École Polytechnique Fédérale de Lausanne

# Leveraging QEMU plugins for fuzzing

by Marwan Azuz

Semester project report

Prof. Mathias Payer
Supervisor

Florian Hofhammer
Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 10, 2025

What's reality? I don't know...

— Terry A. Davis

No one to dedicate this work to, except to the ones that beared my existence.

# Acknowledgments

This work would not have been possible without the amazing team at HexHive, and those that have seen me yap about how I loved working on this project.

I would like to thank as well Florian Hofhammer for his guidance and support throughout the project.

As well, I am scared of Prof. Mathias Payer.

*Lausanne, November 4, 2024* Marwan

# Abstract

No idea to write here

# Contents

Chapter 0

# Introduction

*American-Fuzzy-Lop plus plus* (AFL++) is a popular fuzzer that has been used to find numerous security vulnerabilities in software. The most effective way to fuzz is to have coverage-guided fuzzing: when source code is available, AFL++ provides compiler plugins to compile the target software with instrumentation that allows feedback to the fuzzer on interesting inputs. However, for closed-source software, this is not possible. There are several ways to address this issue, one of them is to use CPU emulation and virtual machines. Therefore, AFL++ can be configured to use *Quick Emulator* (QEMU) using that has been patched to support AFL++, namely `qemuafl`.

However, the starting point of the fork of `qemuafl` is 4 years old (commit 5c65b1f) and having a more recent of QEMU would be beneficial. Using a fork is not ideal as fixes and improvements in the mainline QEMU are not available. For end-users, it is not ideal to install twice the same software, each having different features for the same goal, and each requiring another `make all`. QEMU 4.2 has introduced a plugin system letting users extend QEMU with custom code. Instead of using a fork of QEMU, it would be beneficial to use the plugin system to integrate AFL++ with QEMU, distinctively and independently from the mainline QEMU.

The challenges of this project are numerous, as QEMU is a complex piece of software, maintained by a large community. The plugin system is sometimes not well documented, and the interaction with AFL++ or QEMU is not straightforward. The goal of this project is to add a piece of peace to the puzzle, by providing a plugin that can be gently added to QEMU, filling the gap between a powerful emulator and a powerful fuzzer. How powerful and limiting a plugin is, is yet to be read by the reader currently reading this sentence.

In this report, we present the design and implementation of a QEMU plugin that integrates AFL++ with QEMU, with very little to no effort. We will show that our current implementation is able to fuzz simple programs, that room for improvements is still there and that the plugin system remains to be improved.

# Chapter 1

# Background

Fuzzing is like a lab experiment with monkeys: if they discover that doing a certain action gives them a banana, they will keep doing it; replace the banana by a program crash and the monkey by a fuzzer and you have a very simple analogy of what fuzzing is. To be more formal and to put words on the analogy, fuzzing's goal is to find crashes. It follows a simple routine: generate an input to a program, run the program with the input, and observe the behavior of the program. If it crashes, it likely indicate an issue in the program. The end-user is then free to investigate the crash and fix the issue, as some of these crashes can turn out to be security vulnerabilities.

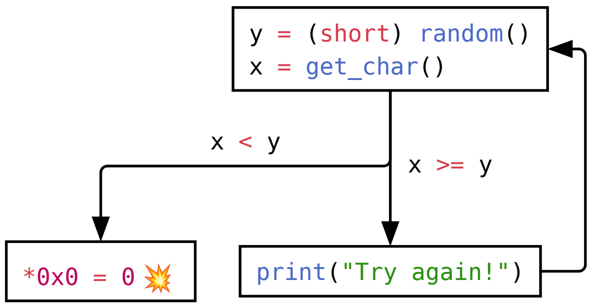
However, the fuzzer might take a while to find what action gives them the ~~banana~~ crash. To guide the fuzzer in the right direction, we forward to the fuzzer some feedback. This feedback is usually a coverage map of the program, that tells the fuzzer which parts of the program have been executed. Upon finding an input that triggers a new part of the program, the fuzzer will keep this input and try to mutate it to possibly explore more parts of the program. Many mutators exist, and a fuzzer can be as simple as a random mutator or as complex as a genetic algorithm.

"**How coverage is obtained?**" you might ask. Many tools out there exist but they essentially boil down to: at desired instructions in the program, before executing, do *something*. The *something* can be as simple as incrementing a counter, or as complex as recording the path taken by the program. The simplest way taken by AFL++'s instrumentation is to increment a counter at each basic block of the program. A basic block is a sequence of instructions that is always executed in sequence; a program is a list of basic blocks that are connected by jumps.

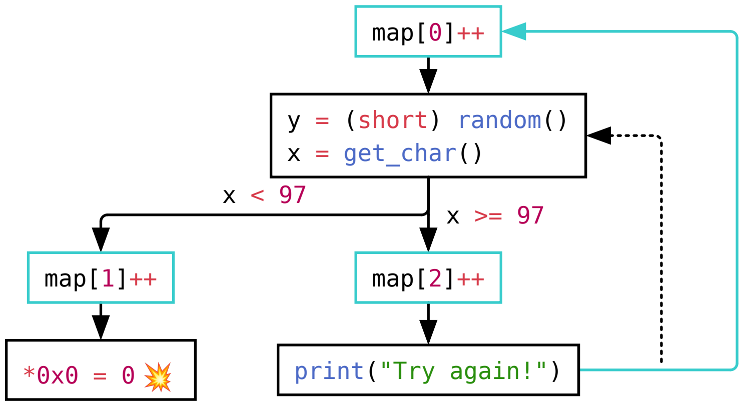
In the following subsections, we will devise the requirements for the plugin.

### Instrumentation under the hood

Let's take a simple program that reads a character from the standard input and either crashes or prints "Try again!" depending on the character read:



Compile-time instrumentation will insert the following code (highlighted in teal) at the beginning of each basic block:



Therefore upon an execution, the fuzzer will have information on which basic block has been inserted.

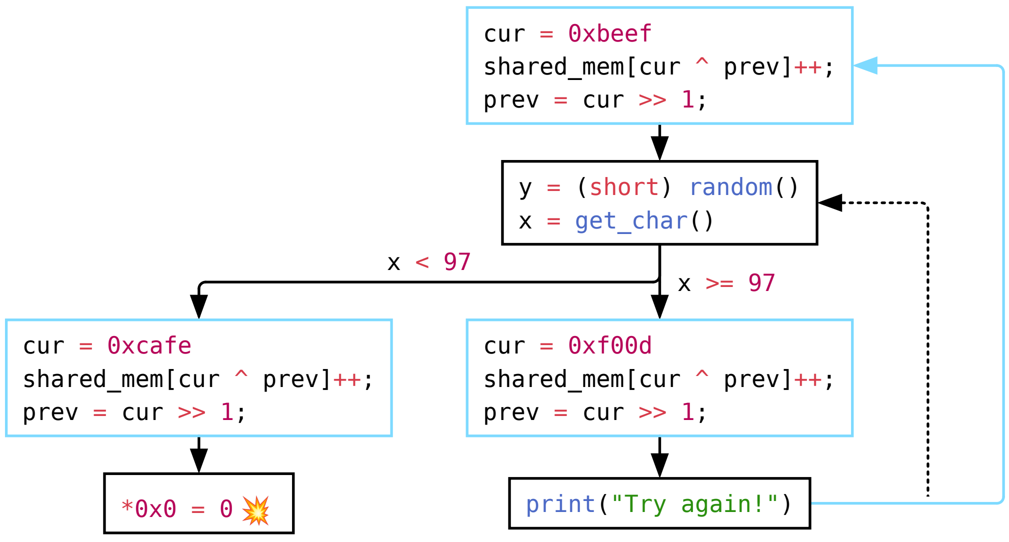
However, with such a `map[0…n]`, one must have an array in memory as long as the number of basic blocks in the program. This is not ideal, as the memory consumption can be high. To address this issue, AFL++ uses the following code, where types have been explicitly added for clarity:

```c
(unsigned short) cur_location = <COMPILE_TIME_RANDOM>;
(unsigned short*) shared_mem[cur_location ^ prev_location]++;

(unsigned short) prev_location = cur_location >> 1;
```

AFL technical details, `AFLplusplus/instrumentation/afl-compiler-rt.o.c:248-251`, `AFLplusplus/instrumentation/afl-gcc-pass.so.cc:217-334`

To get back on our running example, it would look like the following:



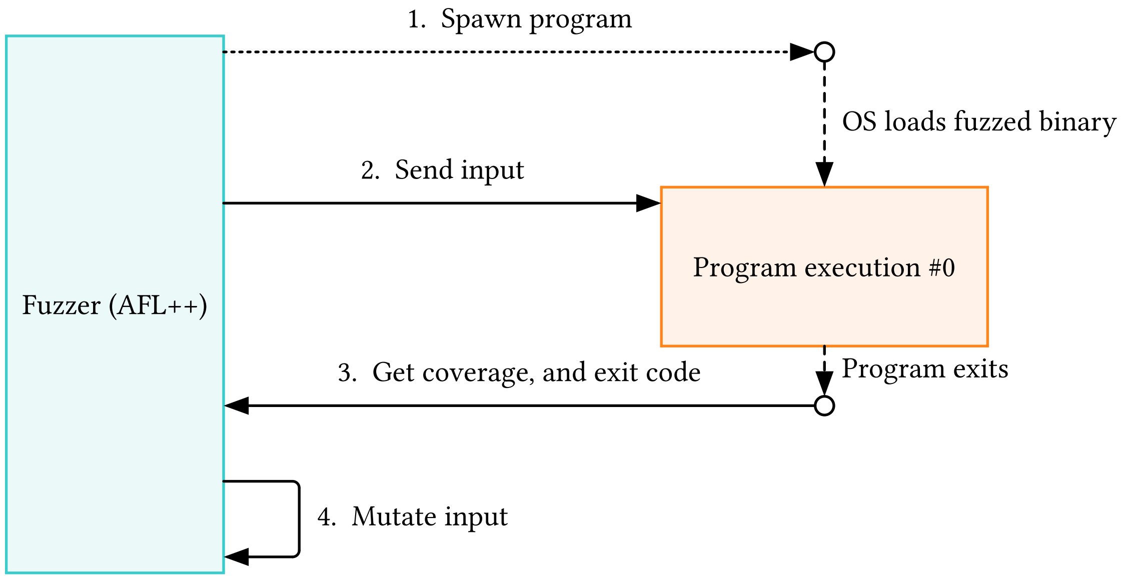
There is notable remarks to make: (1) the map is 64 kilo-bytes; (2) the constant `cur` are determined at compile-time; (3) the `prev` is a global variable, i.e. it is stateful; (4) it is lossy.

The implementation details in the [AFL technical details] highlights that it can effortlessly fit the L2 cache of the CPU, and that its stateful nature permits to distinguish between different executions of the same program, that would still invoke the same basic blocks.

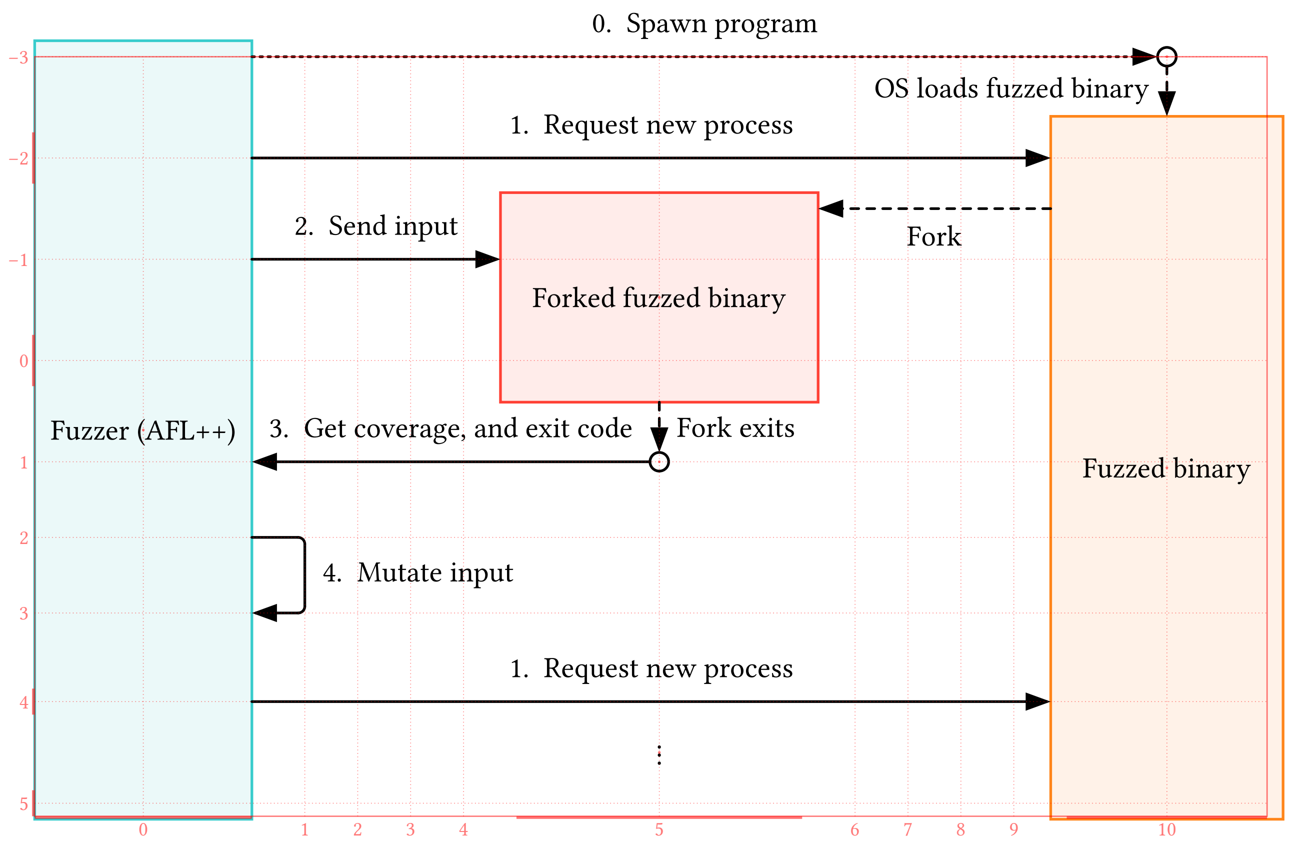
This rather simple and elegant solution lets us devise a minimal specification to provide the coverage map: indices of the basic blocks are random, but should be deterministic across executions. Essentially, the fuzzer will just notice a difference in the coverage map and mark it as interesting [citation needed]

## Performance

The general idea as we have described until now is to do the following in a loop:



However, this has its drawbacks: loading the executable is actually a procedure can take a while, and the fuzzer is not doing anything during this time. The devised solution is to use a fork server [citation needed]: instead of spawning the program, let the fuzzed binary be loaded by the operating system, and upon a request from the fuzzer, fork the process to be fuzzed. This way, instead of starting from scratch at each iteration, the forked process will be already initialized. Therefore, the actual protocol is as follows:



To implement the plugin rather effectively, the `fork` syscall should happen in the plugin, as close as possible before the execution of the fuzzed binary.

## Conclusion

We have devised rather simple requirements for the plugin: it should provide a coverage map that is deterministic across forks, and it **should** fork. Not filling the requirements will result in a fuzzer that is, by design, less effective than `qemuafl`.

## QEMU

“QEMU is a binary translator” [citation needed] is the most effective way to describe how it works. It is a piece of software that reads a binary and translates it to the host’s architecture. It can also simulate a whole system, but we will focus only on userspace emulation, `x86_64-linux-user` for the purpose of this project. However, we will define the relevant parts.

QEMU reads the binary, basic block per basic block, and translates them into an *intermediate representation* (IR) called TCG (for Tiny Code Generator). There is two parts for the translation: the frontend, that reads the binary and translates it into TCG, and the backend, that translates the TCG into the host’s architecture. The frontend is target-specific, and the backend is host-specific.

QEMU will translate each basic block (here, a node) to TCG, and then to the host’s architecture. The translation is done lazily (i.e. done when it is required to be done), and the translation is cached.

With plugins, here is a non-exhaustive list of what callbacks can be added:
- intercepting syscalls
- instrumenting instructions (either add an inline `add` instruction

or a callback to a plugin-defined function)

This section is usually 3-5 pages.

Chapter 2

# Coverage and forking design tentatives

### Coverage

The coverage map is a key component of AFL++‘s feedback loop, and can be implemented in easy

### Forking, first try

Introduce and discuss the design decisions that you made during this project. Highlight why individual decisions are important and/or necessary. Discuss how the design fits together.

This section is usually 5-10 pages.

Chapter 3

# Implementation

The implementation covers some of the implementation details of your project. This is not intended to be a low level description of every line of code that you wrote but covers the implementation aspects of the projects.

This section is usually 3-5 pages.

Chapter 4

# Evaluation

In the evaluation you convince the reader that your design works as intended. Describe the evaluation setup, the designed experiments, and how the experiments showcase the individual points you want to prove.

This section is usually 5-10 pages.

# Chapter 5

# Related Work

The related work section covers closely related work. Here you can highlight the related work, how it solved the problem, and why it solved a different problem. Do not play down the importance of related work, all of these systems have been published and evaluated! Say what is different and how you overcome some of the weaknesses of related work by discussing the trade-offs. Stay positive!

This section is usually 3-5 pages.

Chapter 6

# Conclusion

In the conclusion you repeat the main result and finalize the discussion of your project. Mention the core results and why as well as how your system advances the status quo.

## Bibliography