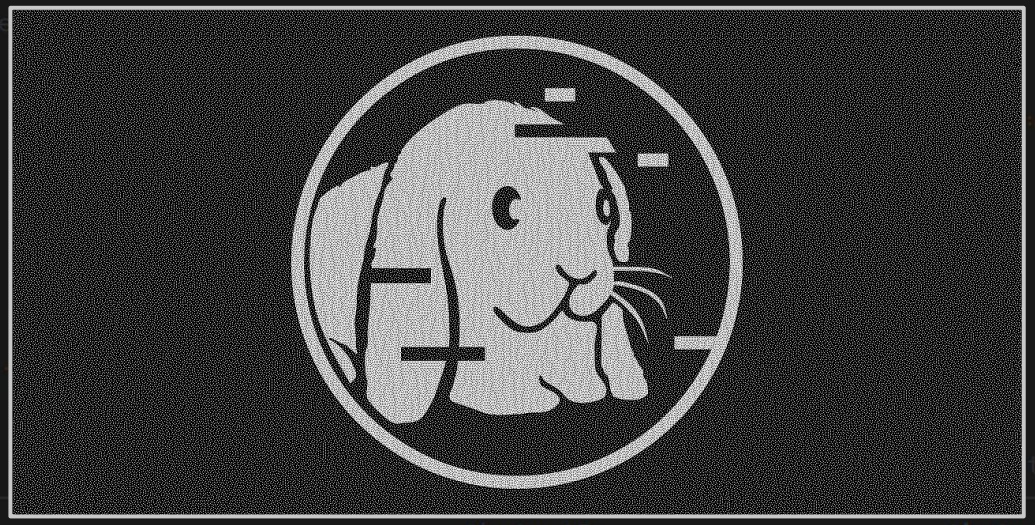
Fuzz Everything, Everywhere, All at Once

Advanced QEMU-based fuzzing

Addison Crump <research@addisoncrump.info>
Andrea Fioraldi <andreafioraldi@gmail.com>
Dominik Maier <mail@dmnk.co>
Donjia “toka” Zhang <toka@aflplus.plus>
Marc “vanHauser” Heuse <marc@srlabs.de>

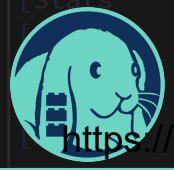
AFLplusplus Project

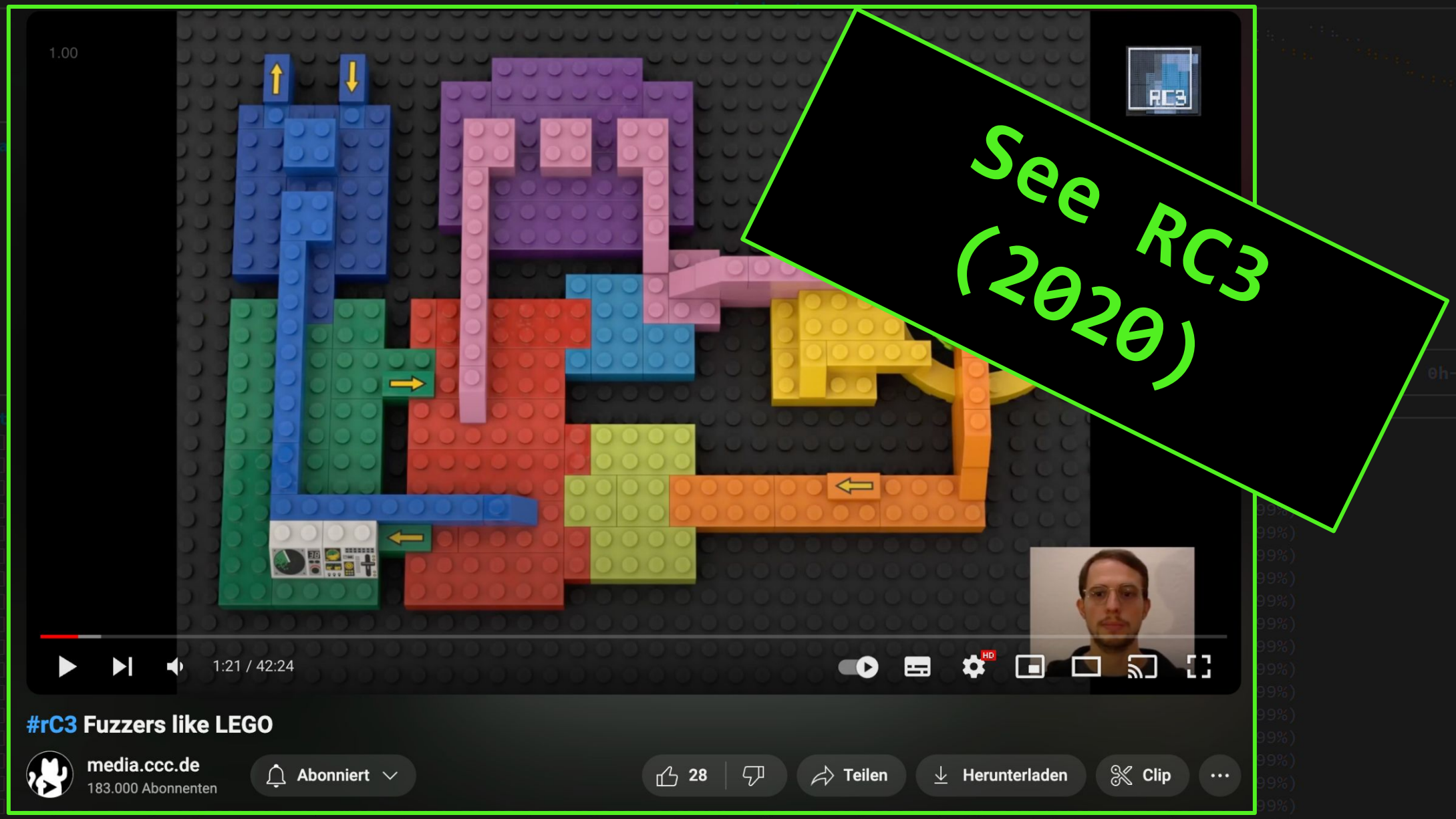
- Started with the AFL fork AFL++
- In 2019
- Added a ton of community features



LibAFL is rewritten from scratch in Rust 🦀🦀🦀

Today we will talk about Fuzzing, LibAFL, and QEMU(lation)





The AFLplusplus Project

Marc “vanHauser” Heuse

Andrea Fioraldi

Dominik Maier

Donjia “toka” Zhang

Addison Crump

Shmarya Rubenstein

Heiko "hexcoder-" Eissfeldt

voluntary contributors,
Full-time working/
researching at:

Security Research Labs

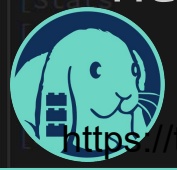
EURECOM
Sophia Antipolis

CISPA

Google

and more

and a large community!



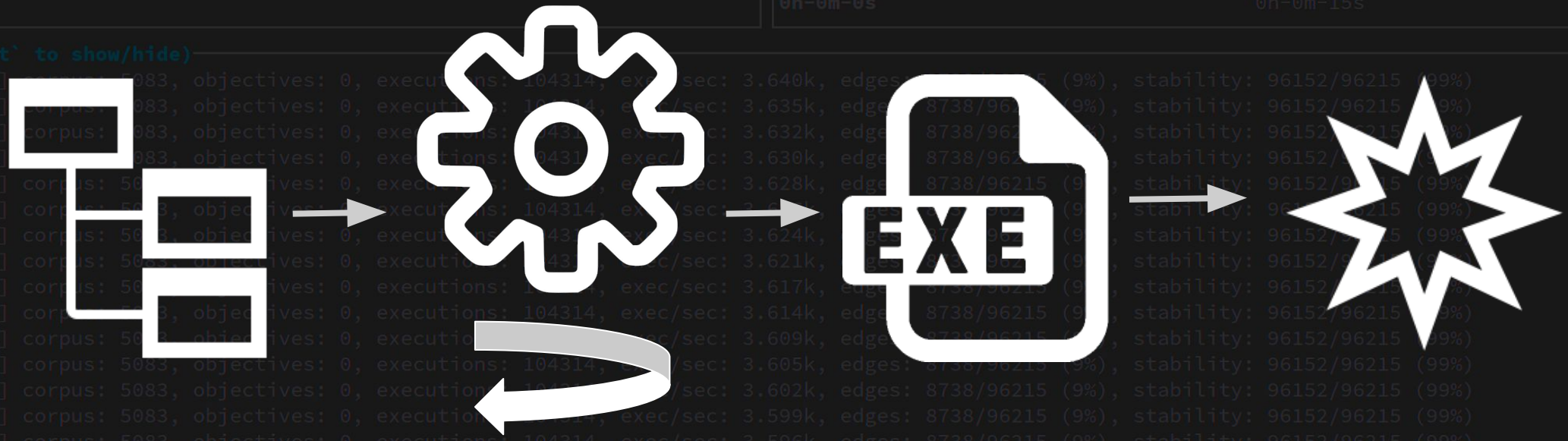
In This Talk

- Quick Fundamentals of
  - Fuzzing
  - QEMU
  - Binary Instrumentation
- Snapshot-Fuzzing an Android Library on a 80 core server
- Adding sanitizers for Injections to binaries at runtime

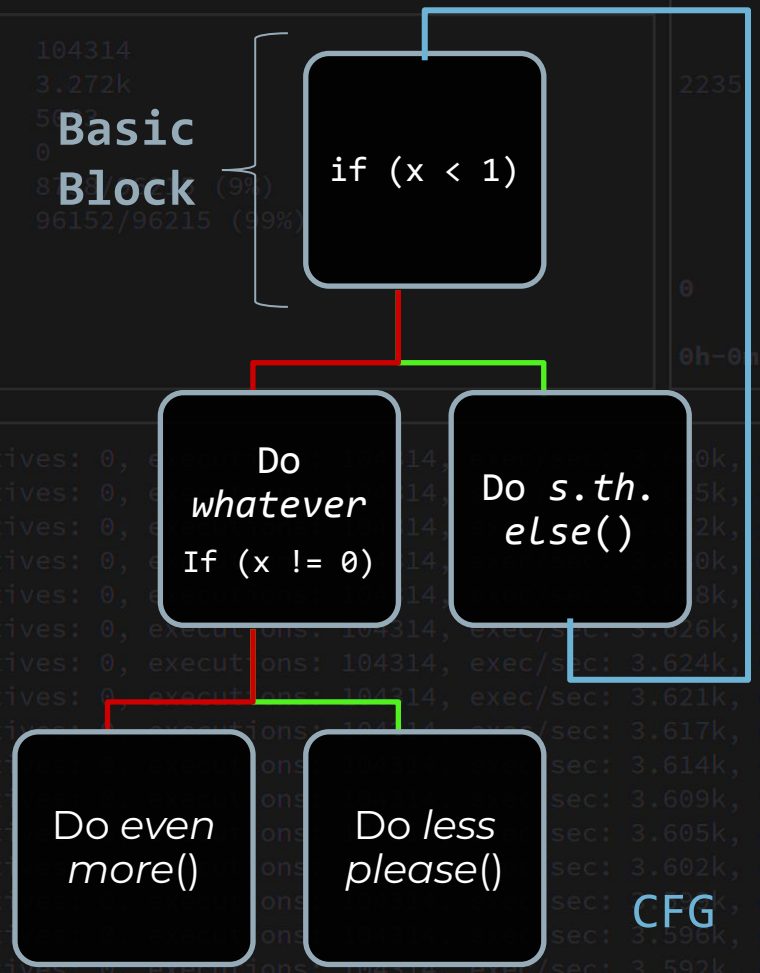
Fuzz Everything, Everywhere, All at Once

Fuzzing in a Nutshell

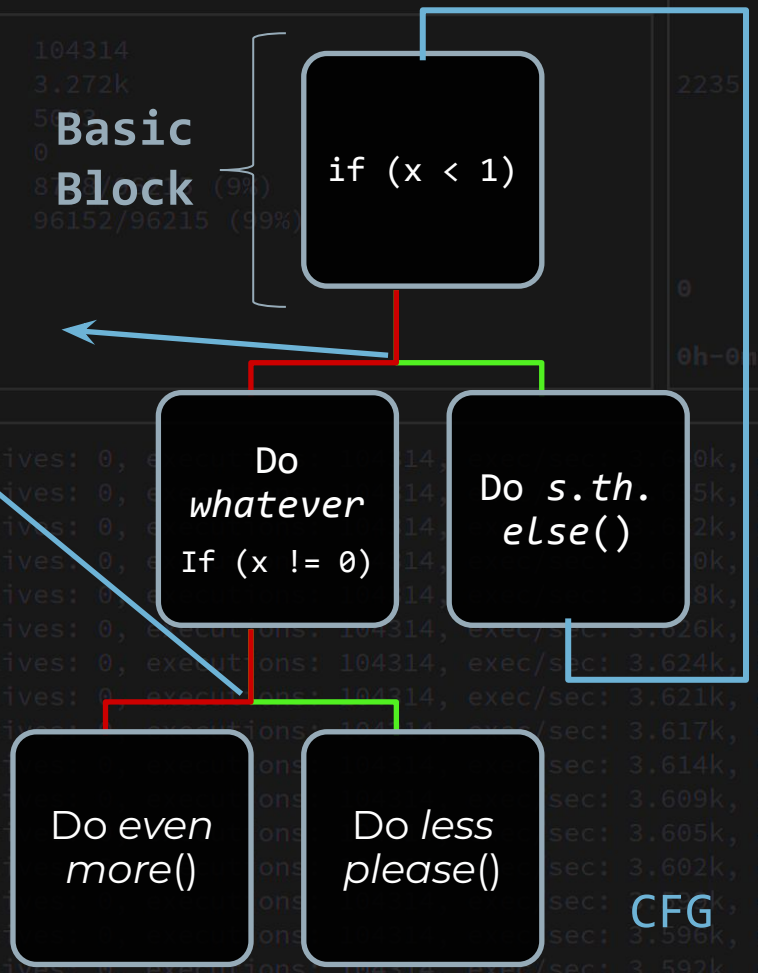
Fuzzing delivers a large amount of machine-generated inputs as quickly as possible to the target in order to find some objectives.



Coverage-Guided Fuzzing



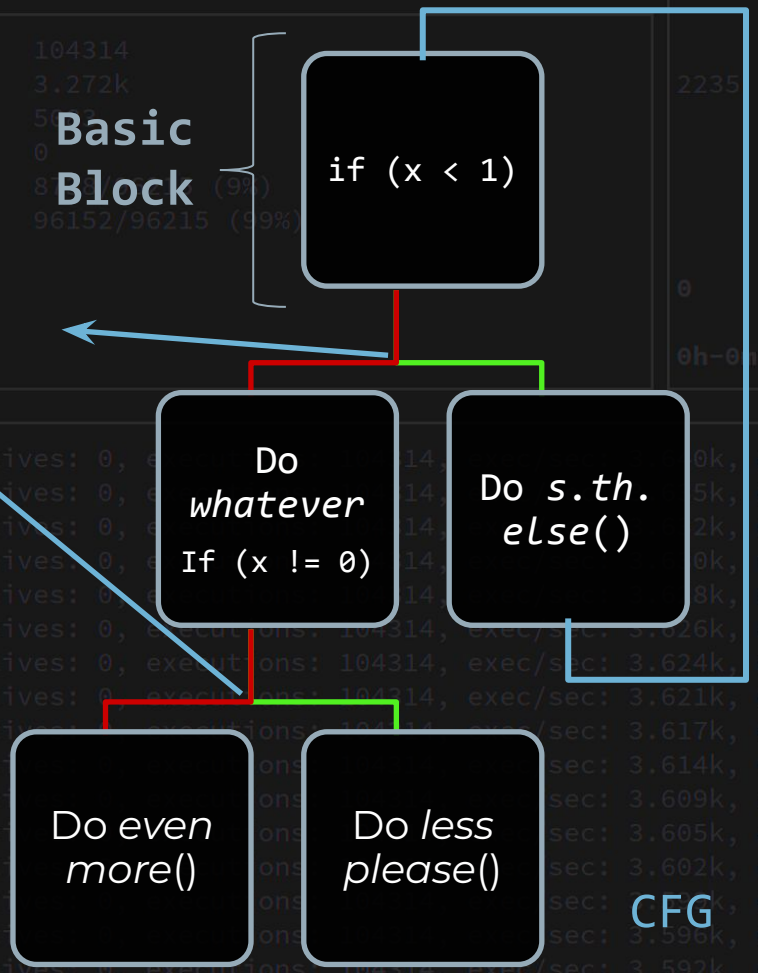
Coverage-Guided Fuzzing



- Fuzzer takes reached coverage as feedback

Count how often these happen.

Coverage-Guided Fuzzing



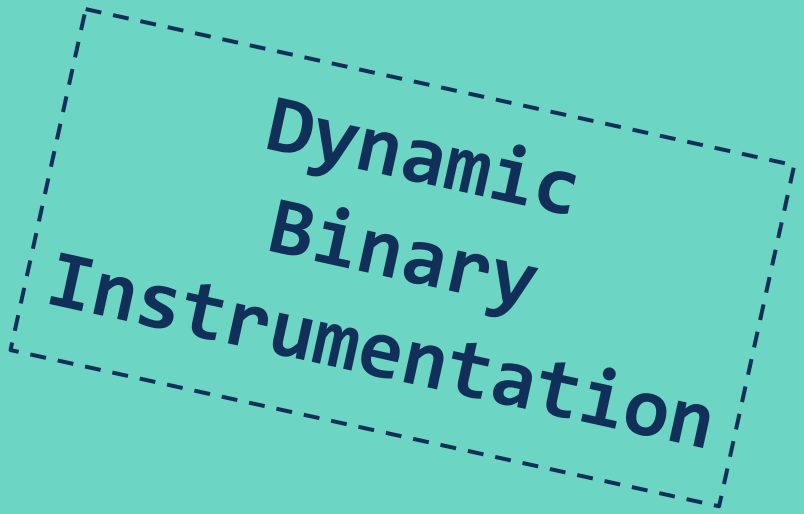
Count how often these happen.

↳ **Feedback**

↳ Favor Inputs leading to new edges

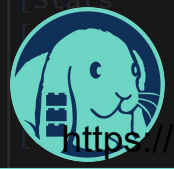
- Fuzzer takes reached coverage as feedback
- Keeps the track of edges and favors new coverage
- Orders of magnitude faster!

Fuzz
Everything,
Everywhere,
All at Once

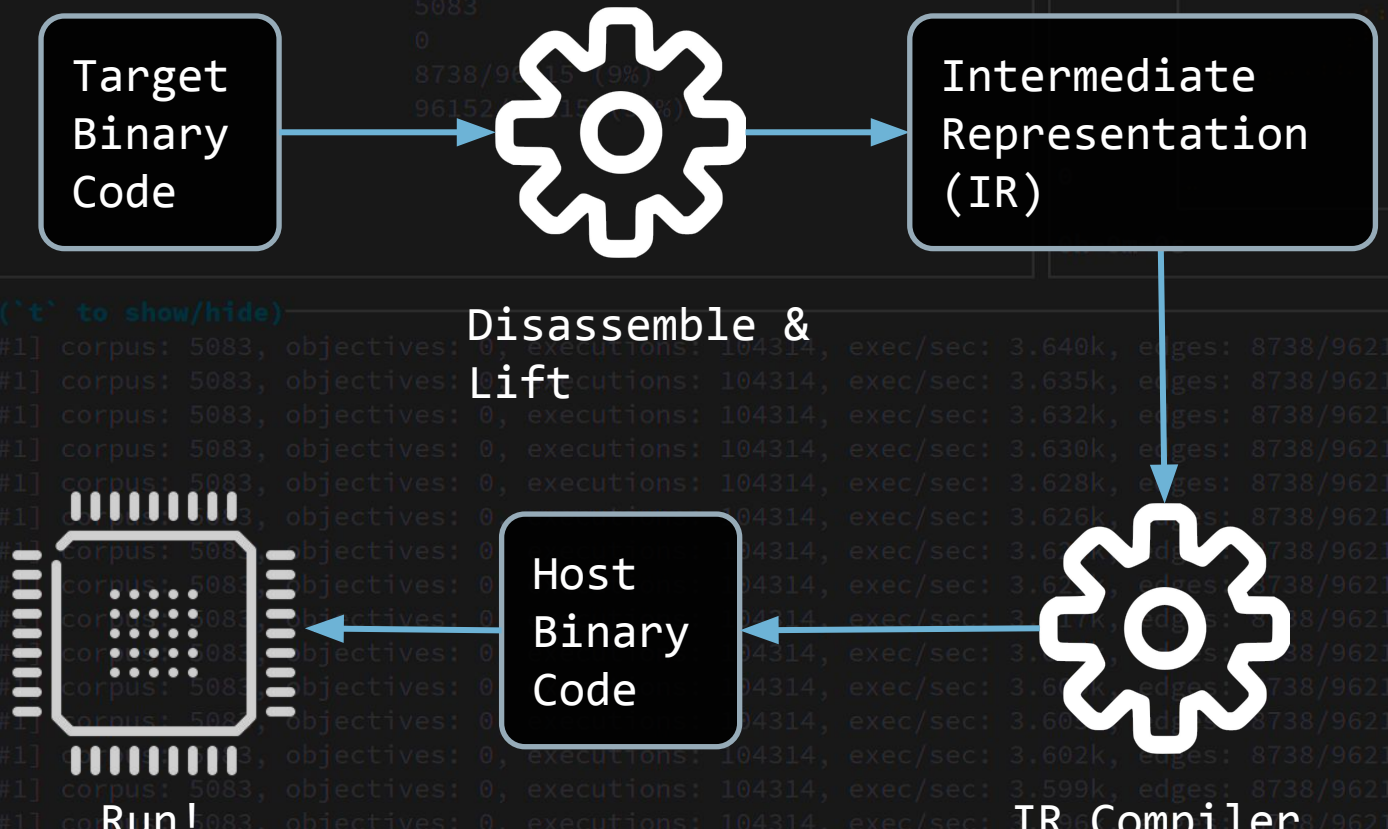


Meet QEMU, the Quick Emulator

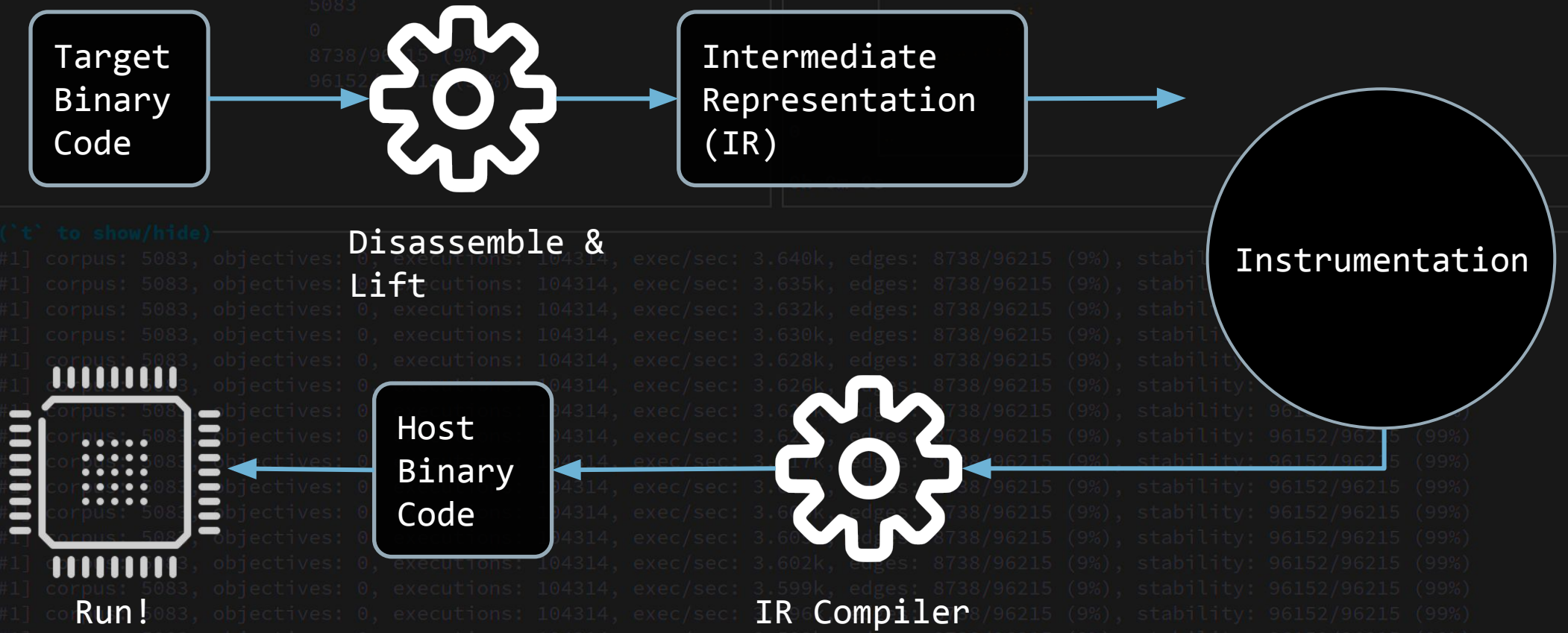
- Very popular full-system emulator
  - CPU
  - Memory
  - Peripherals
- Support for a variety of ISAs (x86, aarch64, ..., even hexagon)
- user-mode emulation support:
  - Emulation of userspace software
  - System call translation layer
  - Can be (ab-)used to change syscall behavior :)



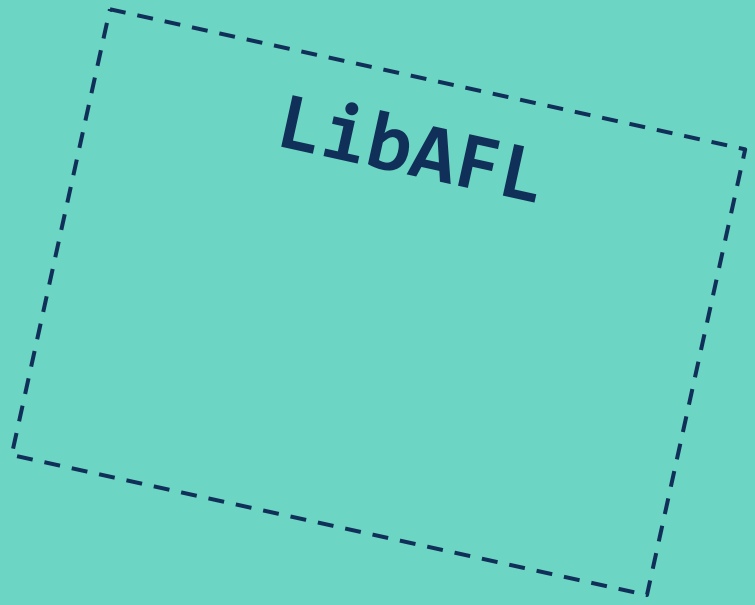
JIT Code Rewriting



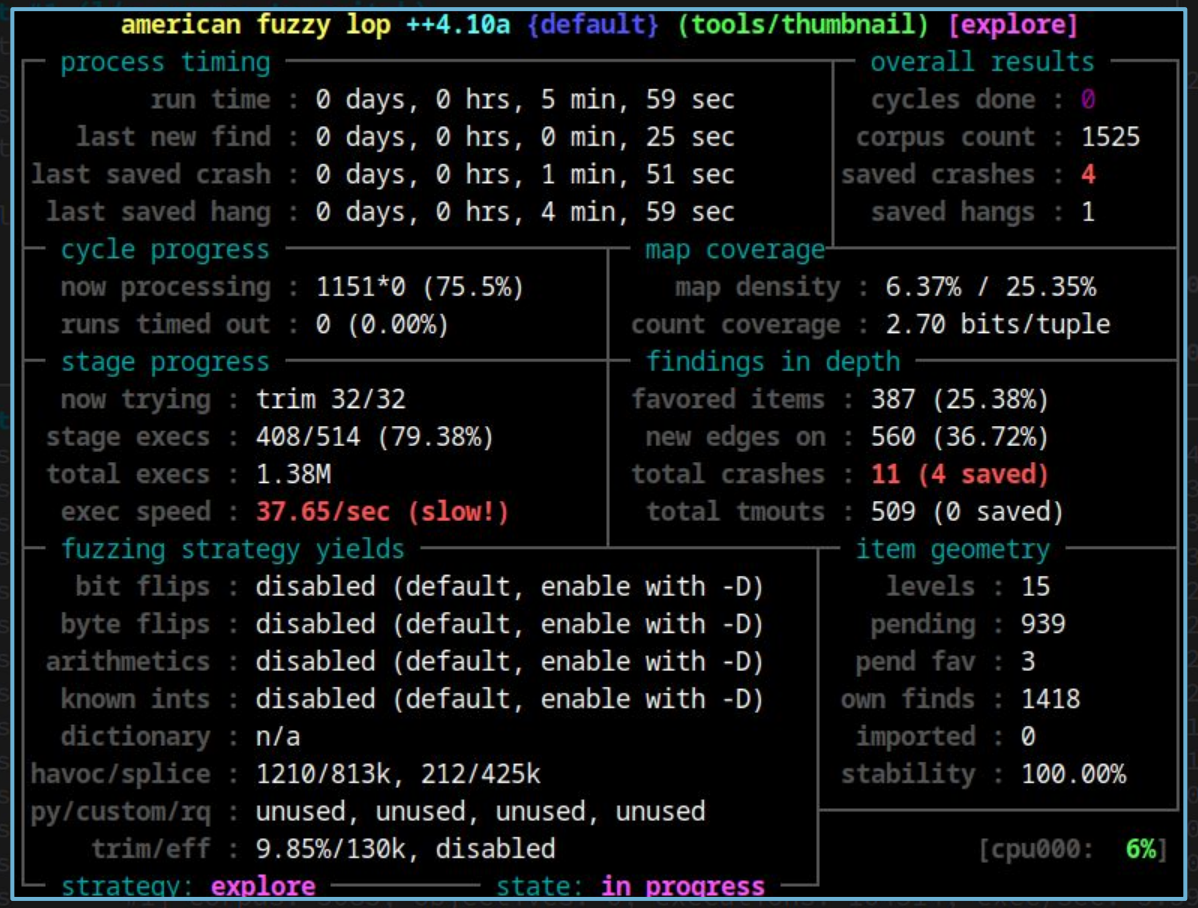
JIT Code Instrumentation

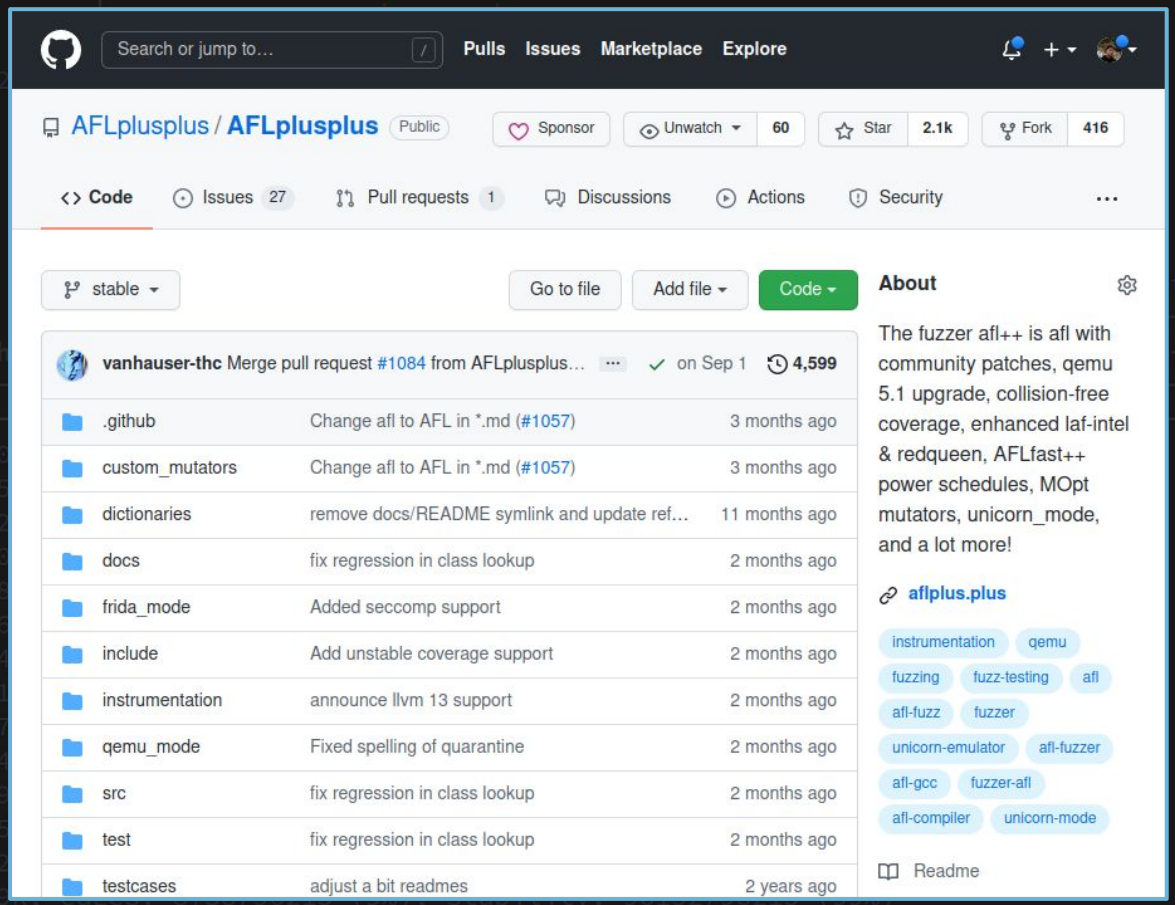


Fuzz Everything, Everywhere, All at Once



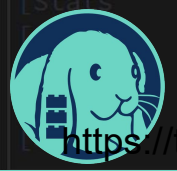
AFL++ pew pew! bugs!





Custom Fuzzers Issues

- Reinventing the wheel: code the same code to do that same thing, again and again
- Naive design: typically just a mutator
- Scaling: scaling to multi-core or -machine is hard



LibAFL

- State-of-the-Art
- Portable
- Extensible
- Scalable
- Performant, thanks to compile-time abstractions in Rust

High level design

- LibAFL Core, the main library
- LibAFL Targets, the runtime code that lives in the target
- LibAFL CC, the library to write compiler wrappers

+

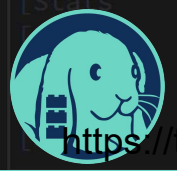
Many instrumentation options

LibAFL QEMU

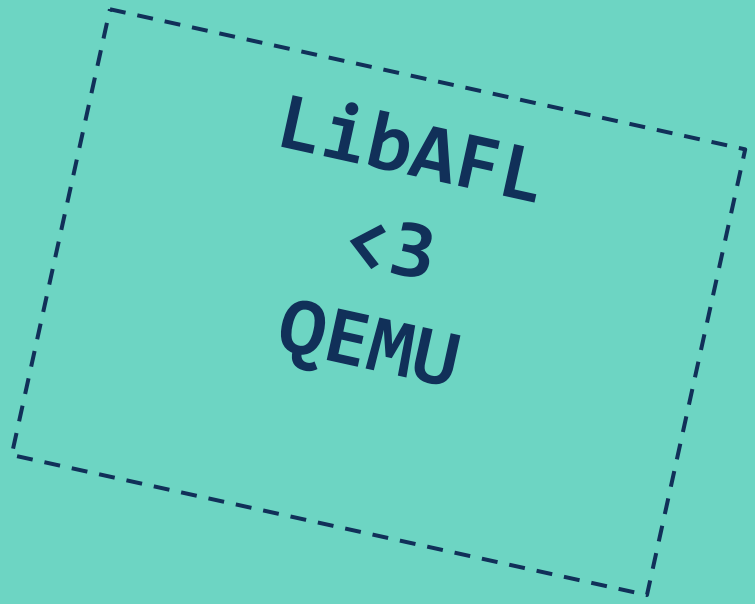
Support for user- *and* system-mode (WIP), based on QEMU 8

Why Emulate?

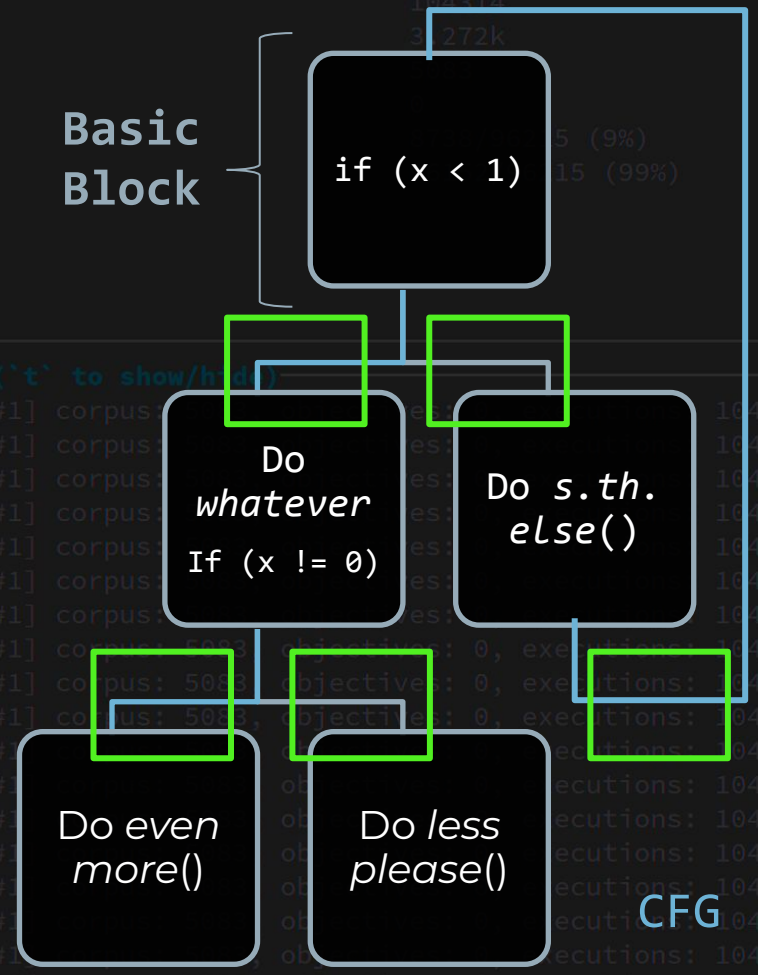
- Why not Compile-Time Instrumentation?
  - Compiling is hard
  - Toolchains are hard
  - Source not always available
  - Change instrumentation at runtime
- Advantages over other dynamic binary instrumentation:
  - Cross architecture
  - Reasonably fast while being stable



Fuzz Everything, Everywhere, All at Once



LibAFL QEMU Hooks



Example: Edge Hooks

Instrumentation (in JIT) that is running callback functions

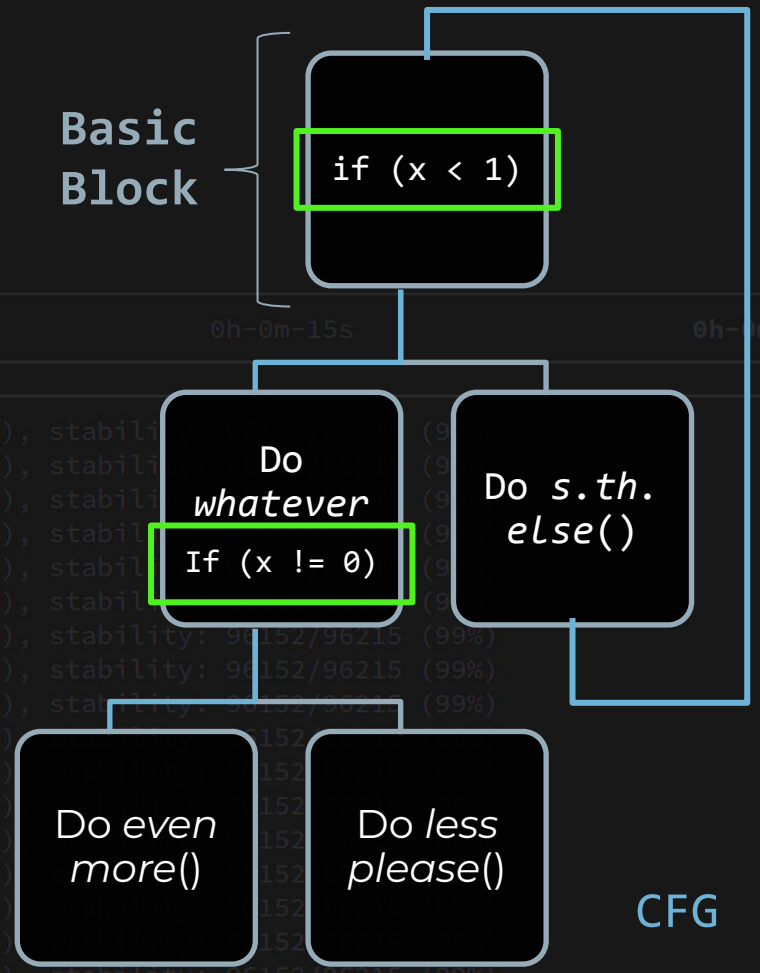
Before any jump, reports a unique id for the taken edge to a hook

```
Generation Hook:
fn(&mut Self, Option<&mut S>, src: GuestAddr, dest: GuestAddr) ->
Option<u64> { … }

Execution Hook:
FnMut(&'a mut Self, Option<&'a mut S>, u64)
```

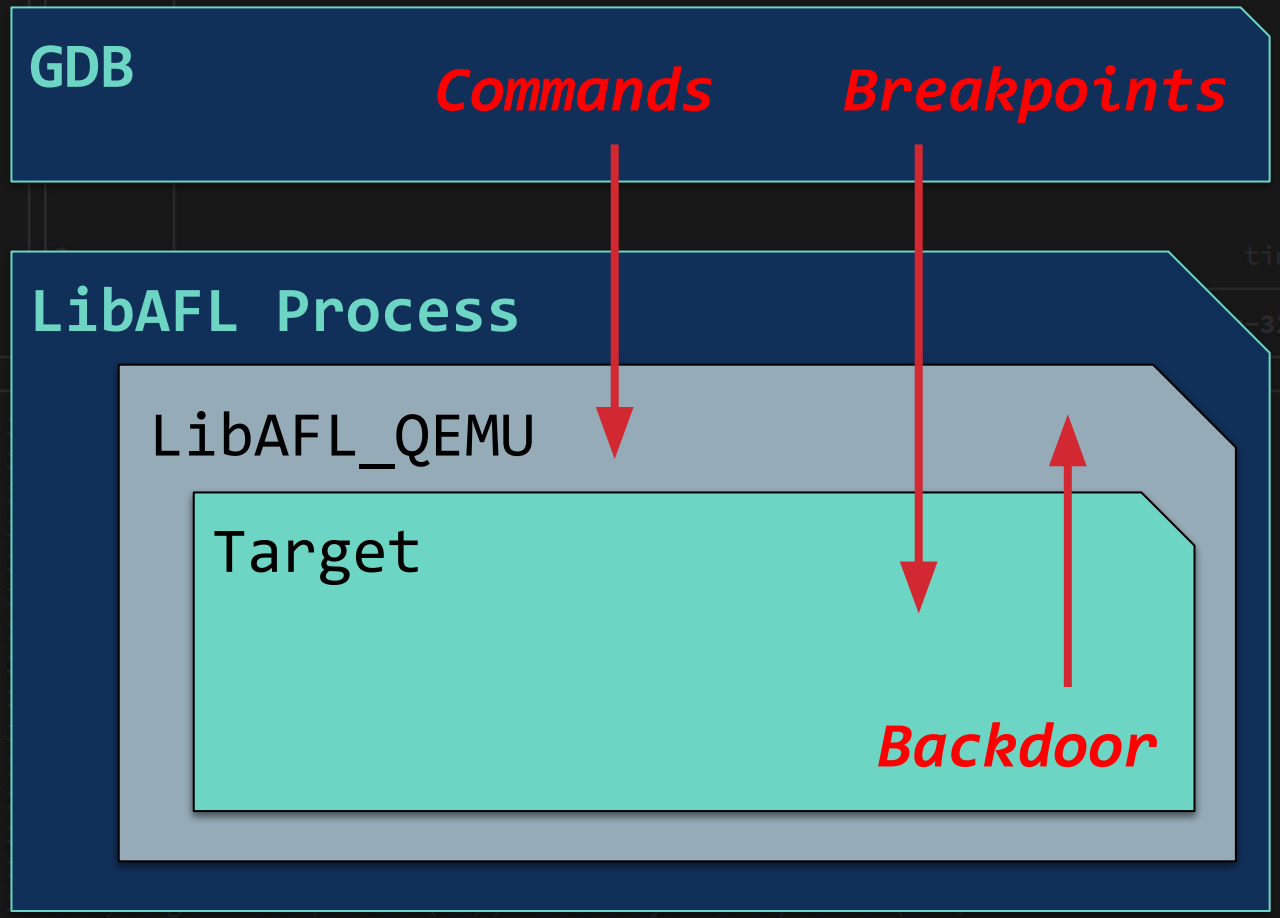
LibAFL QEMU Hooks

- Instructions
- Blocks
- Edges
- Read and write
- Comparisons
- Threads
- Syscalls
- Crashes



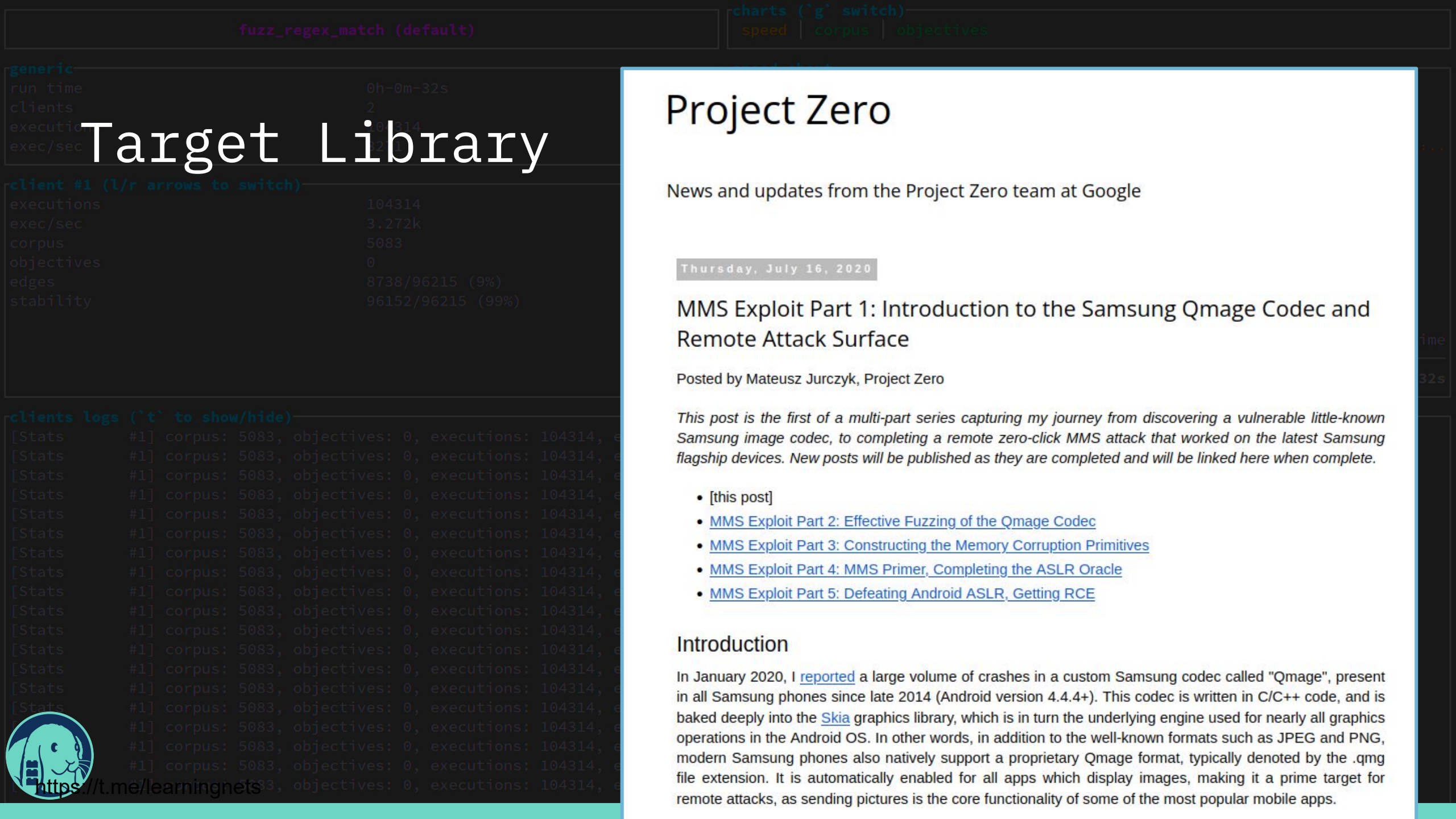
Fuzzing With QEMU: Execution Control

- Backdoor: *target-defined* point at which execution halts
- Breakpoint: *fuzzer-defined* point at which execution halts
- Commands: *custom GDB commands* to interact with the target state



Fuzz Everything, Everywhere, All at Once

Target Library



Project Zero

News and updates from the Project Zero team at Google

Thursday, July 16, 2020

MMS Exploit Part 1: Introduction to the Samsung Qmage Codec and Remote Attack Surface

Posted by Mateusz Jurczyk, Project Zero

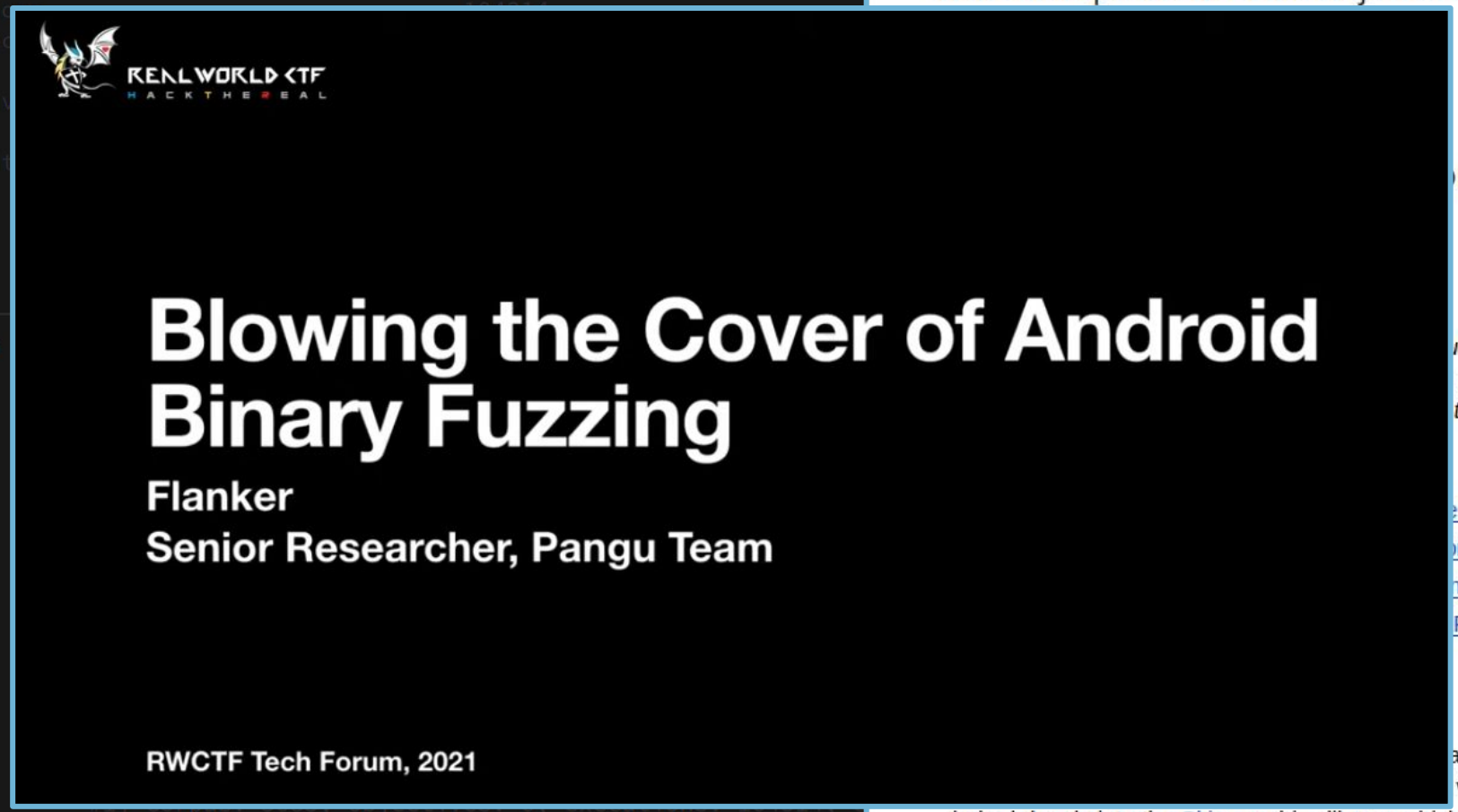
This post is the first of a multi-part series capturing my journey from discovering a vulnerable little-known Samsung image codec, to completing a remote zero-click MMS attack that worked on the latest Samsung flagship devices. New posts will be published as they are completed and will be linked here when complete.

- [this post]
- MMS Exploit Part 2: Effective Fuzzing of the Qmage Codec
- MMS Exploit Part 3: Constructing the Memory Corruption Primitives
- MMS Exploit Part 4: MMS Primer, Completing the ASLR Oracle
- MMS Exploit Part 5: Defeating Android ASLR, Getting RCE

Introduction

In January 2020, I reported a large volume of crashes in a custom Samsung codec called "Qmage", present in all Samsung phones since late 2014 (Android version 4.4.4+). This codec is written in C/C++ code, and is baked deeply into the Skia graphics library, which is in turn the underlying engine used for nearly all graphics operations in the Android OS. In other words, in addition to the well-known formats such as JPEG and PNG, modern Samsung phones also natively support a proprietary Qmage format, typically denoted by the .qmg file extension. It is automatically enabled for all apps which display images, making it a prime target for remote attacks, as sending pictures is the core functionality of some of the most popular mobile apps.

Target Library



Project Zero

News and updates from the Project Zero team at Google

…n to the Samsung Qmage Codec and

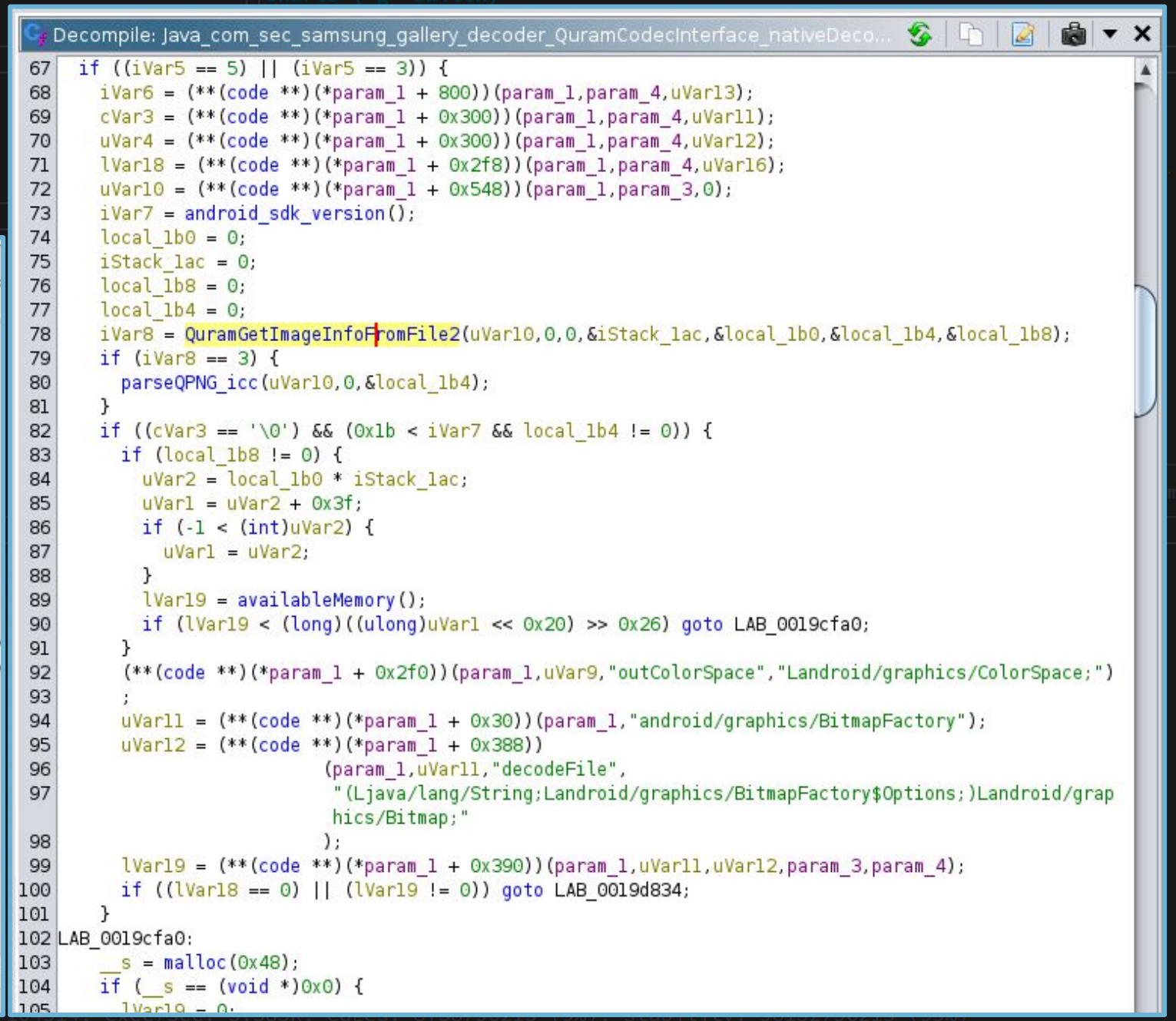
…uring my journey from discovering a vulnerable little-known zero-click MMS attack that worked on the latest Samsung they are completed and will be linked here when complete.

…e Qmage Codec
…ory Corruption Primitives
…ng the ASLR Oracle
…R, Getting RCE

…ashes in a custom Samsung codec called "Qmage", present version 4.4.4+). This codec is written in C/C++ code, and is baked deeply into the Skia graphics library, which is in turn the underlying engine used for nearly all graphics operations in the Android OS. In other words, in addition to the well-known formats such as JPEG and PNG, modern Samsung phones also natively support a proprietary Qmage format, typically denoted by the .qmg file extension. It is automatically enabled for all apps which display images, making it a prime target for remote attacks, as sending pictures is the core functionality of some of the most popular mobile apps.

Reversing

Reversing

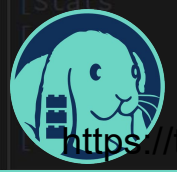


Harness

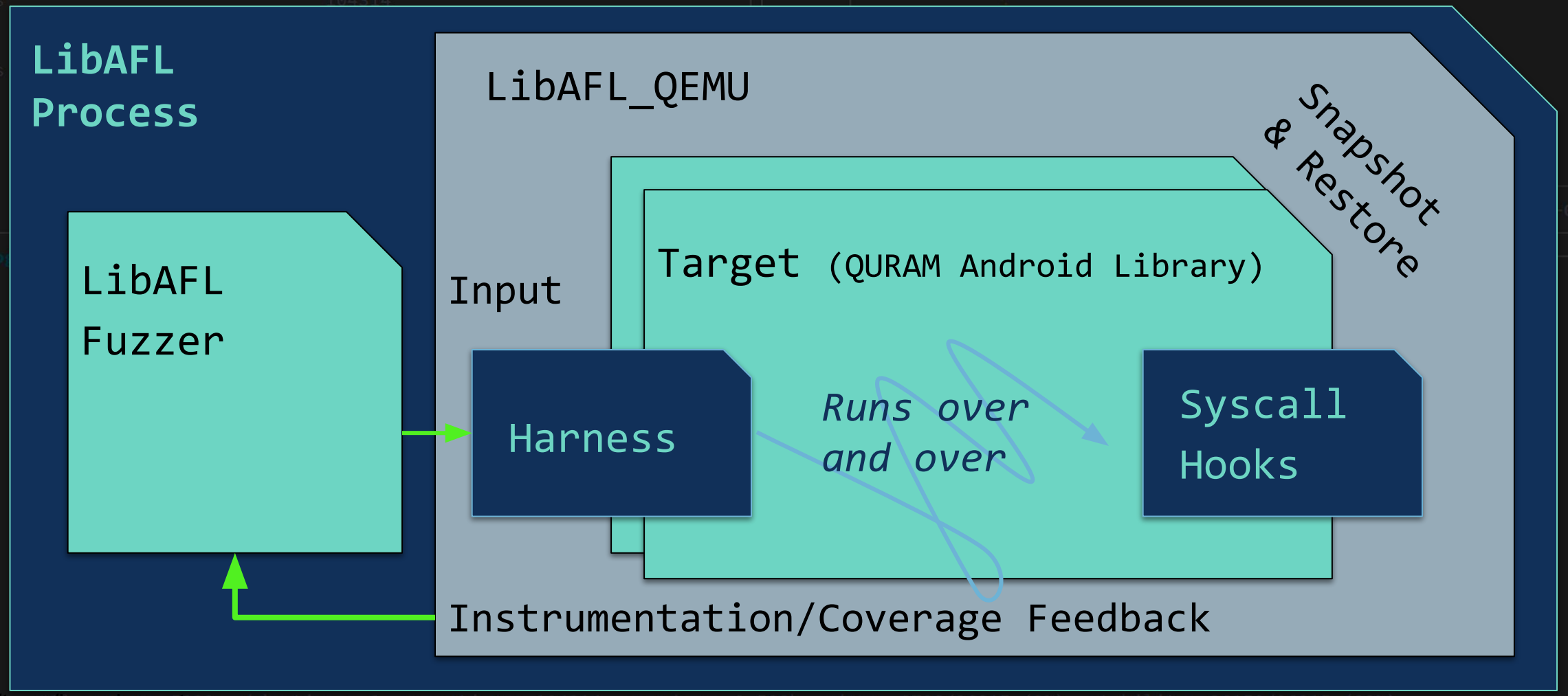
```
int QuramGetImageInfoFromFile2(char *filename, int zero1, int zero2,
                               int *w, int *h, int* getImageOut1, int* getImageOut2);
int QrParseMetadata(char *filename, unsigned int* metadata);


void harnessSimple(char* filename) {
  int w, h, a, b;
  unsigned int metadata[71] = {0};

  if (QuramGetImageInfoFromFile2(filename, 0, 0, &w, &h, &a, &b) == 0) {
    QrParseMetadata(filename, metadata);
  }
}
```



Fuzzing Android Libs on a Host



A Simple Fuzzer

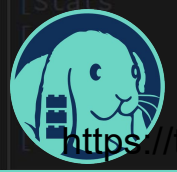
```rust
let mut args = vec!["qemu".into(), "./harness".into(), MAGIC_FILENAME.into()];
let mut env: Vec<(String, String)> = env::vars().collect();

let emu = Emulator::new(&mut args, &mut env);

let mut elf_buffer = Vec::new();
let elf = EasyElf::from_file(emu.binary_path(), &mut elf_buffer).unwrap();

let harness_ptr = elf
    .resolve_symbol(HARNESS_NAME, emu.load_addr())
    .expect(&format!("Symbol {} not found", HARNESS_NAME));
println!("{} @ {:#x}", HARNESS_NAME, harness_ptr);

emu.set_breakpoint(harness_ptr);
unsafe { emu.run() };
```



A Simple Fuzzer

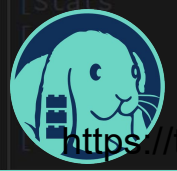
```
let mut args = vec!["qemu".into(), "./harness".into(), MAGIC_FILENAME.into()];
let mut env: Vec<(String, String)> = env::vars().collect();

let emu = Emulator::new(&mut args, &mut env);

let mut elf_buffer = Vec::new();
let elf = EasyElf::from_file(emu.binary_path(), &mut elf_buffer).unwrap();

let harness_ptr = elf
    .resolve_symbol(HARNESS_NAME, emu.load_addr())
    .expect(&format!("Symbol {} not found", HARNESS_NAME));
println!("{} @ {:#x}", HARNESS_NAME, harness_ptr);

emu.set_breakpoint(harness_ptr);
unsafe { emu.run() };
```



A Simple Fuzzer

```
let ret_addr: u64 = emu.read_reg(Regs::Lr).unwrap();
println!("Return address = {:#x}", ret_addr);

emu.remove_breakpoint(harness_ptr);
emu.set_breakpoint(ret_addr);

let saved_cpu_states: Vec<_> = (0..emu.num_cpus())
    .map(|i| emu.cpu_from_index(i).save_state())
    .collect();
```

A Simple Fuzzer

```
let ret_addr: u64 = emu.read_reg(Regs::Lr).unwrap();
println!("Return address = {:#x}", ret_addr);
```

```
emu.remove_breakpoint(harness_ptr);
emu.set_breakpoint(ret_addr);
```

```
let saved_cpu_states: Vec<_> = (0..emu.num_cpus())
    .map(|i| emu.cpu_from_index(i).save_state())
    .collect();
```

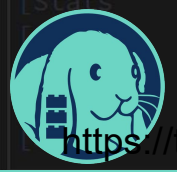
A Simple Fuzzer

```
let mut harness = |input: &BytesInput| {
    input.to_file(MAGIC_FILENAME).unwrap();

    unsafe { let _ = emu.run() };

    for (i, s) in saved_cpu_states.iter().enumerate() {
        emu.cpu_from_index(i).restore_state(s);
    }

    ExitKind::Ok
};
```



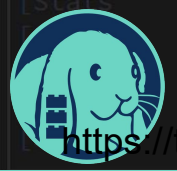
A Simple Fuzzer

```
let mut harness = |input: &BytesInput| {
    input.to_file(MAGIC_FILENAME).unwrap();

    unsafe { let _ = emu.run() };

    for (i, s) in saved_cpu_states.iter().enumerate() {
        emu.cpu_from_index(i).restore_state(s);
    }

    ExitKind::Ok
};
```



A Simple Fuzzer

```
let mut hooks = QemuHooks::new(
    emu.clone(),
    tuple_list!(
        QemuEdgeCoverageHelper::default(),
        QemuCmpLogHelper::default(),
    ),
);

let executor = QemuExecutor::new(
    &mut hooks,
    &mut harness,
    tuple_list!(edges_observer, time_observer),
    &mut fuzzer,
    &mut state,
    &mut mgr,
)
.expect("Failed to create QemuExecutor");
```

A Simple Fuzzer

```
let mut hooks = QemuHooks::new(
    emu.clone(),
    tuple_list!(
        QemuEdgeCoverageHelper::default(),
        QemuCmpLogHelper::default(),
    ),
);

let executor = QemuExecutor::new(
    &mut hooks,
    &mut harness,
    tuple_list!(edges_observer, time_observer),
    &mut fuzzer,
    &mut state,
    &mut mgr,
)
.expect("Failed to create QemuExecutor");
```

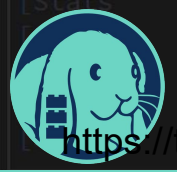
A Simple Fuzzer

```
[Stats      #1]  (GLOBAL) run time: 47h-59m-32s, clients: 2, corpus: 56011, objectives: 0, executions: 881082769, exec/sec: 5.100k
                 (CLIENT) corpus: 56011, objectives: 0, executions: 881082769, exec/sec: 5.100k, edges: 44549/44706 (99%), stability: 43610/44600 (97%)
[Testcase   #1]  (GLOBAL) run time: 47h-59m-33s, clients: 2, corpus: 56012, objectives: 0, executions: 881092795, exec/sec: 5.100k
                 (CLIENT) corpus: 56012, objectives: 0, executions: 881092795, exec/sec: 5.100k, edges: 44549/44706 (99%), stability: 43610/44600 (97%)
[Stats      #1]  (GLOBAL) run time: 47h-59m-33s, clients: 2, corpus: 56012, objectives: 0, executions: 881094649, exec/sec: 5.100k
                 (CLIENT) corpus: 56012, objectives: 0, executions: 881094649, exec/sec: 5.100k, edges: 44549/44706 (99%), stability: 43610/44600 (97%)
[Stats      #1]  (GLOBAL) run time: 47h-59m-35s, clients: 2, corpus: 56012, objectives: 0, executions: 881094649, exec/sec: 5.100k
                 (CLIENT) corpus: 56012, objectives: 0, executions: 881094649, exec/sec: 5.100k, edges: 44550/44707 (99%), stability: 43610/44600 (97%)
[Testcase   #1]  (GLOBAL) run time: 47h-59m-35s, clients: 2, corpus: 56013, objectives: 0, executions: 881103227, exec/sec: 5.100k
                 (CLIENT) corpus: 56013, objectives: 0, executions: 881103227, exec/sec: 5.100k, edges: 44550/44707 (99%), stability: 43610/44600 (97%)
[Stats      #1]  (GLOBAL) run time: 47h-59m-40s, clients: 2, corpus: 56013, objectives: 0, executions: 881103227, exec/sec: 5.100k
                 (CLIENT) corpus: 56013, objectives: 0, executions: 881103227, exec/sec: 5.100k, edges: 44551/44708 (99%), stability: 43610/44600 (97%)
[Testcase   #1]  (GLOBAL) run time: 47h-59m-40s, clients: 2, corpus: 56014, objectives: 0, executions: 881127950, exec/sec: 5.100k
                 (CLIENT) corpus: 56014, objectives: 0, executions: 881127950, exec/sec: 5.100k, edges: 44551/44708 (99%), stability: 43610/44600 (97%)
[Stats      #1]  (GLOBAL) run time: 47h-59m-42s, clients: 2, corpus: 56014, objectives: 0, executions: 881127950, exec/sec: 5.100k
                 (CLIENT) corpus: 56014, objectives: 0, executions: 881127950, exec/sec: 5.100k, edges: 44552/44708 (99%), stability: 43610/44600 (97%)
[Testcase   #1]  (GLOBAL) run time: 47h-59m-43s, clients: 2, corpus: 56015, objectives: 0, executions: 881137484, exec/sec: 5.100k
                 (CLIENT) corpus: 56015, objectives: 0, executions: 881137484, exec/sec: 5.100k, edges: 44552/44708 (99%), stability: 43610/44600 (97%)
[Stats      #1]  (GLOBAL) run time: 47h-59m-43s, clients: 2, corpus: 56015, objectives: 0, executions: 881137484, exec/sec: 5.100k
                 (CLIENT) corpus: 56015, objectives: 0, executions: 881137484, exec/sec: 5.100k, edges: 44552/44708 (99%), stability: 43610/44600 (97%)
[Testcase   #1]  (GLOBAL) run time: 47h-59m-43s, clients: 2, corpus: 56016, objectives: 0, executions: 881141139, exec/sec: 5.100k
                 (CLIENT) corpus: 56016, objectives: 0, executions: 881141139, exec/sec: 5.100k, edges: 44552/44708 (99%), stability: 43610/44600 (97%)
[Stats      #1]  (GLOBAL) run time: 47h-59m-48s, clients: 2, corpus: 56016, objectives: 0, executions: 881163879, exec/sec: 5.100k
                 (CLIENT) corpus: 56016, objectives: 0, executions: 881163879, exec/sec: 5.100k, edges: 44552/44708 (99%), stability: 43610/44600 (97%)
```

5.1k executions per second

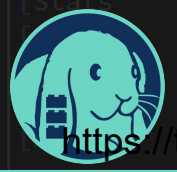
A More Complex Fuzzer

- Snapshot-based Fuzzing
- AddressSanitizer to uncover silent heap corruptions
- Scalability over cores

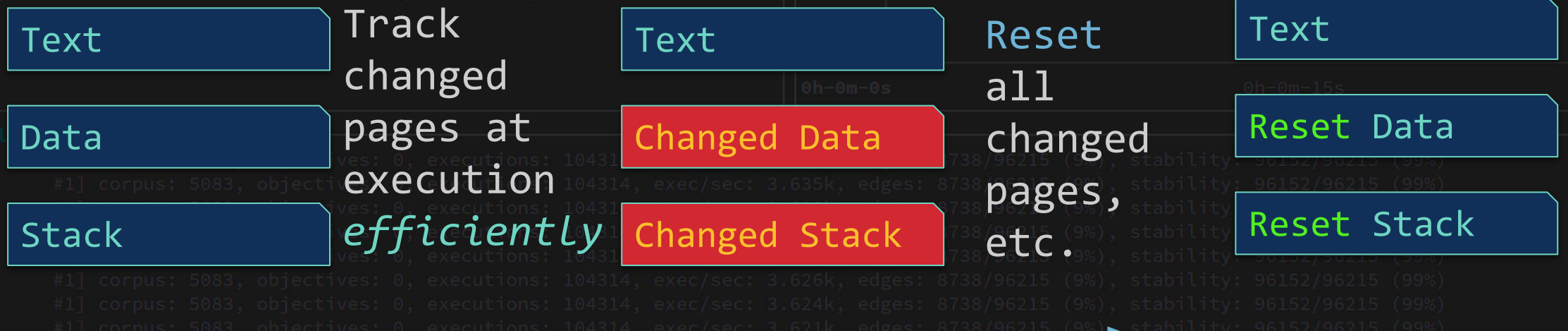


QASAN Sanitization

- Sanitizers checks wider range of errors at runtime
  - e.g. Illegal memory access
- Track all memory accesses
- Hook known libc/allocation functions (malloc/free, strcpy, …)
- Crash on out-of-bounds accesses, uaf, etc.



Userspace Snapshot



```
andrea@libaflexp:~/android_fuzzing/demo$
```

Profit!

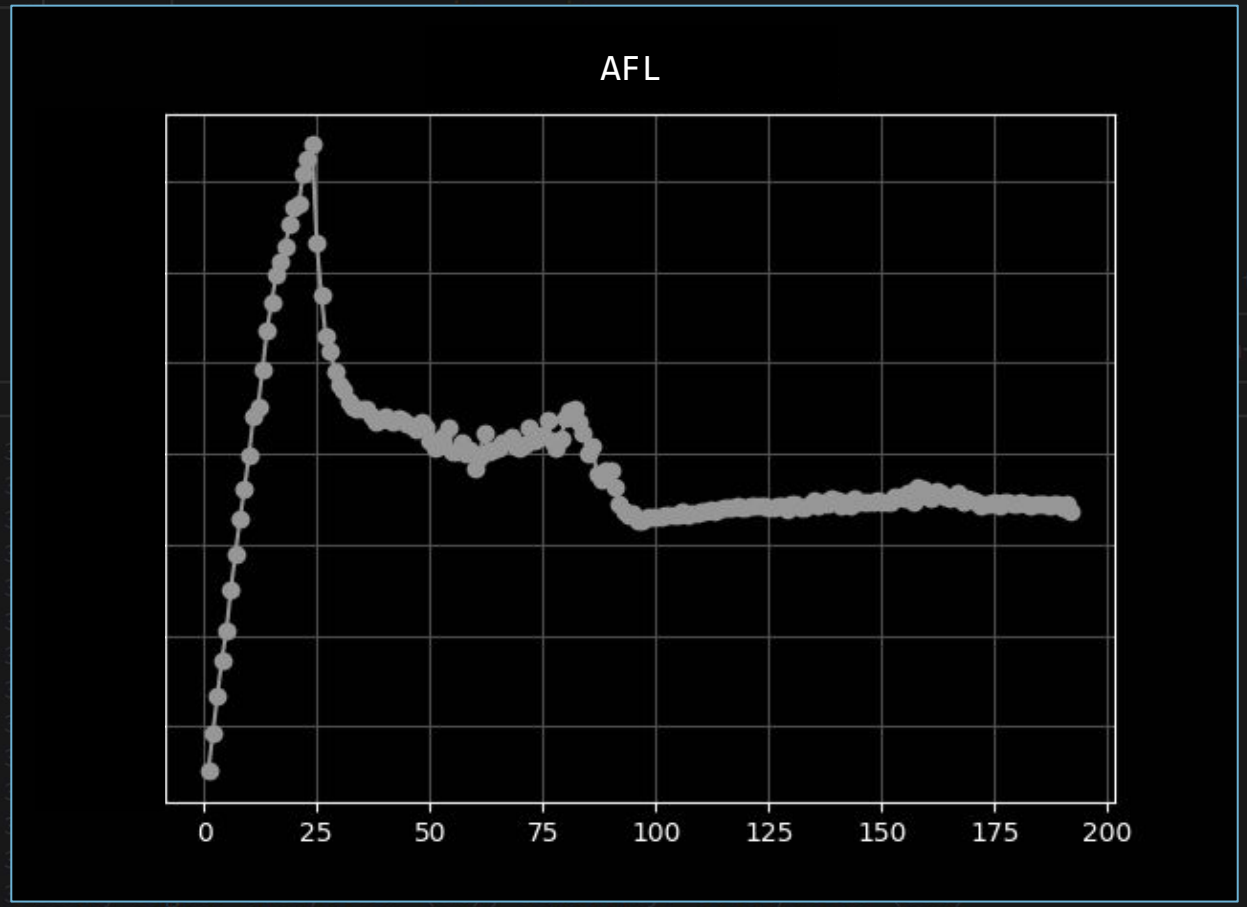
```
=================================================================
AddressSanitizer Error: Invalid 4 bytes write at 0x4000893f[redacted]
        #0 0x400006ea284c in quram_resize_[redacted] (/home/andrea/Desktop/hand_on_2/system/system/lib64/libimagecodec.quram.so+0x7[redacted])
        #1 0x400002091b38 in __libqasan_malloc /home/andrea/Desktop/LibAFL/libafl_qemu/libqasan/malloc.c:184
        #2 0x40000208f78c in malloc /home/andrea/Desktop/LibAFL/libafl_qemu/libqasan/hooks.c:88
        #3 0x400007025f9c in _Znwm (/home/andrea/Desktop/hand_on_2/system/system/lib64/libimagecodec.quram.so+0x1fef9c)
        #4 0x400007008a98 in _ZN14QuramDngRender8doRenderEP15QuramDngDecoder (/home/andrea/Desktop/hand_on_2/system/system/lib64/libimagecodec.quram.so+0x1e1a98)
Address 0x4000893f5c40 is 118 bytes to the right of the 42-byte chunk [0x4000893f5ba0,0x4000893f5bca)
Allocated at:
        #0 0x400004cb1370 in syscall (/home/andrea/Desktop/hand_on_2/system/system/lib64/libc.so+0x1f370)
        #1 0x400002091b98 in __libqasan_malloc /home/andrea/Desktop/LibAFL/libafl_qemu/libqasan/malloc.c:197
        #2 0x400002091ddc in __libqasan_calloc /home/andrea/Desktop/LibAFL/libafl_qemu/libqasan/malloc.c:258
        #3 0x40000208f83c in calloc /home/andrea/Desktop/LibAFL/libafl_qemu/libqasan/hooks.c:98
        #4 0x7ffff7fc2e50 in harnessDecode (/home/andrea/Desktop/hand_on_2/harness+0xe50)
Context:
    X0: 0x00000000000018     X1: 0x00000000000214     X2: 0x00000000000026     X3: 0x000000fffffe16
    X4: 0x0000000000022c     X5: 0x00000000000241     X6: 0x00000000000000     X7: 0x00000000000029
    X8: 0x00000000000008     X9: 0x00000000000029    X10: 0x0000000000002a    X11: 0x000000fcbbbbbb
   X12: 0x0000000000001d    X13: 0x000000bd8c8c8c    X14: 0x00000000000018    X15: 0x000000000000a8
   X16: 0xffffffffffffff58    X17: 0x000000000000a0    X18: 0x0000000000002a    X19: 0x00000000000003
   X20: 0x0000000000000e    X21: 0x00000000000009    X22: 0x0000000000002a    X23: 0x004000893f5ba0
   X24: 0x00400089401598    X25: 0x0000000000002d    X26: 0x00000000000001    X27: 0x000000000000bc
   X28: 0x00000000000000    X29: 0x004000007fe9f0    X30: 0x00400006ea26f0     Sp: 0x004000007fe990
    Pc: 0x00400006ea[redacted] Pstate: 0x00555520000000
```

Fuzz Everything, Everywhere, All at Once

Scaling to cores and machines

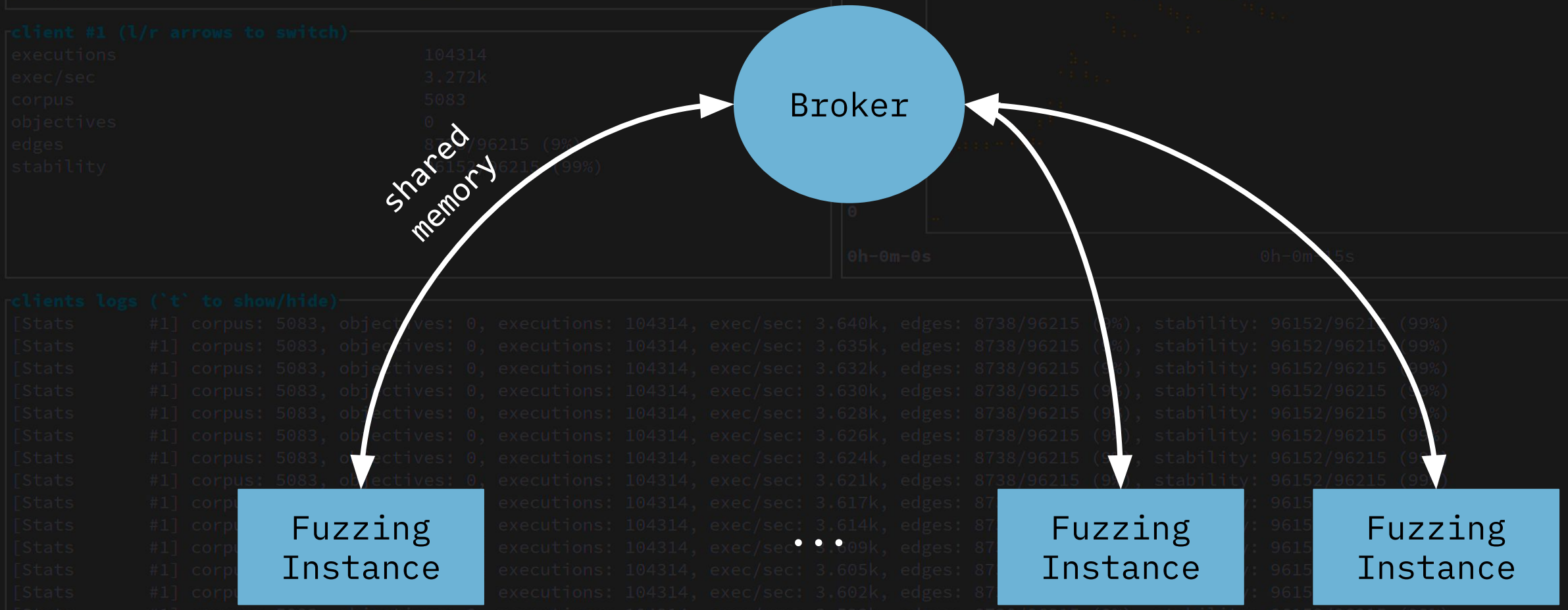
Fuzzer Scaling

- Scaling is *hard*
- Not sharing events means lots of duplicated effort
- Communication slows them down
- Communication via:
  - disk?
  - network?
  - intermittent restarts?
  - something else?

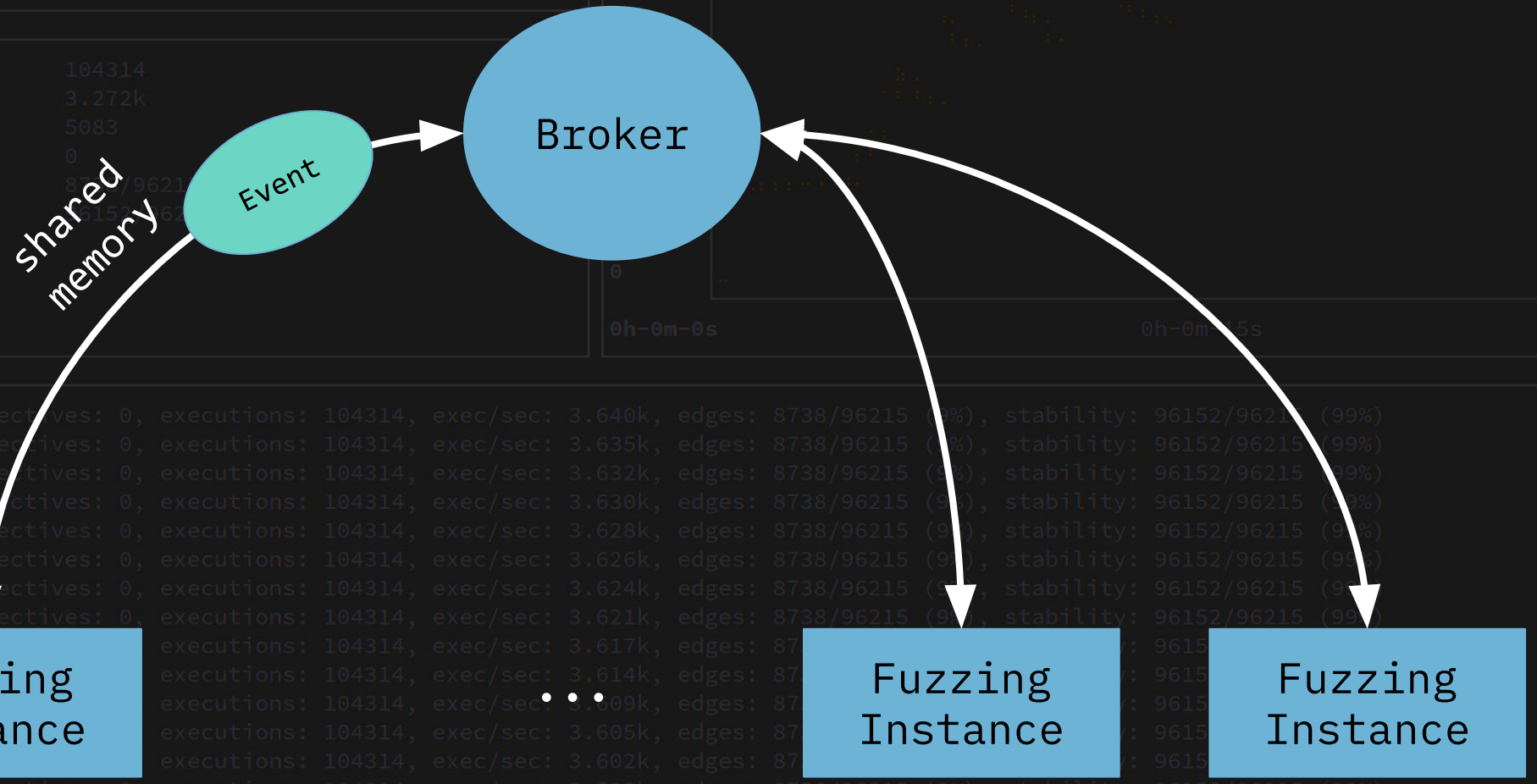


From: https://github.com/gamozolabs/aflbench

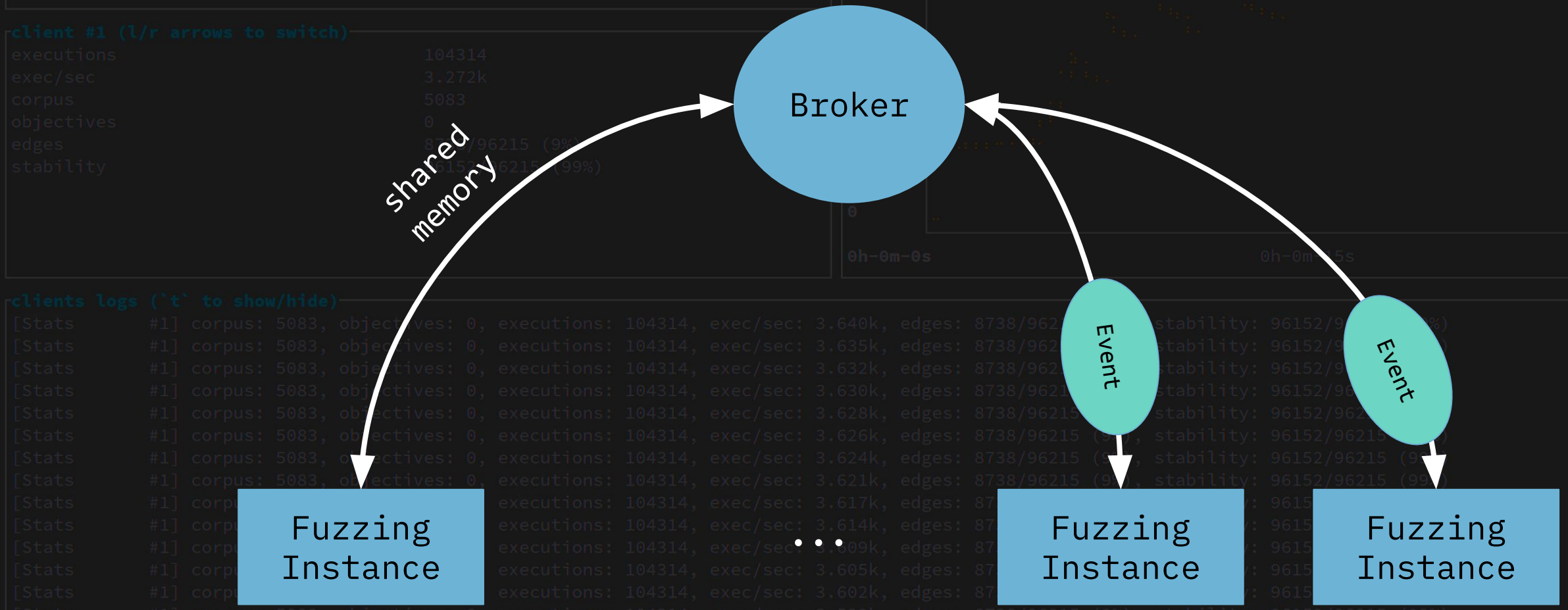
Multi-Node Fuzzing: LLMP



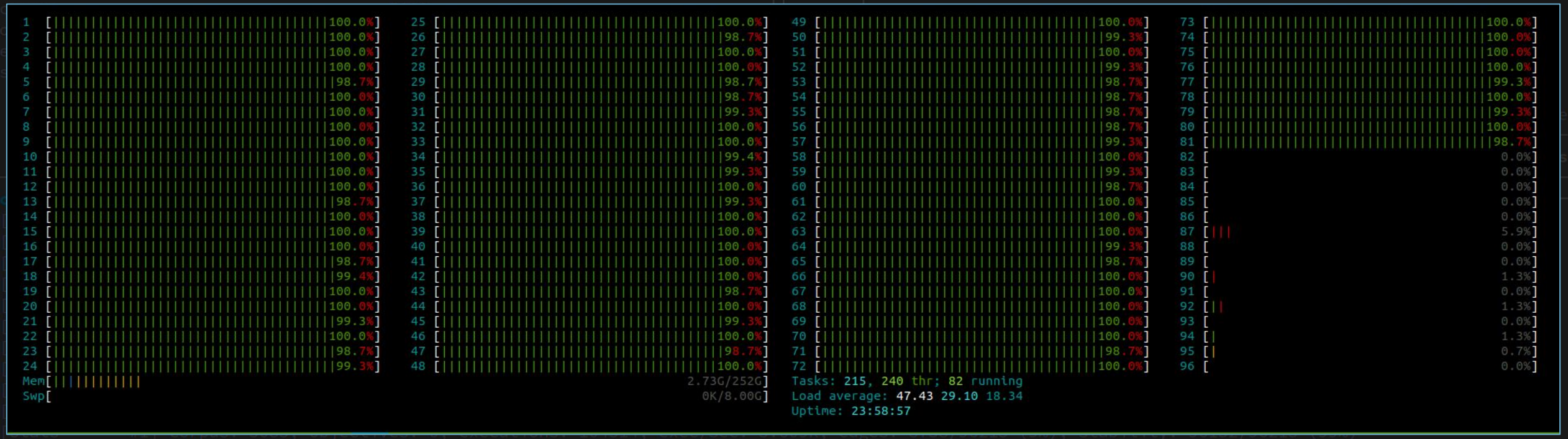
Multi-Node Fuzzing: LLMP



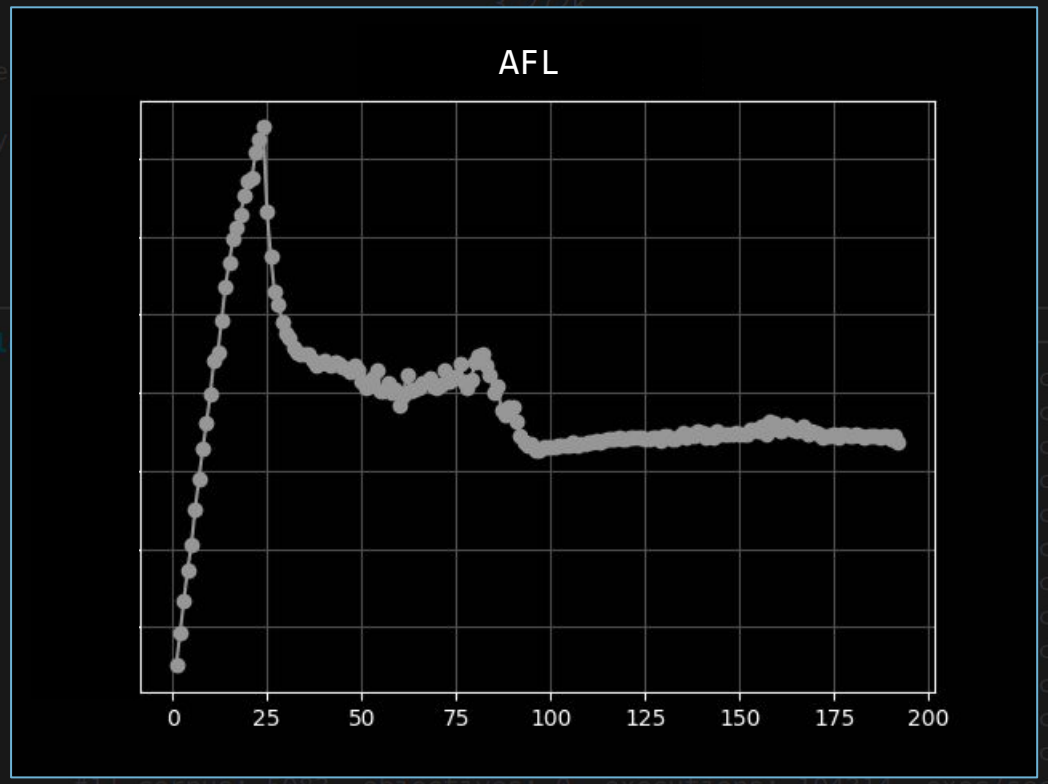
Multi-Node Fuzzing: LLMP

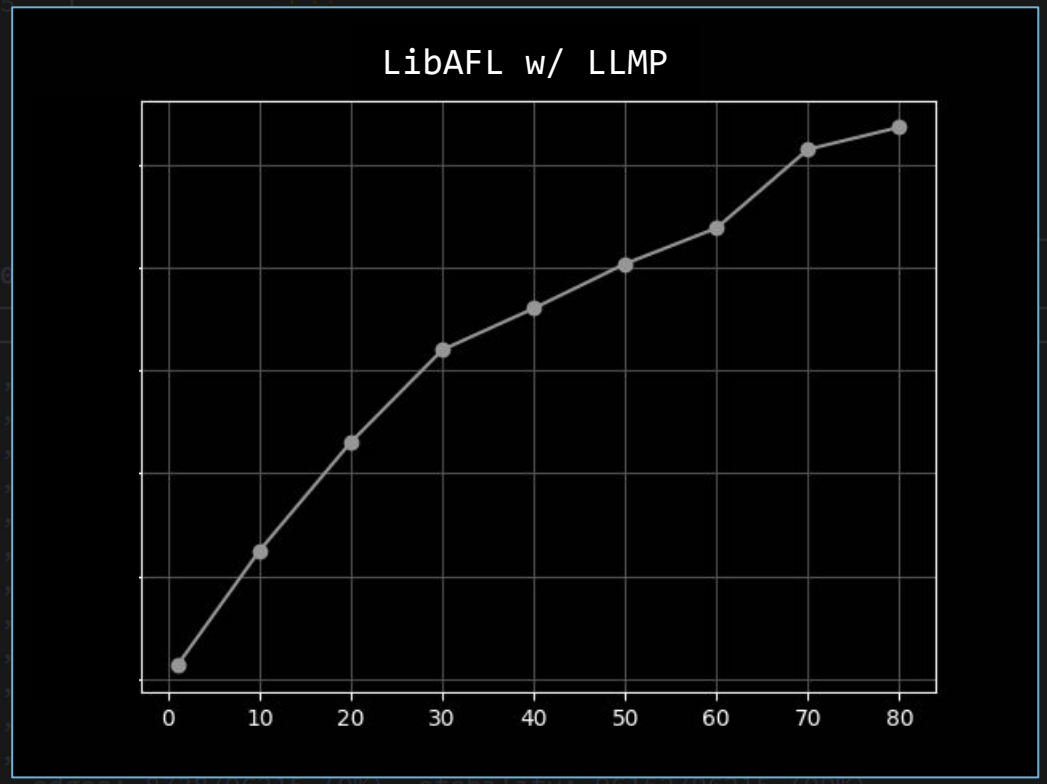


Scaling to 80 cores



Scaling Comparison: AFL and LLMP





Scaling to 80 cores

```
[Stats      #46]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 1007, objectives: 0, executions: 322104, exec/sec: 6.026k, edges: 4755/65536 (7%), stability: 65532/65536 (99%)
[Stats      #50]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 973, objectives: 0, executions: 315802, exec/sec: 5.908k, edges: 4757/65536 (7%), stability: 65532/65536 (99%)
[Stats      #51]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 890, objectives: 0, executions: 294240, exec/sec: 5.505k, edges: 4757/65536 (7%), stability: 65532/65536 (99%)
[Stats      #53]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 913, objectives: 0, executions: 305663, exec/sec: 5.719k, edges: 4755/65536 (7%), stability: 65534/65536 (99%)
[Stats      #53]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 913, objectives: 0, executions: 305663, exec/sec: 5.719k, edges: 4757/65536 (7%), stability: 65534/65536 (99%)
[Stats      #53]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 913, objectives: 0, executions: 305663, exec/sec: 5.719k, edges: 4757/65536 (7%), stability: 65534/65536 (99%)
[Stats      #55]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 884, objectives: 0, executions: 297382, exec/sec: 5.565k, edges: 4750/65536 (7%), stability: 65532/65536 (99%)
[Stats      #55]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 884, objectives: 0, executions: 297382, exec/sec: 5.565k, edges: 4755/65536 (7%), stability: 65532/65536 (99%)
[Stats      #55]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 884, objectives: 0, executions: 297382, exec/sec: 5.565k, edges: 4757/65536 (7%), stability: 65532/65536 (99%)
[Stats      #60]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 854, objectives: 0, executions: 281700, exec/sec: 5.273k, edges: 4757/65536 (7%), stability: 65532/65536 (99%)
[Stats      #63]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 1005, objectives: 0, executions: 311398, exec/sec: 5.830k, edges: 4749/65536 (7%), stability: 65532/65536 (99%)
[Stats      #63]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 1005, objectives: 0, executions: 311398, exec/sec: 5.830k, edges: 4750/65536 (7%), stability: 65532/65536 (99%)
[Stats      #63]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 1005, objectives: 0, executions: 311398, exec/sec: 5.830k, edges: 4750/65536 (7%), stability: 65532/65536 (99%)
[Stats      #71]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 834, objectives: 0, executions: 267731, exec/sec: 5.013k, edges: 4757/65536 (7%), stability: 65534/65536 (99%)
[Stats      #71]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
                  (CLIENT) corpus: 834, objectives: 0, executions: 267731, exec/sec: 5.013k, edges: 4757/65536 (7%), stability: 65534/65536 (99%)
[Stats      #71]  (GLOBAL) run time: 0h-0m-54s, clients: 82, corpus: 74661, objectives: 0, executions: 24985314, exec/sec: 462.8k (Aggregated): edges: 0.726% stability: 99.994%
```

462.8k executions per second

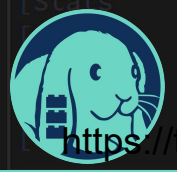
Fuzz Everything, Everywhere, All at Once

catch injections
& corruptions
at the same time

Feedback Fuzzing == Only Crashes(?)

- Coverage-based fuzzing is good at finding crashes like memory corruptions
- Unguided fuzzers like sqlmap are great at finding injection vulnerabilities but only work on network targets and have no coverage

IDEA: Find injection vulnerabilities while doing normal AFL++/libafl style fuzzing!

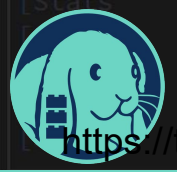


The Idea

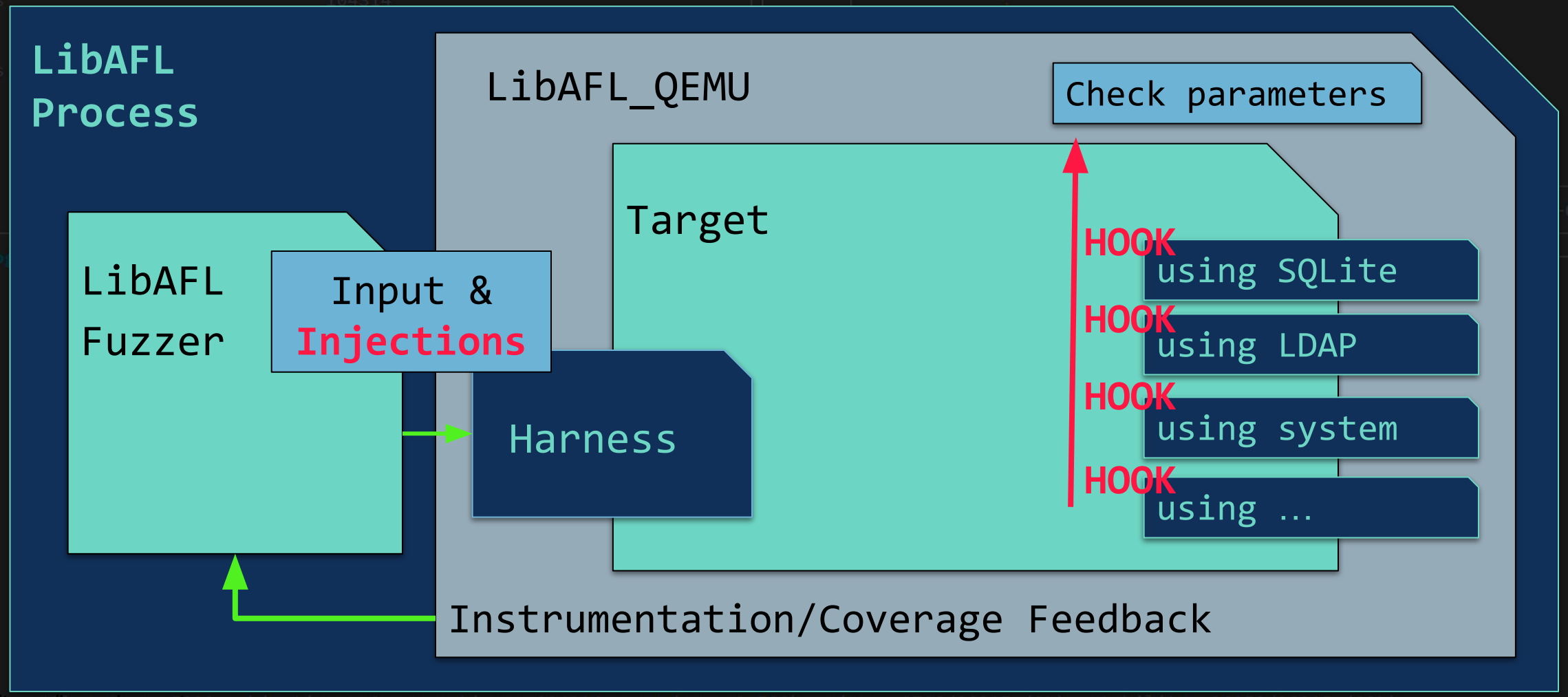
Use LibAFL-QEMU (usermode) hooks to:

- add injection tests to our mutation engine
- hook injection susceptible functions and analyze all queries
- crash if injection test is found unsanitized

Bonus: Having a flexible configuration script so users can easily modify what they want hunt for - and how



Fuzzing for Injections



Example: SQL injection configuration

injections.yml

```
- name: "sql"
  functions:
    - function: "sqlite3_exec"
      parameter: 1
    - function: "mysql_query"
      parameter: 1
  tests:
    - input_value: "'\"\"'"
      match_value: "'\"\"'"
    - input_value: "1\"' OR \""
      match_value: "1\"' OR"
```

```
sqlite3_exec() - Execute SQL
statements

Definition:
int sqlite3_exec( sqlite3 *db,
const char* sql, ... )
```

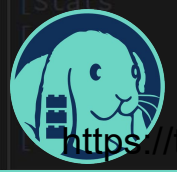
Injection into the 2nd parameter!

Advantages/Disadvantages

- False positives unlikely
- False negatives can happen - depending on your input + match config
- You can hunt for all kinds of injection vulnerabilities

  ⇒ CMD, LDAP, SQL, CSV, XML, XSS, …
- … all while doing coverage-guided fuzzing!

  All implemented using LibAFL QEMU APIs



```
marc /prg/libafl/fuzzers/qemu_injections (vhqemu) $
```

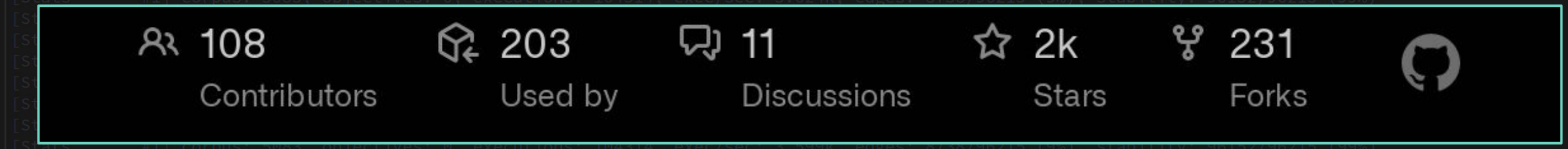
Fuzz Everything, Everywhere, All at Once

Final Words

LibAFL is FOSS!

JOIN THE CULT

https://github.com/AFLplusplus/LibAFL



Conclusion

- Fuzz everything, everywhere, all at once
- Extremely scalable fuzzers
  - QEMU is amazing
- We can fuzz Android libraries on a desktop machine
- We can hunt injections instead of boring crashes!
- with _mad speed_

```
while (questions());

char buf[16];
strncpy(buf, ""
    "Thank you for your attention."
    "\n", sizeof(buf));
printf("%s", buf);
```

Thanks y’all