LibAFL

The Advanced Fuzzing Library

Dominik Maier & Andrea Fioraldi

@domenuk, @andreafioraldi

{dominik, andrea}@aflplus.plus

Who We Are

- Hackademics (both PhD students)

Technische Universität Berlin

EURECOM Sophia Antipolis

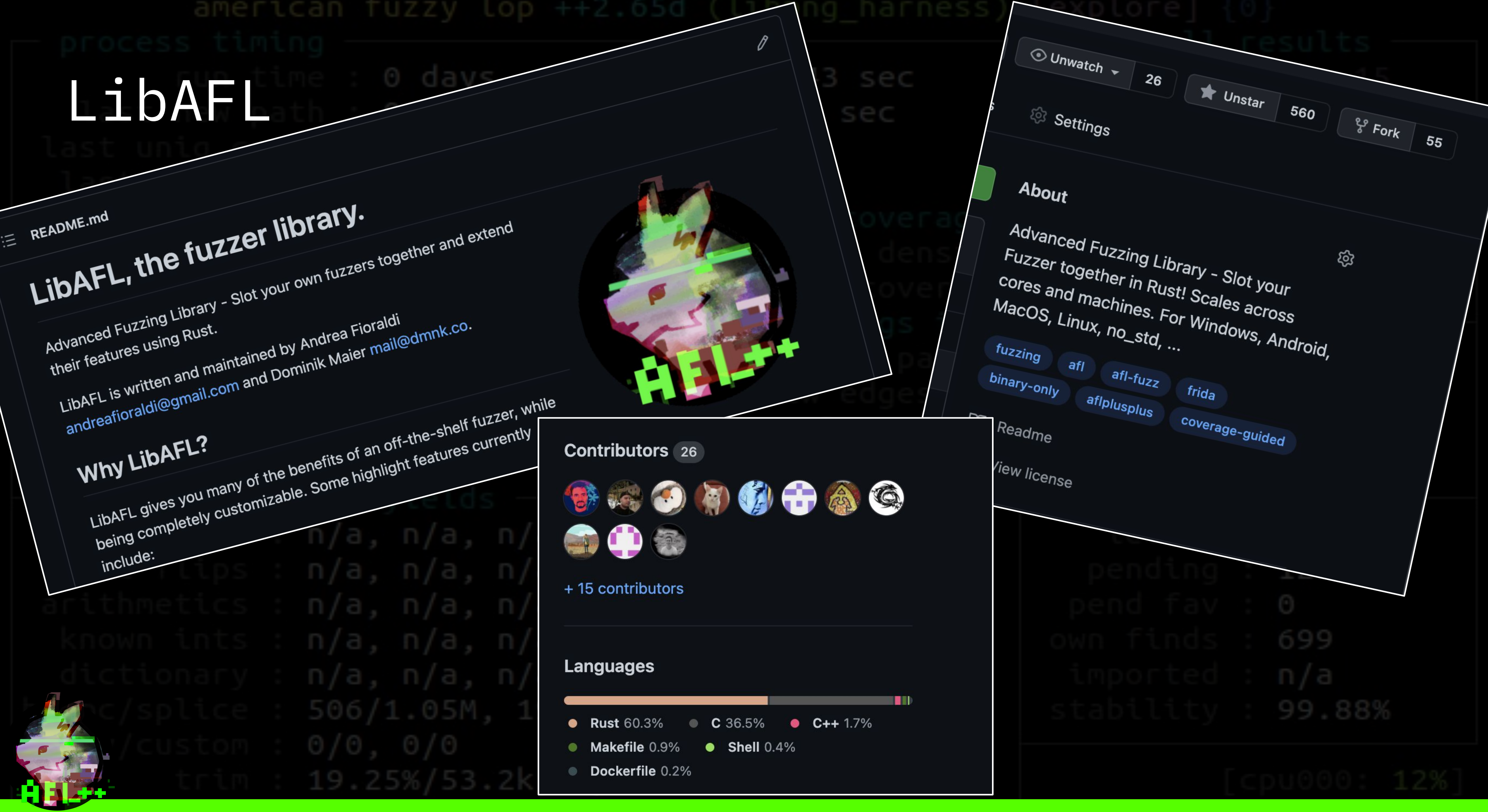
Who We Are

- Hackademics (both PhD students)
- CTFers

Who We Are

- Hackademics (both PhD students)
- CTFers
- Part of the AFL++ team

LibAFL



README.md
LibAFL, the fuzzer library.
Advanced Fuzzing Library - Slot your own fuzzers together and extend their features using Rust.
LibAFL is written and maintained by Andrea Fioraldi andreafioraldi@gmail.com and Dominik Maier mail@dmnk.co.
Why LibAFL?
LibAFL gives you many of the benefits of an off-the-shelf fuzzer, while being completely customizable. Some highlight features currently include:
AFL++
Unwatch 26
Unstar 560
Fork 55
Settings
About
Advanced Fuzzing Library - Slot your Fuzzer together in Rust! Scales across cores and machines. For Windows, Android, MacOS, Linux, no_std, ...
fuzzing
afl
afl-fuzz
frida
binary-only
aflplusplus
coverage-guided
Readme
View license
Contributors 26
+ 15 contributors
Languages
Rust 60.3%
C 36.5%
C++ 1.7%
Makefile 0.9%
Shell 0.4%
Dockerfile 0.2%

This Talk

We present a library for fuzzers that are

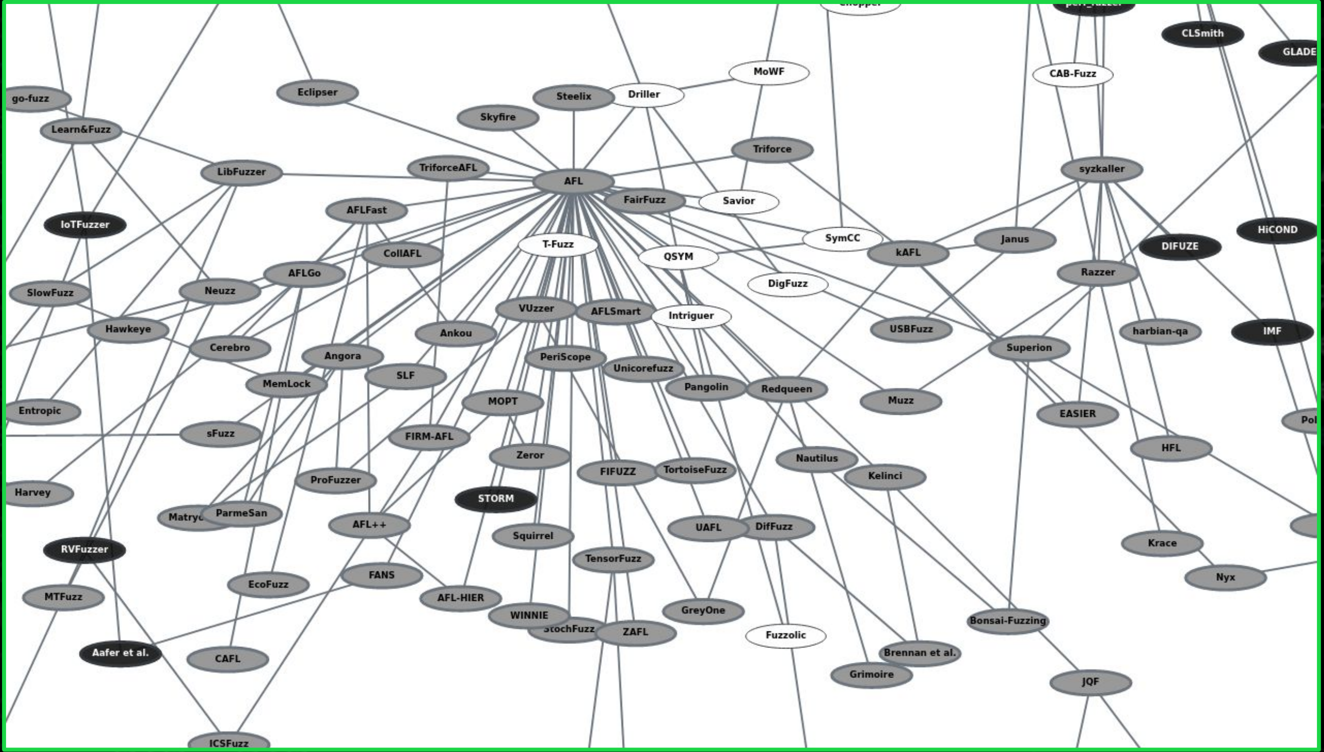
- **Fast** (low IPC, runtime overhead)
- **Scalable** (almost linearly to 200+ cores)
- **Portable** (Android, Windows, MacOS, Linux, Kernels, …)
- **State-of-the-Art** (Hybrid-, Grammar-, Token-, Feedback-Fuzzing)
- **Multi-instrumentation** (binary-only Frida & Qemu, Clang, Python,…)

And, most importantly, very extendable with your own components.

We will show a simple, and a very advance example in code.

(see /fuzzers, the book, and the API docs to get started)

Problem: Fuzzer Fragmentation



From https://fuzzing-survey.org/

Cause: Monolithic Codebases

Fuzzers are

⇒ Designed to be tools

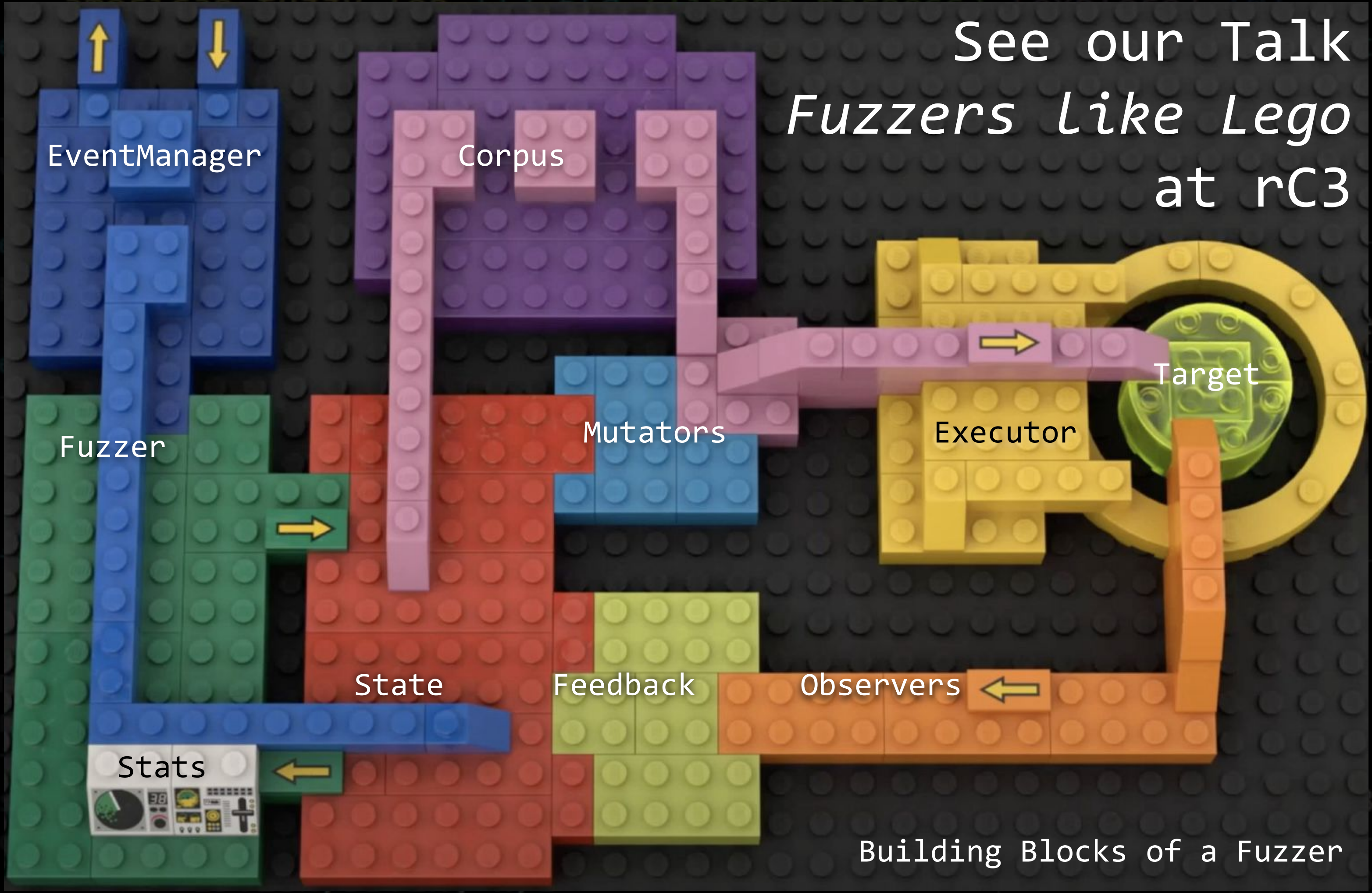
⇒ Not designed with code reuse in mind

⇒ Hard to extend

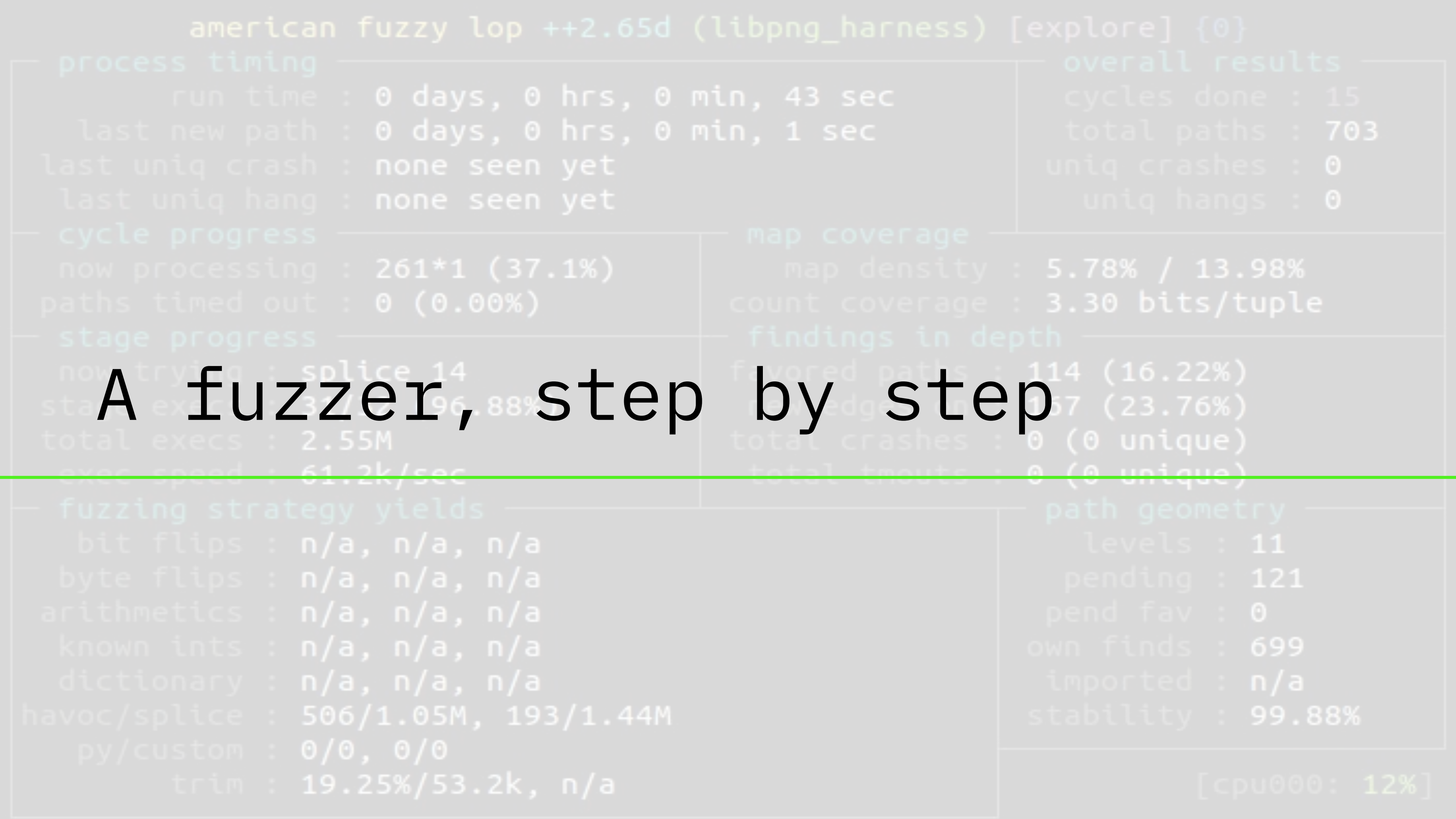
Many fuzzers are incompatible forks of others (usually AFL)

This makes them incompatible with orthogonal techniques

Solution:
Abstract Fuzzing Concepts


See our Talk
Fuzzers like Lego
at rC3
EventManager
Corpus
Fuzzer
Mutators
Target
Executor
State
Feedback
Observers
Stats
Building Blocks of a Fuzzer

A fuzzer, step by step



The Function Under Test

Let's look at a simple function we want to fuzz, first.

The Function Under Test

```rust
let mut harness = |input: &BytesInput| {
    let target = input.target_bytes();
    let buf = target.as_slice();
    if buf.len() > 0 && buf[0] == 'a' as u8 {
        if buf.len() > 1 && buf[1] == 'b' as u8 {
            if buf.len() > 2 && buf[2] == 'c' as u8 {
                panic!("=)");
            }
        }
    }
};
// To test the panic:
// let input = BytesInput::new("abc".as_bytes());
// harness(&input);
```

A Skeleton to Generate and Run Inputs

We will build the target and generate random input for it.

Let's see how to do this simple task in LibAFL.

A Skeleton to Generate and Run Inputs

```rust
// create a State from scratch
let mut state = StdState::new(
    // RNG
    StdRand::with_seed(current_nanos()),
    // Corpus that will be evolved, we keep it in memory for performance
    InMemoryCorpus::new(),
    // Corpus in which we store solutions (crashes in this example),
    // on disk so the user can get them after stopping the fuzzer
    OnDiskCorpus::new(PathBuf::from("./crashes")).unwrap(),
    (),
);
```

A Skeleton to Generate and Run Inputs

```
// The Stats trait define how the fuzzer stats are reported to the user
let stats = SimpleStats::new(|s| println!("{}", s));

// The event manager handle the various events generated during fuzzing
// such as the notification of the addition of a new item to the corpus
let mut mgr = SimpleEventManager::new(stats);
```

A Skeleton to Generate and Run Inputs

```
// A queue policy to get testcasess from the corpus
let scheduler = QueueCorpusScheduler::new();

// A fuzzer with feedbacks and a corpus scheduler
let mut fuzzer = StdFuzzer::new(scheduler, (), ());
```

A Skeleton to Generate and Run Inputs

```rust
// Create the executor for an in-process function
let mut executor = InProcessExecutor::new(
    &mut harness,
    (),
    &mut fuzzer,
    &mut state,
    &mut mgr,
)
.expect("Failed to create the Executor");
```

A Skeleton to Generate and Run Inputs

```rust
// Generator of printable bytearrays of max size 32
let mut generator = RandPrintablesGenerator::new(32);

// Generate 8 initial inputs
state
    .generate_initial_inputs(&mut executor, &mut generator,
                             &mut mgr, &scheduler, 8)
.expect("Failed to generate the initial corpus".into());
```

A Skeleton to Generate and Run Inputs

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.04s
     Running `target/debug/baby_fuzzer`
[LOG Debug]: Loaded 0 over 8 initial testcases
```

A Skeleton to Generate and Run Inputs

Recap, we created a Fuzzer out of:

- a State: everything LibAFL keeps, including Corpus, Metadata, RNG state, … - it get serialized and re-read on crash
- a Stats instance that simply prints the current stats. Advanced stats may print a complete afl-like screen.
- an EventManager: fires events, for example new Testcases and, depending on implementation, propagates them to other nodes.
- the Executor: this calls the Function under Test
- a Generator: Generators emit bytes, according to rules.

Evolving the Corpus With Feedbacks

We now have everything in place to call the function with random input.

However, modern fuzzing uses Feedback.

By observing the target, the fuzzer learns about interesting behavior.

It sees which Testcases are interesting, and can focus on them.

While most fuzzers instrument the target to get coverage as metric, for this simple fuzzer, we build it manually.

Evolving the Corpus With Feedbacks

```rust
// Coverage map with explicit assignments due to the lack of instrumentation
static mut SIGNALS: [u8; 16] = [0; 16];
fn signals_set(idx: usize) {
    unsafe { SIGNALS[idx] = 1 };
}

let mut harness = |input: &BytesInput| {
    let target = input.target_bytes();
    let buf = target.as_slice();
    signals_set(0);
    if buf.len() > 0 && buf[0] == 'a' as u8 {
        signals_set(1);
        if buf.len() > 1 && buf[1] == 'b' as u8 {
            signals_set(2);
            if buf.len() > 2 && buf[2] == 'c' as u8 {
                panic!("=)");
            }
        }
    }
    ExitKind::Ok
};
```

Evolving the Corpus With Feedbacks

```rust
// Coverage map with explicit assignments due to the lack of instrumentation
static mut SIGNALS: [u8; 16] = [0; 16];
fn signals_set(idx: usize) {
    unsafe { SIGNALS[idx] = 1 };
}

let mut harness = |input: &BytesInput| {
    let target = input.target_bytes();
    let buf = target.as_slice();
    signals_set(0);
    if buf.len() > 0 && buf[0] == 'a' as u8 {
        signals_set(1);
        if buf.len() > 1 && buf[1] == 'b' as u8 {
            signals_set(2);
            if buf.len() > 2 && buf[2] == 'c' as u8 {
                panic!("=)");
            }
        }
    }
    ExitKind::Ok
};
```

Evolving the Corpus With Feedbacks

```rust
// Create an observation channel using the signals map
let observer = StdMapObserver::new("signals", unsafe { &mut SIGNALS });

// Create the executor for an in-process function with just one observer
let mut executor =
    InProcessExecutor::new(&mut harness, tuple_list!(observer),
                           &mut state, &mut mgr)
    .expect("Failed to create the Executor".into());
```

Evolving the Corpus With Feedbacks

```rust
// Create an observation channel using the signals map
let observer = StdMapObserver::new("signals", unsafe { &mut SIGNALS });

// Create the executor for an in-process function with just one observer
let mut executor =
    InProcessExecutor::new(&mut harness, tuple_list!(observer),
                           &mut state, &mut mgr)
    .expect("Failed to create the Executor".into());
```

Evolving the Corpus With Feedbacks

```rust
// The state of the map feedback.
let feedback_state = MapFeedbackState::with_observer(&observer);

// Feedback to rate the interestingness of an input
let feedback = MaxMapFeedback::new(&feedback_state, &observer);

// A feedback to choose if an input is a solution or not
let objective = CrashFeedback::new();
```

Evolving the Corpus With Feedbacks

```
// create a State from scratch
let mut state = StdState::new(
    // RNG
    StdRand::with_seed(current_nanos()),
    // The Corpus that will be evolved
    InMemoryCorpus::new(),
    // The Corpus in which we store solutions (crashes in this example), on
disk so the user can get them after stopping the fuzzer
    OnDiskCorpus::new(PathBuf::from("./crashes")).unwrap(),
    // States of the feedbacks, the data related to the feedbacks that  you
want to persist in the State.
    tuple_list!(feedback_state),
);

// A fuzzer with feedbacks and a corpus scheduler
let mut fuzzer = StdFuzzer::new(scheduler, feedback, objective);
```

Evolving the Corpus With Feedbacks

```
// create a State from scratch
let mut state = StdState::new(
    // RNG
    StdRand::with_seed(current_nanos()),
    // The Corpus that will be evolved
    InMemoryCorpus::new(),
    // The Corpus in which we store solutions (crashes in this example), on
disk so the user can get them after stopping the fuzzer
    OnDiskCorpus::new(PathBuf::from("./crashes")).unwrap(),
    // States of the feedbacks, the data related to the feedbacks that  you
want to persist in the State.
    tuple_list!(feedback_state),
);

// A fuzzer with feedbacks and a corpus scheduler
let mut fuzzer = StdFuzzer::new(scheduler, feedback, objective);
```

Evolving the Corpus With Feedbacks

Recap, we

- Manually added a coverage map to the target
- Added a *panic!* (rust for "crash") to the target
- Use a MapObserver that points LibAFL to this map
- Use a MapMaxFeedback: with it, the fuzzer tries to increase the map
- Use a CrashFeedback that will tell the fuzzer when it his our crash
- Added a Corpus for new interesting Testcases and one for Solutions
- We store the MapFeedbacks's state in the fuzzer State

Observers and Feedbacks are decoupled, so we can use the same feedback for multiple targets, and even have feedbacks over multiple observers.

The Actual Fuzzing

All that's left to do now is:

Loop the fuzzer until it finds a crash.

The Actual Fuzzing

```rust
// Setup a mutational stage with a basic bytes mutator
let mutator = StdScheduledMutator::new(havoc_mutations());
let mut stages = tuple_list!(StdMutationalStage::new(mutator));

fuzzer
    .fuzz_loop(&mut stages, &mut executor, &mut state, &mut mgr)
.expect("Error in the fuzzing loop");
```

The Actual Fuzzing

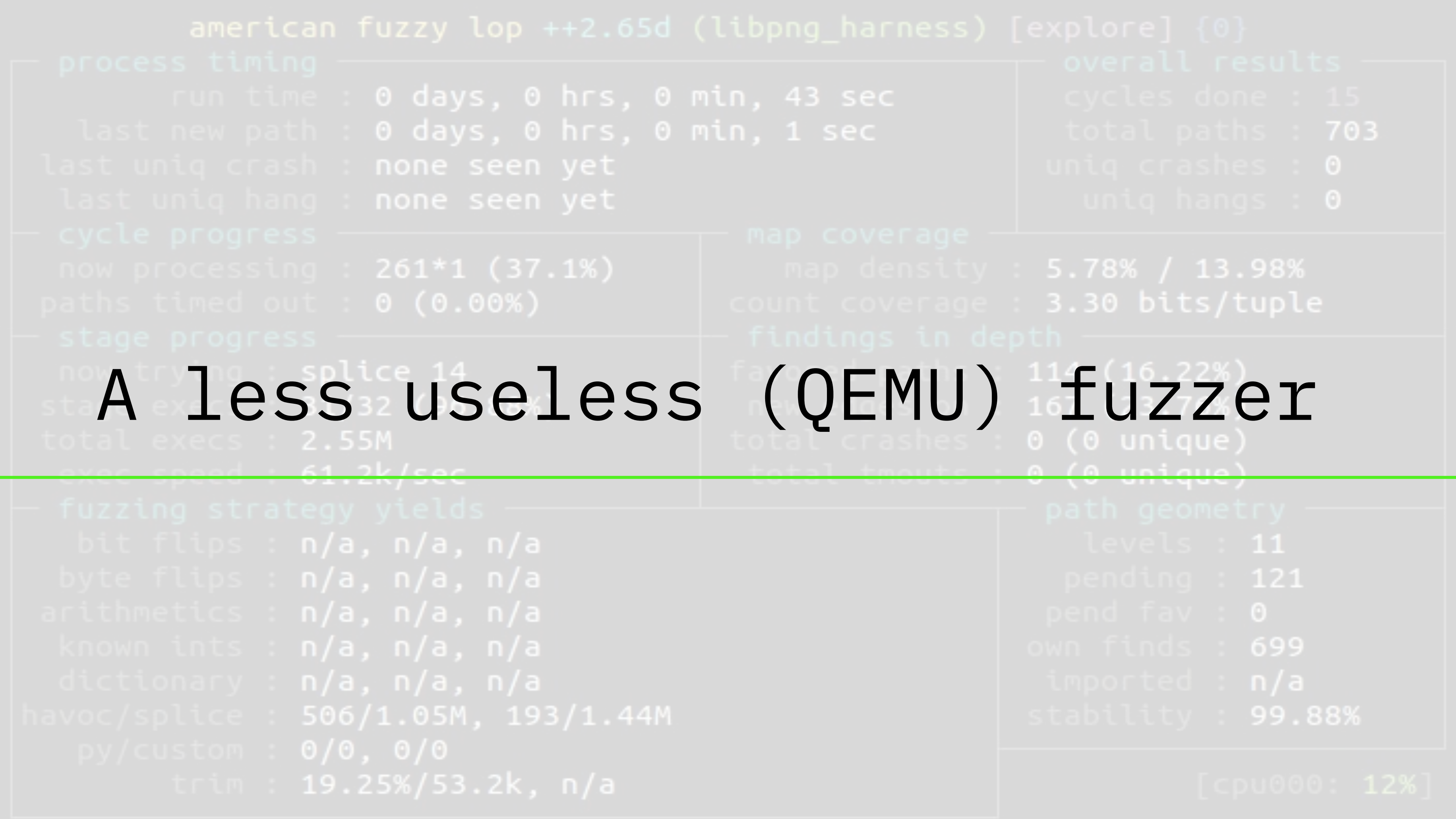
```
$ cargo run --release
   Compiling baby_fuzzer v0.1.0 (/home/andrea/Desktop/baby_fuzzer)
    Finished release [optimized] target(s) in 1.56s
     Running `target/release/baby_fuzzer`
[New Testcase] clients: 1, corpus: 2, objectives: 0, executions: 1, exec/sec: 0
[LOG Debug]: Loaded 1 over 8 initial testcases
[New Testcase] clients: 1, corpus: 3, objectives: 0, executions: 804, exec/sec: 0
[New Testcase] clients: 1, corpus: 4, objectives: 0, executions: 1408, exec/sec: 0
thread 'main' panicked at '=)', src/main.rs:35:21
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
Crashed with SIGABRT
Child crashed!
[Objective] clients: 1, corpus: 4, objectives: 1, executions: 1408, exec/sec: 0
Waiting for broker...
Bye!
```

The Actual Fuzzing

```
$ cargo run --release
   Compiling baby_fuzzer v0.1.0 (/home/andrea/Desktop/baby_fuzzer)
    Finished release [optimized] target(s) in 1.56s
     Running `target/release/baby_fuzzer`
[New Testcase] clients: 1, corpus: 2, objectives: 0, executions: 1, exec/sec: 0
[LOG Debug]: Loaded 1 over 8 initial testcases
[New Testcase] clients: 1, corpus: 3, objectives: 0, executions: 804, exec/sec: 0
[New Testcase] clients: 1, corpus: 4, objectives: 0, executions: 1408, exec/sec: 0
thread 'main' panicked at '=)', src/main.rs:35:21
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
Crashed with SIGABRT
Child crashed!
[Objective] clients: 1, corpus: 4, objectives: 1, executions: 1408, exec/sec: 0
Waiting for broker...
Bye!
```

It crashes too fast to even calculate the execs/sec stat

A less useless (QEMU) fuzzer



QEMU

QEMU is the Quick EMUlator

We built an API around QEMU to set hooks (e.g. on each block execution or memory operation) and control the guest state.

You can load ELFs in libafl_qemu and fuzz binary-only with coverage, CmpLog, snapshots and more.

QEMU setup

```rust
// Initialize QEMU
let args: Vec<String> = env::args().collect();
let env: Vec<(String, String)> = env::vars().collect();
emu::init(&args, &env);

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

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

emu::set_breakpoint(test_one_input_ptr); // LLVMFuzzerTestOneInput
emu::run();
```

QEMU setup

```
// Initialize QEMU
let args: Vec<String> = env::args().collect();
let env: Vec<(String, String)> = env::vars().collect();
emu::init(&args, &env);

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

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

emu::set_breakpoint(test_one_input_ptr); // LLVMFuzzerTestOneInput
emu::run();
```

QEMU setup

```rust
// Get the return address
let stack_ptr: u64 = emu::read_reg(Amd64Regs::Rsp).unwrap();
let mut ret_addr = [0u64];
emu::read_mem(stack_ptr, &mut ret_addr);
let ret_addr = ret_addr[0];

println!("Stack pointer = {:#x}", stack_ptr);
println!("Return address = {:#x}", ret_addr);

emu::remove_breakpoint(test_one_input_ptr); // LLVMFuzzerTestOneInput
emu::set_breakpoint(ret_addr); // LLVMFuzzerTestOneInput ret addr

let input_addr = emu::map_private(0, 4096, MmapPerms::ReadWrite).unwrap();
println!("Placing input at {:#x}", input_addr);
```

The Harness

```
// The wrapped harness function, calling out to the LLVM-style harness
let mut harness = |input: &BytesInput| {
    let target = input.target_bytes();
    let mut buf = target.as_slice();
    if buf.len() > 4096 {
        buf = &buf[0..4096];
    }

    emu::write_mem(input_addr, buf);
    emu::write_reg(Amd64Regs::Rdi, input_addr).unwrap();
    emu::write_reg(Amd64Regs::Rsi, len).unwrap();
    emu::write_reg(Amd64Regs::Rip, test_one_input_ptr).unwrap();
    emu::write_reg(Amd64Regs::Rsp, stack_ptr).unwrap();

    emu::run();
    ExitKind::Ok
};
```

The Launcher

```rust
// The shared memory allocator
let shmem_provider = StdShMemProvider::new().expect("Failed to init shared memory");
// The stats reporter for the broker
let stats = MultiStats::new(|s| println!("{}", s));
// Build and run a Launcher
match Launcher::builder()
    .shmem_provider(shmem_provider)
    .broker_port(broker_port)
    .configuration(EventConfig::from_build_id())
    .stats(stats)
    .run_client(&mut run_client)
    .cores(&cores)
    .stdout_file(Some("/dev/null"))
    .build()
    .launch()
{
    Ok(()) => (),
    Err(Error::ShuttingDown) => println!("Fuzzing stopped by user. Good bye."),
    Err(err) => panic!("Failed to run launcher: {:?}", err),
}
```

The Launcher

```rust
// The shared memory allocator
let shmem_provider = StdShMemProvider::new().expect("Failed to init shared memory");
// The stats reporter for the broker
let stats = MultiStats::new(|s| println!("{}", s));
// Build and run a Launcher
match Launcher::builder()
    .shmem_provider(shmem_provider)
    .broker_port(broker_port)
    .configuration(EventConfig::from_build_id())
    .stats(stats)
    .run_client(&mut run_client)
    .cores(&cores)
    .stdout_file(Some("/dev/null"))
    .build()
    .launch()
{
    Ok(()) => (),
    Err(Error::ShuttingDown) => println!("Fuzzing stopped by user. Good bye."),
    Err(err) => panic!("Failed to run launcher: {:?}", err),
}
```

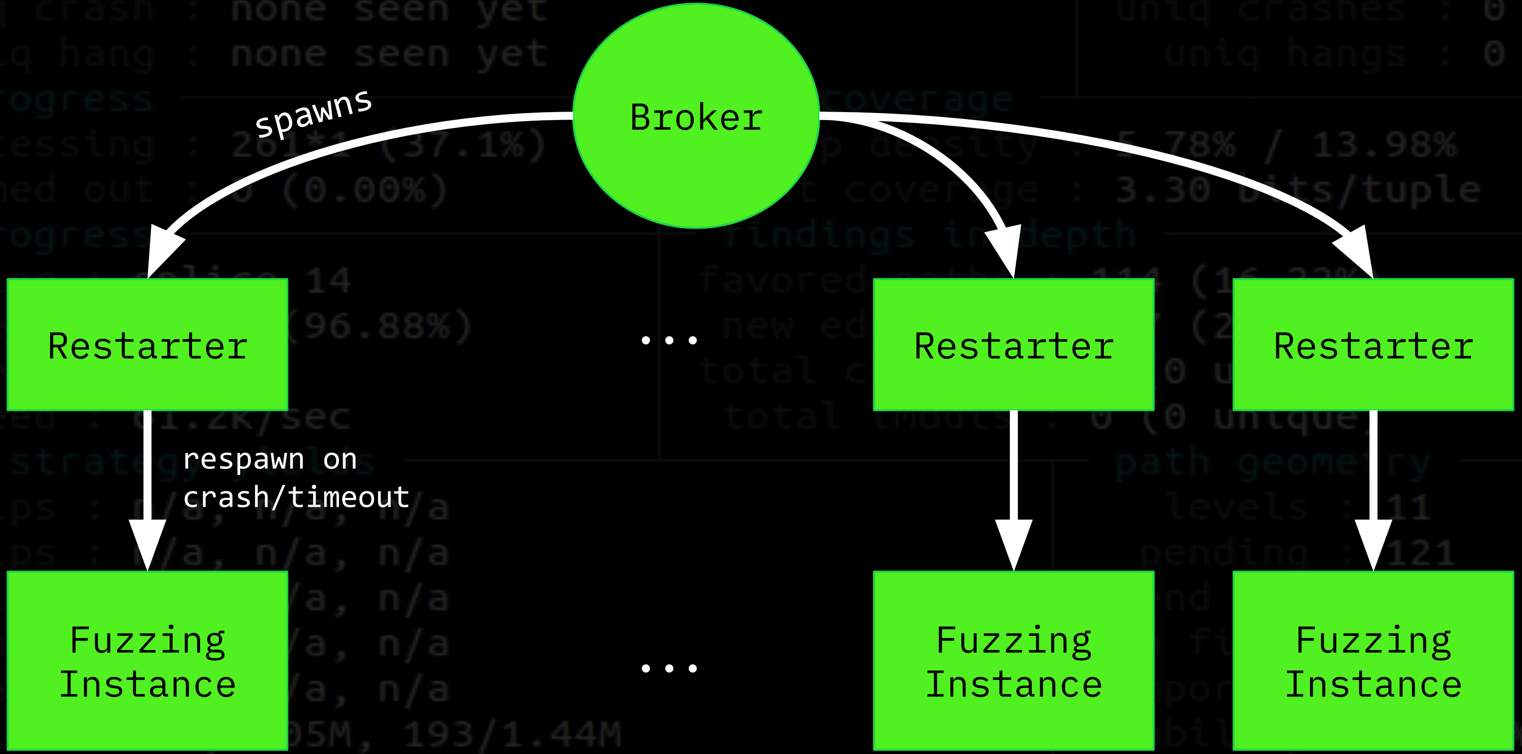
The Launcher

```rust
// The shared memory allocator
let shmem_provider = StdShMemProvider::new().expect("Failed to init shared memory");
// The stats reporter for the broker
let stats = MultiStats::new(|s| println!("{}", s));
// Build and run a Launcher
match Launcher::builder()
    .shmem_provider(shmem_provider)
    .broker_port(broker_port)
    .configuration(EventConfig::from_build_id())
    .stats(stats)
    .run_client(&mut run_client)
    .cores(&cores)
    .stdout_file(Some("/dev/null"))
    .build()
    .launch()
{
    Ok(()) => (),
    Err(Error::ShuttingDown) => println!("Fuzzing stopped by user. Good bye."),
    Err(err) => panic!("Failed to run launcher: {:?}", err),
}
```

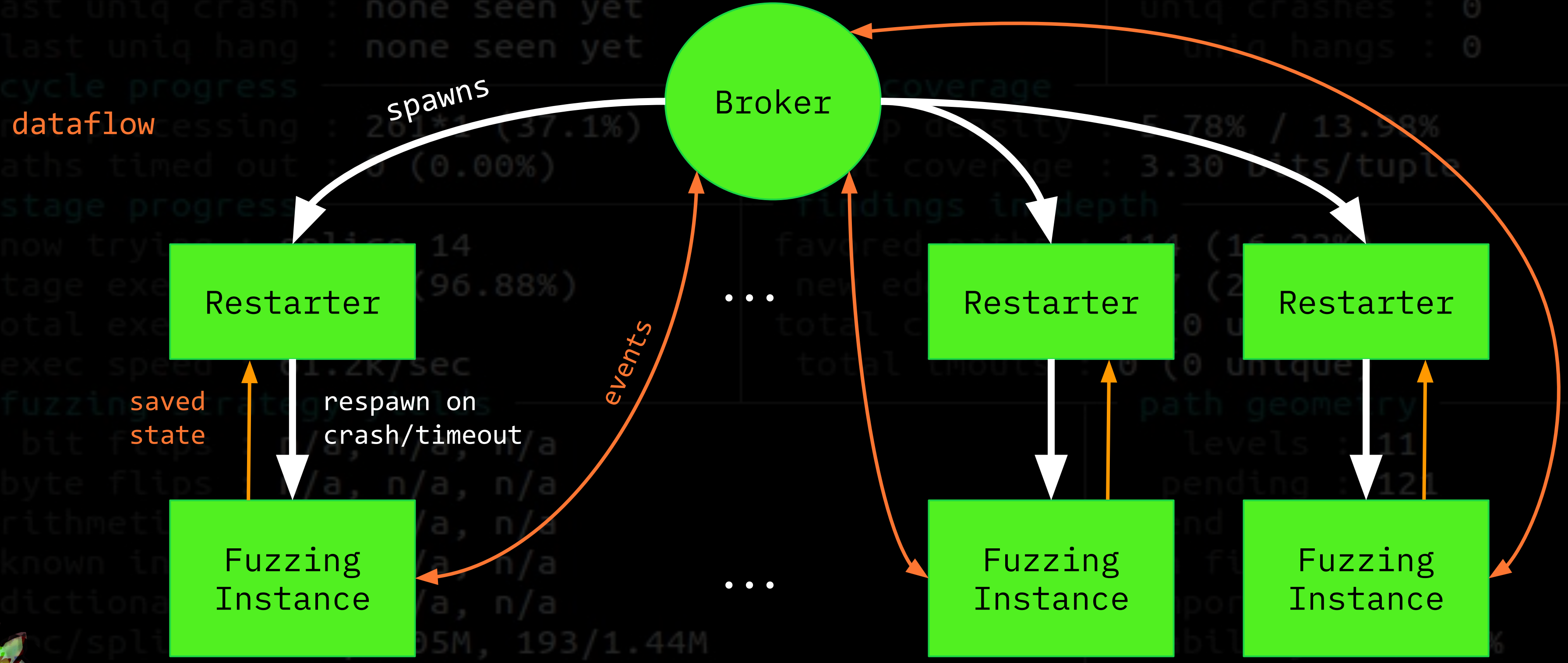
The Launcher

```rust
// The shared memory allocator
let shmem_provider = StdShMemProvider::new().expect("Failed to init shared memory");
// The stats reporter for the broker
let stats = MultiStats::new(|s| println!("{}", s));
// Build and run a Launcher
match Launcher::builder()
    .shmem_provider(shmem_provider)
    .broker_port(broker_port)
    .configuration(EventConfig::from_build_id())
    .stats(stats)
    .run_client(&mut run_client)
    .cores(&cores)
    .stdout_file(Some("/dev/null"))
    .build()
    .launch()
{
    Ok(()) => (),
    Err(Error::ShuttingDown) => println!("Fuzzing stopped by user. Good bye."),
    Err(err) => panic!("Failed to run launcher: {:?}", err),
}
```

The Launcher



The Launcher



The Observers

```rust
let mut run_client = |state: Option<_>, mut mgr, _core_id| {
    // Create an observation channel using the coverage map
    let edges = unsafe { &mut hooks::EDGES_MAP };
    let edges_counter = unsafe { &mut hooks::MAX_EDGES_NUM };
    let edges_observer =
        HitcountsMapObserver::new(
          VariableMapObserver::new("edges", edges, edges_counter)
         );

    // Create an observation channel to keep track of the execution time
    let time_observer = TimeObserver::new("time");

    // …
}
```

The Feedbacks

```rust
// The state of the edges feedback, in this case the coverage seen so far.
let feedback_state = MapFeedbackState::with_observer(&edges_observer);

// Feedback to rate the interestingness of an input
// This one is composed by two Feedbacks in OR
let feedback = feedback_or!(
    // New maximization map feedback linked to the observer and the feedback state
    MaxMapFeedback::new_tracking(&feedback_state, &edges_observer, true, false),
    // Time feedback, this one does not need a feedback state
    TimeFeedback::new_with_observer(&time_observer)
);

// A feedback to choose an input as solution if it is a crash or a timeout
let objective = feedback_or_fast!(CrashFeedback::new(), TimeoutFeedback::new());
```

The State on restart

```
let mut run_client = |state: Option<_>, mut mgr, _core_id| {
    // ...

    // If not restarting, create a State from scratch
    let mut state = state.unwrap_or_else(|| {
        StdState::new(
            StdRand::with_seed(current_nanos()),
            InMemoryCorpus::new(),
            OnDiskCorpus::new(objective_dir.clone()).unwrap(),
            // The feedback state persist in the State
            tuple_list!(feedback_state),
        )
    });

    // ...
}
```

AFL-like Corpus Culling

```
// A minimization+queue policy to get testcasess from the corpus
let scheduler =
IndexesLenTimeMinimizerCorpusScheduler::new(QueueCorpusScheduler::new());

// A fuzzer with feedbacks and a corpus scheduler
let mut fuzzer = StdFuzzer::new(scheduler, feedback, objective);
```

The QEMU Executor

```rust
// Create a QEMU in-process executor
let executor = QemuExecutor::new(
    &mut harness,
    // The QEMU helpers define common hooks like coverage tracking hooks
    // In the specific, QemuEdgeCoverageHelper hooks every executed edge
    // for binary-only collision free edge coverage
    tuple_list!(QemuEdgeCoverageHelper::new()),
    tuple_list!(edges_observer, time_observer),
    &mut fuzzer,
    &mut state,
    &mut mgr,
)
.expect("Failed to create QemuExecutor");

// Wrap the executor to keep track of the timeout
let mut executor = TimeoutExecutor::new(executor, timeout);
```

The QEMU Executor

```rust
// Create a QEMU in-process executor
let executor = QemuExecutor::new(
    &mut harness,
    // The QEMU helpers define common hooks like coverage tracking hooks
    // In the specific, QemuEdgeCoverageHelper hooks every executed edge
    // for binary-only collision free edge coverage
    tuple_list!(QemuEdgeCoverageHelper::new()),
    tuple_list!(edges_observer, time_observer),
    &mut fuzzer,
    &mut state,
    &mut mgr,
)
.expect("Failed to create QemuExecutor");

// Wrap the executor to keep track of the timeout
let mut executor = TimeoutExecutor::new(executor, timeout);
```

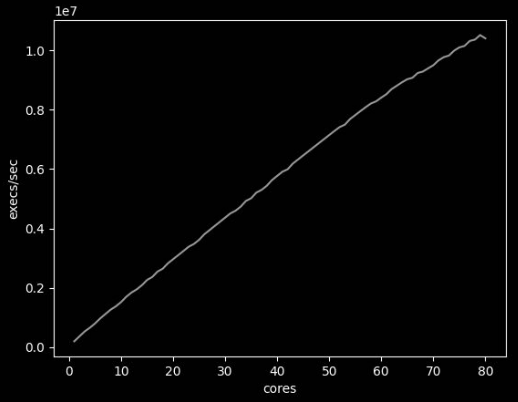
The last bits...

```
    //...

    if state.corpus().count() < 1 {
        state
            .load_initial_inputs(&mut fuzzer, &mut executor, &mut mgr, &corpus_dirs)
            .unwrap_or_else(|_| {
                println!("Failed to load initial corpus at {:?}", &corpus_dirs);
                process::exit(0);
            });
        println!("We imported {} inputs from disk.", state.corpus().count());
    }

    // Setup an havoc mutator with a mutational stage
    let mutator = StdScheduledMutator::new(havoc_mutations());
    let mut stages = tuple_list!(StdMutationalStage::new(mutator));

    fuzzer.fuzz_loop(&mut stages, &mut executor, &mut state, &mut mgr)?;
    Ok(())
}
```

```
$ cargo run --release ./libpng_harness
warning: CPU_TARGET is not set, default to x86_64
warning: Qemu not found, cloning with git (dd66ee9d0ec90221eb6476c63800f2671ad2a0c8)...
     Compiling qemu_launcher v0.6.1 (/home/andrea/Desktop/LibAFL/fuzzers/qemu_launcher)
      Finished release [optimized + debuginfo] target(s) in 27.84s
       Running `target/release/qemu_launcher ./libpng_harness`
LLVMFuzzerTestOneInput @ 0x400000277b
Break at 0x400000277b
Stack pointer = 0x4001832908
Return address = 0x400000314f
Placing input at 0x4001de2000
spawning on cores: [0, 1, 2, 3]
...
[/home/andrea/Desktop/LibAFL/libafl/src/bolts/llmp.rs:2191] "New connection" = "New connection"
[/home/andrea/Desktop/LibAFL/libafl/src/bolts/llmp.rs:2191] addr = 127.0.0.1:59192
[/home/andrea/Desktop/LibAFL/libafl/src/bolts/llmp.rs:2191] stream.peer_addr().unwrap() = 127.0.0.1:59192
[/home/andrea/Desktop/LibAFL/libafl/src/bolts/llmp.rs:2191] "New connection" = "New connection"
[/home/andrea/Desktop/LibAFL/libafl/src/bolts/llmp.rs:2191] addr = 127.0.0.1:59194
[/home/andrea/Desktop/LibAFL/libafl/src/bolts/llmp.rs:2191] stream.peer_addr().unwrap() = 127.0.0.1:59194
[/home/andrea/Desktop/LibAFL/libafl/src/bolts/llmp.rs:2191] "New connection" = "New connection"
[/home/andrea/Desktop/LibAFL/libafl/src/bolts/llmp.rs:2191] addr = 127.0.0.1:59196
[/home/andrea/Desktop/LibAFL/libafl/src/bolts/llmp.rs:2191] stream.peer_addr().unwrap() = 127.0.0.1:59196
[Stats       #3]  (GLOBAL) clients: 4, corpus: 0, objectives: 0, executions: 0, exec/sec: 0
                   (CLIENT) corpus: 0, objectives: 0, executions: 0, exec/sec: 0, edges: 752/752 (100%)
[Testcase    #3]  (GLOBAL) clients: 4, corpus: 1, objectives: 0, executions: 1, exec/sec: 0
                   (CLIENT) corpus: 1, objectives: 0, executions: 1, exec/sec: 0, edges: 752/752 (100%)
[Stats       #3]  (GLOBAL) clients: 4, corpus: 1, objectives: 0, executions: 1, exec/sec: 0
                   (CLIENT) corpus: 1, objectives: 0, executions: 1, exec/sec: 0, edges: 821/821 (100%)
[Testcase    #3]  (GLOBAL) clients: 4, corpus: 2, objectives: 0, executions: 2, exec/sec: 0
                   (CLIENT) corpus: 2, objectives: 0, executions: 2, exec/sec: 0, edges: 821/821 (100%)
```

Scaling

- Little Kernel Load
- Fast message passing
- libpng: > 10mio execs/s on 80 cores



A less verbose (QEMU) fuzzer

```
        american fuzzy lop ++2.65d (libpng_harness) [explore] {0}
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 43 sec       │  cycles done : 15    │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec        │  total paths : 703   │
│ last uniq crash : none seen yet                      │ uniq crashes : 0     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 261*1 (37.1%)     │    map density : 5.78% / 13.98%       │
│ paths timed out : 0 (0.00%)         │ count coverage : 3.30 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 14             │ favored paths : 114 (16.22%)          │
│ stage execs : 31/32 (96.88%)        │  new edges on : 167 (23.76%)          │
│ total execs : 2.55M                 │ total crashes : 0 (0 unique)          │
│  exec speed : 61.2k/sec             │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ───────┤
│   bit flips : n/a, n/a, n/a                         │    levels : 11        │
│  byte flips : n/a, n/a, n/a                         │   pending : 121       │
│ arithmetics : n/a, n/a, n/a                         │  pend fav : 0         │
│  known ints : n/a, n/a, n/a                         │ own finds : 699       │
│  dictionary : n/a, n/a, n/a                         │  imported : n/a       │
│havoc/splice : 506/1.05M, 193/1.44M                  │ stability : 99.88%    │
│   py/custom : 0/0, 0/0                              ├───────────────────────┘
│        trim : 19.25%/53.2k, n/a                     │          [cpu000: 12%]
└─────────────────────────────────────────────────────┘
```

```rust
libafl_sugar::QemuBytesCoverageSugar::builder()
    .input_dirs(&[PathBuf::from("./corpus")])
    .output_dir(PathBuf::from("./crashes"))
    .broker_port(1337)
    .cores(&[0, 1, 2, 3])
    .harness(|buf| {
        let mut len = buf.len();
        if len > 4096 {
            buf = &buf[0..4096];
            len = 4096;
        }

        emu::write_mem(input_addr, buf);
        emu::write_reg(Amd64Regs::Rdi, input_addr).unwrap();
        emu::write_reg(Amd64Regs::Rsi, len).unwrap();
        emu::write_reg(Amd64Regs::Rip, test_one_input_ptr).unwrap();
        emu::write_reg(Amd64Regs::Rsp, stack_ptr).unwrap();

        emu::run();
    })
    .build()
    .run();
```

Wait, so only Rust?

```python
from pylibafl import sugar, qemu

# ...

def harness(b):
    if len(b) > MAX_SIZE:
        b = b[:MAX_SIZE]
    qemu.write_mem(inp, b)
    qemu.write_reg(qemu.amd64.Rsi, len(b))
    qemu.write_reg(qemu.amd64.Rdi, inp)
    qemu.write_reg(qemu.amd64.Rsp, sp)
    qemu.write_reg(qemu.amd64.Rip, test_one_input)
    qemu.run()

fuzz = sugar.QemuBytesCoverageSugar(['./corpus'], './crashes',
                                    1337, [0,1,2,3])

fuzz.run(harness)
```

Wait, so only Rust?

You can already use LibAFL from python!

(And yes, you can also use it to fuzz python ;))

LibAFL is FOSS!

LibAFL is young but the community is growing

- 2 GSOC students this summer
- 26 contributors
- >500 stars on GitHub

https://github.com/AFLplusplus/LibAFL

LibAFL is FOSS!

LibAFL is young but the community is growing

- 2 GSOC students this summer
- 26 contributors
- >500 stars on GitHub

https://github.com/AFLplusplus/LibAFL

Fuzzers written in it already exist, for example https://github.com/tlspuffin/tlspuffin

Other reasons to use LibAFL

- Easy-to-use Mutators
- Platform independent Shared Map implementation (incl. Android)
- Serializable "AnyMap" implementation
- Low-level Message Passing: Scalable Many-to-many communication
- QEMU bindings

Conclusion

LibAFL is a library to build fuzzers

Example fuzzers outperform State-of-the-Art fuzzers in scalability and execution speed

Modern features like CmpLog, Grammar Fuzzing, Hybrid Fuzzing, Frida-Mode, Binary-Only ASAN

It's FOSS, with a growing community

Thanks y’all

Have a nice FuzzCon Europe