code review is often used to find application-specific vulnerabilities, where specialized personnel manually review code for vulnerabilities. This process is often time-consuming and costly and requires highly qualified personnel in shortage in the market.

In addition, companies with fewer than 500 employees often do not have access to security experts internally or through third-party contractors or providers [Cob16]. Therefore, fuzzing can be a great way to provide good levels of security in a cost-effective and automated manner via *continuous integration/continuous deployment* (*CI/CD*) pipelines.

However, what is fuzzing? *Fuzzing* is an approach to software testing that involves running numerous tests on a target program by a program called *fuzzer*. Inputs can be generated following numerous approaches, but all methods share a random part. The program under test is then monitored to detect any type of flaw caused by the input provided.

Compared to other vulnerability discovery techniques such as static analysis, dynamic analysis, and symbolic execution, fuzzing is applicable with "off-the-shelf", scalable, and modular instruments. Although the approach may seem too simplistic, fuzzing has shown considerable potential by revealing numerous vulnerabilities. Honggfuzz[1], an open-source fuzzer developed by Google researchers, managed to find a critical vulnerability within OpenSSL [ope16]. The combination of simplicity and effectiveness has contributed to the

[1] `https://github.com/google/honggfuzz`

widespread adoption of these tools in both industry and the academic community.

Academic interest has driven the creation of numerous new fuzzers and new analysis techniques to improve the performance of fuzzers by creating several thousand papers since the inception of the term [Sch].

This thesis's objective is to make an overview of this field, accompanying the reader in creating a personal knowledge graph that allows a subsequent autonomous study. In Chapter 2, after a brief opening on the history of fuzzing, we introduce the basic concepts of fuzzing and taxonomize these tools. In Chapter 3 we describe the techniques and optimizations that are used by modern fuzzers, and in Chapter 4 we give an overview of binary-only fuzzing and some of the main techniques used in this field.

Chapter 5 presents a quantitative and qualitative analysis of some of the main and most famous fuzzers. In Chapter 6, a demonstration of the use of the fuzzers and techniques presented in the thesis is made.

This document assumes that the reader has a basic understanding of the following topics: Executable and Linkable Format (ELF), C compilation process, Binary instrumentation and Linux process creation.

Chapter 2

The basics of fuzzing

In this chapter, we introduce fuzzing and give a brief overview of its evolution through the narration of its history in Section 2.1.

In Subsection 2.2.1, we taxonomize the fuzzers and explain their fundamental differences.

Subsections 2.2.2 to 2.2.4 introduce the basic concepts necessary for the reader to understand how these tools work and their obstacles.

2.1 The evolution of fuzzing

The birth of fuzzing and the initial development of the concept is attributable to a class project by Professor Barton Miller in 1988 at the University of Wisconsin-Madison, whose results were published in 1990 [MFS90]. The class project consisted in developing a tool capable of generating a stream of random outputs to test the robustness of different UNIX utilities. The implementation of the fuzzer was extremely simple by modern standards; the output of the fuzzer was fed to the target utility either via pipe or using a support program that permitted to simulate console input. The program under test was then monitored, and, in case of crash or hang, a core file was generated. Their results found that, over the ninety different utility programs on seven versions of UNIX, more than 24% of these crashed.

Since Miller's early work, fuzzing has gained more attention in academia and the industry thanks to an important step: the release of the *SPIKE* fuzzing tool. This tool was presented at the 2002 Blackhat conference by Dave Aitel [DA02] in conjunction with his paper [Ait02] and a framework for network protocol fuzzing that allows the creation of models of network protocols and use them to send generated traffic. SPIKE's approach was an important

milestone because it was the first to tie random data to the regular input the application expected.

In April 2015, it was demonstrated that *AFL* [LLCa], an open-source fuzzer developed by Google, could be used to find the Hearthbleed vulnerability in the OpenSSL software library [Bö15].

In December 2016, Google announced *OSS-FUZZ* [Ser17], a platform that provides continuous fuzzing for selected core open-source programs that found over thirty thousand bugs in five hundred projects.

Academic research is currently focusing on discovering and refining techniques to: overcome complex checks to penetrate deeper parts of the program [ASB+19, laf16, SGS+16], improve static and dynamic binary instrumentation for closed-source programs fuzzing [DGR20, LCC+17, FMEH20], and standardize fuzzers in common frameworks [MSS+21, Lea04, Ser17, HHP20].

2.2 Fundamentals

2.2.1 Types of fuzzers

There are several classifications for fuzzers. According to the need of the source code and its input structure awareness, three categories can be distinguished: *white-box*, *black-box*, and *gray-box* [MHH+19].

A black-box fuzzer is utterly unaware of the program's internal state, and it can only observe the input/output behaviors. Given these minimal requirements, these tools have a relatively simple implementation, are very fast, easily parallelized, scalable, and their main focus is on generating input mutations that can reveal corner cases. On the other hand, they are often unable to reach the innermost parts of a program, and they frequently test only the surface of these. For this reason, academic efforts are focusing on tools that, through observation of the output, can understand the internal state of the program.

A white-box fuzzer leverages program analysis, such as *symbolic execution* and *constraint solving*, to create test cases that increase code coverage. Unfortunately, white-box fuzzing turns out to be limited by the defects of the techniques that compose it, such as the *path explosion* problem for symbolic execution. These issues can arise when the program under test needs broadly structured inputs.

Gray-box fuzzing is placed on a middle ground between the two types mentioned above, and it aims to integrate the black-box fuzzing simplicity with the effectiveness of white-box fuzzing. These fuzzers use instrumentation instead of program analysis to obtain

information about the program's internal state during test execution.

Based on the methodology for generating the test cases, we can distinguish between fuzzers *generation-based* and *mutation-based*. Generation-based fuzzers often need a configuration file that specifies the input structure and leverages this model to generate a more significant proportion of valid inputs. Mutation-based fuzzer, instead, generate test cases by modifying an input *corpus*, which can also be empty.

Fuzzers using generated test cases could often reach deeper parts of code than mutation-based fuzzers [SGA07, Ney08]. Miller and Peterson in [MP+07] show how, in the case of *Libpng*[1], the results of the generation-based SPIKE fuzzer are far better than a mutation-based fuzzer. On the other hand, this type of fuzzers cannot be used "off-the-shelf" as creating an input generator can be problematic and time-consuming. For these reasons, the use of mutation-based fuzzer is quicker to set up and easier to use, and this has allowed its widespread diffusion and use by the community.

There are other ways to taxonomize fuzzers; for example, fuzzers can be classified as *coverage-based* or *direct* [LZZ18], according to the strategies applied to explore the target program. A coverage-based fuzzer aims to generate test cases that cover as much as possible of the target code, while the direct fuzzer aims to reach and test in-depth particular paths and portions of the program under test.

2.2.2 Instrumentation

In gray box fuzzing, the instrumentation is applied to the target program to detect a crash and calculate code coverage. Based on the availability of the source code, the instrumentation can be added at compile-time or, in the case of a binary-only target, via *dynamic binary instrumentation* or *static binary rewriting*.

Each fuzzer injects instrumentation in different ways but it essentially produces the same result. Instrumentation functions are injected in specific code positions, such as the beginning of a basic block or inside a branch of a conditional instruction. Figure 2.1 shows the instrumentation injected by AFL++ during compilation at the beginning of each basic block.

2.2.3 Code coverage

Instrumentation provides the data necessary to calculate the code coverage metric that tracks which part of the program has been effectively executed. In fact, vulnerabilities

[1] `http://www.libpng.org/pub/png/libpng.html`

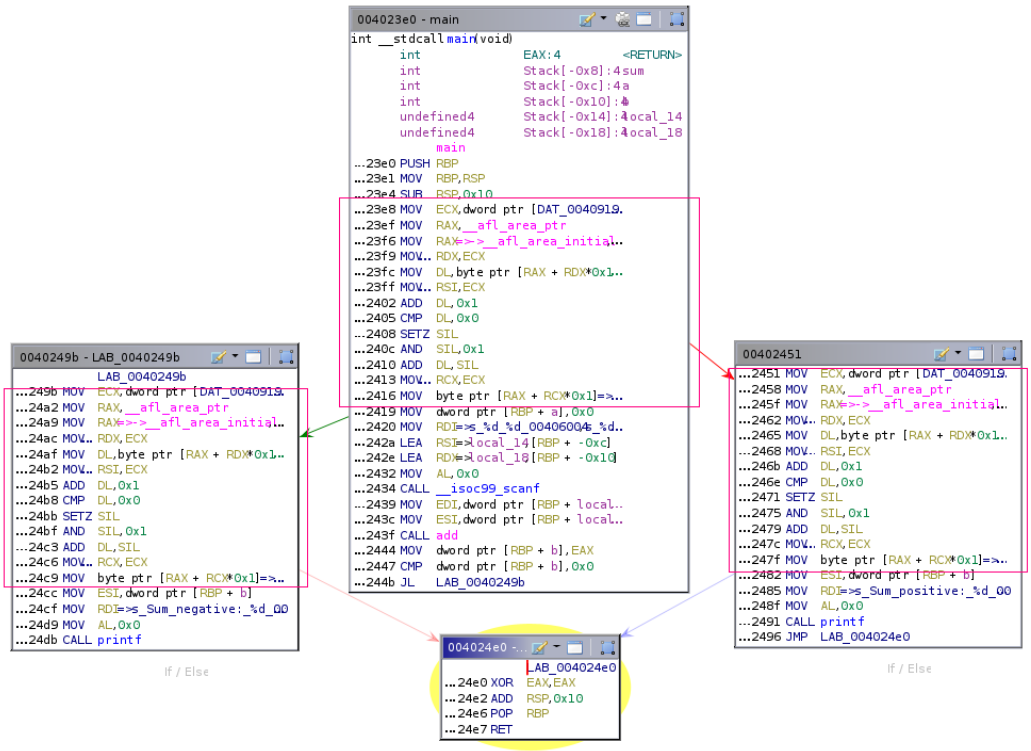


Figure 2.1

cannot be found in a program section if that section has not been executed. Based on the change in code coverage during the execution of test cases, gray-box fuzzers can understand if a test case has managed to penetrate deeper into the program and thus select it for a subsequent mutation.

Code coverage metrics can be defined in numerous ways; some of the most important ones are: *Block Coverage*, *Branch Coverage*, *Full Path Coverage*, *N-Gram Branch Coverage*.

Block coverage is one of the most basic technique and measures how many code block have beeen visited.

In branch coverage, the basic unit is the tuple *(prev, cur)*, where *prev* and *cur* stand for the previous and current block IDs, respectively.

Full path coverage is infeasible, and for a program of n reachable branches, it will require a 2^n test cases while for the branch coverage $2 \cdot n$ [Mil].

N-Gram Branch Coverage sits between full path coverage and branch coverage. `N` is the parameter that describes how many previous blocks are taken into account. When `N` tends to infinity, n-gram branch coverage is equivalent to full path coverage, while when n is 0,

it is reduced to block coverage [WDS+19].

Different fuzzers use various approaches to measure coverage, for instance, Vuzzer [RJK+17] and Honggfuzz [Swi] use basic block coverage metric, AFL [Zal] uses branch coverage while LibFuzzer [Pro] can use either branch or block coverage.

2.2.4 Road-blocks

Roadblock is a term used to refer to patterns that make it difficult for fuzzers to create suitable inputs to reach specific branches. Mutation-based fuzzers applying random mutations have little chance of guessing the correct input. The two most common roadblocks are *magic numbers* and *checksum tests*. Magic numbers are often found in header fields, and an example is shown in Listing 2.1.

Listing 2.1: Magic number road-block

```
if input == "MAGIC"
  # buggy code
```

Checksum tests can be found in the portions of code where data integrity is checked; an example is shown in Listing 2.2.

Listing 2.2: Checksum road-block

```
  if input[-1] == sum(input[:-1]):
    # buggy code
```

In order to overcome these problems, different techniques have been proposed; most of them use *symbolic execution* [PSP18], *taint tracking* [PSP18], *splitting multi-byte comparisons* [laf16, LCC+17], and *input to state correspondence* [ASB+19]. These techniques will be presented in the following chapter.

Chapter 3

Techniques

Fuzzers use additional techniques and optimizations to improve their effectiveness, code coverage, and pass roadblocks. This chapter presents in sections 3.1 to 3.5 some of the techniques applied by fuzzers in the compilation phase of the target program or at run time.

Finally, sections 3.6 to 3.8 show some optimizations used to improve the performance of fuzzing campaigns.

3.1 Sub-instruction profiling

Sub-instruction profiling divides multi-byte comparisons into single-byte comparisons to more effectively track the progress of mutations to match constant values.

Listing 3.1: Original code

```
if(input == 0xDEADBEEF){
  // buggy code
}
// secure code
```

By dividing complex comparisons into more straightforward nested comparisons, there is a greater likelihood that the fuzzer can discover a test case that allows the discovery of a new branch. This optimization, in turn, will signal the fuzzer that the current input should be used again in further fuzzing attempts. Again, this last event will repeat for the other if-statements and increase the probability of guessing the correct value.

Listing 3.2: Sub-instruction profiling code

```
if(input>>24 == 0xDE){
  if((input & 0xff0000) >> 16 == 0xAD){
    if((input & 0xff00) >> 8 == 0xBE){
      if((input & 0xff) == 0xFF){
        // buggy code
        goto end;
      }
    }
  }
}
end:
  // secure code
```

This technique is excellent against magic numbers, but it is ineffective with checksums [FDC20]. It is implemented as a compiler extension in *CompareCoverage* [Zer], and *laf-intel* [laf16] and it is integrated into fuzzers such as *AFL++* [FMEH20] and *Honggfuzz* [Swi].

3.2 Taint Analysis

Dynamic taint analysis tracks information flow in the program by running and observing which computations are affected by taint sources. This technique identifies inputs that could reach internal parts of the target program without proper sanitization, hence creating potential security issues such as SQL injections.

Angora [CC18] implements a form of scalable byte-level taint tracking, which can identify noncontiguous magic numbers in the input. VUzzer [RJK+17] and TaintScope [WWGZ10] identify magic numbers by tracking comparison instruction where one operand is independent by the input.

3.3 Symbolic execution

Symbolic execution is a static program analysis technique that systematically explores many possible execution paths at the same time by abstractly representing variables as symbols and by using constraint solvers [BCD+18]. This technique suffers from the path explosion problem: a symbolic executor may fork off a new state at every branch of the program, and the total number of states may quickly become exponential in the number of

branches. Path explosion usually occurs in loops and function calls; each loop interaction can be generalized as an if-goto statement, leading to a conditional branch in the execution tree. Thereby, symbolic execution does not scale well for large programs.

T-Fuzz [PSP18] is one of the fuzzers that employs symbolic execution. When fuzzing does not improve coverage, T-Fuzz temporarily patches the checks that cause the roadblock and uses symbolic execution to filter out false positives.

3.4 Concolic execution

Concolic execution tries to handle the path explosion problem with a hybrid approach by performing symbolic execution along with a concrete execution. Several approaches implement concolic execution; among these, we have *dynamic symbolic execution*, where the symbolic execution is driven by a specific concrete execution, and *selective symbolic execution*, where the efforts of the SMT solver are directed towards specific paths of the target program [BCD+18].

Examples of fuzzers that use concolic execution are *Driller* [SGS+16], *Qsym* [YLX+18], and *TaintScope* [WWGZ10]. Driller switches to symbolic execution when fuzzing does not improve coverage and after reaching a timeout; Qsym implements an instruction-level concolic execution by tightly integrating the symbolic emulation with the native execution using dynamic binary translation; TaintScope fixes checksum values in generated inputs by combining concrete and symbolic execution techniques.

3.5 Input to state correspondence

Input to state correspondence [ASB+19] exploits an idea that stems from an observation: in many cases, part of the input corresponds to parts of memory or registers at runtime. In practice, most programs apply only a few decoding steps to the input before it is used directly in the context of challenging conditions, such as checks for magic bytes and checksum tests. This technique exploits this strong input-to-state correspondence by using *colorization* that inserts random bytes into the input and then checks whether some of these bytes appear, as is or after few simple transformations, in the comparison operands when running the program. This heuristic allows creating a lighter version of a taint tracker.

The latter, in turn, facilitates skipping complex sections of the code such as API calls or unknown instructions, which otherwise would be difficult to handle for taint tracking or symbolic execution. The technique then tries to pass these checks by mutating only the interesting part of the input found by the colorization. As a result, even inputs that

pass through unknown library functions, large data-dependent loops, and floating-point instructions do not significantly reduce the quality of the results.

For these reasons Input to state correspondence could replace the two most complex methods of analysis: taint-tracking and symbolic execution. In fact, compared to the latters, it is easier to implement and scales to large complex targets and diverse environments.

3.6 Fuzzing modes

There are three main methods to fuzz a target application: *simple*, *fork-server*, and *persistence mode*.

In the simple mode, the fuzzer creates the process from scratch, thus making it unnecessary for the security researcher to comprehensively analyze the API offered by the target library and ensure the repeatability and robustness of the method. A particular test case is executed in a separate process, thus ensuring that a possible crash is not due to random memory corruption or an invalid state caused by repeated feeding of test cases. However, this method runs into slowdowns caused by the `execve` syscall and the dynamic-linking process.

The target application runs normally in a fork-server architecture until the input is read and processed. At that point, the fuzzer generates a pre-initialized process for each test input execution and collects its results. The benefits of the fork-server architecture are of two types: flawless stability compared to persistence mode and more speed than the method that reinitializes the process from scratch.

Persistent mode, in which the fuzzer executes multiple inputs in the same fully-initialized process, is the fastest way to fuzz a target program, but it suffers from stability problems. Unless the target program is perfectly reset by the fuzzer, the program's internal state will diverge from the initial one due to the side-effects generated by the execution of the test cases, thus yielding stability degradation. Numerous programs such as web servers and web browsers can run into timeouts, hangs, and crashes, significantly decreasing this metric. Therefore, it is crucial to optimize the number of uses of the same process in persistent mode or to use a fork-server architecture that avoids these problems entirely at the expense of the execution speed.

3.7 Instrumentation optimizations

AFL, a well-known fuzzer, captures edge coverage, along with branch-taken hit counts by injecting a code essentially equivalent to Listing 3.3. AFL generates the `cur_location`

value randomly to keep the `XOR` output distributed uniformly [Zal]. The fuzzer tracks transitions between basic blocks by incrementing a counter referenced by the resulting value of the `XOR` calculation. The one-bit shifting of the basic block address is used to differentiate between the transitions A→B and B→A.

Listing 3.3: AFL instrumentation code [Zal]

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

The standard shared memory region employed by AFL is 65.5 K bytes (64KB) and thus at most 65.5K edges could be stored without conflicts. For example, over 75% of edges collide with other edges in the application libtorrent, which has over 260K edges [GZQ+18].

Applications	Size	#ins.	#BB	#edges	collision
LAVA(base64)	193KB	5570	822	1308	0.8%
LAVA(uniq)	208KB	5285	890	1407	0.92%
LAVA(md5sum)	234KB	7397	1013	1560	1.02%
LAVA(who)	1.52MB	84648	1831	3332	1.8%
catdoc	202KB	6448	841	1322	1.29%
libtasn1	540KB	12511	2163	3820	2.72%
cflow	688KB	24655	4286	7001	5.2%
libncurses	338KB	21486	4646	7883	5.57%
libtiff+tiffset	1.77MB	61119	8974	14826	10.4%
libtiff+tiff2ps	1.97MB	65932	9632	15927	10.84%
libtiff+tiff2pdf	2.1MB	71530	10507	17603	12.31%
libming+listswf	4.04MB	87148	11456	19154	13.61%
libdwarf	3MB	73921	11698	20260	13.7%
tcpdump	4.62MB	127082	18781	32656	21.2%
nm	8.72MB	218326	31611	53652	36.06%
bison	3.28Mb	219268	42856	55658	32.8%
nasm	4.4MB	226665	41691	57411	33.38%
libpspp	5MB	259501	41323	71335	38.9%
objdump	11.88MB	305620	43935	74313	40.17%
clamav	11.35MB	347156	46140	81069	42.48%
exiv2+libexiv2	4.75MB	283284	59650	91287	45.87%
libsass+sassc	32.8MB	593570	68538	106738	50.7%
vim	14.7MB	478402	83877	153689	61.4%
libav	76.7MB	1776730	158009	255212	74.85%
libtorrent	97.5MB	1228513	164325	260485	75.29%

Figure 3.1: [GZQ+18]

AFL++ LTO solves this problem by injecting the instrumentation at link time and by manipulating the intermediate representation using a custom *LLVM Pass*[1]. Each edge is associated with a specific address of the shared memory, thus obtaining a guaranteed non-colliding edge coverage and a shared memory dimension calculated automatically.

[1] `https://llvm.org/docs/Passes.html`

From the sample shown in Listing 3.4, decompiled with Ghidra[2] by reversing a binary compiled with AFL++ LTO, we can notice that, as anticipated, the references became hardcoded rather than determined via the original AFL heuristic.

Listing 3.4: Example of instrumented code by AFL++ (Reversed)

```
int main(void){
  int local_14;
  int local_10;
  int a;
  int b;
  int sum;
  __isoc99_scanf("%d %d",&local_14,&local_10);
  a = add(local_14,local_10);
  if (a < 0) {
    __afl_area_ptr[3] = __afl_area_ptr[3] + '\x01' + ((u8)(__afl_area_ptr[3] + '\x01') == '\0');
    printf("Sum negative: %d\n",(ulong)(uint)a);
  }
  else {
    __afl_area_ptr[2] = __afl_area_ptr[2] + '\x01' + ((u8)(__afl_area_ptr[2] + '\x01') == '\0');
    printf("Sum positive: %d\n",(ulong)(uint)a);
  }
  return 0;
}
```

This solves the collision problem entirely and ensures that shared memory is as large as needed.

3.8 Compiler Sanitizers

Many compilers for C/C++, such as Clang and GCC support the use of so-called *sanitizers*. These are usually inserted into the binary during the build phase and operate at run time.

There are different types of sanitizers; some of the most important are: *AddressSanitizer* (*ASan*), *MemorySanitizer* (*MSan*), and *UndefinedBehaviourSanitizer* (*UBSan*).

The former detects vulnerability addressability issues, including *stack* and *heap* buffer overflow, using after free and after return. The second is a detector of uninitialized memory reads. The third is an undefined behavior detector that catches, for example, integer overflow and conversions between floating-point types that would overflow the destination.

All three allow finding vulnerabilities that may not directly cause a program crash and, therefore, are essential for fuzzing. In addition to speeding up the vulnerability discovery process, sanitizers allow maintaining a deterministic campaign. If a crash does not happen always with the same test case, the coverage is not calculated correctly, and the process becomes non-deterministic.

[2] `https://ghidra-sre.org/`

Chapter 4

Binary-only fuzzing

Binary-only fuzzing is a sub-branch that deals with testing binaries already compiled. The fundamental problem in binary-only fuzzing is that the binary has not been compiled with instrumentation, and therefore, it is not straightforwardly possible to compute coverage.

There are essentially four approaches for binary-only fuzzing: *black-box fuzzing*, *dynamic binary instrumentation*, *static binary rewriting*, and *hardware-based instrumentation*.

The black-box approach directly avoids the problem by not requiring coverage. This method can be successful in the case of targets that have never been tested before, but it cannot uncover more complex vulnerabilities that require a gray-box approach. Black-box fuzzing is often too simplistic and does not improve coverage compared to traditional testing.

Dynamic binary instrumentation guarantees the correctness and the possibility of instrumenting a binary at the price of a performance loss and it is presented in Section 4.1.

The static rewriting approach, introduced in Section 4.2, provides execution times comparable to fuzzing with source code but does not always guarantee the correctness and possibility of doing so.

Finally, the performance of hardware-based fuzzing depends on the technology used. For instance, fuzzing with *Intel PT*[1] is less efficient than using *QEMU-AFL*[2] [CMX+19]. Hardware-based fuzzing is not addressed in this thesis; however, we report that two of the best-known fuzzers have these techniques in their "arsenal". AFL++ makes it available the use of Intel PT, and Honggfuzz supports *hardware-based counters*, *Intel BTS code coverage*, and Intel PT code coverage on Linux.

[1] `https://www.intel.com/content/www/us/en/support/articles/000056730/processors.html`
[2] `https://github.com/google/AFL/tree/master/qemu_mode`

4.1 Dynamic Binary Instrumentation

This section presents some of the main toolkits and hypervisors used for dynamic binary instrumentation.

4.1.1 QEMU

QEMU[3] is an open-source machine emulator and virtualizer that could emulate different CPU architectures; for example, it can emulate ARM, RISC-V, and PowerPC [QEM04]. It works like a JIT compiler but without including an interpreter. QEMU internally is built around the *tiny Code Generator* in short *TCG*, which is separated into three parts: a frontend, a backend, and a core part. The frontend consists of modules used by the TCG to translate from guest code architecture code to an *intermediate representation* (*IR*). The TCG uses the backend to translate the IR code into the host architecture compliant machine code. The core is responsible for orchestrating the frontend/backend translation, managing the cache, and handling exceptions. QEMU dynamically translates from the guest architecture instruction set to the host one up to the next jump or instruction that modifies the static CPU state in a way that cannot be deduced at translation time. These blocks, similar to basic blocks, are called *translated blocks* (*TBs*) and are saved in a QEMU translation cache for later reuse [Bel05].

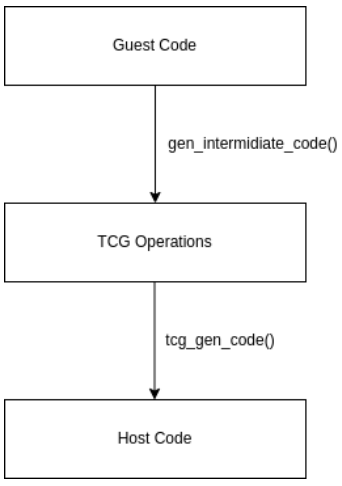


Figure 4.1: QEMU block translation

The process earlier described is repeated through a cycle where TBs' execution is alternated with the execution of the TCG. The execution of the translation blocks is preceded by the

[3] `https://www.qemu.org/`

prologue and followed by the epilogue (Figure 4.2). These two functions are fundamentals for fuzzing because they can be customized and used for passing information between QEMU and the fuzzers.

When a TB has been executed, QEMU uses the simulated Program Counter and other information of the static CPU state to find the next TB using a hash table. If the next TB is already in the cache, QEMU patch the original block to jump directly to the next one instead of jumping to the epilogue. In this way, QEMU removes the significant overhead of returning to the main loop each time a block is executed. This way it creates chains of TBs and gets a significant boost in performances, and it gets closer to native host performances. This technique is referred to as *block chaining* and in the QEMU forks for fuzzing, it has been manipulated in different ways. In the qualitative analysis of the fuzzers, we will state for each tool if this optimization is enabled.

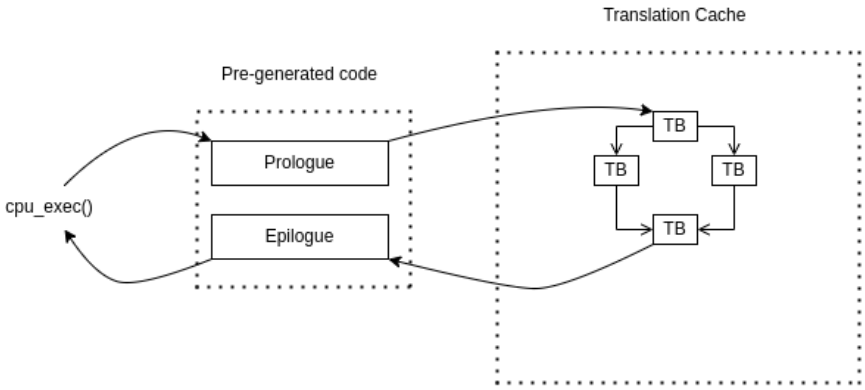


Figure 4.2: Part of QEMU execution flow

4.1.2 Unicorn engine

Unicorn engine[4] is a lightweight multi-platform, multi-architecture CPU emulator framework. It is based on core components of QEMU, and it focuses on emulating CPU operations without managing other parts of the virtualized computer machine. The removal of all subsystems not related to CPU emulation makes Unicorn lighter and with optimizations not present in QEMU. Unlike the latter, in fact, it is able to emulate multiple CPUs simultaneously while remaining thread-safe, natively supports dynamic instrumentation, and is able to emulate chunks of raw binary code without any context.

The Unicorn engine has been successfully employed in fuzzers such as AFL and AFL++ to perform fuzzing of specific portions of binaries and Linux kernel modules [MRH19].

[4] `https://www.unicorn-engine.org/`

Compared to using QEMU with fuzzers such as Honggfuzz and AFL, this emulator requires creating a Unicord-based test harness. It is possible to create this harness with one of the languages for which Unicorn provides a binding such as Rust, Python and C. The harness must create memory map regions, load target code, set initial registers and memory state, and load mutated data from the fuzzer. It then emulates the target binary code, and if it detects that a crash or error has occurred, it fires a signal. This signal will be intercepted by the fuzzer, which could then associate the test case with a crash [Vos04].

4.1.3 Other Tools

There are many tools for dynamic instrumentation that are used by fuzzers. Each of these has characteristics that differentiate it from the others. Some of these tools are *Dyninst*[5], *Frida*[6], and *Dynamorio*[7]. Among these noteworthy is FRIDA, which can be used effectively to fuzz Linux, macOS, Windows, Android, and iOS even remotely.

4.2 Static Binary rewriting

Many tools exist for binary rewriting, and in this section, we will present some of the most famous or promising.

There are three fundamental techniques to rewrite binaries: *recompilation*, *reassemble assembly*, and *trampolines*.

The first technique attempts to generate an intermediate representation but in order to do so, it needs to recover type information from binary, which is an open problem.

The second approach generates an assembly file with relocation symbols. A tool that leverages this technique is *Retrowrite* [DBXP20]. This tool loads the text and the data section from the binary to recover a best-effort control-flow graph which is used to identify symbolizable constants, which are subsequently converted to assembler labels. It then generates a reassemble assembly which is subsequently manipulated by instrumentation passes; the paper shows ASAN insertion and the necessary instrumentation for AFL. After other intermediate steps, Retrowrite then outputs the newly generated binary that could be used with AFL. This tool has numerous limitations; it does not handle non-PIE binaries and C++ compiled binaries that are using exceptions.

The trampoline approach relies on indirection to insert new code segments without chang-

[5] `https://www.dyninst.org/`
[6] `https://frida.re/`
[7] `https://dynamorio.org/`

ing the size of basic blocks. The downside of this approach is that it may significantly increase code size and diminish performances through the extra indirections. This technique is used by *E9Patch* [DGR20], which can be used on x86_64 PIE and non-PIE ELF binaries as well as libraries and shared objects. This tool does not need to recover any information about the control flow graph because it uses binary rewriting methodologies that are control flow agnostic. For this reason, it can work with stripped binaries and does not assume that the target binary was compiled with a specific type of programming language or compiler. The authors created an algorithm that attempts to patch the target instruction through five techniques ordered by performance. E9Patch does not work with self-modifying code and does not guarantee perfect coverage, in fact, for particular hard cases, it can fail. The authors cite three hard cases which are: virtual address space shortages, single-byte instructions, and attempting to patch many instructions.

Along with E9Patch, the same authors published *E9AFL* [GDR21], which is implemented as a plugin for the E9Patch frontend. The plugin takes the disassembly of the input binary from the frontend and outputs an AFL instrumentation trampoline template, the AFL runtime, and a set of instrumentation locations. Finally, the information is passed to the static binary rewriter that generates the AFL instrumented binary. Interestingly, the authors generate trampolines with AFL's instrumentation which is subject to the collision problem as described in Section 3.7; in addition, the authors test the tool on Firefox and Chrome, two large binaries. Therefore it would be interesting to test trampolines generated with an approach similar to the one illustrated in Section 3.7 instead of using the AFL heuristic approach.

Chapter 5

Analysis

Fuzzer benchmarking is a complex research field that requires computational power, research funding and time. Both independent researchers and creators of fuzzer variants have performed numerous evaluations, but, despite the great interest from the industry and the academic community, there was no tool to perform correct and effective evaluations of the fuzzers performances until recently.

A survey conducted by Klees et al. [KRC+18] over thirty-two fuzzing research papers found that most papers failed to perform a statistically sound analysis. In particular, many papers failed to perform multiple experiment runs, measure performance using an insecure and not independent metric from the fuzzer under test, use short trials, and do not use a large and diverse set of benchmarks. Experiments must be performed on many benchmark programs, run multiple times, and each tool-benchmark pair trial needs to run at least 24 hours to perform a correct evaluation.

As described by [MSS+21], an evaluation that respects all the points mentioned above would require an amount of time in the order of CPU-years with a cost of tens of thousand dollars.

Therefore we approached our analysis from a different perspective; we reported the most recent and most structured quantitative analysis available and created a qualitative analysis.

The former exposes the data of [MSS+21] in Section 5.1, whose service is quickly gaining ground in the community, and that is establishing itself as the new "gold standard" for evaluating the performance of fuzzers.

The qualitative research in Section 5.2 is inspired by the *Open Maturity Model* [Qua04, Vim15] and developed around the constraints derived from the target user that we have taken as a reference.

5.1 FuzzBench

FuzzBench is a free service that evaluates fuzzers on various open-source programs, such as those contained in the OSS-Fuzz project. The project is hosted on a GitHub repository[1] where the community can integrate fuzzers and benchmarks using docker containers. The experiments are performed in Google's cloud, and a dedicated portal provides statistics and results. This platform guarantees repeatability, replicability, and reproducibility. Repeatability and reproducibility are guaranteed through the repository and the version control system; being able to download the framework and run the test locally guarantees replicability.

The researchers presented, together with the FuzzBunch platform, the comparison of the main fuzzers with the availability of the source code. They benchmarked eleven fuzzers that are important academic works or are popular in industry, on twenty-two open source projects. For each fuzzer-benchmark pair, they ran twenty trials, each trial lasting approximately twenty-three hours. The critical difference diagram of the experiment is shown in Figure 5.1. AFL++ came out to be the best, followed by Honggfuzz, Entropic, and Eclipser. The diagram shows how the first seven fuzzers have no significant statistical differences. This is indicated by the bold line connecting these fuzzers. AFL++, however, is significantly better than the last four fuzzers, Honggfuzz is better than the last three, and so on.

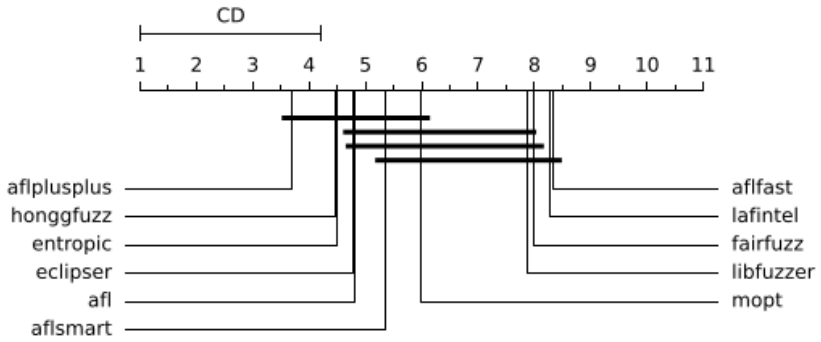


Figure 5.1: Critical difference diagram [MSS+21]

The experiment also measured which parts of the code each fuzzer covers, thus creating a "differential coverage" that tracks how many unique regions were covered by one fuzzer relative to any other one.

The experiment found that of the 2,182,118 regions covered by any fuzzer, just 3,566 regions, or .163% were covered by only one fuzzer. No fuzzer found many regions that

[1] `https://github.com/google/fuzzbench`

Fuzzer	Total Unique Regions
Honggfuzz	1121
entropic	1119
AFL++	870
fairfuzz	113
libfuzzer	69
Eclipser	65
mopt	49
AFL	47
aflfast	46
lafintel	38
aflsmart	29

Table 5.1: Fuzzers Unique Regions [MSS+21]

other fuzzers could not find. It also showed that, in general, the fuzzers that cover more regions of code are the ones that performed better. Table 5.1 shows the total number of unique regions covered by each fuzzer across all FuzzBench's benchmarks.

5.2 Qualitative Research

To build a qualitative research, we partly followed the *Open Source Maturity Model* (*OMM*) approach [PNS09]. The Open Source Maturity Model is a methodology for evaluating *Free/Libre Open Source Software* (*FLOSS*) and uses *Trustworthy elements* (*TWEs*) as central components. A trustworthy element is a specific factor or aspect of the software development process or a product result that indirectly influences the perception of the trustworthiness of the FLOSS development process. OMM is organized into levels; each is composed of its TWEs plus those of the lower level. The basic level collects the simplest TWEs while the highest-level groups the most difficult ones.

This model can be used by different categories of users, each with specific and different objectives. In our case, our target user is an agile development team, who need fuzzers that can be used in a versatile way to test internal products, possibly even to test closed source products that need to be integrated. For the part concerning the integration in the software development processes, they would like to install the fuzzers in their CI/CD pipeline; thus, they need fuzzers that can be easily integrated into software developed according to the *Test Driven Development* (*TDD*) methodology and which are: interchangeable, integrable and executable in cluster or multithreaded. They aim to find the most trivial security bugs to make their software free of low-hanging fruits bugs.

Compared to the original methodology we have simplified and added some TWEs, moreover we have not provided a score for each practice but we went briefly to describe whether the analyzed fuzzer could satisfy the requirement or not. Also, the TWEs linked to the evolution of the software have lost importance since there is no strict requirement for an evolving fuzzer.

5.2.1 First Level

In the first level were placed factors that describe the ease of use and essential integration of the software. The tool must be documented enough to allow easy integration with the target program and guarantee adequate support for at least one operating system. We evaluate if the repository is actively maintained and if the maintainers are actively managing the pull requests and issues raised by the community.

5.2.1.1 Product Documentation (PDOC)

The potential fuzzer integrator wants the product documentation to be comprehensive and easy to understand. In particular, the documentation of the fuzzer must describe the installation process and how to use the software in a basic way. Finally, the documentation must be easily accessible from a web page.

5.2.1.2 Licenses (LCS)

Licensing is not very binding in the fuzzing world as fuzzers are not commercialized but solely used to test the target program. However, licenses must meet minimum requirements. The license must allow using and modifying the software, which most open source licenses do with the copyright notice. Therefore this element evaluates what type of license the software is released with and whether it meets three minimum requirements: open-source, permits modifications, and private use.

5.2.1.3 Number of Commits and Bug Reports (DFCT)

The number of commits and bug reports are influential for fuzzer evaluation since they could be used as indicators of fuzzer popularity. They also may indicate that the product is actively developed and supported and that further change requests and bug reports will be undertaken.

5.2.1.4 Maintainability and Stability (MST)

This element assesses whether the project is actively developed, whether the maintainers evaluate pull requests created by third parties and whether new software versions are released periodically.

5.2.1.5 Technical Environment (ENV1)

This element verifies whether the fuzzer could be easily integrated with minimum constraints. The fuzzer installation procedure must be simple enough to not discourage its use. It should be runnable in a regular desktop computer running one of the primary operating systems used in the IT community. The tool must guarantee an interface for monitoring the fuzzing campaign or that at least there is a method for saving the test cases that have generated a crash.

5.2.2 Second Level

In the second level, we evaluate whether the fuzzer includes extensive documentation describing advanced features of the tool. For example, we analyze if the tool could run in parallel, clusters, on binary-only targets and if it implements standards that allow the latter's use in conjunction with other fuzzers. Finally, we describe if the tool has received contributions from third-party companies and enjoys popularity in the community.

5.2.2.1 Extensive documentation (EPDOC)

This element extends the PDOC element of the first level. In particular, it evaluates whether the fuzzer was released along with an academic paper or whether it includes extensive documentation. The latter should describe the internal workings of the fuzzer and, if any, how to use the advanced techniques made available.

5.2.2.2 Technical Environment (ENV2)

This element checks whether the fuzzer could be easily run in multithreaded and clustered mode on different machines. It is then checked whether it exposes an API to customize the fuzzing process directly and whether it makes available advanced techniques such as those described in Chapter 3.

5.2.2.3 Popularity of the SW Product (REP)

The more popular the software product is, the more likely the fuzzer is of good quality. Such popularity can be indicated by the number of users who have downloaded the product and are using it. Discussions in mailing lists, forums, bug reporting systems, and other communication environments are also relevant to indicate the popularity of a project.

5.2.2.4 Results of third party product evaluation (RASM)

This element verifies whether the fuzzer has been evaluated by at least one of the following entities: a company, a systematic review paper, or by the open-source community.

5.2.2.5 Use of Established and Widespread Standards (STD)

In the field of fuzzing, there are only a few de facto standards that derive from the great diffusion of some software over time. In our case, we will consider two elements: the *LibFuzzer fuzzer harness* is the former is, while the latter is the *AFL compatibility*.

Libfuzzer is a fuzzer created within the *LLVM project*[2] and which uses the instrumentation of their compiler to perform the tests. To fuzz a target with LibFuzzer is necessary to implement a *fuzz target*, a function that accepts an array of bytes and pass them to a function of the target library.

```
// fuzz_target.cc
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;  // Non-zero return values are reserved for future use.
}
```

The latter does not depend on LibFuzzer in any way and so it is possible to use it with other fuzzing engines such as AFL and Radamsa. To integrate the fuzzers with the *fuzz target*, it is therefore sufficient to use or write small drivers that combine the two components. This harness allows you to test the target application in persistent mode and is widely used to integrate open source projects within the OSSFuzz repository. The fuzzer that has gained the greatest popularity is certainly AFL, therefore research has often focused on creating variants. These can be integrated with each other thanks to the master-slave mode made available by AFL, and hence the *afl-compatible* term is derived.

[2] `https://llvm.org/`

5.2.2.6 Contribution from SW Companies (CONT)

For a potential integrator, the participation of a major IT company in the development or use of the fuzzer can be a positive sign. It can indicate that the fuzzer is a quality product and that it has already been effectively integrated into the development and testing environments of leading IT companies.

5.2.3 Fuzzers

5.2.3.1 AFL

American Fuzzy Lop (*AFL*) is a mutation-based, coverage-guided fuzzer that uses genetic algorithms to create new test cases.

AFL starts from an initial set of input (*Corpus*) and removes unnecessary ones while maintaining the same code coverage (*Corpus pruning*). If the corpus is not provided, then the fuzzer will create one itself; it is possibile, for example, to create jpeg images from simple text files by testing programs such as Libjpeg [Zal14b]. AFL then mutates test cases using splicing methods, including sequential bit flip, insertion of interesting integers such as `INT_MAX`, `INT_MIN`, `0`, and others. Finally, test cases are executed on the target input and selected for the subsequent execution, according to the coverage generated.

AFL uses a fork-server architecture that improves performance and makes fuzzing available in persistent mode [Zal, Zal14a]. AFL could also run in parallel or distributed mode with a master-slave architecture.

Test in binary-only mode relies on a patched version of QEMU that writes the coverage feedback data to a shared memory region. This way, each executed basic block is registered, and when QEMU exits, the fuzzer can recover the coverage feedback data. AFL in QEMU mode leverages a fork-server architecture instead of running the target from the beginning for each test case. Furthermore, it creates a channel between the emulator and the fork server, thus maintaining the translation block cache instead of invalidating it for each process created. As a result of these two optimizations, the overhead of the QEMU mode is roughly 2-5x, compared to 100x+ for Intel's dynamic binary instrumentation tool PIN [Zal]. Since the jumps between chained blocks do not call back the emulator, AFL disables translation block chaining, losing an essential optimization of QEMU.

TWEs	Description
PDOC	Yes. AFL makes available extensive documentation[3] that describes how to use the tool in standard and specific cases.
LCS	Yes. AFL is protected by Apache License 2.0, a permissive license whose main conditions require preservation of copyright and license notices.
DFCT	No. AFL is not actively developed aside from modifications to maintain the repository. For example, in the period of one month starting from 25 October 2021, 2 unresolved issues were created and no activity on the branches.
MST	No. Aside from minor maintenance changes, the project is not maintained.
ENV1	Yes. AFL can be run on a normal desktop computer with a Linux distribution.
EPDOC	Yes. The technical white paper [Zal] is also accessible which gives a quick overview on the internal functioning of the tool.
ENV2	Partially. Provides the possibility of using several fuzzers in parallel. it does not provide advanced fuzzing techniques and the fuzzing in binary only mode is limited and not optimized.
REP	Yes. AFL enjoys great popularity with over one thousand citations reported by Google Scholar and more than two-thousand stars on the official github repository.
RASM	Yes. AFL has been subjected to numerous third-party benchmarks [MSS+21, HHP20].
STD	Yes. Due to its popularity it has created, de facto, an "afl-compatible fuzzers" standard which allows the synchronization of different fuzzers. However, it is not compliant with libfuzzer `LLVMFuzzerTestOneInput` harnesses.
CONT	Yes. It was initially released by Google.

[3] `https://afl-1.readthedocs.io/`

5.2.3.2 AFL++

AFL++ is a fork of AFL that implements numerous additional features and optimizations. The user can choose from numerous configurations and plug-ins of techniques derived from academic research. It was common practice for researchers to create minor variations of AFL that demonstrated a particularly new technique for fuzzing but did not keep the repository active and evolving; to name a few: AFLFast [BPR17], FairFuzz [LS18], AFLSmart [PBS+19]. By providing a custom mutator API, AFL++ makes it easy to integrate any new technique, thus eliminating the numerous one-time-forks of AFL.

In contrast to AFL, AFL++ has managed to keep TCG block chaining by modifying the point where QEMU carries out the instrumentation [FMEH20, Bio18, Fio19]. The TCG instrumentation has been moved into the translated code by injecting a snippet of TCG intermediate representation at the beginning of every translation block. This way, a call back to the emulator for each TB executed is no longer necessary. QEMU package of AFL++ also provides the possibility to use Compare Coverage, and binary-only fuzzing could also be performed using Frida or the Unicorn engine.

TWEs	Description
PDOC	Yes. AFL++ makes available extensive documentation that describes how to use the tool in standard and specific cases.
LCS	Yes. AFL++ is protected by Apache License 2.0
DFCT	Yes. AFL++ is currently being updated with new features and issues are being resolved, for example, in the one-month period starting from 25 October 2021, 29 pull requests were merged and 13 issues closed.
MST	Yes. Over four hundred thirty seven issues have been closed and currently it is in its twenty-first release.
ENV1	Yes. AFL++ can be run on a normal desktop computer with a Linux distribution, Mac OS X and Windows/-CygWin.
EPDOC	Yes, AFL++ has been detailed in [FMEH20].
ENV2	Yes, AFL++ can run in parallel, custom mutators can be created, and provides various methods for binary-only fuzzing.
REP	Yes, AFL++ enjoys great popularity with AFL-like metrics.
RASM	Yes. AFL++ has been subjected to numerous third-party benchmarks [MSS+21, HHP20].
STD	Yes, AFL++ deriving from AFL is "afl-compatible" and is also compatible with Libfuzzer.
CONT	Yes, it is a fork of AFL++ that was initially released by Google.

5.2.3.3 Eclipser

Eclipser is a fuzzer that implements gray-box concolic testing through the use of lightweight instrumentation and, at the same time, does not use an SMT solver(Section 3.3). Eclipser builds on ideas developed in KLEE and MAYHEM [CJHC19] to maintain an independent subset for each input byte of a seed to effectively approximate path constraints. These conditions being less precise and easier to solve, allow Eclipser not to use an SMT solver and assume that they are linear. If the actual constraints do not satisfy these assumptions, Eclipser fails to generate relevant inputs. In practice, often, even SMT solvers fail on these conditions [Gro].

Starting from v2.0, Eclipser only performs gray-box concolic testing for test case generation and relies on AFL to perform random-based fuzzing.

TWEs	Description
PDOC	Yes. The GitHub repository [4] provides a concise, but sufficient documentation.
LCS	Yes. Eclipser is released under MIT license.
DFCT	No. The project does not seem active and many issues are open and inactive.
MST	No.
ENV1	No. It support only x86 and x64 architectures.
EPDOC	Yes. Eclipser has been described in detail in [CJHC19].
ENV2	Yes. Eclipser can fuzz in binary-only mode and being AFL compatible, it can be used in cluster mode with other fuzzers.
REP	Partially. The academic community has shown interest in Eclipser, but diffusion to the general public is still in its initial phase.
RASM	Yes. Eclipser has been subjected to numerous third-party benchmarks [LJC+21, MSS+21].
STD	Yes. It is AFL compatible.
CONT	No. Eclipser has not received any official contributions from software companies.

5.2.3.4 Honggfuzz

Honggfuzz is a mutation-based, feedback-driven fuzzer that uses the POSIX interface to monitor processes and detect crashes. It can use both compile-time and sanitizer-coverage instrumentation on the target binary and supports several hardware-based and software-based coverage feedback modes. Test in binary-only mode relies on QEMU, which provides the dynamic instrumentation; like AFL, it has TB chaining disabled.[5]

[4] `https://github.com/SoftSec-KAIST/Eclipser`

[5] `https://github.com/thebabush/honggfuzz-qemu/issues/1`

TWEs	Description
PDOC	Yes. Honggfuzz makes the documentation available in its repository.[6]
LCS	Yes. It is released under Apache-2.0 License.
DFCT	Yes. Honggfuzz is currently being updated and issues are opened by the community.
MST	Yes. Over two hundred issues have been closed and currently Honggfuzz is in its twenty-first release.
ENV1	Yes. It works under GNU/Linux, FreeBSD, NetBSD, Mac OS X, Windows/CygWin and Android.
EPDOC	Partially. The documentation describes in detail and with examples the use of the fuzzer, but no specific paper for Honggfuzz has been published.
ENV2	Yes. It is a multi-threaded and multi-process fuzzer, implements hardware-based coverage feedback, support persistent fuzzing mode and binary-only fuzzing.
REP	Yes. Honggfuzz has been widely adopted by the community [LJC+21].
RASM	Yes. Honggfuzz has been subjected to third-party benchmarks [LJC+21, MSS+21].
STD	Yes, it is compatible with Libfuzzer
CONT	Yes, it was initially released by Google.

5.2.3.5 LibFuzzer

LibFuzzer is an in-process, coverage-guided, evolutionary fuzzing engine. LLVM's SanitizerCoverage instrumentation provides the information to calculate the code coverage, while LibFuzzer uses the Listing 5.2.2.5 in Section 5.2 as an entrypoint. The target created by the user must follow some constraints including: it must not use `exit` on any input, threads must be joined at the end of the function, and it should not modify any global state. For these reasons, LibFuzzer is a suitable choice when dealing with API fuzzing, while it may not be the soundest option for entire programs testing.

[6] `https://github.com/google/honggfuzz`

TWEs	Description
PDOC	Yes. LibFuzzer makes the documentation available in the LLVM project site.[7]
LCS	Yes. It is released within the LLVM project which is licensed with Apache License, Version 2.0.
DFCT	Yes. LibFuzzer is currently being updated and issues are opened by the community.
MST	Yes. Being part of the llvm project, it indirectly enjoys the stability of the latter.
ENV1	Yes. It works under Linux, Mac OS X, Android, and Windows with some limitations.[8]
EPDOC	Partially. The documentation describes in detail and with examples the use of the fuzzer, but no specific paper for LibFuzzer has been published.
ENV2	Partially. LibFuzzer does not support binary only fuzzing.
REP	Yes. LibFuzzer and in particular the *fuzz target* has been widely adopted by the community. OSS-Fuzz use the LLVM's *fuzz target* as standard for integrating external repositories.
RASM	Yes. LibFuzzer has been subjected to third-party benchmarks [MSS+21].
STD	Yes. The *fuzz target* is widespread across the fuzzing community and LibFuzzer could be used together with AFL-like fuzzers.
CONT	Yes. The project is sponsorized by third party companies and its is used in important project, such as Chromium [LLCb].

5.2.3.6 Radamsa

Radamsa is a mutation-based, general-purpose black-box fuzzer created to test the resilience of programs to malformed inputs smoothly. Being a black-box type fuzzer, it does not need information about the program's source code or the input structure, thus making available remarkable agility and ease of use. On the other hand, Radamsa does not obtain performances comparable to other fuzzers, nor does it save interesting test cases and restart the target program after a crash. Thereby, it is often used in scripts that involve

[7] `https://llvm.org/docs/LibFuzzer.html`
[8] `https://llvm.org/docs/LibFuzzer.html`

complementary tools such as GDB[9] and Valgrind.[10]

There are tools related to Radamsa that extend its functionality; one example is Mutiny,[11] a network fuzzer that operates by sending mutated PCAPs packets thanks to Radamsa.

TWEs	**Description**
PDOC	Yes. Radamsa makes the documentation available in its repository. [12]
LCS	Yes. It is released under MIT license.
DFCT	Yes. Radamsa is actively developed and issues are being resolved.
MST	Yes. Over seventy issues have been resolved and currently Radamsa is in its twenty-first release.
ENV1	Yes. It works under GNU/Linux, FreeBSD, OpenBSD, Mac OS X, Windows/CygWin [Hel18].
EPDOC	Yes. Being a black-box fuzzer that makes simplicity its strong point, the repository documentation can be considered sufficient.
ENV2	No. Radamsa is a simple but versatile black-box fuzzer; therefore it does not make more complex tools available.
REP	Yes. Radamsa enjoys a good reputation with over three hundred citations reported by Google Scholar and more than one thousand two hundred stars on the github repository.
RASM	No. Radamsa was compared in a minor comparative evaluation [RTG+19].
STD	No. Radamsa is not compatible with other fuzzers nor implements any standards.
CONT	No.

[9] `https://www.sourceware.org/gdb/`
[10] `https://valgrind.org/`
[11] `https://github.com/Cisco-Talos/mutiny-fuzzer`
[12] `https://gitlab.com/akihe/radamsa`

Chapter 6

Fuzzing Cases

As test cases, we undertook test campaigns on two different pieces of software: the former is the *Apache HTTP* server, while the latter is proprietary software that implements a *Phasor Data Concentrator* (*PDC*) and communicates using the *IEEE C37.118* protocol. We carried out the PDC's fuzzing campaign in collaboration with a local company that allowed us to analyze and test the sources of one of their programs under active development.

6.1 Apache HTTP Server Fuzzing

As of January 2021, Netcraft estimated [Net04] that Apache served 24.63% of the millions of most trafficked websites, thus making it a core technology of the web. Thereby, we chose Apache HTTP server 2.4.49 as the first target. We compiled the target in conjunction with the *APR*[1], *APR-util*[2], *Nghttp2*[3], and *PCRE*[4] library dependencies.

AFL++ can fuzz in two different ways: if the program under test directly accepts the input from standard input, through the command shown in Listing Listing 6.1, if the program accepts input from files, using the command shown in Listing Listing 6.2, where `@@` indicates the position of the name of the test file, that AFL++ will replace.

Listing 6.1: fuzz via stdin

```
$ ./afl-fuzz -i test_dir -o findings_dir /path/to/program [params...]
```

Listing 6.2: fuzz via file

[1] `https://apr.apache.org/`
[2] `https://apr.apache.org/`
[3] `https://nghttp2.org/`
[4] `https://www.pcre.org/`

```
$ ./afl-fuzz -i test_dir -o findings_dir /path/to/program @@
```

To fuzz the server, we have to provide test data through the socket on which it is listening. The approach we adopted was to modify the source code of the target program in order to create a thread that reads and forwards the fuzzing input to the server listening on port 8080.

To have multiple processes with the same settings and listening on the same loopback interface and port, the added code (Listing 6.3) uses the `unshare` function that disassociates parts of the process execution context, such as mount and network namespaces. The source code modification empowered us to further optimize the fuzzing campaign by using AFL++ in persistent mode (Section 3.6), which permits testing a single forked process multiple times instead of forking a new process for each test execution.

Listing 6.3: Disassociation of the execution context to fuzz Apache HTTP

```
unshare(CLONE_NEWUSER | CLONE_NEWNET | CLONE_NEWNS);
if (mount("tmpfs", "/tmp", "tmpfs", 0, "") == -1) {
    perror("tmpfs");
    _exit(1);
}
netIfaceUp("lo");
int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
if (sock == -1) {
    perror("socket(AF_INET, SOCK_STREAM, IPPROTO_IP)");
    _exit(1);
}
struct ifreq ifr;
memset(&ifr, '\0', sizeof(ifr));
snprintf(ifr.ifr_name, IF_NAMESIZE, "%s", "lo");
if (ioctl(sock, SIOCGIFFLAGS, &ifr) == -1) {
    perror("ioctl(iface='lo', SIOCGIFFLAGS, IFF_UP)");
    _exit(1);
}
ifr.ifr_flags |= (IFF_UP | IFF_RUNNING);
if (ioctl(sock, SIOCSIFFLAGS, &ifr) == -1) {
    perror("ioctl(iface='lo', SIOCSIFFLAGS, IFF_UP)");
    _exit(1);
}
close(sock);
```

Only a fraction of the internal state of the HTTP server can be easily restored; thus, to improve stability, we limited the number of iterations for each forked process to one

thousand, and we excluded the APR, APR-util, Nghttp2, and PCRE libraries from the instrumentation. We excluded those libraries because they are external to Apache HTTP.

The fuzzing campaign took place on an Intel Core i7-4770 with 32GB DDR3 RAM, where several AFL++ fuzzers were executed in master-slave mode, each with different enabled options and on targets compiled with different options. Specifically, each binary was compiled using `afl-clang-lto` (Section 3.7), and each had MSAN or ASAN (Section 3.8) or laf-intel (Section 3.1) enabled respectively. We also utilized instances of AFL++ with MOPT [LJZ+19] and CmpLog (Section 3.5) in round-robin because the number of virtual cores was insufficient to run all at once without performance degradation.

The fuzzing campaign ran for eight days but we did not find any crashes. This is probably due to the fact that Apache HTTP is constantly being fuzzed by several security teams, researchers and automatic fuzzers such as OSS Fuzz.

During this period, the *CVE-2021-41773*[5] was published. This CVE describes a path-traversal vulnerability affecting the version we tested. Path traversal vulnerabilities are implicitly complex for a fuzzer to find; they do not generate any crash but only an unwanted behavior. Despite this, we have shown that it is possible to find this type of vulnerability by developing a shared library to be loaded on the target program. This shared library is loaded with the `AFL_PRELOAD` environment variable, which causes AFL++ to set `LD_PRELOAD` for the target binary without disrupting the `afl-fuzz` process itself. In the shared library we created two implementations for the functions `__xstat` and `writev` that modified the behavior of the two original functions using a technique known as the "`LD_PRELOAD` trick". In our particular case, we extended the default behaviour of `__xstat` in order to compare the requested path with the path of the server base folder to check if the requested resource was outside of it, and `writev` to throw an error if the server was actually writing the HTTP response for the theoretically unreachable file.

We used this method for test purposes, and we are aware that it is difficult to use in a real fuzzing campaign, nevertheless it allowed us to try the preload functionality of AFL++ that we used in the second experiment successfully.

6.2 Phasor Data Concentrator Fuzzing

In modern Smart Grids, *Supervisory control and data acquisition* (*SCADA*) systems are widely used to gather information, monitor and control the infrastructure of a distributed system. A SCADA system is usually composed of several *Remote terminal units*(*RTUs*) and *Master Terminal Units*(*MTUs*). The former provides the physical system to measure and control the smart-grid physical equipment, while the latter are intermediate nodes

[5] `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-41773`

responsible for collecting the RTUs' data and feeding RTUs with control messages.

In this context, the *Phasor Measurement Unit* (*PMU*) is the application-specific RTU that measures the electrical waves of the power distribution infrastructure to solve faults and evaluate the system status.

Phasor Data Concentrator's (*PDC*) primary purpose is to collect and combine synchrophasor measurements from many phasor measurement units into a single synchronized data stream [65113]. However, due to the increasing size of measurement systems, usually, PDCs include data handling, processing, and storage, thus making them fundamental nodes within smart-grids.

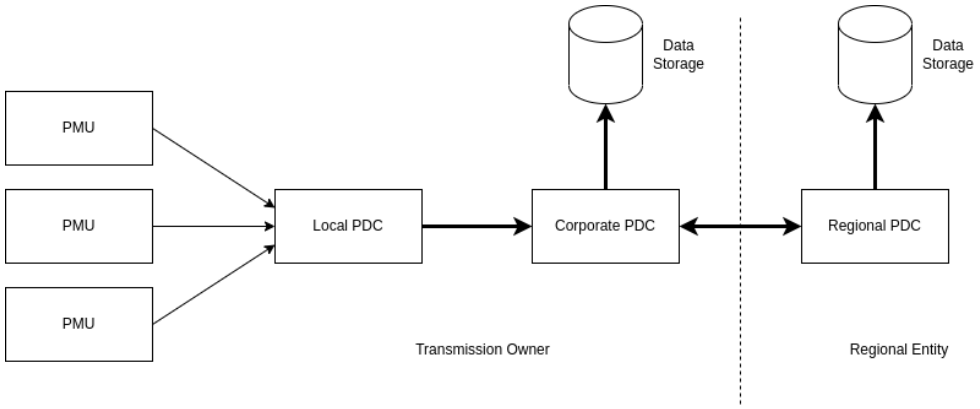


Figure 6.1: Synchrophasor data collection network

Currently, the *IEEE C37.118 standard* [61111] is widely used for the communication between these nodes, and it provides standardized message formats for real-time data transmission and control of the PDCs/PMUs.

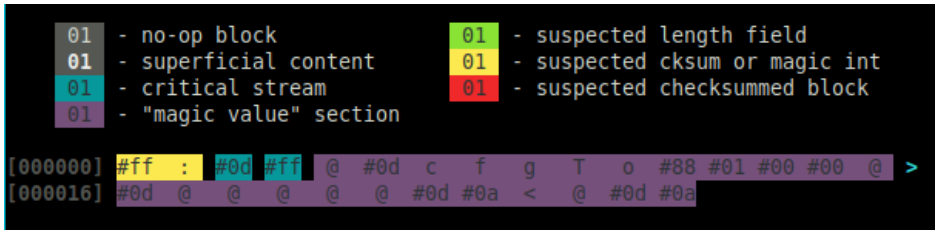
The standard does not provide any mechanism to ensure data integrity and confidentiality nor specify the underlying data communication protocol; therefore, these messages are usually transmitted over TCP/IP and LANs. Companies usually employ VPNs and VLANs to mitigate these issues, but the protocol's lack of encryption and authentication remains.

The C37.118 protocol is usually used with the client/server paradigm, where the PDC connects to one or more PMUs listening for connections and provides five types of messages: *Header Frame*, *Configuration Frame*, *Data Frame*, and *Command Frame*. Each of these frames implements a specific function: the Header Frame contains human-readable information about the PMU, the Data Frame contains synchrophasor and frequency measurements in binary format, the Command Frame is sent by the PDC to PMUs to toogle the data transmission and to retrieve PMU's configuration information, and the Configuration Frame describes the structure and fields that the Data Frame will use.

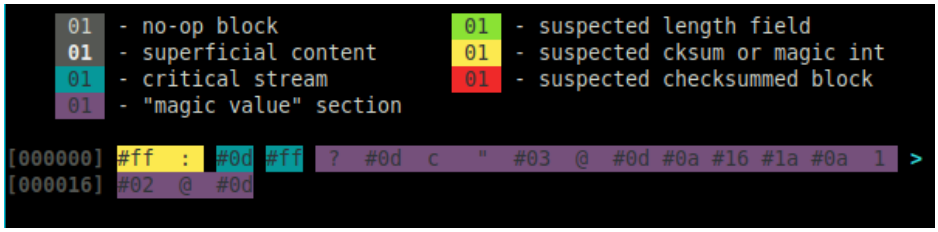
In collaboration with a company, we tested an in development software that will implement

a PDC's requirements and uses the C37.118 standard. We tested the target program for UDP and TCP connections made by both PMUs and upper layer PDCs. In all of the approaches used, a white list had to be modified to avoid packet rejection by the PDC. Compared to the approach used to fuzz Apache HTTP, we have not used AFL++ in persistent mode, but instead, we have created a shared library, which allowed us to emulate the functioning of a real network.

The fuzzing campaign detected various unique crashes in the handling of frames; their size starts from just 19 bytes up to 158KB in size. Figure 6.2 shows two inputs that cause the error, analyzed with the tool.



(a) Crash one



(b) Crash two

Figure 6.2: Crashes

Comparing our input with the message structure of the standard Figure 6.3, we can say that the critical section corresponds to the `FRAMESIZE` field. All message frames start with a two-byte `SYNC` word followed by a two-byte `FRAMESIZE` word and terminate in a check word (`CHK`) which is a *CRC-CCITT*.

By analyzing the inputs and the source code we found that there are two specific families of inputs causing a crash. The former occurs during the calculation of the CRC of the *Configuration frame 2* (*CFG2*), while the latter is specific to incorrect handling of an invalid field in the frame.

The target program calculates the CRC's position starting from the `FRAMESIZE` value; therefore if it is greater than the actual size of the message frame, the target program will read in an undefined memory page and, if not allocated, generates a segmentation fault. Hence, the non-sanitization of the `FRAMESIZE` value is the cause of this first vulnerability.

Field	Size (bytes)	Comments
SYNC	2	Frame synchronization word. Leading byte: AA hex Second byte: Frame type and version, divided as follows: Bit 7: Reserved for future definition, must be 0 for this standard version. Bits 6–4: 000: Data Frame 001: Header Frame 010: Configuration Frame 1 011: Configuration Frame 2 101: Configuration Frame 3 100: Command Frame (received message) Bits 3–0: Version number, in binary (1–15) Version 1 (0001) for messages defined in IEEE Std C37.118-2005 [B6]. Version 2 (0010) for messages added in this revision, IEEE Std C37.118.2-2011.
FRAMESIZE	2	Total number of bytes in the frame, including CHK. 16-bit unsigned number. Range = maximum 65535
IDCODE	2	Data stream ID number, 16-bit integer, assigned by user, 1–65534 (0 and 65535 are reserved). Identifies destination data stream for commands and source data stream for other messages. A stream will be hosted by a device that can be physical or virtual. If a device only hosts one data stream, the IDCODE identifies the device as well as the stream. If the device hosts more than one data stream, there shall be a different IDCODE for each stream.
SOC	4	Time stamp, 32-bit unsigned number, SOC count starting at midnight 01-Jan-1970 (UNIX time base). Range is 136 years, rolls over 2106 AD. Leap seconds are not included in count, so each year has the same number of seconds except leap years, which have an extra day (86 400 s).
FRACSEC	4	Fraction of second and Time Quality, time of measurement for data frames or time of frame transmission for non-data frames. Bits 31–24: Message Time Quality as defined in 6.2.2. Bits 23–00: FRACSEC, 24-bit integer number. When divided by TIME_BASE yields the actual fractional second. FRACSEC used in all messages to and from a given PMU shall use the same TIME_BASE that is provided in the configuration message from that PMU.
CHK	2	CRC-CCITT, 16-bit unsigned integer.

Figure 6.3: Word definitions common to all frame types [61111]

AFL++ found the second vulnerability in a very fascinating and lucky way. Initially the target program was not compiled with any kind of sanitization; nevertheless the first vulnerability was found thanks to the segmentation fault generated by the operating system. If we had used MSAN all the out of bounds reads would have generated a crash, and this would have paradoxically slowed down the discovery of the second vulnerability. In fact, AFL++ was able to find inputs that generated a checksum equal to the CRC value read from uninitialized memory areas. Using GDB with the generated test cases we could see that the CRC read from uninitialized memory areas was, in most cases, bytes set to 0. This is due to the operating system that, for security reasons, resets the pages to zero before assigning them to processes. If we had instead used MSAN it would have been necessary patching the program, disabling checksum testing, or using a generative approach to continue fuzzing. After analyzing the two vulnerabilities, we patched the target program to

eliminate the undefined behaviors and continue the fuzzing campaign in a deterministic manner. Non-determinism makes fuzzing ineffective and should be avoided as much as possible.

The second vulnerability, found despite the undefined behavior, is located in the function responsible for parsing CFG2 sent from a PMU to a PDC. The CFG2 has a specific structure represented by the Figure 6.4.

No	Field	Size (bytes)	Short description
1	SYNC	2	Sync byte followed by frame type and version number.
2	FRAMESIZE	2	Number of bytes in frame, defined in 6.2.
3	IDCODE	2	Stream source ID number, 16-bit integer, defined in 6.2.
4	SOC	4	SOC time stamp, defined in 6.2.
5	FRACSEC	4	Fraction of Second and Message Time Quality, defined in 6.2.
6	TIME_BASE	4	Resolution of FRACSEC time stamp.
7	NUM_PMU	2	The number of PMUs included in the data frame.
8	STN	16	Station Name—16 bytes in ASCII format.
9	IDCODE	2	Data source ID number identifies source of each data block.
10	FORMAT	2	Data format within the data frame.
11	PHNMR	2	Number of phasors—2-byte integer (0 to 32 767).
12	ANNMR	2	Number of analog values—2-byte integer.
13	DGNMR	2	Number of digital status words—2-byte integer.
14	CHNAM	16 × (PHNMR + ANNMR + 16 × DGNMR)	Phasor and channel names—16 bytes for each phasor, analog, and each digital channel (16 channels in each digital word) in ASCII format in the same order as they are transmitted. For digital channels, the channel name order will be from the least significant to the most significant. (The first name is for bit 0 of the first 16-bit status word, the second is for bit 1, etc., up to bit 15. If there is more than 1 digital status, the next name will apply to bit 0 of the second word and so on.)
15	PHUNIT	4 × PHNMR	Conversion factor for phasor channels.
16	ANUNIT	4 × ANNMR	Conversion factor for analog channels.
17	DIGUNIT	4 × DGNMR	Mask words for digital status words.
18	FNOM	2	Nominal line frequency code and flags.
19	*CFGCNT*	2	Configuration change count.
	Repeat 8–19		Fields 8—19, repeated for as many PMUs as in field 7 (NUM_PMU).
20+	DATA_RATE	2	Rate of data transmissions.
21+	CHK	2	CRC-CCITT.

Figure 6.4: IEEE C37.118 Configuration Frame 2 [61111]

Using GDB, we have found that the interesting part of the input are the bytes corresponding to `NUM_PMU` which determines the number of PMUs included in the data frame. The target program reads out of bound with values large enough or not in line with the frame size thus making a vulnerability.

The second approach used to fuzz the PDC was to use the black-box fuzzer Radamsa in conjunction with the open source *Pypmu*[6] software which implements in Python the IEEE C37.118.2 standard for a Synchrophasor module.

[6] `https://github.com/iicsys/pypmu`

Since this software manages the protocol entirely, we have also been able to fuzz the Data Frames provided by the protocol without any modification of the target program's source code. The PDC accepts a Data Frame only after receiving a Configuration Frame, and in the case of AFL++ this would have meant modifying the source code of the target program to simulate the reception of a configuration frame.

No.	Field	Size (bytes)	Comment
1	SYNC	2	Sync byte followed by frame type and version number.
2	FRAMESIZE	2	Number of bytes in frame, defined in 6.2.
3	IDCODE	2	Stream source ID number, 16-bit integer, defined in 6.2.
4	SOC	4	SOC time stamp, defined in 6.2, for all measurements in frame.
5	FRACSEC	4	Fraction of Second and Time Quality, defined in 6.2, for all measurements in frame.
6	STAT	2	Bit-mapped flags.
7	PHASORS	4 × PHNMR or 8 × PHNMR	Phasor estimates. May be single phase or 3-phase positive, negative, or zero sequence. Four or 8 bytes each depending on the fixed 16-bit or floating-point format used, as indicated by the FORMAT field in the configuration frame. The number of values is determined by the PHNMR field in configuration 1, 2, and 3 frames.
8	FREQ	2 / 4	Frequency (fixed or floating point).
9	DFREQ	2 / 4	ROCOF (fixed or floating point).
10	ANALOG	2 × ANNMR or 4 × ANNMR	Analog data, 2 or 4 bytes per value depending on fixed or floating-point format used, as indicated by the FORMAT field in configuration 1, 2, and 3 frames. The number of values is determined by the ANNMR field in configuration 1, 2, and 3 frames.
11	DIGITAL	2 × DGNMR	Digital data, usually representing 16 digital status points (channels). The number of values is determined by the DGNMR field in configuration 1, 2, and 3 frames.
	Repeat 6–11		Fields 6–11 are repeated for as many PMUs as in NUM_PMU field in configuration frame.
12+	CHK	2	CRC-CCITT

Figure 6.5: IEEE C37.118 Data Frame [61111]

During the fuzzing campaign with Radamsa, we found five vulnerabilities: the first two are those already found with AFL++ in the CFG, the other three are related to the Data Frame. The first vulnerability is triggered by sending a Data Frame with incorrect fields, which the target program will handle badly by running `free` system call on uninitialized pointers. The second vulnerability is an out-of-bounds read that occurs when the number of PMUs defined in the CFG frame, used to calculate offsets within the data frame, is not aligned with the actual packet size. The third vulnerability is similar to the one found in the CFG frame, and concerns the non-sanitization of the `FRAMESIZE` field.

Although this approach is very simple to apply, it suffers from the disadvantages of black-box fuzzing: the highly structured input makes most of the test cases generated by the fuzzer unusable because the checksum check blocks them.

The third and last approach used was generative and is based on *Scapy* [Bio10], a powerful and interactive packet manipulation program, which we have extended to manage the

C37.118 protocol.

The framework provides classes and fields to quickly and easily define all the package components for a specific protocol. Listing 6.4 shows part of the code that implements the CFG2. The complete code is available in the repository[7] and once completed in detail a pull request will be made to be integrated into Scapy.

Listing 6.4: CFG-Frame2 Packet class

```
class ConfigurationFrame2(Packet):
  name = "ConfigurationFrame2"
  fields_desc = [
      XByteField("syncHead", 0xaa),
      XBitField("syncReserved", 0b0, 1),
      BitEnumField("syncFrameType", 0b011, 3,
                   CommonFieldsConstants.SyncFieldConstants.FRAME_TYPES),
      BitEnumField("syncVersion", 0b0001, 4,
                   CommonFieldsConstants.SyncFieldConstants.VERSIONS),
      ShortField("framesize", None),
      ShortField("idcode", 7734),
      IntField("soc", 1149577200),
      XBitField("fracsecReserved", 0b0, 1),
      BitEnumField("fracsecLeapSecDirection", 0b1, 1,
                   CommonFieldsConstants.FracSecFieldConstants.LEAP_SEC_DIRECTION),
      BitEnumField("fracsecLeapSecOccured", 0b0, 1,
                   CommonFieldsConstants.FracSecFieldConstants.LEAP_SEC_OCCURED),
      BitEnumField("fracsecLeapSecPending", 0b1, 1,
                   CommonFieldsConstants.FracSecFieldConstants.LEAP_SEC_PENDING),
      BitEnumField("fracsecTimeQuality", 0b0110, 4,
                   CommonFieldsConstants.FracSecFieldConstants.TQ_CODE),
      ThreeBytesField("fracsecValue", 0x071098),
      ByteField("timeBaseFlags", 0x00),
      ThreeBytesField("timeBaseValue", 0x0f4240),
      ShortField("numPmu", 1),
      PacketListField("configurationFrame2ListEntries",
                      [ConfigurationFrame2Entry(), ],
                      ConfigurationFrame2Entry),
      SignedShortField("dataRate", 30),
      ShortField("chk", None)
  ]

  def post_build(self, pkt: bytes, pay: bytes) -> bytes:
      if self.framesize is None:
          pkt = pkt[: 2] + struct.pack("!H", len(pkt)) + pkt[4:]
      if self.chk is None:
          pkt = pkt[: -2] + struct.pack("!H", crc16xmodem(pkt[:-2], 0xffff))
      return pkt + pay
```

To check the correct implementation of the protocol, we have reproduced the example (Listing 6.5) provided by the IEEE standard.

[7] https://github.com/Strafo/scapy

Listing 6.5: CFG2 Frame example using Scapy

```
###[ ConfigurationFrame2 ]###
  syncHead  = 0xaa
  syncReserved= 0x0
  syncFrameType= Cfg Frame 2
  syncVersion= Version 1
  framesize = None
  idcode    = 7734
  soc       = 1149577200
  fracsecReserved= 0x0
  fracsecLeapSecDirection= Leap second delete
  fracsecLeapSecOccured= No leap second occured
  fracsecLeapSecPending= Leap second pending
  fracsecTimeQuality= Time within 10^-4 s of UTC
  fracsecValue= 463000
  timeBaseFlags= 0
  timeBaseValue= 1000000
  numPmu    = 1
  \configurationFrame2ListEntries\
   |###[ ConfigurationFrame2Entry ]###
   |  stn       = 'Station A       '
   |  idCode    = 7734
   |  formatUnused= 0x0
   |  formatFreqSize= 16-bit Integer
   |  formatAnalogsSize= Floating point
   |  formatPhasorsDataSize= 16-bit Integer
   |  formatPhasorsType= Rectangular
   |  phnmr     = 4
   |  annmr     = 3
   |  dgnmr     = 1
   |  chnamList = ['VA              ', [...] ,'BREAKER G STATUS']
   |  \phUnitList\
   |   |###[ PhUnitField ]###
   |   |  type      = voltage
   |   |  scaling   = 915527
   |   |###[ PhUnitField ]###
   |   |  type      = voltage
   |   |  scaling   = 915527
   |   |###[ PhUnitField ]###
   |   |  type      = voltage
   |   |  scaling   = 915527
   |   |###[ PhUnitField ]###
   |   |  type      = current
   |   |  scaling   = 45776
   |  \anUnitList\
   |   |###[ AnUnitField ]###
   |   |  type      = single point-on-wave
   |   |  scaling   = 1
   |   |###[ AnUnitField ]###
   |   |  type      = rms of analog input
   |   |  scaling   = 1
   |   |###[ AnUnitField ]###
   |   |  type      = peak of analog input
   |   |  scaling   = 1
   |  \digUnitList\
   |   |###[ DigUnitField ]###
   |   |  normalStatus= 0x0
   |   |  validInputs= 0xffff
   |  fnomReserved= 0
   |  fnomHead  = Frequency 60Hz
   |  cfgCnt    = 22
  dataRate  = 30
  chk       = None
```

Fuzzing through Scapy enabled us to create complex and standard-compliant packets that the PDC accepts. If in the first two approaches most of the packets created were blocked by the checksum and, in later versions, by the `FRAMESIZE` checks, now with the generative approach, all packets can pass them. The framework allows the definition of *computable fields*; these values are determined only after the packet fuzzing process, which modifies packet fields by substituting random values consistent with the field type.

We have fuzzed the target program in both the original and patched version. The vul-

nerabilities found in the former are those already discussed; for the latter version, no vulnerabilities were found.

Chapter 7

Conclusion

In the first part, we taxonomized fuzzers and introduced basic fuzzing concepts to build the knowledge base for the novice reader in this field.

The core of this thesis illustrates the techniques and optimizations used for fuzzing and an analysis of the main open-source available solutions. We explained the internal working of each technique and described the advantages and disadvantages it can bring.

We then showed an overview of the principal methods for binary-only fuzzing and analyzed some of the optimizations created to improve performance. The world of fuzzing is constantly evolving and often scattered; therefore, we showed some projects attempting to create a more mature research field.

We compared some of the major open-source fuzzers showing each of their main features and created an analysis to get a quick overview of them.

In the final part, we showed examples of the use of the discussed theory. We empirically showed the advantages and disadvantages of using different approaches, such as the generative, the fork-server, and the differences between black-box and gray-box fuzzing.

Bibliography

[61111] Ieee standard for synchrophasor measurements for power systems. *IEEE Std C37.118.1-2011 (Revision of IEEE Std C37.118-2005)*, pages 1–61, 2011. doi:10.1109/IEEESTD.2011.6111219.

[65113] Ieee guide for phasor data concentrator requirements for power system protection, control, andmonitoring. *IEEE Std C37.244-2013*, pages 1–65, 2013. doi:10.1109/IEEESTD.2013.6514039.

[Ait02] Dave Aitel. The advantages of block-based protocol analysis for security testing. *Immunity Inc., February*, 105:106, 2002.

[ASB+19] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[BCD+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[Bio10] Philippe Biondi. Scapy documentation. *vol*, 469:155–203, 2010.

[Bio18] Andrea Biondo. Andrea Biondo improving afl qemu mode. https://abiondo.me/2018/09/21/improving-afl-qemu-mode/, 2018. Accessed: 2021-12-04.

[BPR17] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[Bö15] Hanno Böck. Hanno Böck how heartbleed could've been found. `https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html`, 2015. Accessed: 2021-12-04.

[CC18] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[CJHC19] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *Proceedings of the International Conference on Software Engineering*, pages 736–747, 2019.

[CMX+19] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 633–645, 2019.

[Cob16] Stephen Cobb. Mind this gap: criminal hacking and the global cybersecurity skills shortage, a critical analysis. In *Virus Bulletin Conference*, pages 1–8, 2016.

[DA02] Inc Dave Aitel, Immunity. Dave Aitel blackhat 2002 presentation. `https://www.blackhat.com/html/bh-usa-02/bh-usa-02-speakers.html#Aitel`, 2002. Accessed: 2022-01-07.

[DBXP20] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.

[DGR20] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–163, 2020.

[FDC20] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. Weizz: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13, 2020.

[Fio19] Andrea Fioraldi. Andrea Fioraldi afl++ qemu compcov. `https://andreafioraldi.github.io/articles/2019/07/20/aflpp-qemu-compcov.html`, 2019. Accessed: 2021-12-04.

[FMEH20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[GDR21] Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. Scalable fuzzing of program binaries with e9afl. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1247–1251. IEEE, 2021.

[Gro] Alex Groce. Fuzzing unit tests with deepstate and eclipser. https://blog.trailofbits.com/2019/05/31/fuzzing-unit-tests-with-deepstate-and-eclipser/. Accessed: 2021-12-18.

[GZQ+18] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018. doi:10.1109/SP.2018.00040.

[Hel18] Aki Helin. Radamsa-a general purpose fuzzer. *URl: https://gitlab. com/akihe/radamsa*, 32, 2018.

[HHP20] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, 2020.

[KRC+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[laf16] laf-intel blog. https://lafintel.wordpress.com/, 2016. Accessed: 2021-12-18.

[LCC+17] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

[Lea04] Advanced Fuzzing League. Libafl. https://github.com/AFLplusplus/LibAFL, Accessed: 2021-12-04.

[LJC+21] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. {UNIFUZZ}: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[LJZ+19] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.

[LLCa] Google LLC. Google LLC afl. `https://github.com/google/AFL`. Accessed: 2021-12-04.

[LLCb] Google LLC. Google LLC getting started with fuzzing in chromium. `https://chromium.googlesource.com/chromium/src/testing/libfuzzer/+/HEAD/getting_started.md`. Accessed: 2021-12-04.

[LS18] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[LZZ18] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[MFS90] 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.

[MHH+19] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[Mil] Charlie Miller. Fuzzing with code coverage by example. `https://www.ise.io/wp-content/uploads/2019/11/cmiller_toorcon2007.pdf`.

[MP+07] Charlie Miller, Zachary NJ Peterson, et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 4, 2007.

[MRH19] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association. URL: `https://www.usenix.org/conference/woot19/presentation/maier`.

[MSS+21] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1393–1403, 2021.

[Net04] Netcraft. December 2020 web server survey. `https://news.netcraft.com/archives/2020/12/22/december-2020-web-server-survey.html`, Accessed: 2021-12-04.

[Ney08] John Neystadt. Automated penetration testing with white-box fuzzing. *MSDN Library*, 2008.

[ope16] openssl.org. CVE-2016-6309 cve-2016-6309. `https://www.openssl.org/news/secadv/20160926.txt`, 2016. Accessed: 2021-12-04.

[PBS+19] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[PNS09] Etiel Petrinja, Ranga Nambakam, and Alberto Sillitti. Introducing the opensource maturity model. In *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 37–41, 2009. `doi:10.1109/FLOSS.2009.5071358`.

[Pro] LLVM Project. Libfuzzer website. `https://llvm.org/docs/LibFuzzer.html`. Accessed: 2021-12-18.

[PSP18] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[QEM04] QEMU. Qemu documentation. `https://www.qemu.org/docs/master/`, Accessed: 2021-12-04.

[Qua04] Qualipso. Qualipso open maturity model. `http://qualipso.icmc.usp.br/OMM/`, Accessed: 2021-12-04.

[RJK+17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[RTG+19] A Romanovsky, E Troubitsyna, I Gashi, E Schoitsch, and F Bitsch. Comparative evaluation of security fuzzing approaches. 2019.

[Sch] Google Scholar. Google fuzzing. https://scholar.google.com/scholar?q=fuzzing&hl=en&as_sdt=0%2C5&as_ylo=2014&as_yhi=. Accessed: 2021-12-04.

[Ser17] Kostya Serebryany. Oss-fuzz-google's continuous fuzzing service for open source software. 2017.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[SGS+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.

[Swi] R. Swiecki. honggfuzz. https://github.com/google/honggfuzz. Accessed: 2021-12-18.

[Vim15] Mikko Vimpari. An evaluation of free fuzzing tools. *Master's Thesis, University of Oulu*, 2015.

[Vos04] Nathan Voss. afl-unicorn: Fuzzing arbitrary binary code. https://medium.com/hackernoon/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf, Accessed: 2021-12-04.

[WDS+19] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 1–15, 2019.

[WWGZ10] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

[YLX+18] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

[Zal] Michal Zalewski. Michal Zalewski afl technical whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2021-12-04.

[Zal14a] Michał Zalewski. Fuzzing random programs without execve (), 2014.

[Zal14b] Michal Zalewski. Pulling jpegs out of thin air. *lcamtuf's blog*, 7, 2014.

[Zer] Google Project Zero. Comparecoverage. `https://github.com/googleprojectzero/CompareCoverage`. Accessed: 2021-12-18.