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, December 19, 2024 Marwan

Abstract

No idea to write here

Contents

Chapter 1

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 2

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.

To get coverage, 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

As a small running example, consider a simple program that reads a character from the standard input and either crashes or prints "Try again!" depending on the read character as depicted in Figure 1.

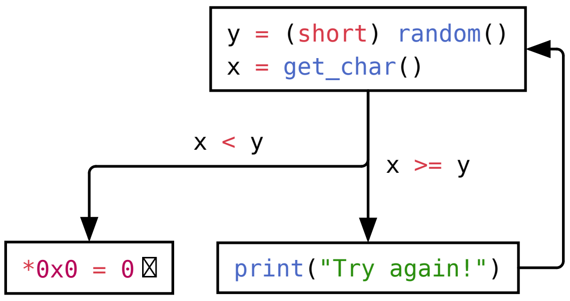


Figure 1: The Control Flow Graph of the running example: reads a character and crashes if it is less than a random number

Compile-time instrumentation will insert the following code (highlighted in teal) at the beginning of each basic block, as depicted in Figure 2.

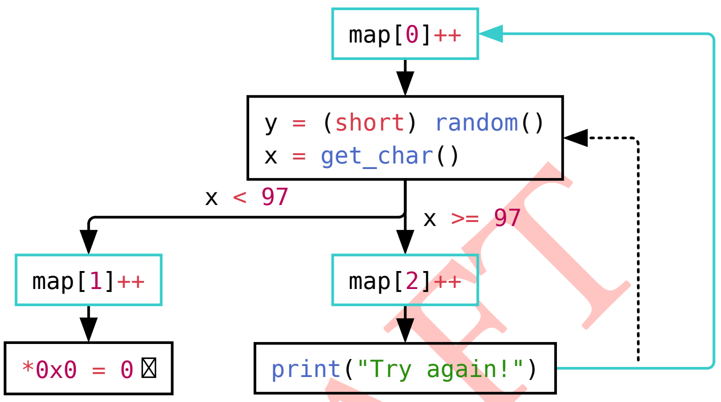


Figure 2: The Control Flow Graph of the running example with instrumentation

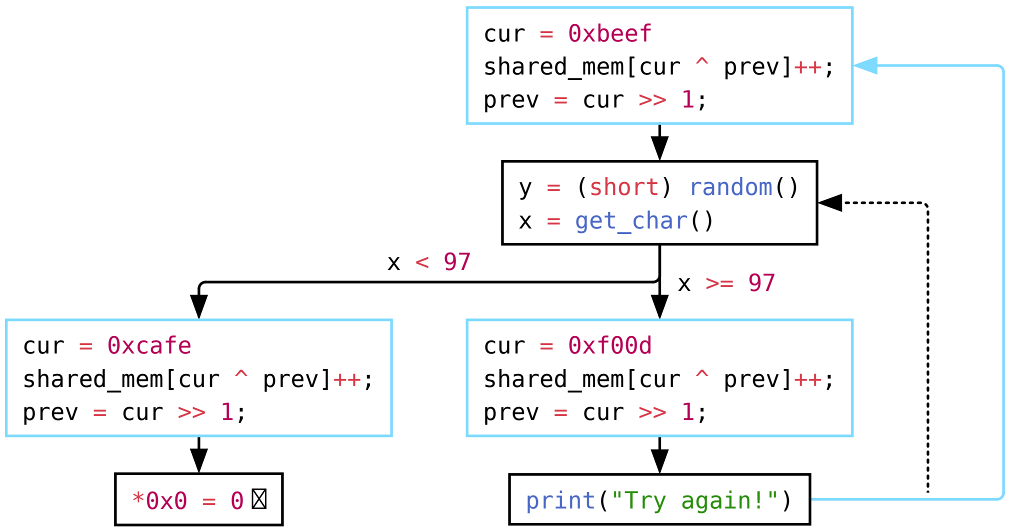
Therefore upon an execution, the fuzzer will have information on which basic block has been inserted.

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
248  (unsigned short) cur_location = <COMPILE_TIME_RANDOM>;
249  ((unsigned short*) shared_mem)[cur_location ^ prev_location]++;
250
251  (unsigned short) prev_location = cur_location >> 1;
```

AFL technical details [1], `AFLplusplus/instrumentation/afl-compiler-rt.o.c:248-251`, `AFLplusplus/instrumentation/afl-gcc-pass.so.cc:217-334`

To get back on the 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 these steps:

1.

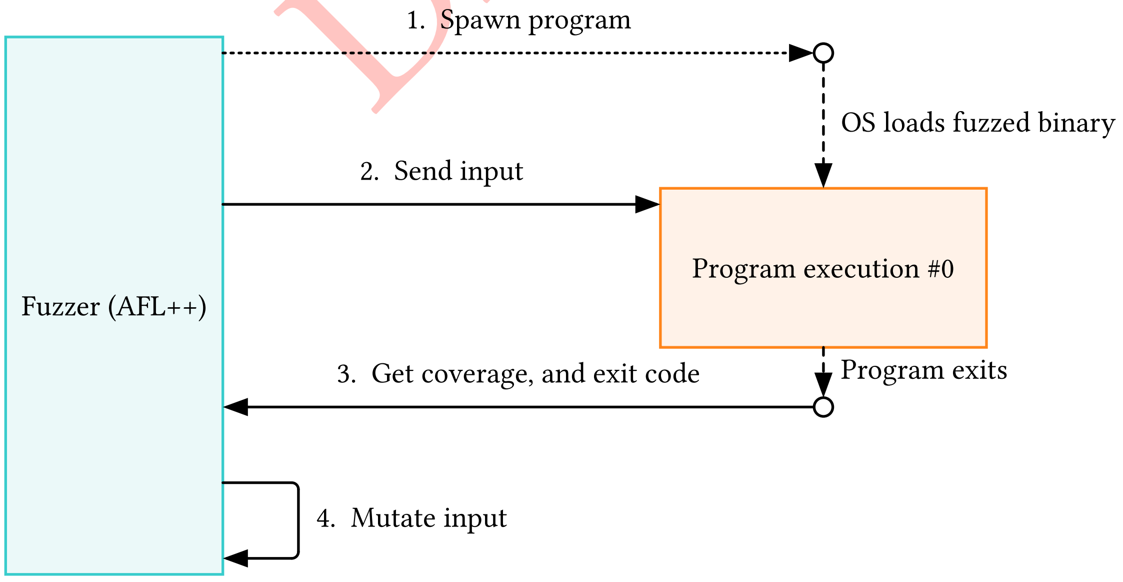


Figure 3: The fuzzing process

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 and ready for input. Therefore, the actual protocol is as follows:

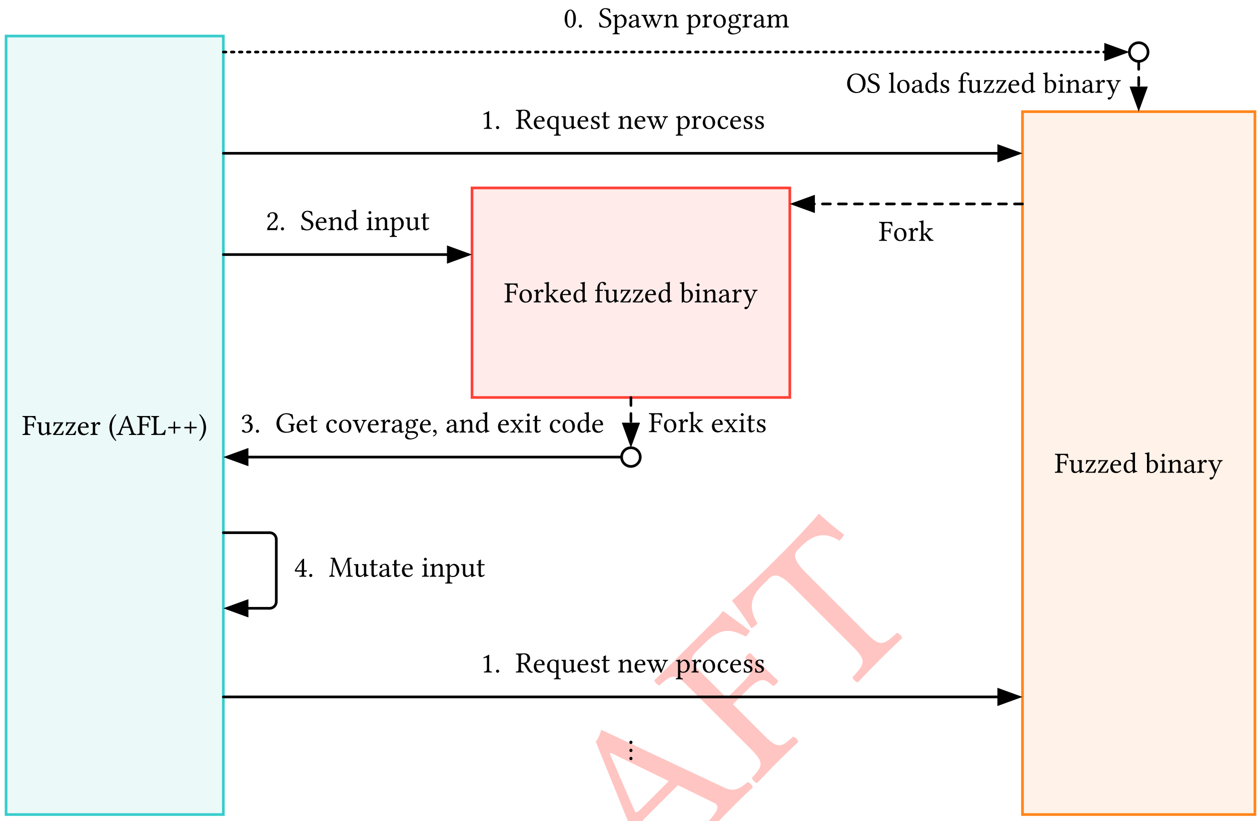


Figure 4: The AFL process with a fork server.

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`. The obvious main requirement is to **not modify a single line of QEMU's code**.

QEMU introduction

"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.

QEMU's internal can be drawn as the following: [TODO]

With plugins, here is a non-exhaustive list of what callbacks can be added:

- on syscalls
- on instructions (either add an inline `add` instruction or a callback to a plugin-defined function)
- on virtual CPU initialization
- on memory accesses

but it also have some drawbacks, notably, one can not inspect the internals of the emulator, e.g. TCG code: QEMU deserves a book on its own, so we will go through each part as needed.

This section is usually 3-5 pages.

Chapter 3

Coverage and forking design tentatives // Towards a PoC

Instrumentation

The instrumentation is a key component of any fuzzing tools, and can be implemented in an easy manner with plugins, in less than 11 lines of C code:

```c
int qemu_plugin_install(…) {
  qemu_plugin_register_vcpu_tb_trans_cb(…, on_tb_trans);
}

// On translation, this function will be executed.
void vcpu_tb_trans(…, struct qemu_plugin_tb *tb) {
  // Get the virtual address (guest) of the first instruction
  uint64_t vaddr = qemu_plugin_insn_vaddr(qemu_plugin_tb_get_insn(tb, 0));
  // Register a callback on execution for the basic block
  qemu_plugin_register_vcpu_tb_exec_cb(tb, tb_exec, QEMU_PLUGIN_CB_NO_REGS,
(void*) vaddr);
}

// On execution of a TB, this function will be executed.
void tb_exec(…, void* user_data) {
  uint64_t address = (uint64_t) user_data;
  // The translation block at `address` has been executed.
  has_been_executed(address);
}
```

This is a really strong point compared to qemuafl: the change is minimal, and does not involve messing around with the TCG translation (at least… indirectly.) To comply with callbacks, QEMU inserts a call instruction to the plugin's tb_exec callback at the beginning of each TB. Design-wise, it is simple, easily maintainable and therefore easy to change and improve!

Note as well that it relies heavily on QEMU's correctness with its internal implementation: a possible doubt that we had was that, for performance reason, QEMU can chain TBs together, and that tb_exec would be called only once for the whole chain. However, from tests that we have conducted, it seems that the unexpected behavior we have described is not happening and that the chaining of TBs is invisible to the plugin. However, note that this has been an issue for qemuafl until 2018, where qemuafl was explicitly disabling the chaining of TBs, but has now been fixed. [2]

But this is only the first part of building the plugin, the second part is to make the fork in fork server happen.

A first attempt to forking

Let play it simple in a pragmatic way of learning: what happens if we fork inside the plugin? Let's make it at initialization:

```c
int qemu_plugin_install(…) {
  fork();
}
```

We obtain the following while running the plugin:

```
qemu: qemu_mutex_unlock_impl: Operation not permitted
```

→ There is a lock taken at plugin initialization, and it's not really too much of a surprise, if one reads the documentation:

> We have to ensure we cannot deadlock, particularly under MTTCG. For this we acquire a lock when called from plugin code. We also keep the list of callbacks under RCU so that we do not have to hold the lock when calling the callbacks. This is also for performance, since some callbacks (e.g. memory access callbacks) might be called very frequently.
>
> — https://www.qemu.org/docs/master/devel/tcg-plugins.html#locking

So, it seems that we can't fork at initialization…

We considered temporarily making a fork in user-space: wait for a signal from the plugin, fork then exec the fuzzed binary. However, if an emulated program in QEMU makes the exec syscall (e.g. exec("sl")), the exec syscall is forwarded to the host's kernel. Therefore, this idea has been dropped.

What about other callbacks? In this setup, a good thought would be to think of places where no TCG is involved. One of these places where plugin callbacks can be registered where the TCG is not involved is on the initialization of a virtual CPU. We will propose a second solution later in a future, yet to be read, chapter.

So, here is where we are:

- ✓ Coverage
- ✓ Fork
- ☐ AFL++

AFL protocol breakdown

AFL and the target binary communicate through a pipe, consisting of two file descriptors, namely 198 and 199. The target binary reads from the file descriptor.

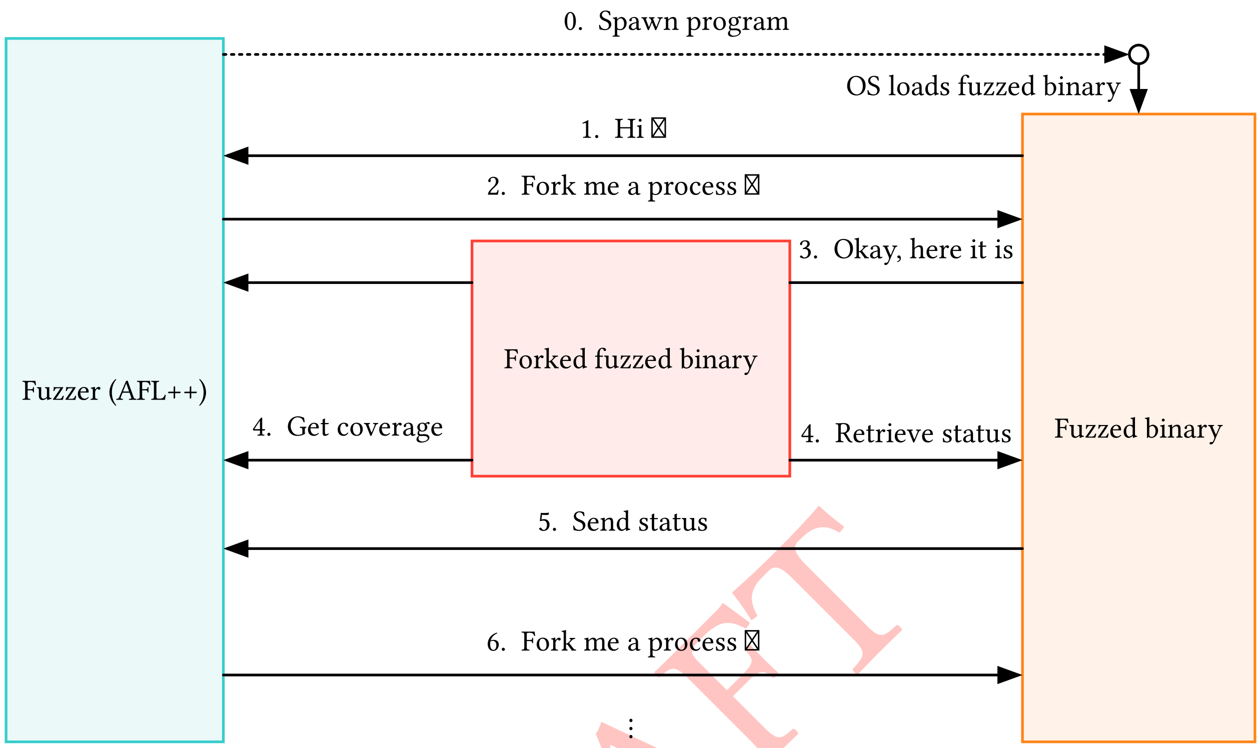


Figure 5: A visual representation between AFL++ and its fuzzed binary

For the current implementation that we have, it is sufficient to follow this protocol. Note however that the protocol has changed, while staying backward compatible and we will explore the new details when introducing the *cmplog* feature.

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 4

Implementation

Development configuration

To devise the plugin,

```c
int main(void) {
  char buf[15];
  read(stdin, 15);
  if (buf[0] == 'f' && buf[1] == 'u' && buf[2] == 'z' && buf[3] == 'z')
    *0x0 = 0;
  return 0;
}
```

The fork server

The implementation can be really straightforward, on a `vcpu_init` callback, initialize the connection with AFL++. AFL++ exposes a pipe with two file descriptors (`199` for binary⟶AFL++, `198` for AFL++⟶binary) for communication.

The 1st version of the protocol described in Figure 5 can be described programatically as:

1. A handshake: The forkserver sends a 4-bytes zero.
   - AFL++ passes via the `__AFL_SHM_ID` environment variable the shared memory ID for the coverage map. In this version, the map size is constant and is 64 kilo-bytes. [TODO] verify
2. For each fork:
   1. AFL++ sends a 4-bytes dummy value.
   2. FS sends the process id of the forked binary.
   3. FS waits for the fork to exits, and sends the status code.
   4. Repeat.

Roughly speaking, the following implements the fork server where, for the sake of simplicity, error handling and system calls to map the shared coverage map have been removed:

```c
 int is_forked = 0;

```

```
void on_vcpu_init() {
  pid_t child;
  if (is_forked) return;
  else {
    is_forked = true;
    afl_write(0, 4); // handshake
    while(true) {
      map_shared_page(getenv("__AFL_SHM_ID")) // maps the coverage map
      afl_read(&dummy, 4); // waits for signal
      if ((child = fork()) == 0) { // forks
        return;
      }
      afl_write(child, 4); // writes child PID
      waitpid(child, &status, 0); // waits for exit
      afl_write(status, 4); // writes the status
    }
  }
}
```

However, this implementation is sub-optimal: the following callback gets executed at an early stage of QEMU's initialization, namely here, which happens before reading the executable itself. There is therefore a non-negligible overhead.

We will show two tricks in the following subsection that allowed for optimizations of the fuzzing loop.

Overcoming the late fork

To overcome the overhead, after carefully analyzing the source code, QEMU uses a user-space read-copy-update mechanism for its TCG, where some part of the execution are defined as critical sections. The vCPU loop of QEMU is (1) find the next translation block to execute; (2) if it is not executed yet, translate it; (3) execute it until the end, which is a system call, or branch; (4) handle system call and interrupts. In this case, step 2 is a critical section. This routine shows that forking could be made closer to the real execution of the program by forking when the first system call happens: this is valid as most fuzzing targets would read a file descriptor before having the part.

[TODO] "show speed-up"

The following trick requires to understand how QEMU bootstraps and loads the binary.

Coverage

The QEMU API exposes the following two primitives to instrument the code, namely:
- `qemu_plugin_install`

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 5

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 6

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 7

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.

Sources

All the project is available under the following GitHub repository: https://github.com/Maeeen/qemu-semester-project

Bibliography

[1] M. Zalewski, “Technical "whitepaper" for afl-fuzz.” 2013.

[2] A. Biondo, “Improving AFL's QEMU mode performance.” 2018.