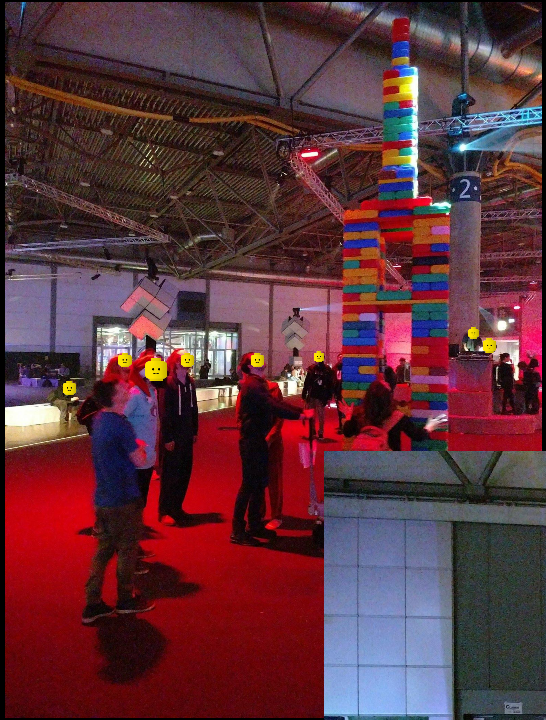
Fuzzers like LEGO

by Andrea Fioraldi & Dominik Maier

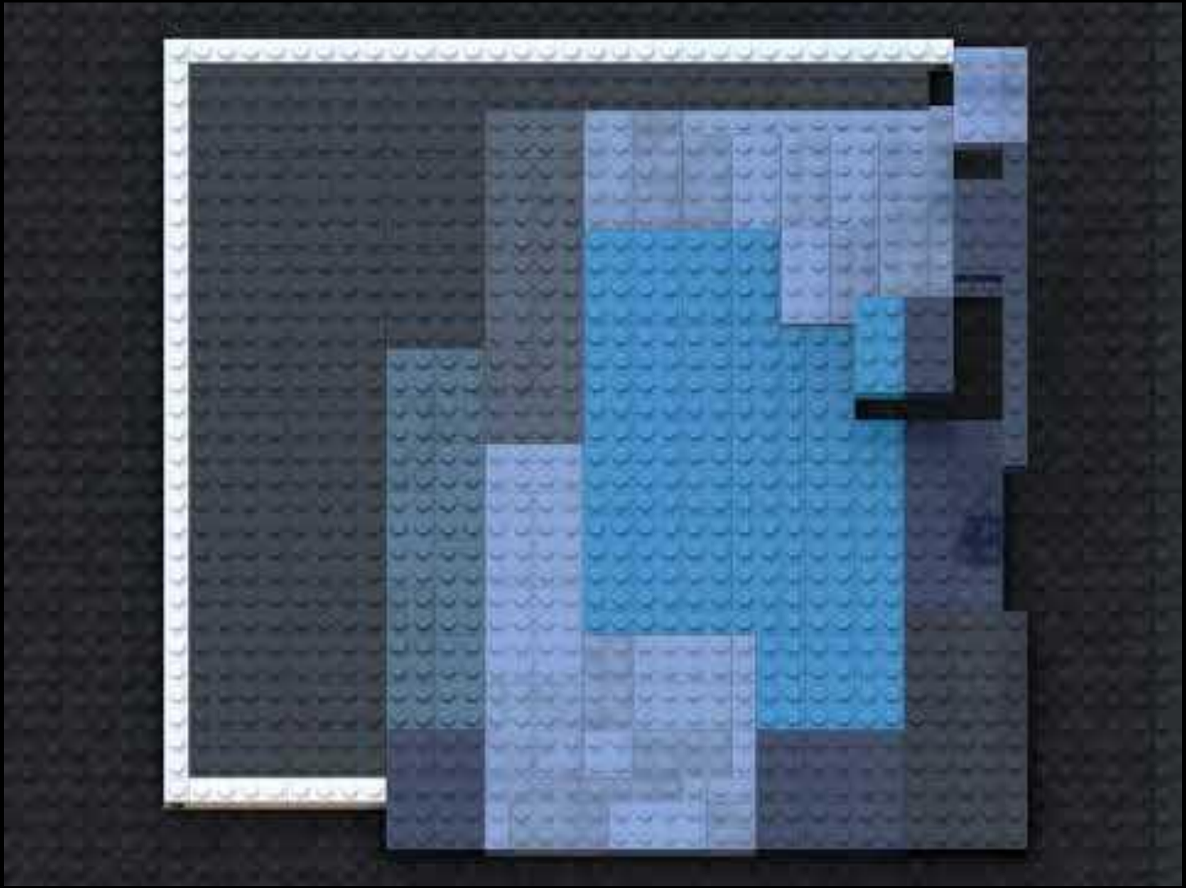
@andreafioraldi, @domenuk

{andrea, dominik}@aflplus.plus



AFL++

LEGO



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

AFL++ 3.0c released in Dec 2020

experiment summary

We show two different aggregate (cross-benchmark) rankings of fuzzers. The first is based on the average of per-benchmarks scores, where the score represents the percentage of the highest reached median coverage on a given benchmark (higher value is better). The second ranking shows the average rank of fuzzers, after we rank them on each benchmark according to their median reached covereges (lower value is better).

By avg. score

	average normalized score
fuzzer	
aflplusplus_300c	99.43
aflplusplus_268c	93.78
aflplusplus_300c_qemu	92.61
aflplusplus_268c_qemu	90.10

By avg. rank

	average rank
fuzzer	
aflplusplus_300c	1.71
aflplusplus_268c	2.48
aflplusplus_300c_qemu	2.64
aflplusplus_268c_qemu	3.17

AFL++ 3.0c released in Dec 2020

experiment summary

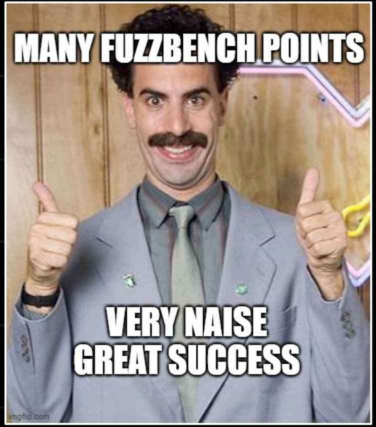
We show two different aggregate (cross-benchmark) rankings of fuzzers. The first is based on the average of per-benchmarks scores, where the score represents the percentage of the highest reached median coverage on a given benchmark (higher value is better). The second ranking shows the average rank of fuzzers, after we rank them on each benchmark according to their median reached covereges (lower value is better).

By avg. score

	average normalized score
fuzzer	
aflplusplus_300c	99.43
aflplusplus_268c	93.78
aflplusplus_300c_qemu	92.61
aflplusplus_268c_qemu	90.10

By avg. rank

	average rank
fuzzer	
aflplusplus_300c	1.71
aflplusplus_268c	2.48
aflplusplus_300c_qemu	2.64
aflplusplus_268c_qemu	3.17



The Truth(™) About Fuzz Testing

The best fuzzer is…

The Truth(™) About Fuzz Testing

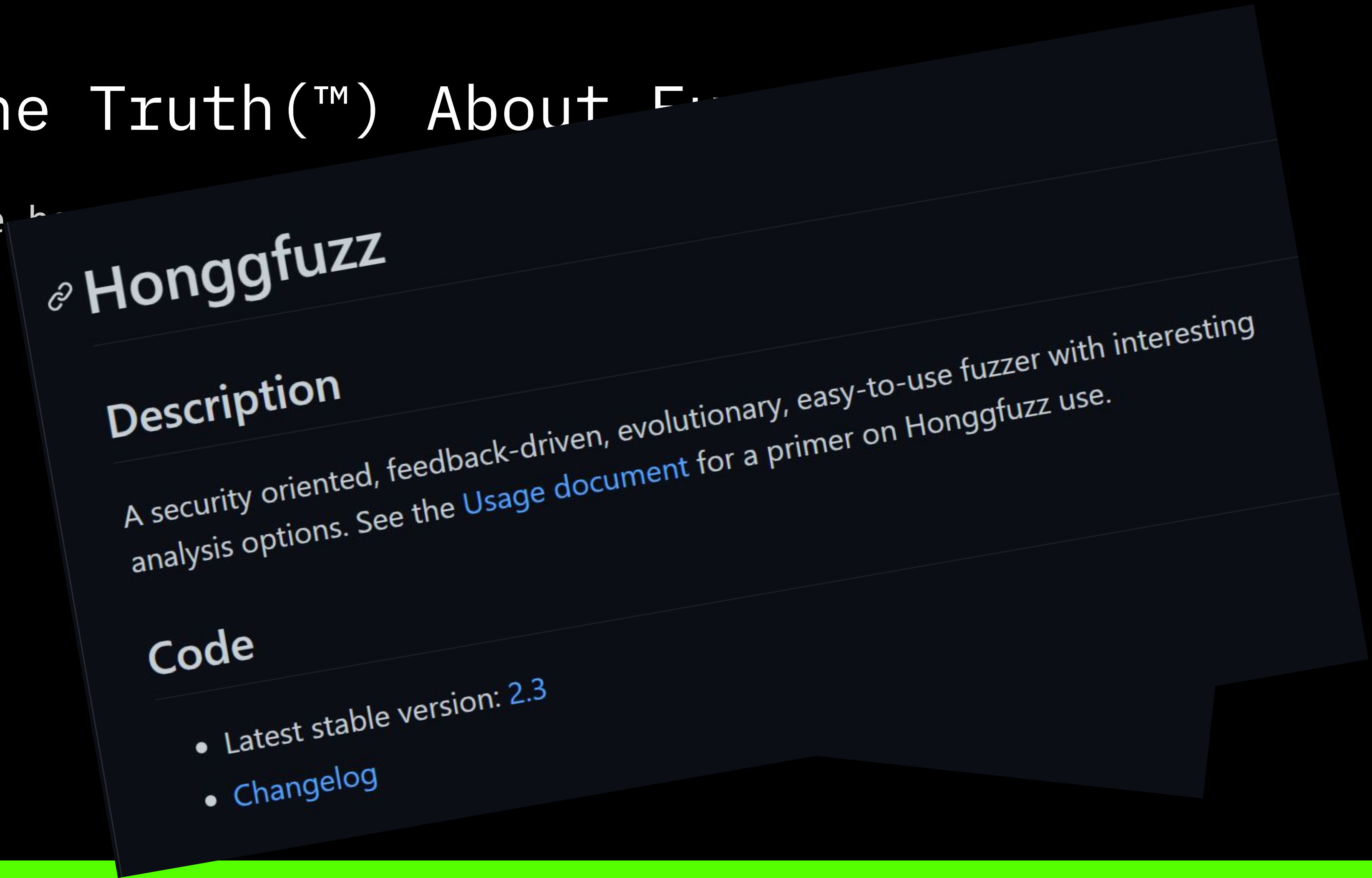
The best fuzzer is…

...

The Truth(™) About

The

...



Honggfuzz

Description

A security oriented, feedback-driven, evolutionary, easy-to-use fuzzer with interesting analysis options. See the Usage document for a primer on Honggfuzz use.

Code

- Latest stable version: 2.3
- Changelog

Frida API Fuzzer

> v1.4 Copyright (C) 2020 Andrea Fioraldi andreafioraldi@gmail.com
>
> Released under the Apache License v2.0

This experimental fuzzer is meant to be used for API in-memory fuzzing.

The design is highly inspired and based on AFL/AFL++.

ATM the mutator is quite simple, just the AFL's havoc and splice stages.

- Latest
- Changelog

Frida API Fuzz

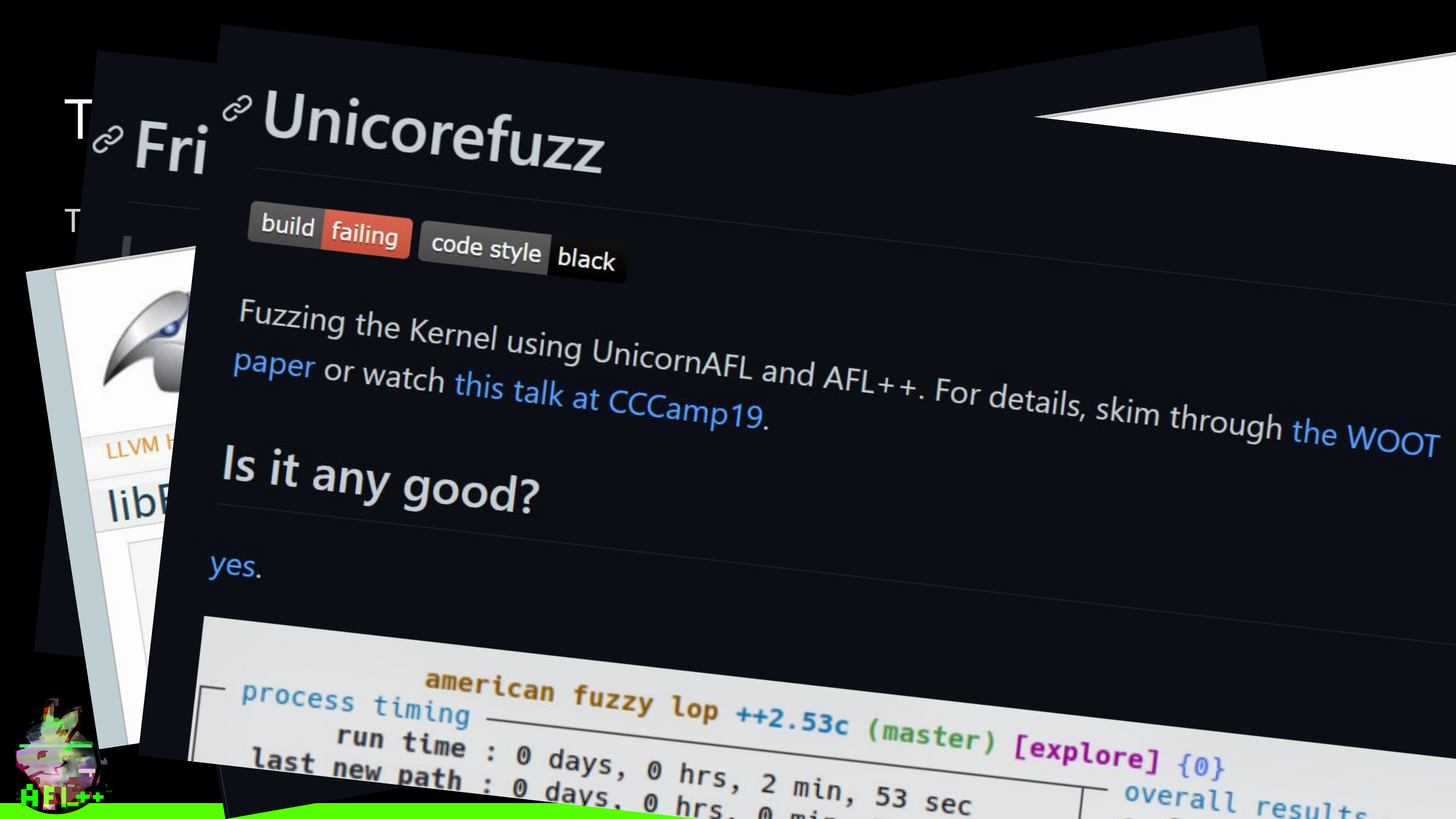
LLVM COMPILER INFRASTRUCTURE

libFuzzer – a library for coverage-guided fuzz testing.

AFL's havoc and splice stages.

- Changelog

AFL++


Fri
Unicorefuzz
build failing
code style black
Fuzzing the Kernel using UnicornAFL and AFL++. For details, skim through the WOOT paper or watch this talk at CCCamp19.
Is it any good?
yes.
LLVM
lib
american fuzzy lop ++2.53c (master) [explore] {0}
process timing
run time : 0 days, 0 hrs, 2 min, 53 sec
last new path : 0 days, 0 hrs,
overall results
AFL++

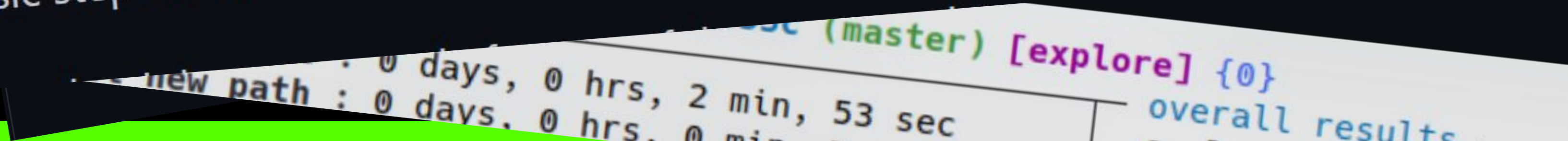
Fuzzilli

A (coverage-)guided fuzzer for dynamic language interpreters based on a custom intermediate language ("FuzzIL") which can be mutated and translated to JavaScript.

Written and maintained by Samuel Groß, saelo@google.com.

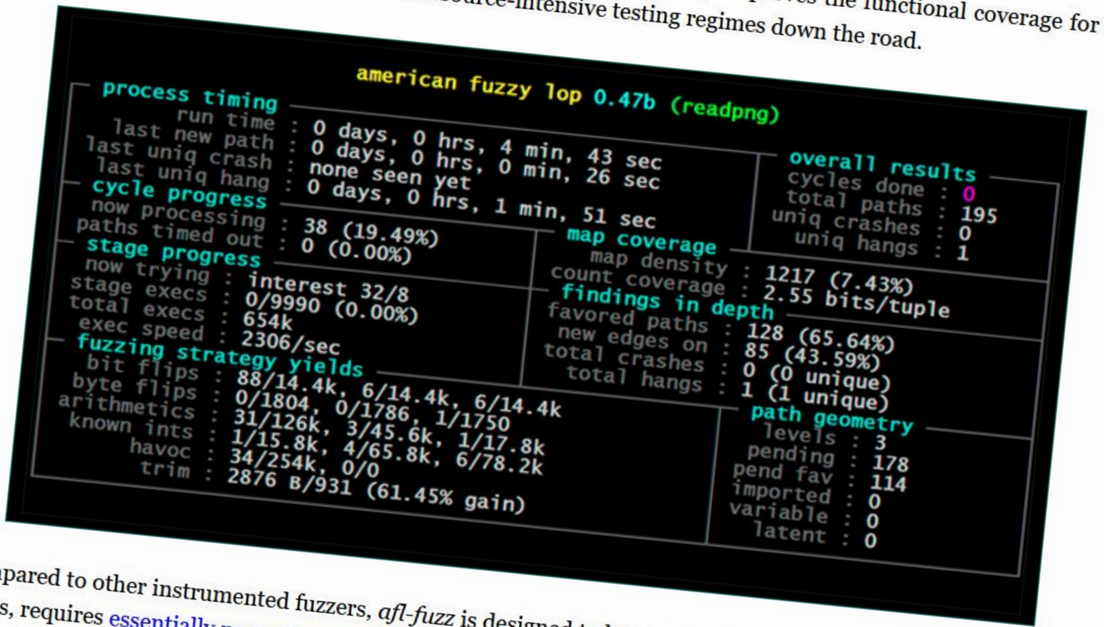
Usage

The basic steps to use this fuzzer are:



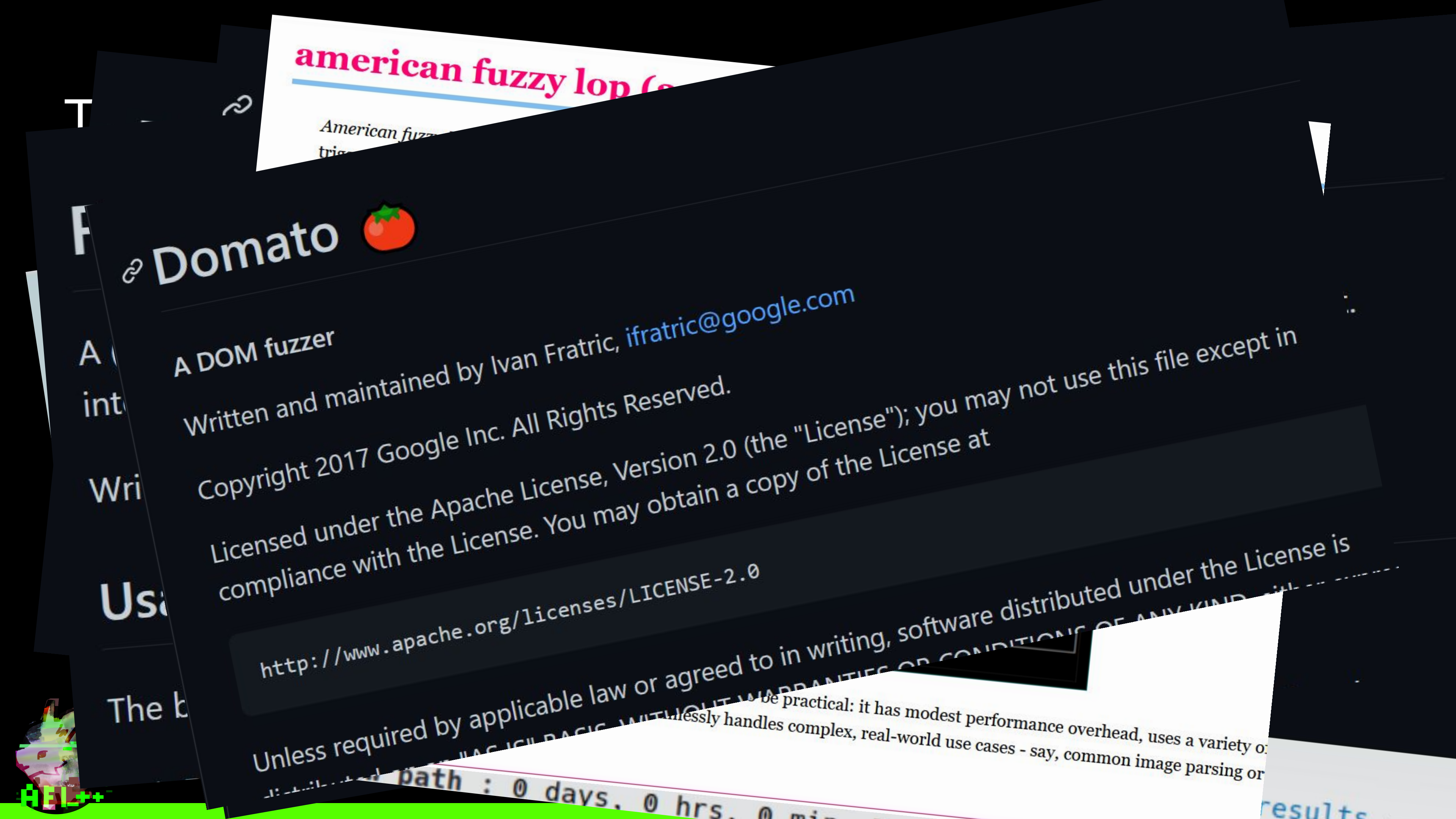
american fuzzy lop (2.52b)

American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms
trigger new internal states in the targeted binary. This substantially improves the functional coverage for the fuzzed code. The c
useful for seeding other, more labor- or resource-intensive testing regimes down the road.



Compared to other instrumented fuzzers, *afl-fuzz* is designed to be practical: it has modest performance overhead, uses a variety o
tricks, requires essentially no configuration, and seamlessly handles complex, real-world use cases - say, common image parsing or

The "sales pitch"


american fuzzy lop
Domato
A DOM fuzzer
Written and maintained by Ivan Fratric, ifratric@google.com
Copyright 2017 Google Inc. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software distributed under the License is
be practical: it has modest performance overhead, uses a variety o
handles complex, real-world use cases - say, common image parsing or
path : 0 days, 0 hrs,
AFL++

Jackalope

```
Copyright 2020 Google LLC

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

What is Jackalope

http://www.apa

Unless required by applicable law or

american fuzzy lop

Domat

A DOM f

Writter

Copy

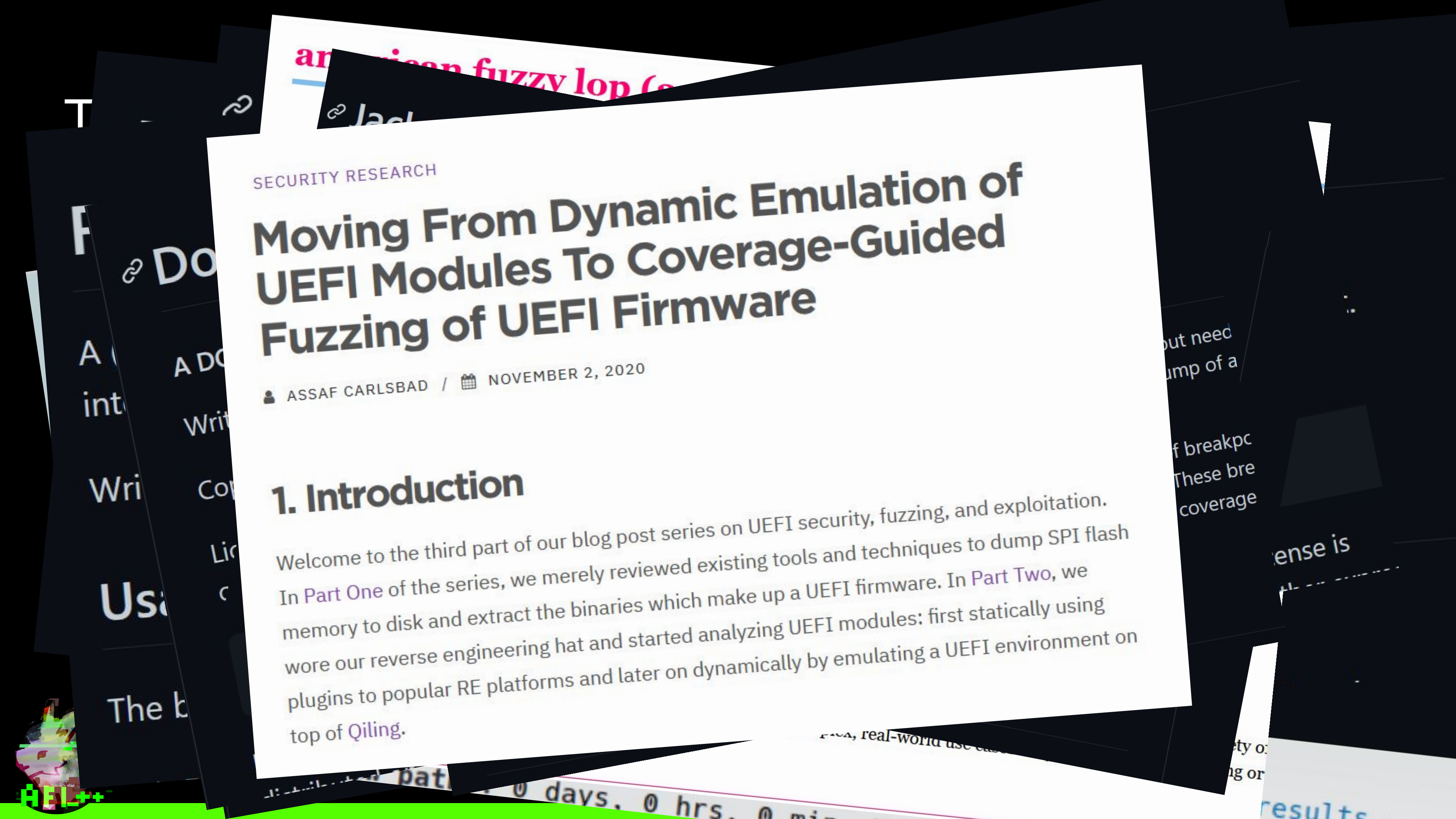
Lic

lessly handles complex, real-world use ca

path : 0 days, 0 hrs, 0 mi

cense is

results



SECURITY RESEARCH

Moving From Dynamic Emulation of UEFI Modules To Coverage-Guided Fuzzing of UEFI Firmware

ASSAF CARLSBAD / NOVEMBER 2, 2020

1. Introduction

Welcome to the third part of our blog post series on UEFI security, fuzzing, and exploitation. In Part One of the series, we merely reviewed existing tools and techniques to dump SPI flash memory to disk and extract the binaries which make up a UEFI firmware. In Part Two, we wore our reverse engineering hat and started analyzing UEFI modules: first statically using plugins to popular RE platforms and later on dynamically by emulating a UEFI environment on top of Qiling.

README.md

Qiling Unicornafl

"Load the cake and fuzz it too"

Fuzz pretty much anything...

Just read on the frontpage what qiling can do, then think you can fuzz all of this with code coverage.

```
american fuzzy lop ++2.57d (python3) [explore] {0}
process timing
       run time : 0 days, 0 hrs, 1 min, 9 sec
  last new path : 0 days, 0 hrs, 0 min, 59 sec
last uniq crash : 0 days, 0 hrs, 0 min, 15 sec
 last uniq hang : none seen yet
cycle progress
 now processing : 0.0 (0.0%)
overall results
  cycles done : 0
  total paths : 11
 uniq crashes : 13
   uniq hangs : 0
map coverage
  map density : 0.94% / 1.00%
```

Intro

`fzero` is a grammar-based fuzzer that generates a Rust application inspired by the paper "Fuzzers" by Rahul Gopinath and Andreas Zeller. https://arxiv.org/pdf/1911.07707.pdf

You can find the F1 fuzzer here:

https://github.com/vrthra/F1

The Truth(™) About Fuzz Testing

The best fuzzer is…

...

The Truth(™) About Fuzz Testing

The best fuzzer is…

your custom fuzzer

tuned for your specific use case & target

adapted to your specific needs

with custom mutations and concepts, ...

How to Create a Fuzzer Then?

- Fork an existing fuzzer (the n-th AFL-something)
- Create a custom fuzzer from scratch

Custom Fuzzer Engineering Issues

- Lack of code reuse, you will have to spend a lot of time in adapting different techniques from different fuzzers

Custom Fuzzer Engineering Issues

- Lack of code reuse, you will have to spend a lot of time in adapting different techniques from different fuzzers
- Reinventing the wheel, you will code the same code to do that same thing that all others do again and again

Custom Fuzzer Engineering Issues

- Lack of code reuse, you will have to spend a lot of time in adapting different techniques from different fuzzers
- Reinventing the wheel, you will code the same code to do that same thing that all others do again and again
- Naive design, typically just a mutator

Custom Fuzzer Engineering Issues

- Lack of code reuse, you will have to spend a lot of time in adapting different techniques from different fuzzers
- Reinventing the wheel, you will code the same code to do that same thing that all others do again and again
- Naive design, typically just a mutator
- Scaling, you cannot adapt it easily to multi-core or -machine

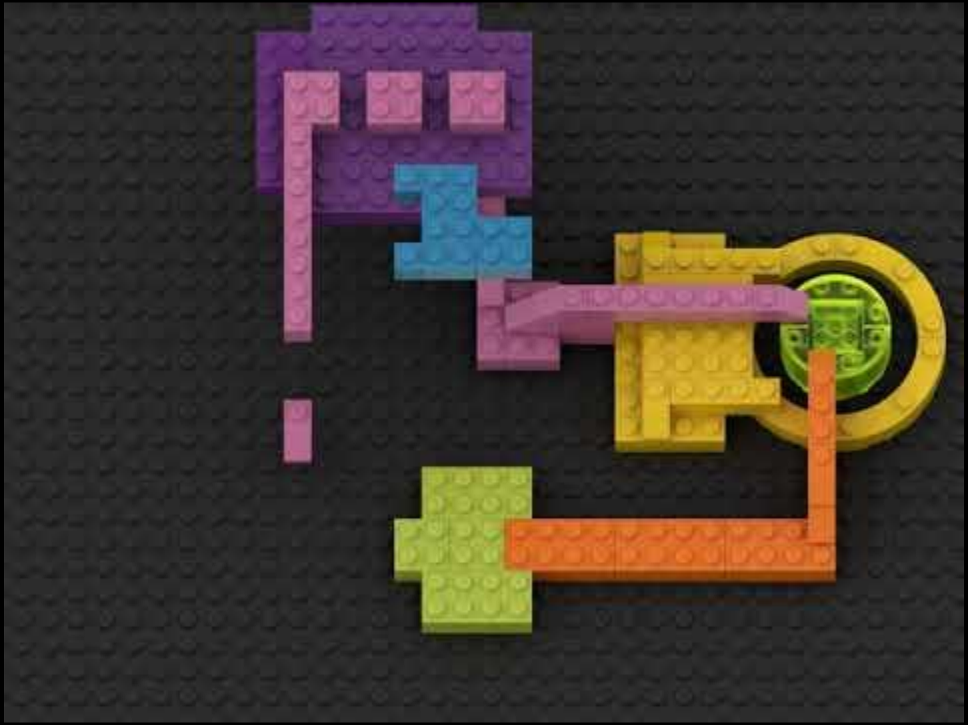
Our Solution: A Fuzzing Library

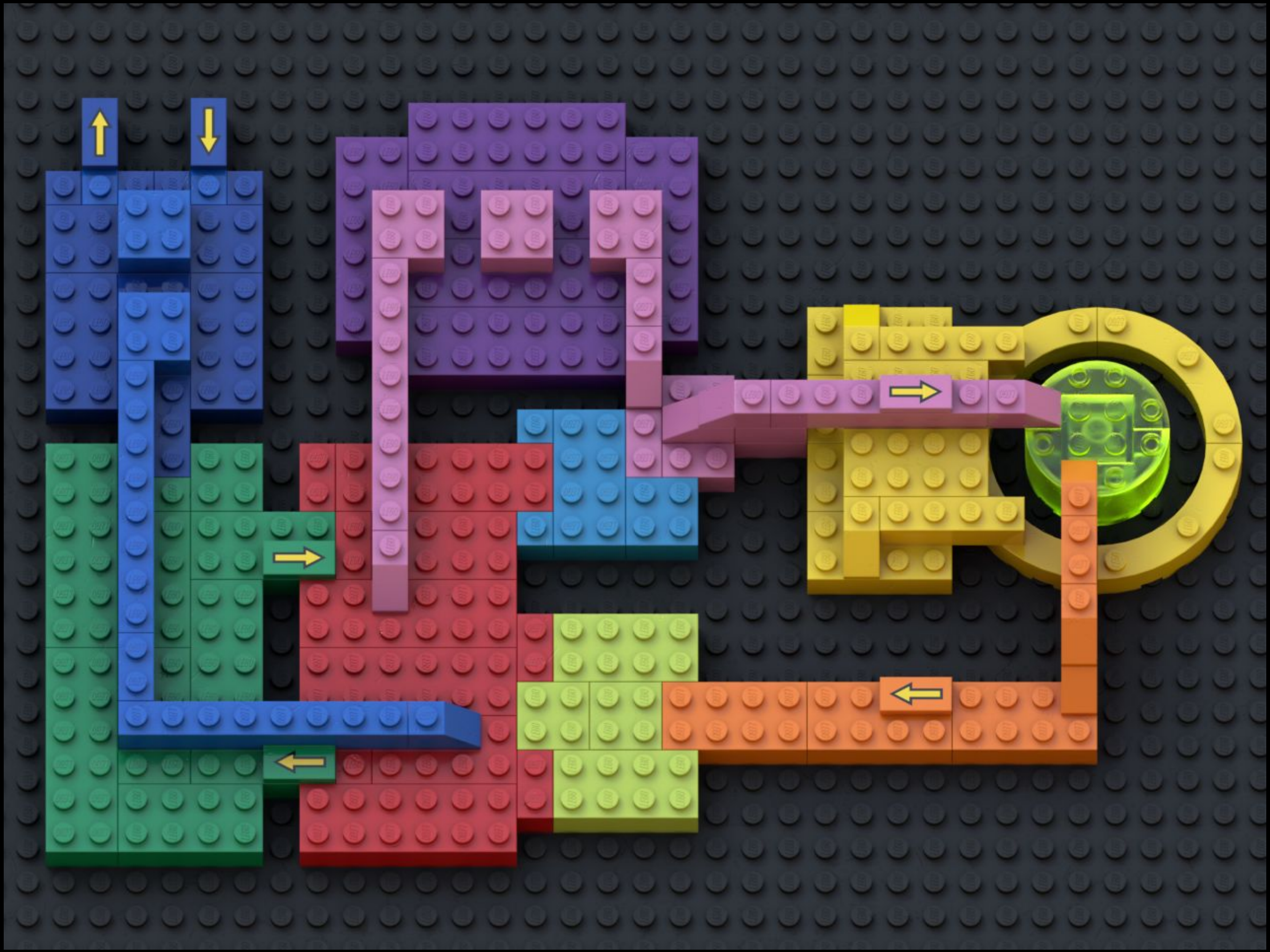
We aim to build a library that can be used to develop custom fuzzers quickly and reusing even complex techniques easily.

Think about Tensorflow or LLVM, but for fuzzers.

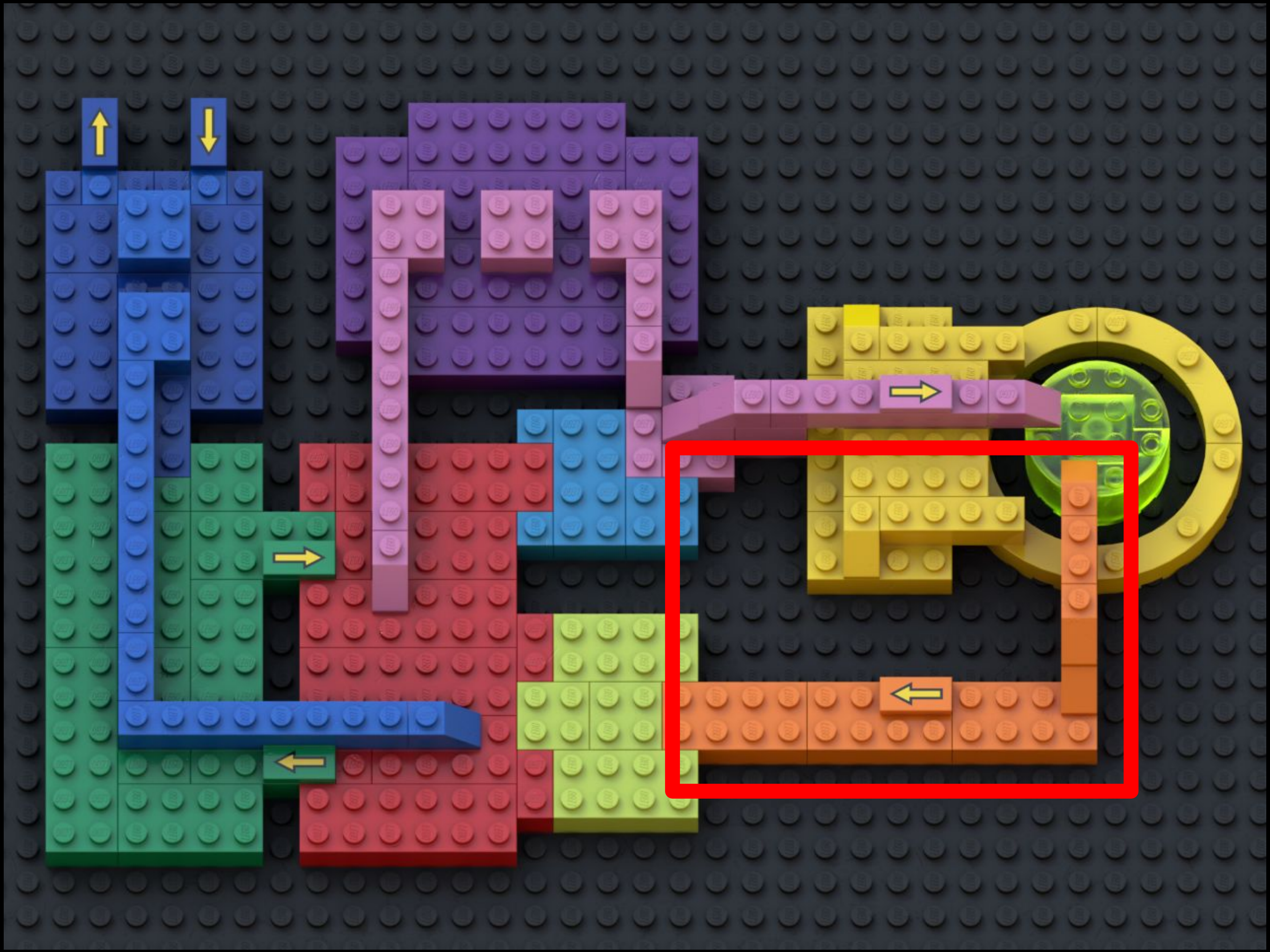
This Talk

- We present concepts that abstract properties of fuzzers
- We give some examples
- We translate them to code
- The community profits





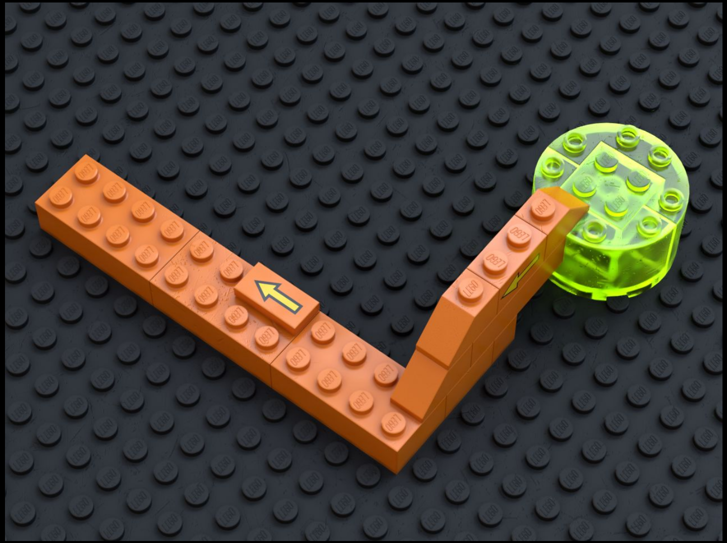
AFL++



Observer

Provides information about some properties of a target run.

Rerunning the target with the same input will (usually*) yield the same Obeserver state.



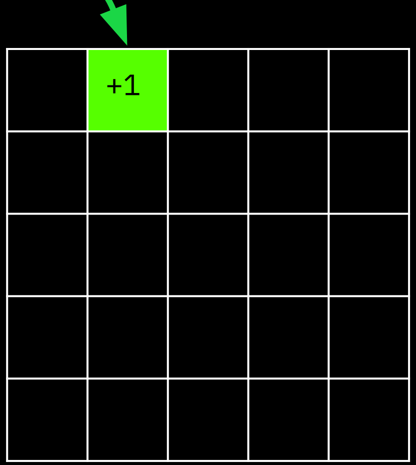
* Some impure observers alter the target to work, for example breakpoint-based coverage

Observer: Coverage Map

AFL-like shared memory that increases a counter at the hashed position of each edge

`hash_edge(0x41414141, 0x42424242)`

`0x41414141: jmp 0x42424242`



Observer: Execution Time

The time needed to execute the testcase.

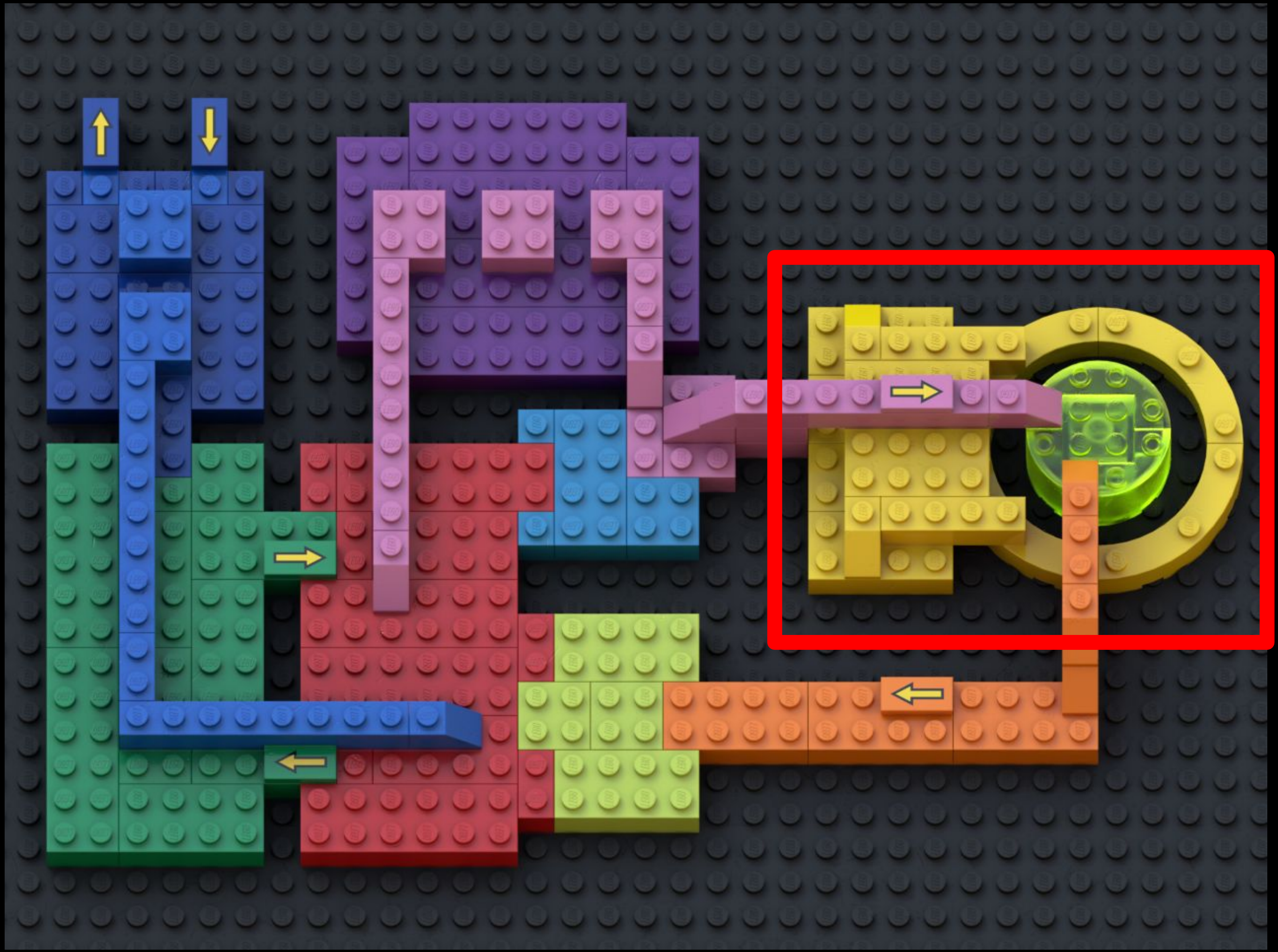
```
let start_time = get_cur_time();
run_target();
let elapsed_time = get_cur_time() - start_time;
```

Observer: Reachability

Sets booleans for each interesting point reached in the target

We can use annotations in the source code like:

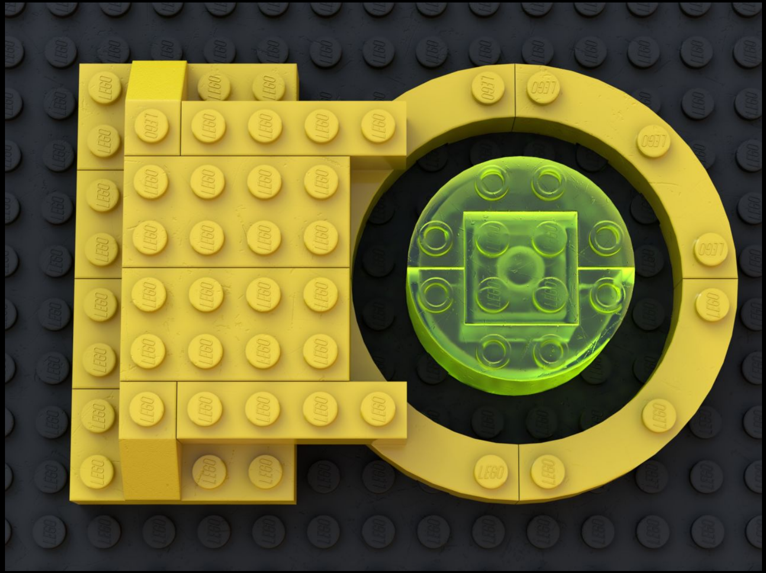
```
void func() {
    ...
    FUZZER_TARGET_POINT();
}
```



Executor

The Executor runs the target.
By its nature, the choice of executor is target-specific.

The executor associates target with observation channels.



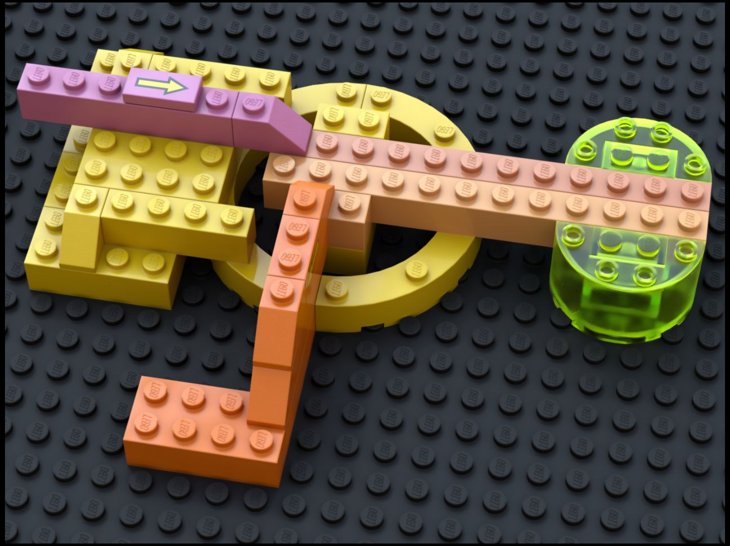
Executor: In Memory

An In-Memory executor simply calls a function or harness in the linked target for each run. Known from Libfuzzer.

```
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    ...
}
```

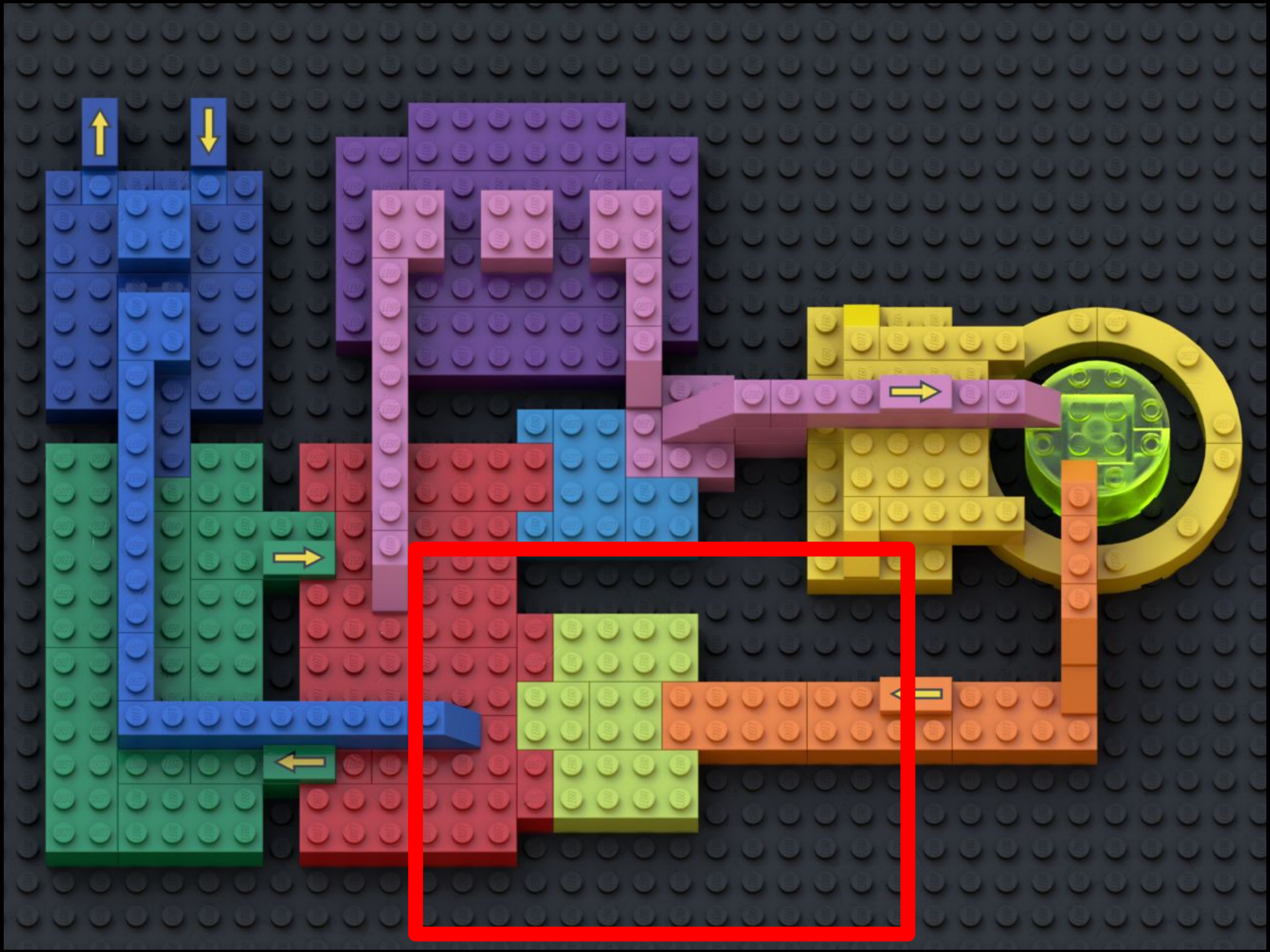
Executor: Forkserver

Another executor is the AFL forkserver. The target runs as (forked) process outside of the actual executor, communicating via pipes, shared maps, files.



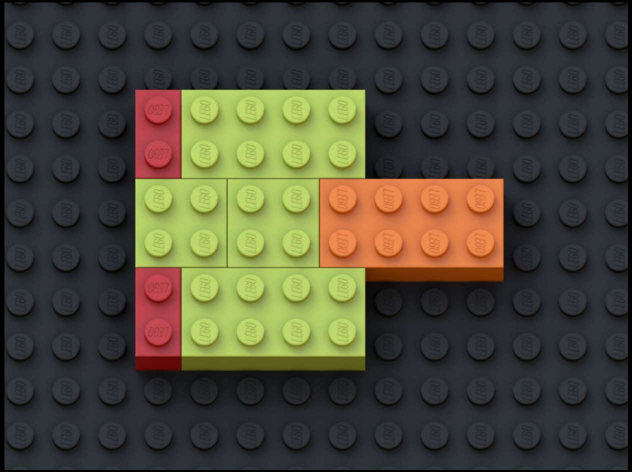
Executor: Forkserver

The AFL-like forkserver executor uses pipes to control the external target.



Feedback

The Feedback reduces the state of the observation channels after a run to an “is_interesting” fitness. The rates given by all the feedbacks are then used to decide if the input is worth keeping.



Feedback: Maximization Map

Feedback that tries to maximize the map entries. It needs a map Observer, and an internal map to keeps track of the maximum values seen so far.

Feedback: Maximization Map (Coverage)

Example: coverage map of AFL

⇒ AFL-like coverage observer + max map feedback = <3

Feedback: Maximization Map (Coverage)

Example: coverage map of AFL

```
let mut fitness = 0;
for i in 0..map_size {
    if observer_map[i] > history_map[i] {
        history_map[i] = observer_map[i];
        fitness += 1;
    }
}
```

Feedback: Maximization Map (Allocs)

Anther usage: maximization of allocation sizes to spot out-of-memory bugs. => same feedback, different observer.

```
void* malloc(size_t size) {
    FUZZER_ALLOC_REPORT(__builtin_return_address(0), size);
    return real_malloc(size);
}
```

Objective Feedback

A normal feedback tells the fuzzer:

"this is interesting, add to the corpus, mutate here!"

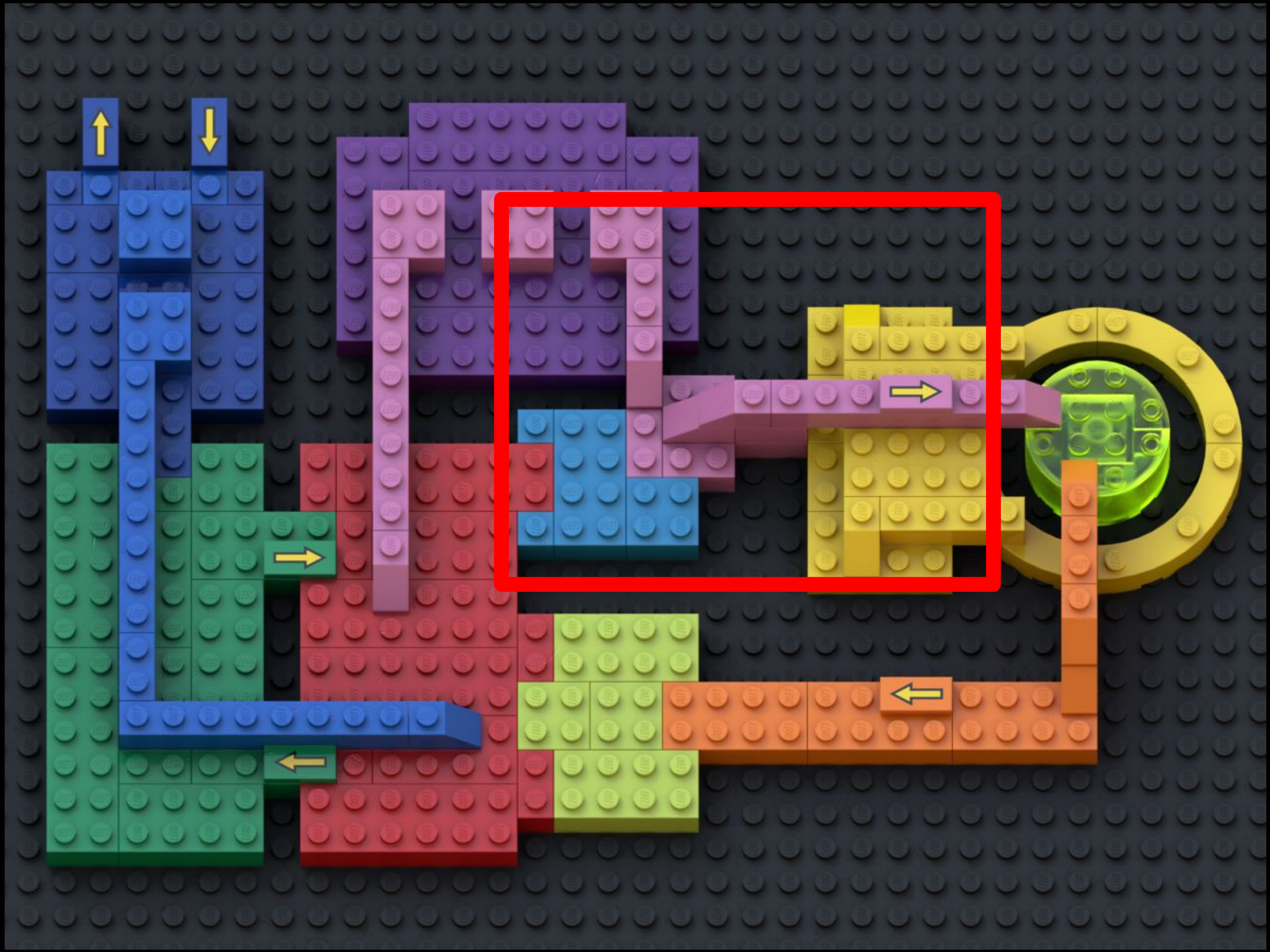
In contrast, the Objective decides if an input satisfies some objective for this run.

Example: finding crashing inputs.

Objective Feedback: Reachability

Another objective: reach a specific program point (using the reachability observation channel).

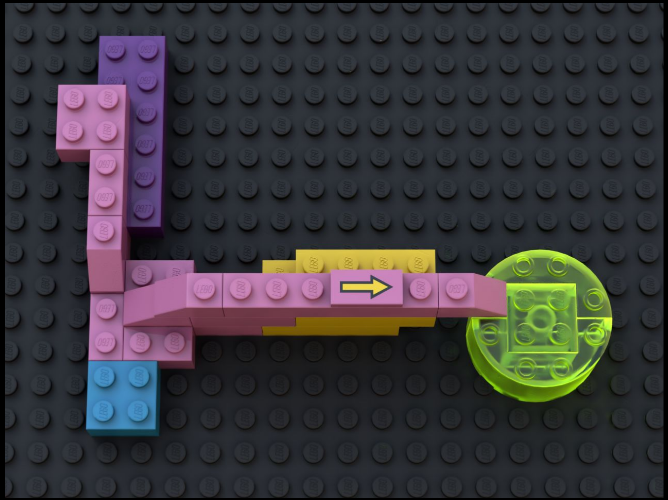
⇒ Inputs that make the program covering that point are added to the objective corpus.



AFL++

Input

An Input is the entity that represents the testcase. It doesn’t have to be in the same format expected by the program, but rather is in a structure that can be easily manipulated.



Input: Abstract Syntax Tree

An example of complex input is an AST, a structure that can be easily handled by a mutator event if the target expect a bytes array

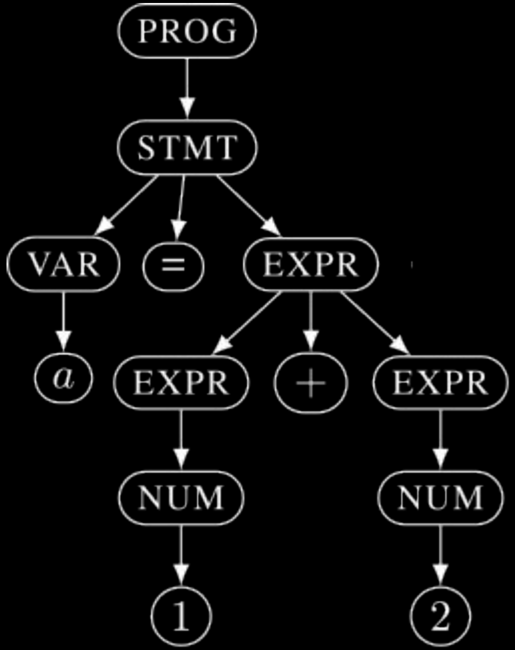
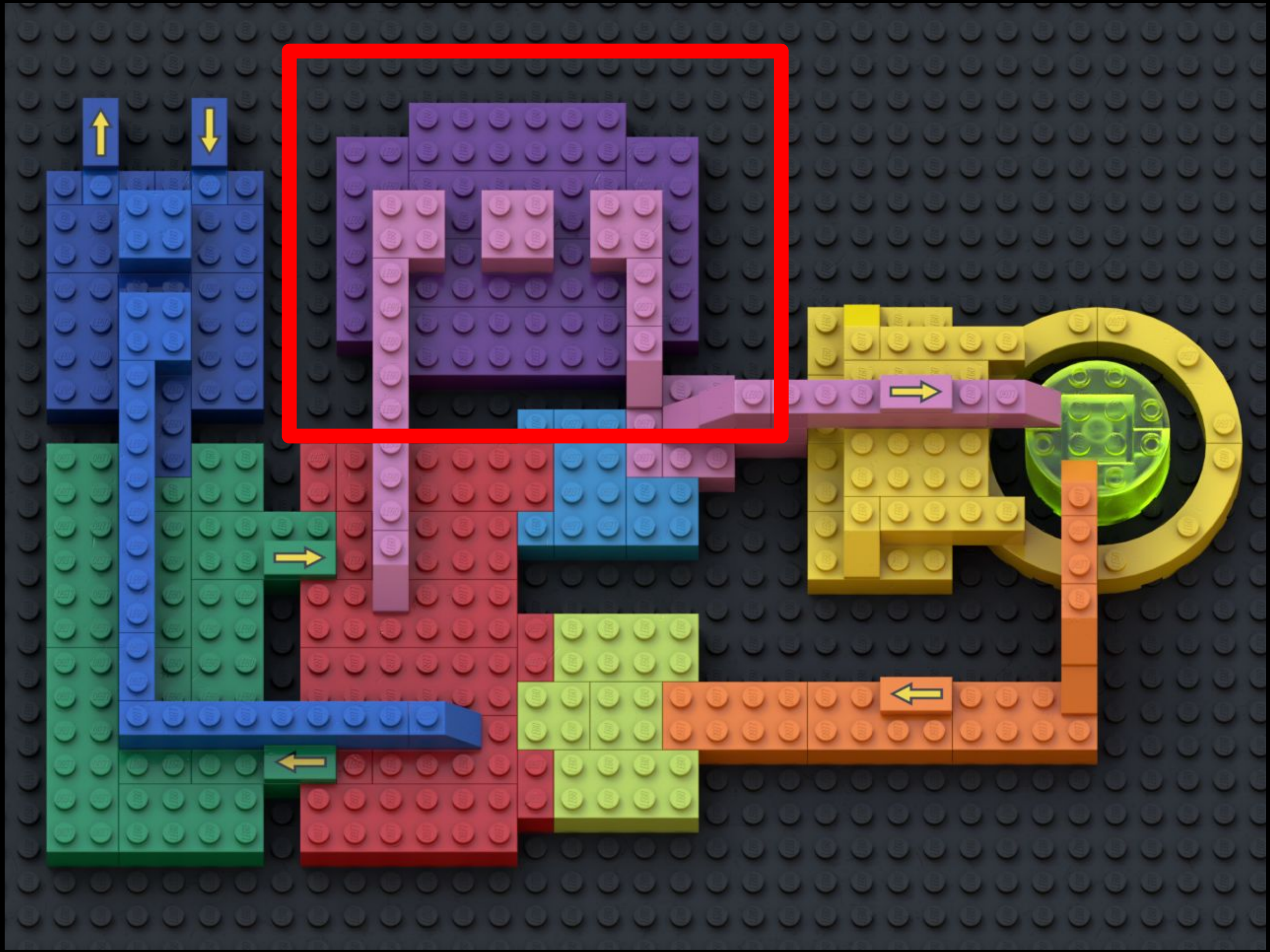


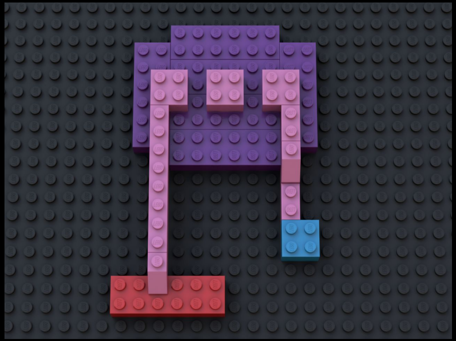
Image from https://github.com/RUB-SysSec/nautilus



AFL++

Corpus

The Corpus is the entity that collects testcases that are interesting for one or more feedbacks, defines how they are related to each other and how to feed the fuzzer with those inputs when requested. The items stored in the corpus are not only the inputs but also the Metadata like execution time.



Corpus: Random Corpus

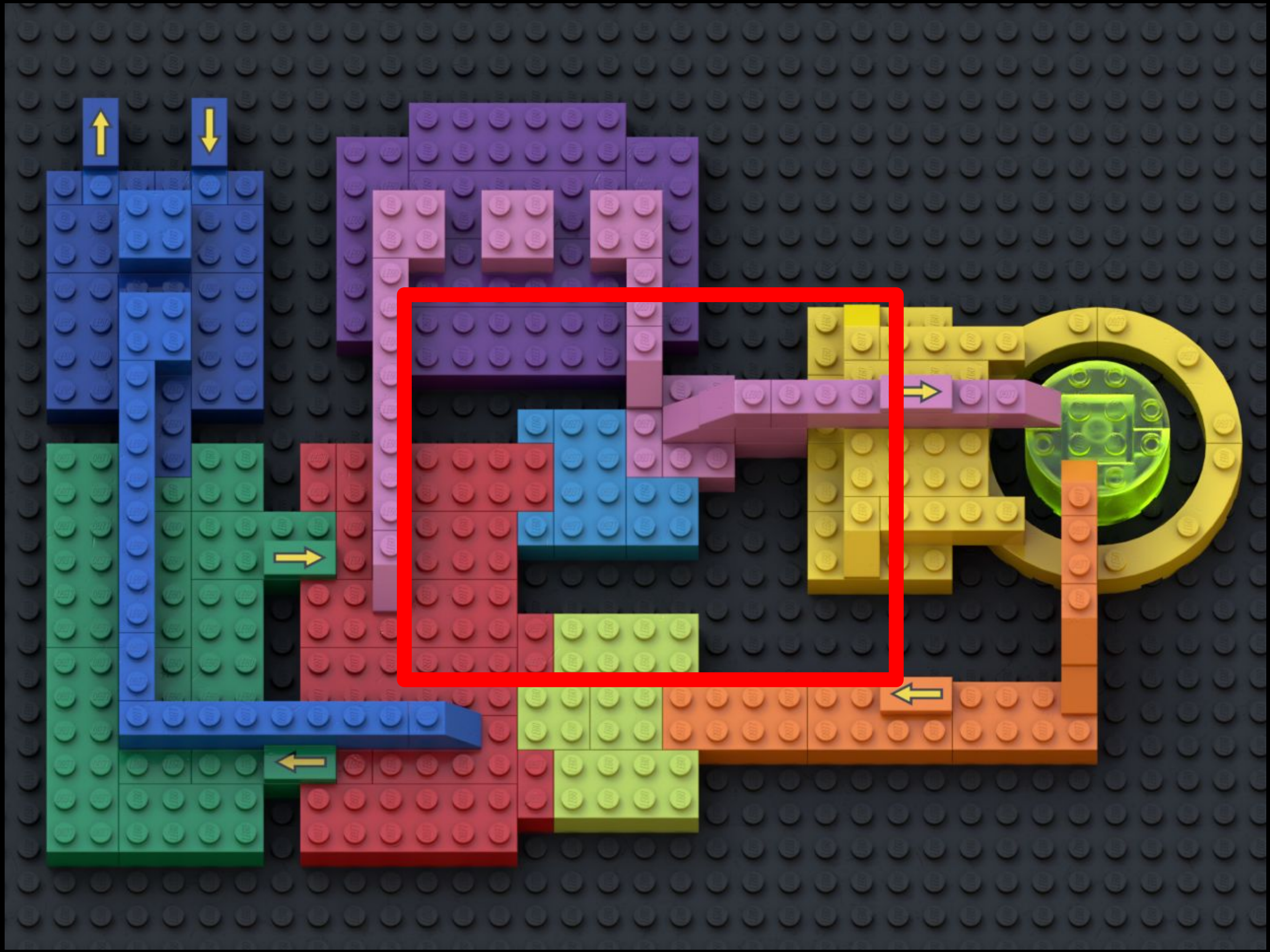
A naive corpus can be just a vector that provides a random entry to the fuzzer when requested.

```
fn get(&self, rand: Random) -> &Testcase {
    self.testcases[rand.below(self.testcases.len())]
}
```

Corpus: Queue Corpus

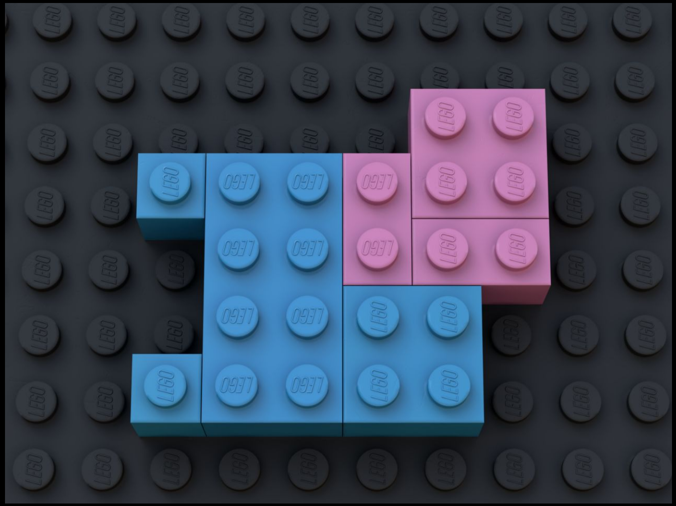
Another example of Corpus is a queue (AFL for instance).

```
fn get(&mut self) -> &Testcase {
    let t = self.testcases[self.pos];
    self.pos = (self.pos +1) % self.testcases.len();
    t
}
```



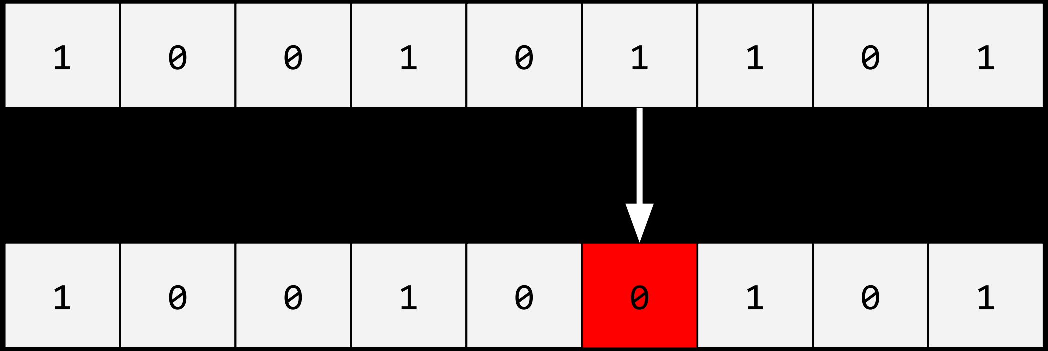
Mutator

A Mutator is an entity that takes one or more inputs and generates a new derived one.



Mutator: Bitflip Mutator

This simple mutator just flip a bit in the input.



Mutator: Scheduled Mutator

- Applies a set of mutations.
- The number and kind of mutations is decided by a scheduler.
- In the old-skool AFL Havoc mutator, the number of mutations is a bounded random number, the chosen mutations are random.
- More advanced solutions like MOpt employ scheduling algorithms.

Generator

A Generator generates new inputs from scratch, according to individual parameters.

⇒ Initial corpus, or part of a mutator.

Generator: Random Array

Initial testcases can be generated simply as random bytes array, or almost random following some rules (e.g. just printable bytes).

Generator: Grammar Generator

A generator using a grammar specification creates valid inputs from scratch. In Nautilus, it is used also as component of the mutator as one of the possible mutations is subtree generation.

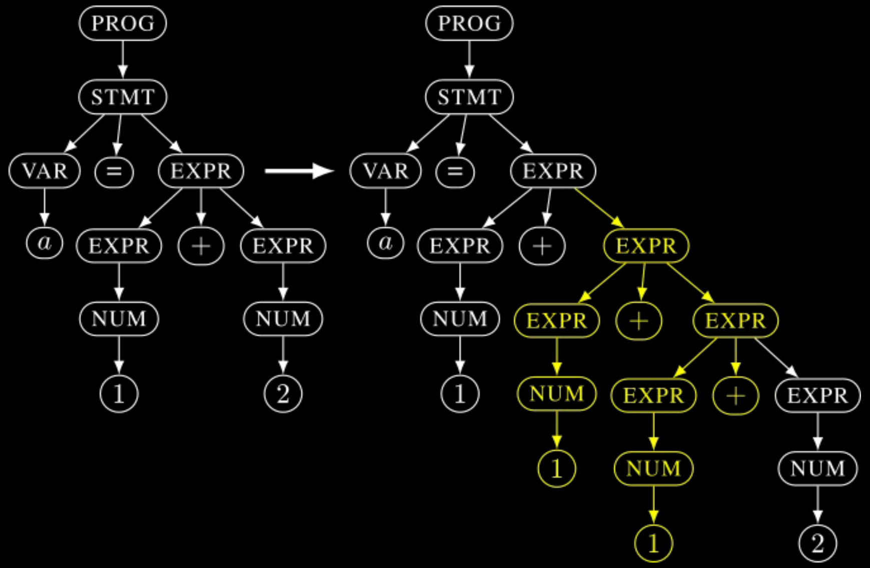
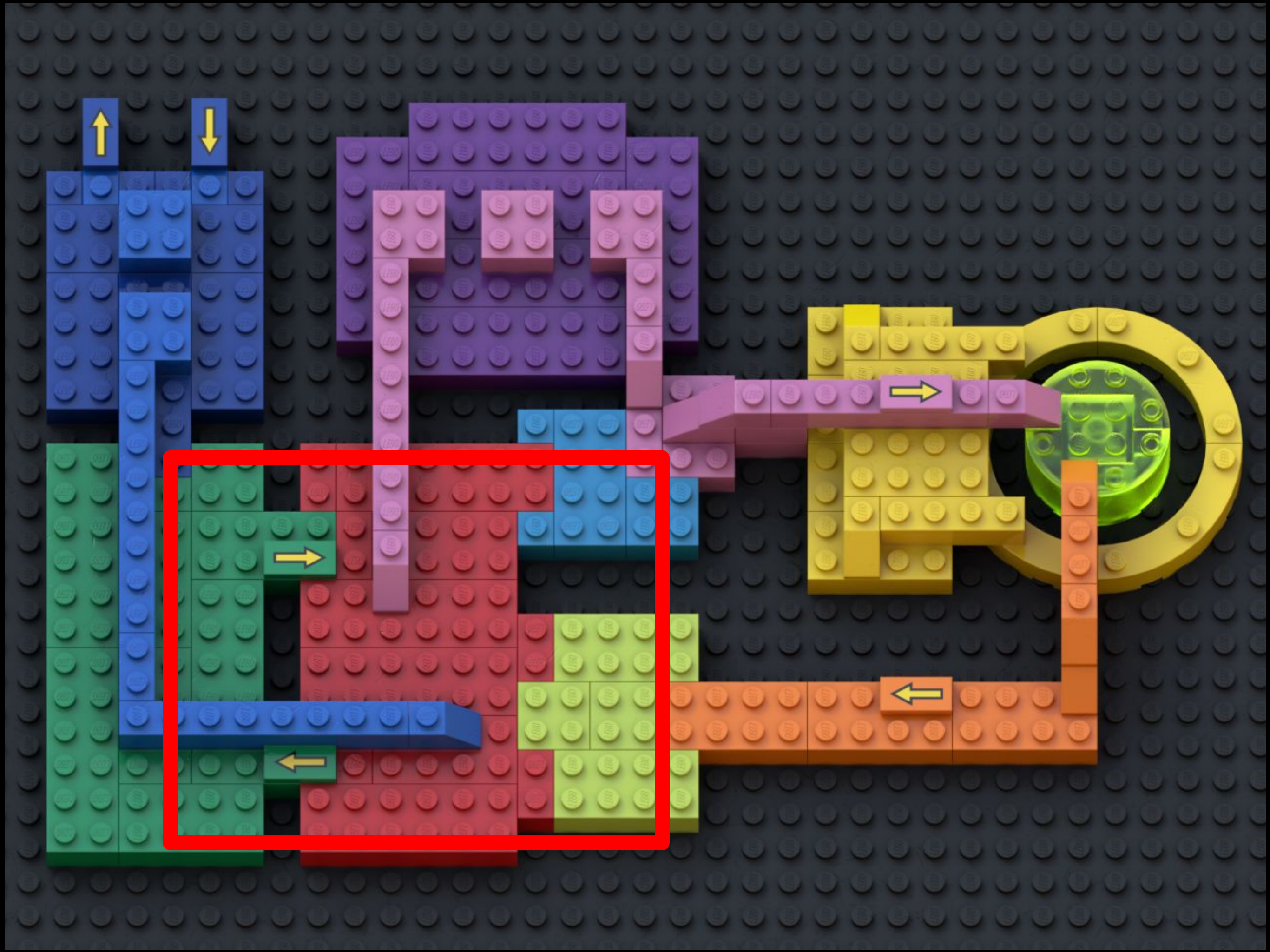
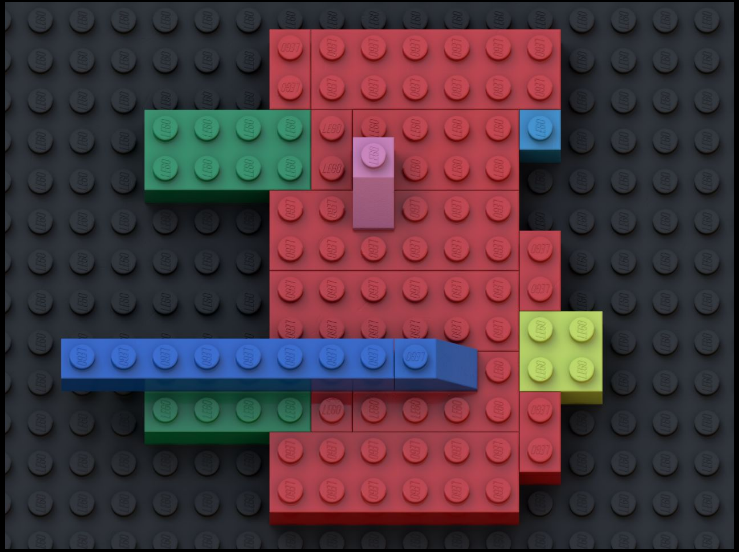


Image from https://github.com/RUB-SysSec/nautilus



Stage

A Stage is an entity that operates some actions on a single testcase.



Stage: Mutational Stage

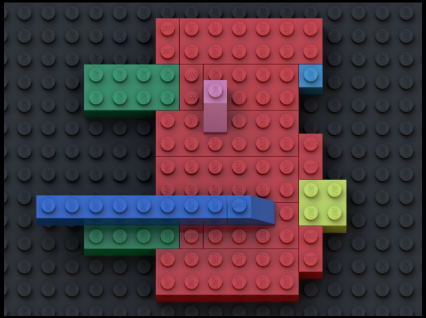
- Evaluates the generated input several times in a loop
- Num Iterations can be controlled with a scheduling algorithm.

=> Havoc stage in AFL applies the Havoc mutator to the current queue entry several times, the number of iteration depends on the perf score.

⇒ Fuzzers like AFLFast employ different algorithms to compute this perf score.

Stage: Analysis Stage

An analysis stage runs the input with an executor that performs taint tracking. The information extracted is then attached to the testcase as Metadata.



+

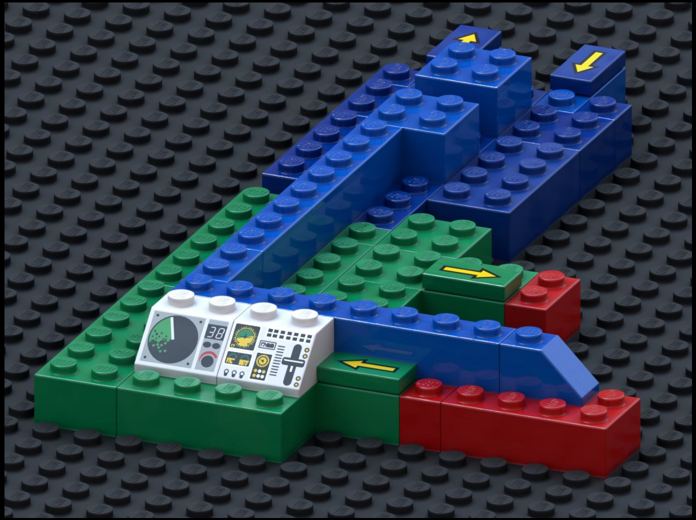
Stage: Trim Stage

Trimming stage of AFL reduces the size of inputs while maintaining the same coverage.

Additional Components

Beside these theoretical entities in Feedback-driven Fuzzing, a modern framework needs more components to glue all the blocks.

RNG, EventMgr, State, ...?



Random Number Generator

As fuzzing is a technique derived from random testing, the generation is random numbers is an important matter. We choose to abstract the implementation to of the PRNG to allow the user to pick the best for their needs, from the faster to the one with more soundness.

State

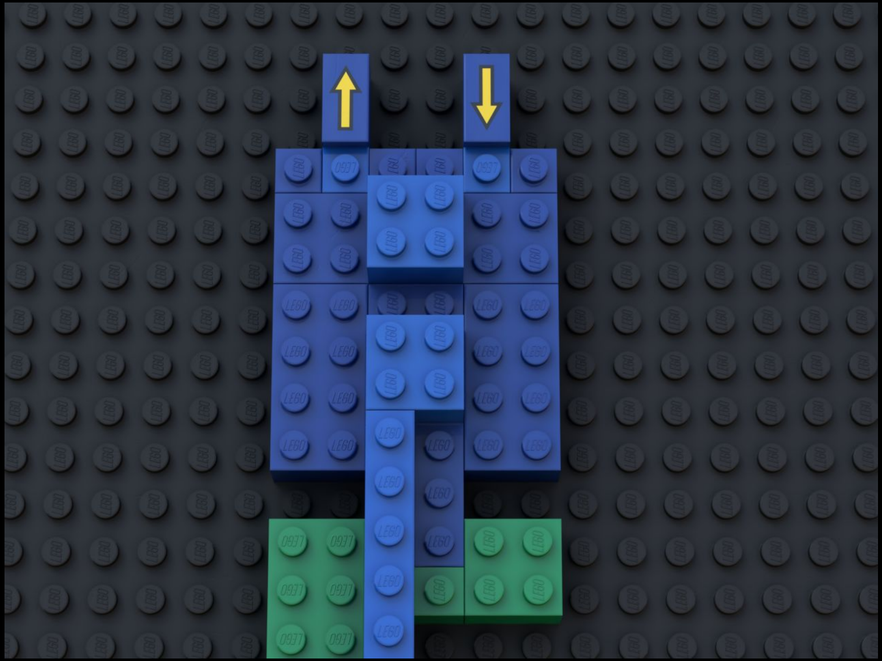
The fuzzer evolves entities, corpus and feedbacks.

We define State as the sum of all of the evolving parts of the fuzzer.

Event Manager

We can define in the fuzzing loop some interesting event that we may want to observe in an abstract way.

So we defined an event manager that provides implementations of event handlers.



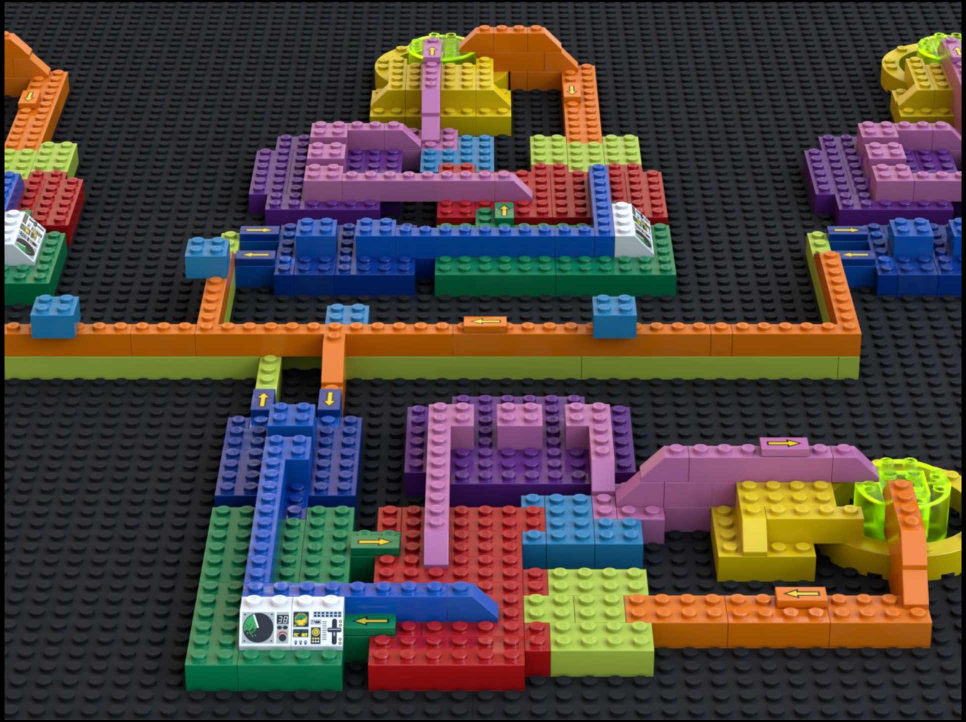
Events

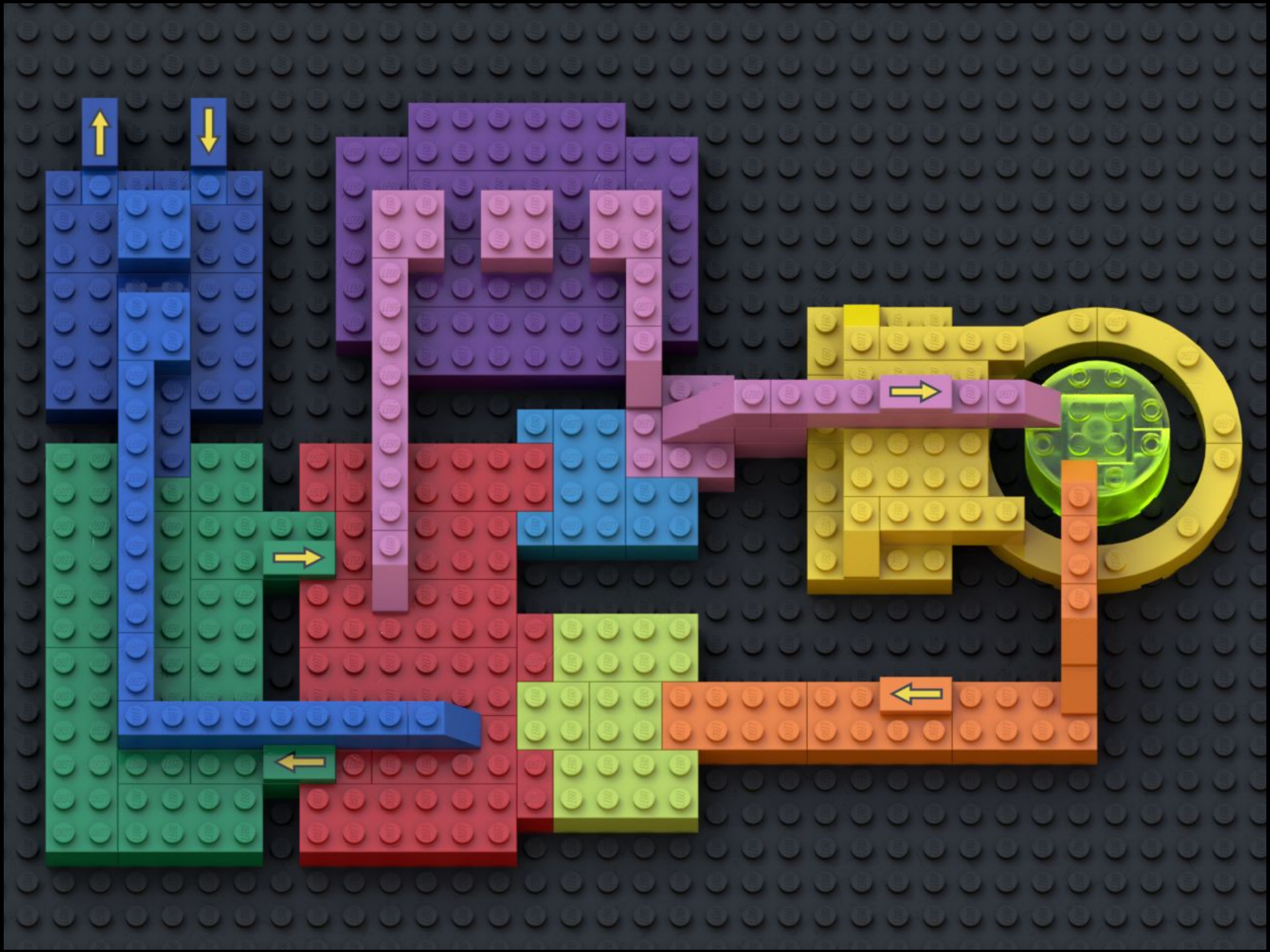
Common events are when a new testcase if added to the corpus, when the program crashes or when there is a timeout.

A very naive event manager just logs these events to inform the user about the changes in the State.

Event Manager: Multicore Sync

A less naive usage of the event manager is for state synchronization between fuzzers. We broadcast information about fuzzing progress. For instance, we exchange testcases between fuzzers in this way.





Then Code?

Initial implementation in C by Rishi Ranjan during AFL++ GsoC

-> No generics, hard to maintain -> Initial rewrite in C++

-> Lots of virtual functions, lots of options, crazy templates

-> 2nd rewrite (initially PoC) in Rust

⇒ Some language features missing, but legible and performant

Then Code?

Observation: Fuzzers have a loop and state, similar to games.

We took inspiration from the RustConf ‘18 Game Dev keynote.

We translated fuzzing the concept to Rust patterns and code.

Abstractions in Rust

Can we model entities as classes like we do in C++ or Java?

The Game State in Rust

```rust
type Entity = GenerationalIndex;
type EntityMap<T> = GenerationalIndexArray<T>;

struct GameState {
    assets: Assets,
    entity_allocator: GenerationalIndexAllocator,
    entity_components: AnyMap,
    players: Vec<Entity>,
    ...
}
```

https://kyren.github.io/2018/09/14/rustconf-talk.html

The Fuzzer State

```
struct State {

    feedbacks: Vec<Box<dyn Feedback>>,
    executor: Box<dyn Executor>,
    corpus: Box<dyn Corpus>,
    stages: Vec<Box<dyn Stage>>,
    // allow the extension of State
    metadatas: AnyMap,

}
```

AnyMap for MetaData

```
struct Testcase<I> {
    input: I,
    metadatas: AnyMap
}

struct State {
    ...,
    metadatas: AnyMap,
}
```

Haskell-like Tuples for static sets

```
struct State {
    feedbacks: FeedbacksTuple,
    metadatas: AnyMap,
}

struct Fuzzer {
    stages: StagesTuple,
}

fn fuzzer_loop<EM, E, C>(&mut EM, &mut E, &mut C,
                          &mut Fuzzer, &mut State);
```

Serde all the things!

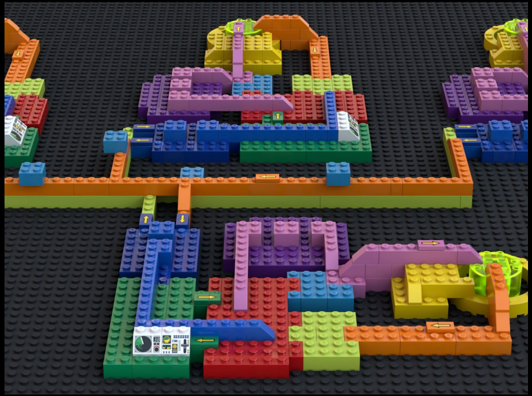
```
trait Input: Serialize + DeserializeOwned {
    …
}

trait Observer: Serialize + DeserializeOwned {
    …
}
```

Send a lot of stuffs on the wire without much effort!

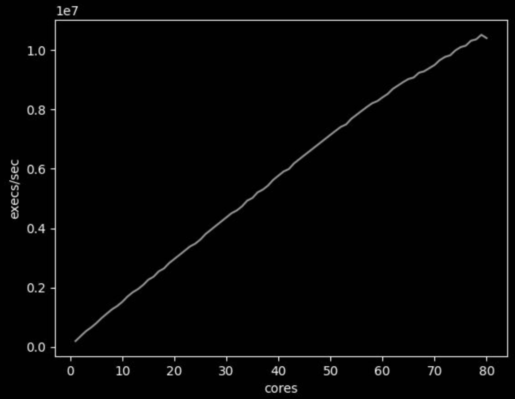
Scaling

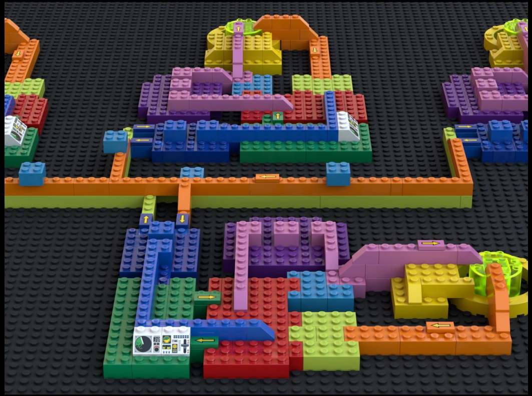
- Just spawning a thread enables glibc mutex
- So each fuzzer instance is a process
- Whenever new interesting testcases are discovered, they are synced (lock-free) over shared map channels
- There is a marginal one-time overhead of serialization per testcase, afterwards no more syncing is involved
- If observers are the same for each client, we can reuse them, else we rerun testcase



Scaling

- Little Kernel Load
- Pretty decend speed
- libpng: > 10mio execs/s





Wait, so only Rust?

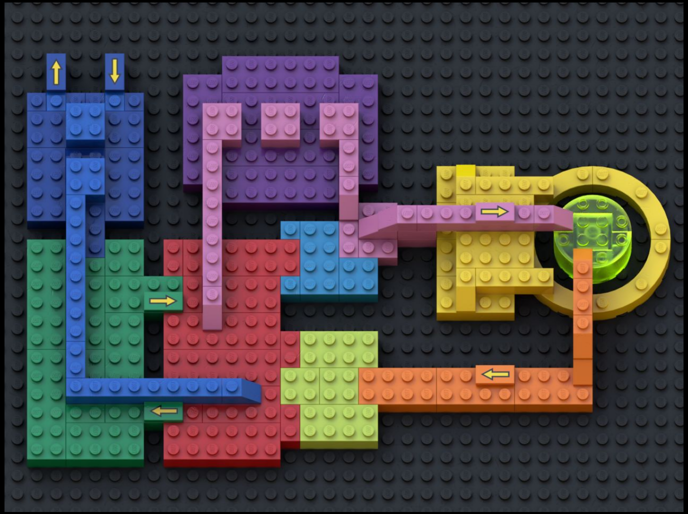
- Of course not, only its core.
- Our test harnesses already include
    - C
    - C++
    - Emulator (QEMU)
- The lib is no_std + alloc
  ⇒ should even run in kernels, embedded, …
- Yes, it needs clang to build once, but can then be linked against gcc, etc.
- Future: Custom LEGO parts may be possible in C, or even Python

Open Source… soon. (sorry :))

- Old C lib at github.com/aflplusplus/libafl
- Rust rewrite _INCOMING_ (q1 2021 ;))
- Let us know if you're interested to test-drive this early
- AFL++ will stay around for "normal" use-cases, potential porting to rust in $future(?)

Conclusion

- We presented some fuzzing building blocks for our lib
- Implementation will be out very soon(™)
- With good defaults
- Follow github.com/aflplusplus
- or us (andreafioraldi, domenukk)



Thanks y'all

Have a nice rC3