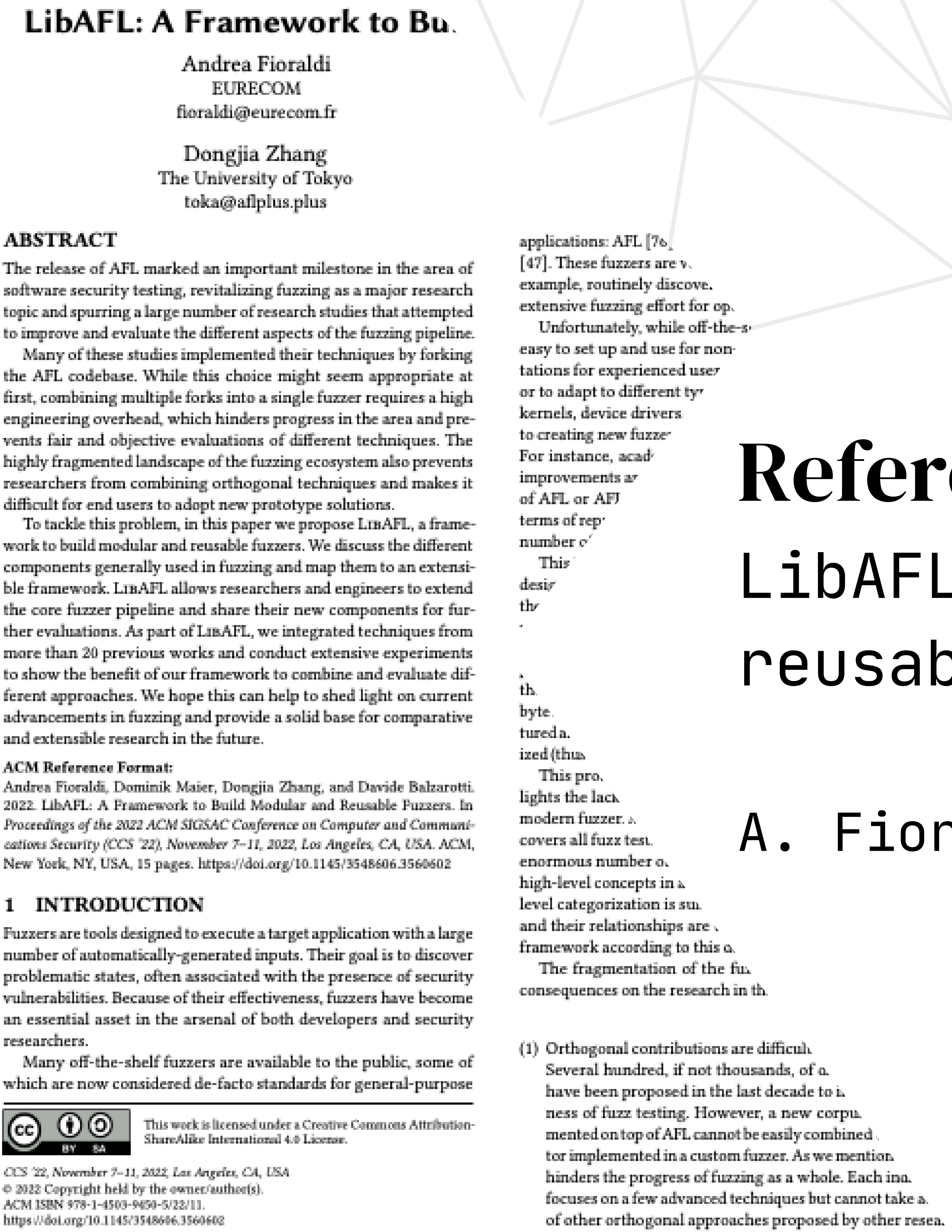
30/04/2024

Fuzz Testing



LibAFL: A Framework to Bu

Andrea Fioraldi
EURECOM
fioraldi@eurecom.fr

Dongjia Zhang
The University of Tokyo
toka@aflplus.plus

ABSTRACT

The release of AFL marked an important milestone in the area of software security testing, revitalizing fuzzing as a major research topic and spurring a large number of research studies that attempted to improve and evaluate the different aspects of the fuzzing pipeline.

Many of these studies implemented their techniques by forking the AFL codebase. While this choice might seem appropriate at first, combining multiple forks into a single fuzzer requires a high engineering overhead, which hinders progress in the area and prevents fair and objective evaluations of different techniques. The highly fragmented landscape of the fuzzing ecosystem also prevents researchers from combining orthogonal techniques and makes it difficult for end users to adopt new prototype solutions.

To tackle this problem, in this paper we propose LibAFL, a framework to build modular and reusable fuzzers. We discuss the different components generally used in fuzzing and map them to an extensible framework. LibAFL allows researchers and engineers to extend the core fuzzer pipeline and share their new components for further evaluations. As part of LibAFL, we integrated techniques from more than 20 previous works and conduct extensive experiments to show the benefit of our framework to combine and evaluate different approaches. We hope this can help to shed light on current advancements in fuzzing and provide a solid base for comparative and extensible research in the future.

ACM Reference Format:
Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560602

1 INTRODUCTION

Fuzzers are tools designed to execute a target application with a large number of automatically-generated inputs. Their goal is to discover problematic states, often associated with the presence of security vulnerabilities. Because of their effectiveness, fuzzers have become an essential asset in the arsenal of both developers and security researchers.

Many off-the-shelf fuzzers are available to the public, some of which are now considered de-facto standards for general-purpose

This work is licensed under a Creative Commons Attribution-ShareAlike International 4.0 License.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560602

applications: AFL [76
[47]. These fuzzers are v
example, routinely discove
extensive fuzzing effort for op

Unfortunately, while off-the-s
easy to set up and use for non
tations for experienced user
or to adapt to different ty
kernels, device drivers
to creating new fuzze
For instance, acad
improvements ar
of AFL or AF
terms of rep
number o

This
desi
th

th
byte
tured a
ized (thus

This pro
lights the lack
modern fuzzer.
covers all fuzz test
enormous number o
high-level concepts in a
level categorization is su
and their relationships are
framework according to this o

The fragmentation of the fu
consequences on the research in th

(1) Orthogonal contributions are difficul
Several hundred, if not thousands, of a
have been proposed in the last decade to i
ness of fuzz testing. However, a new corpu
mented on top of AFL cannot be easily combined
tor implemented in a custom fuzzer. As we mention
hinders the progress of fuzzing as a whole. Each ina
focuses on a few advanced techniques but cannot take a
of other orthogonal approaches proposed by other resea

Reference:

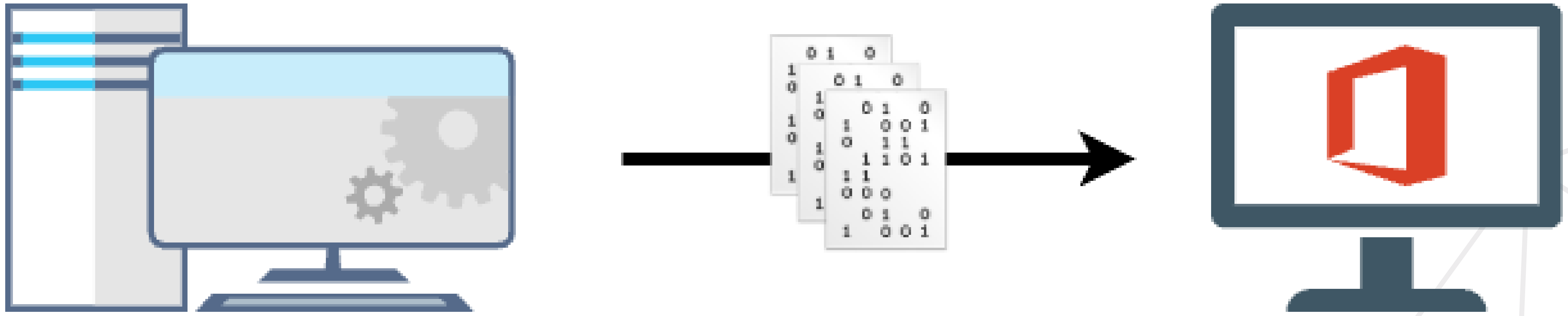
LibAFL: A Framework to build modular and reusable fuzzers.

A. Fioraldi, D. Maier, D. Zhang, D. Balzarotti
CCS 2022

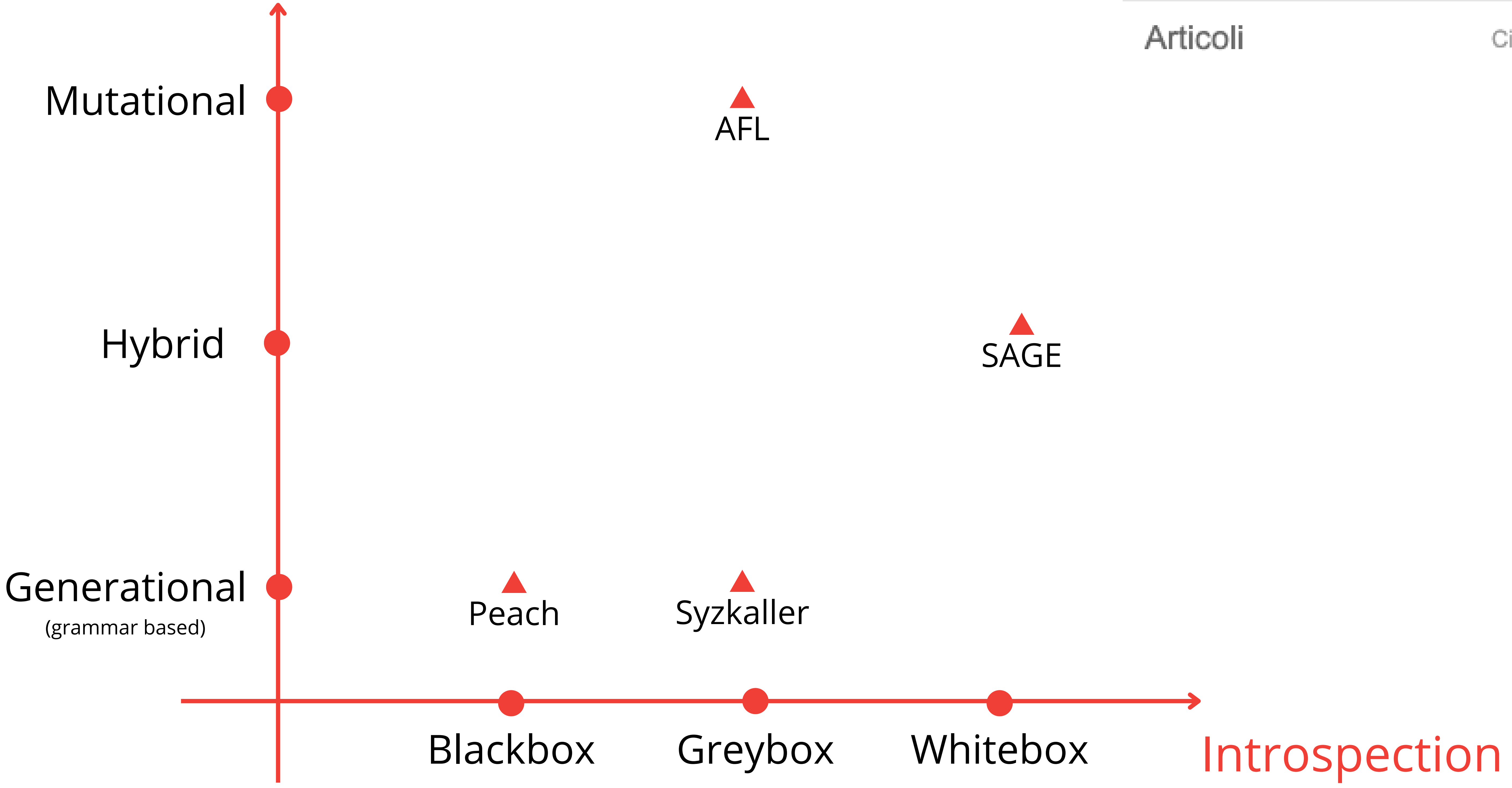
What is Fuzzing

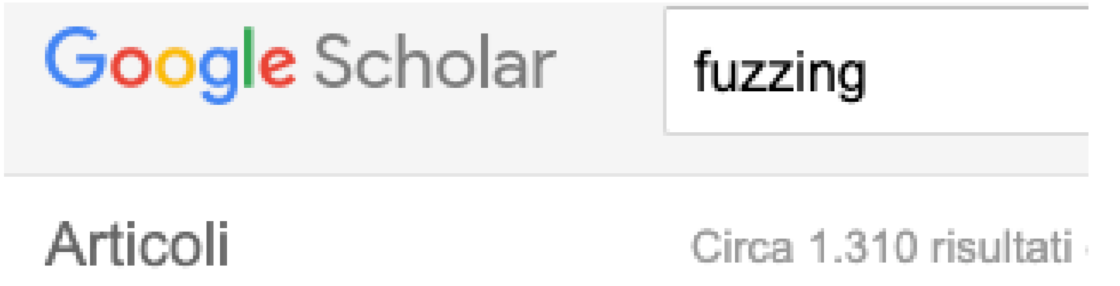
Fuzzers are tools designed to execute a target application with a large number of **automatically-generated inputs**, with the goal of discover problematic states (crashes, exceptions, non termination), often associated with security vulnerabilities.

Evolution of the testcase with subsequent mutations



Fuzzers Taxonomy





Fuzzers Taxonomy

Introspection

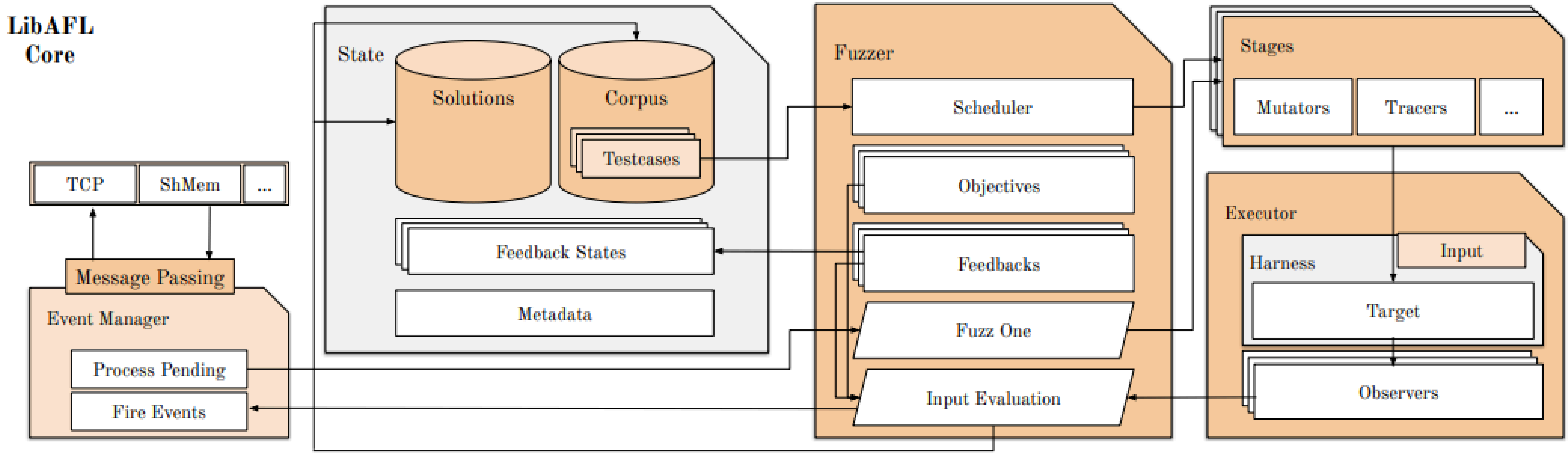
BlackBox	**Do not require any feedback from the target application.** They may still require information about input specification (e.g. Peach).
GreyBox	**Extract minimal information from the target.** Usually information is extracted during execution via instrumentation. (e.g. AFL)
WhiteBox	**Have a complete knowledge of the internal state.** (e.g. SAGE)

Fuzzers Taxonomy

Input Generation

Generational	**Generates new testcases using a model of the input.** Usually via a user provided grammar, generation rules or via learning techniques (e.g. Syzkaller).
Mutational	**Mutates previous inputs to generate new ones.** They require a set of initial seeds to perform the mutation, usually lead by feedback from the target. (e.g. AFL)
Hybrid	**Can perform both input generation and mutation.**

Fuzzer Architecture

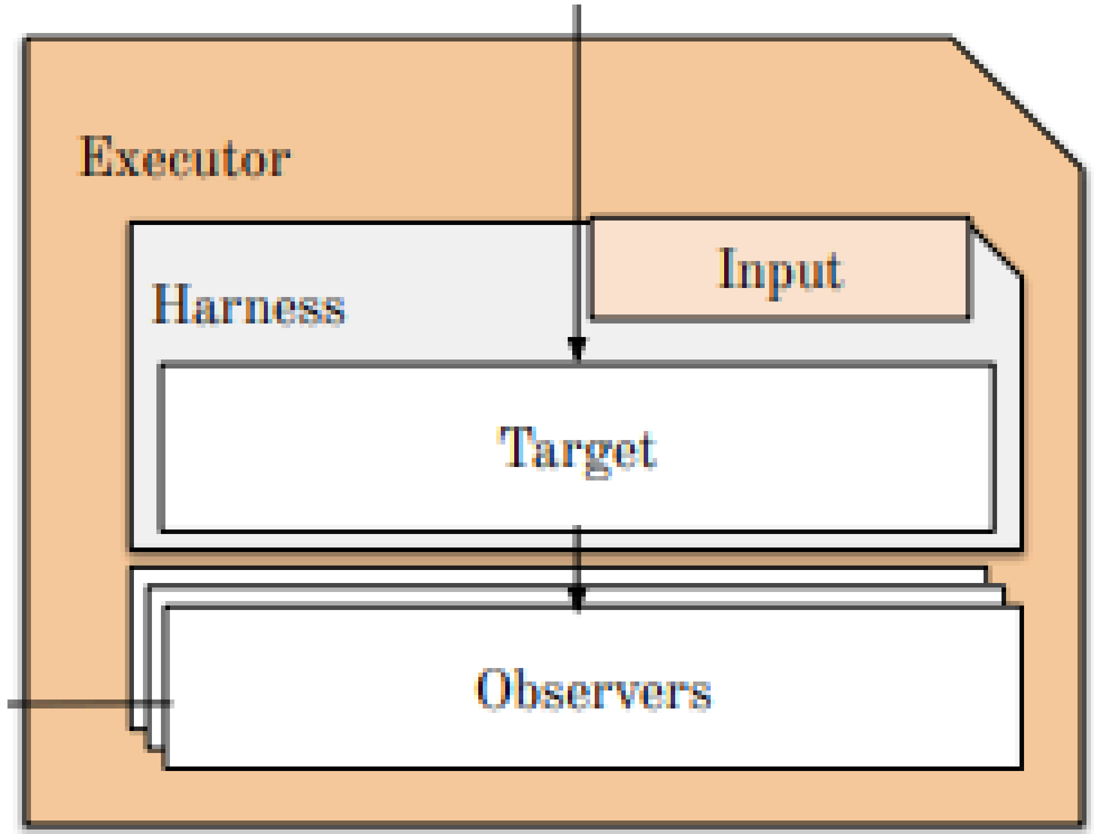


Fuzzer Architecture

Executor

Is the main component responsible for the execution of the testcase

Depending on the target the executor can change a lot, from a simple forkserver, to virtual machine, to a fully fledged hypervisor.



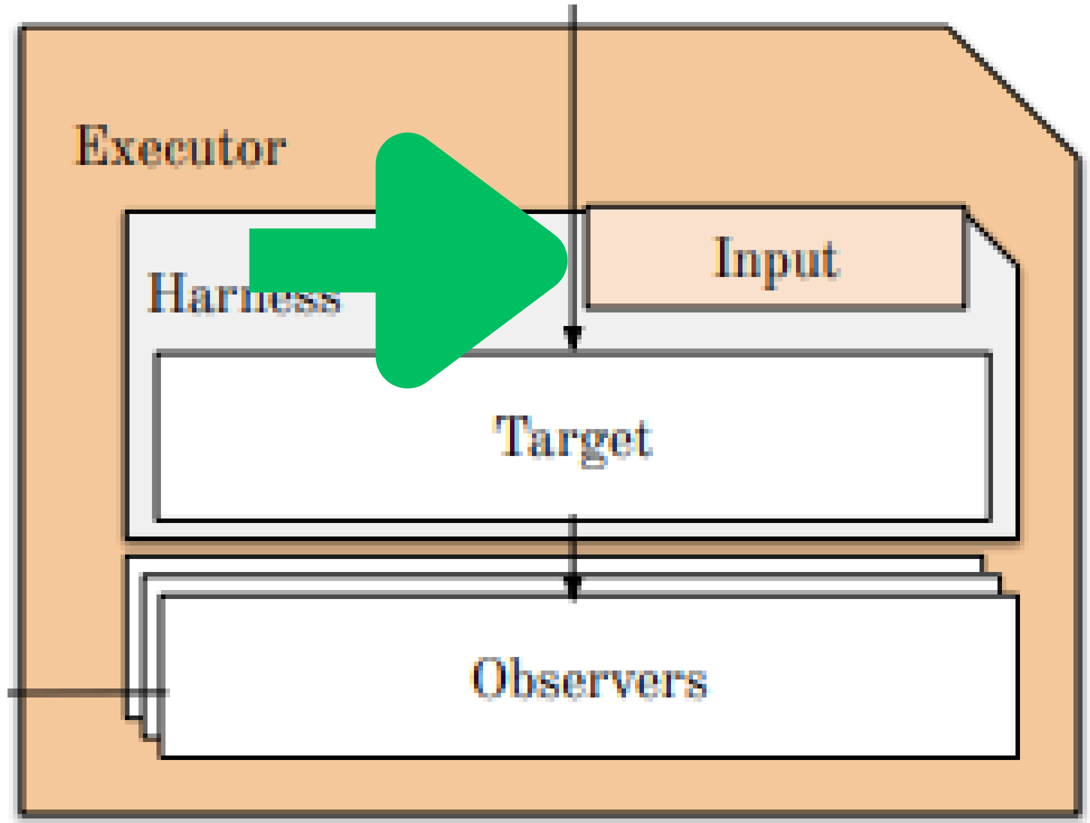
Fuzzer Architecture

Input

The representation of the program input

Could be a:

- A simple ByteArray
- A sequence of System Calls
- A list of transactions
- …



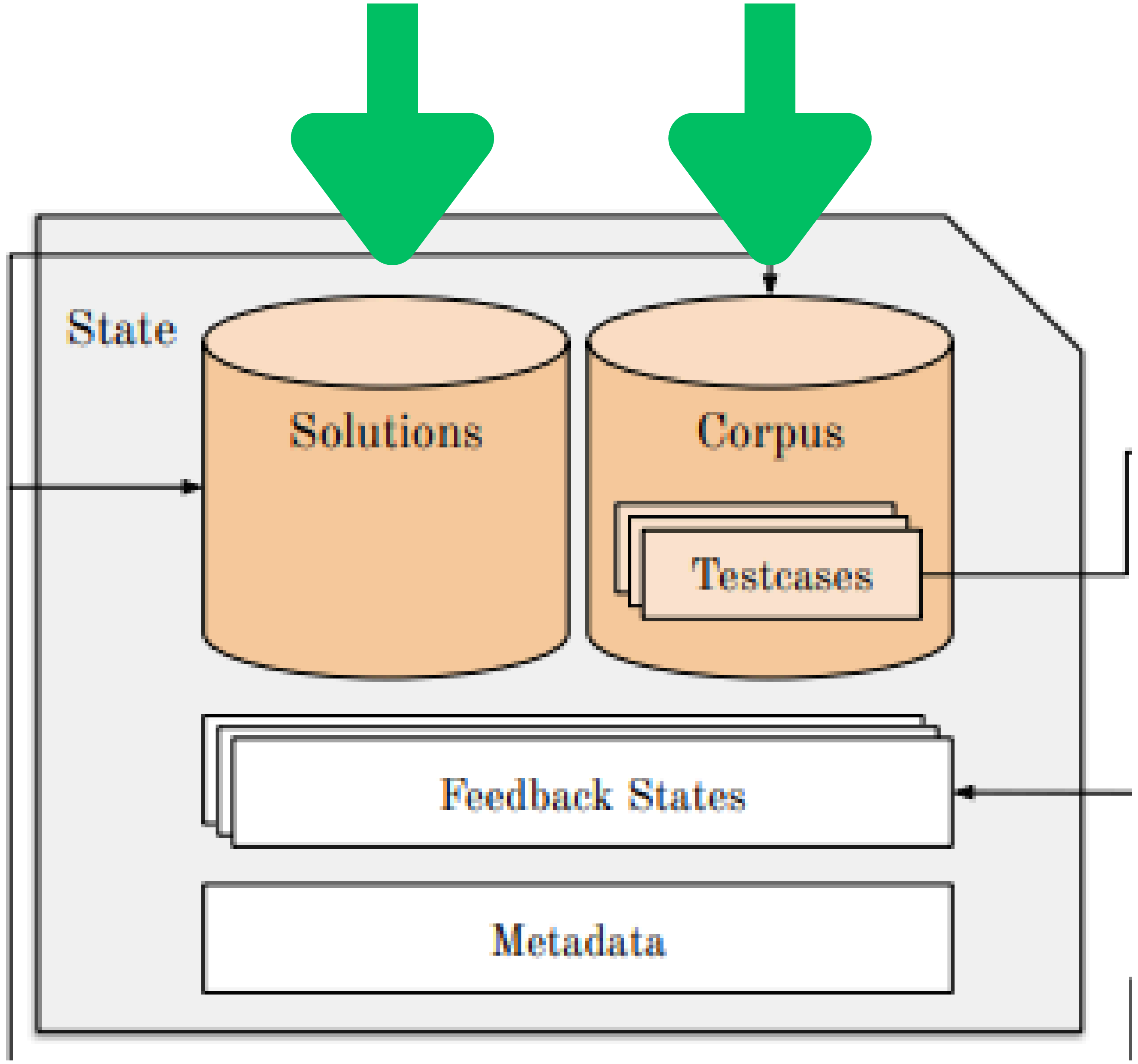
Fuzzer Architecture

Corpus

The storage for the inputs

Contains 2 main classes of inputs:

- *Solutions* (e.g. inputs that make the program crash)
- *Interesting* (e.g. inputs which are queued for mutation)

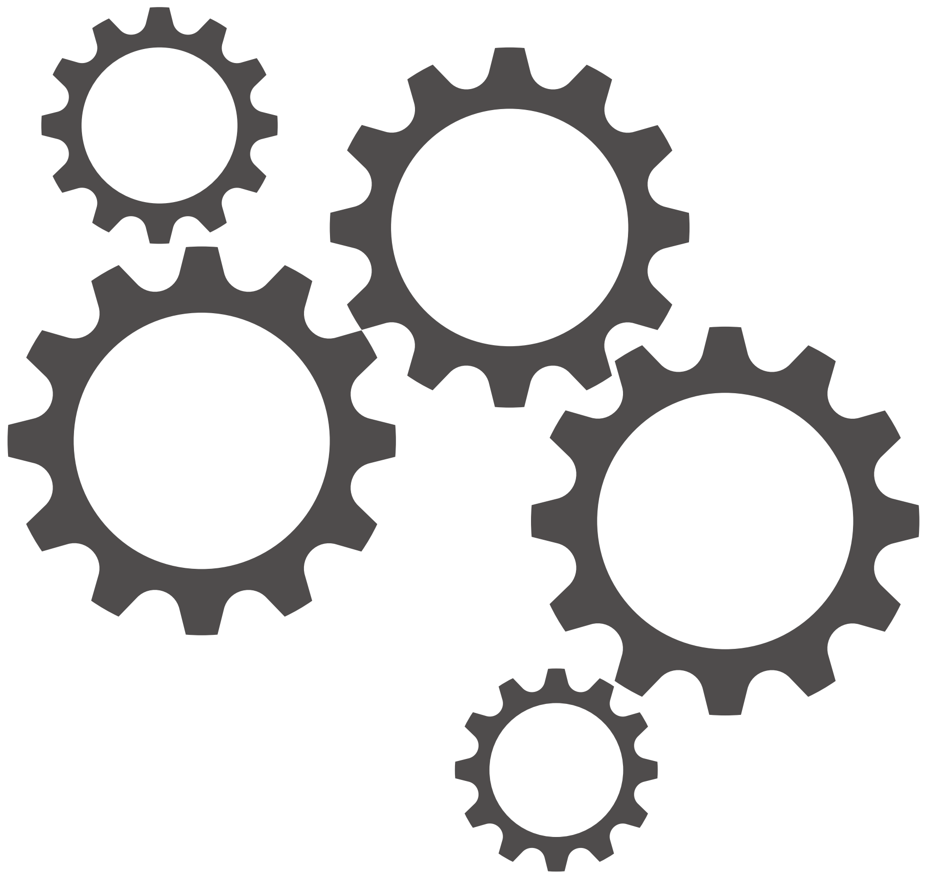


Fuzzer Architecture

Generator

It is the component which performs the testcase generation

The simples generator could be a random generator, however according to the target a grammar or model could be provided which describe the input structure. e.g. Grammar based kernel fuzzing requires a syscall grammar to be provided for testcase generation.

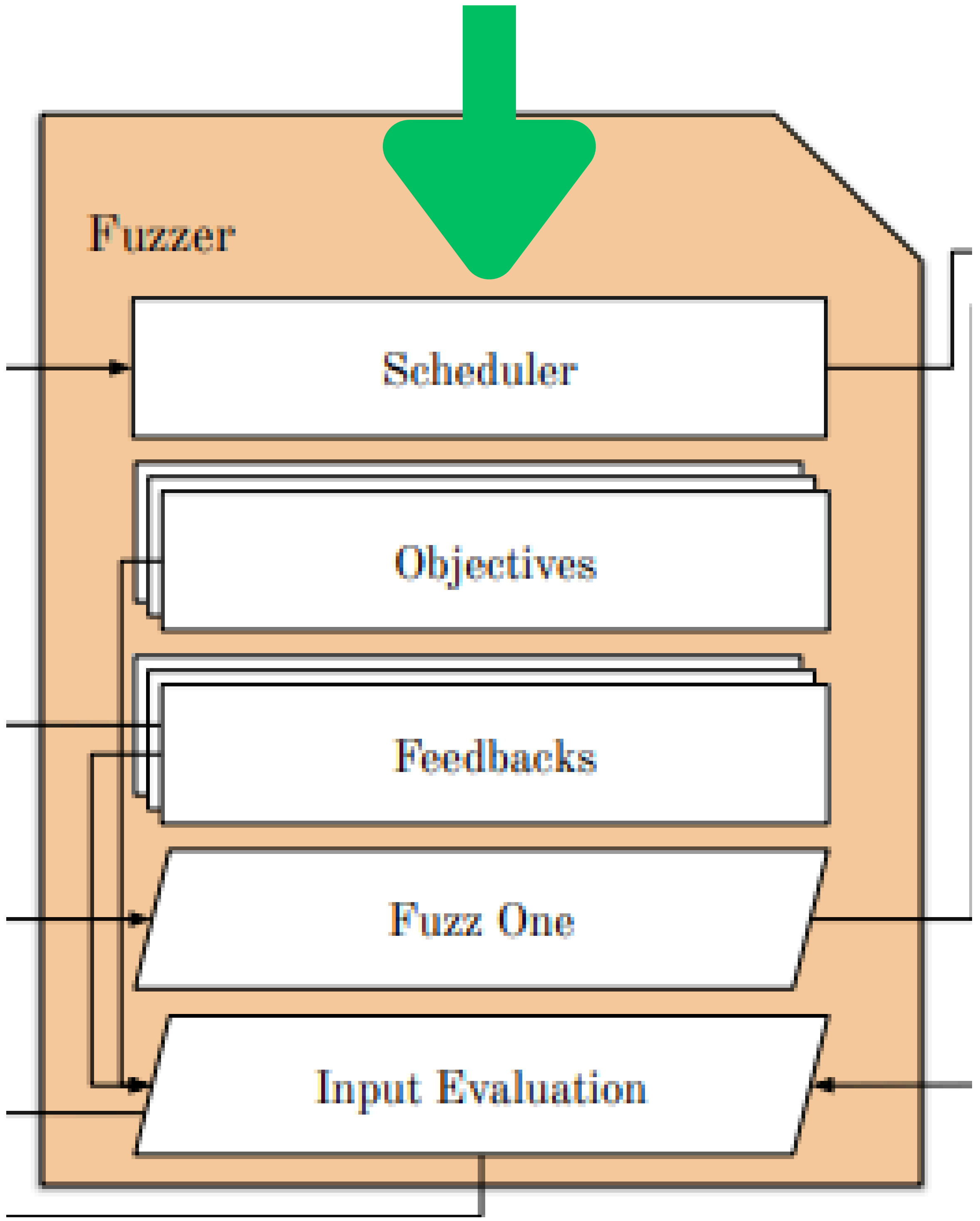


Fuzzer Architecture

Scheduler

Orchestrate the order in which testcases will be selected

Many different strategies could be implemented by a scheduler.
The simplest possible implementation could be a FIFO queue, however one branch of fuzzing research focuses on developing highly optimised scheduling algorithm.

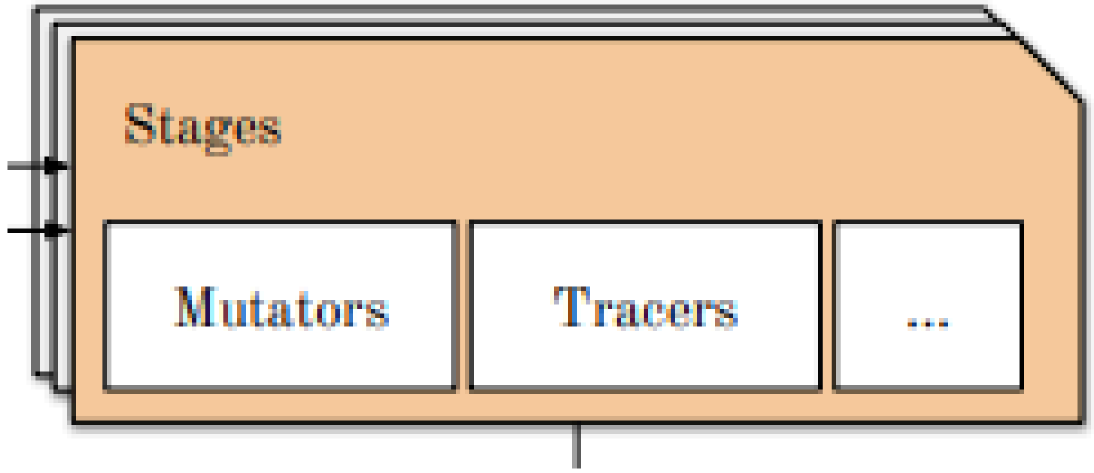


Fuzzer Architecture

Stage

Defines the operation to be performed on the testcase

Is a very generic entity which receives the testcase selected by the scheduler and performs a series of operation. (e.g. mutation, taint tracking, ...)



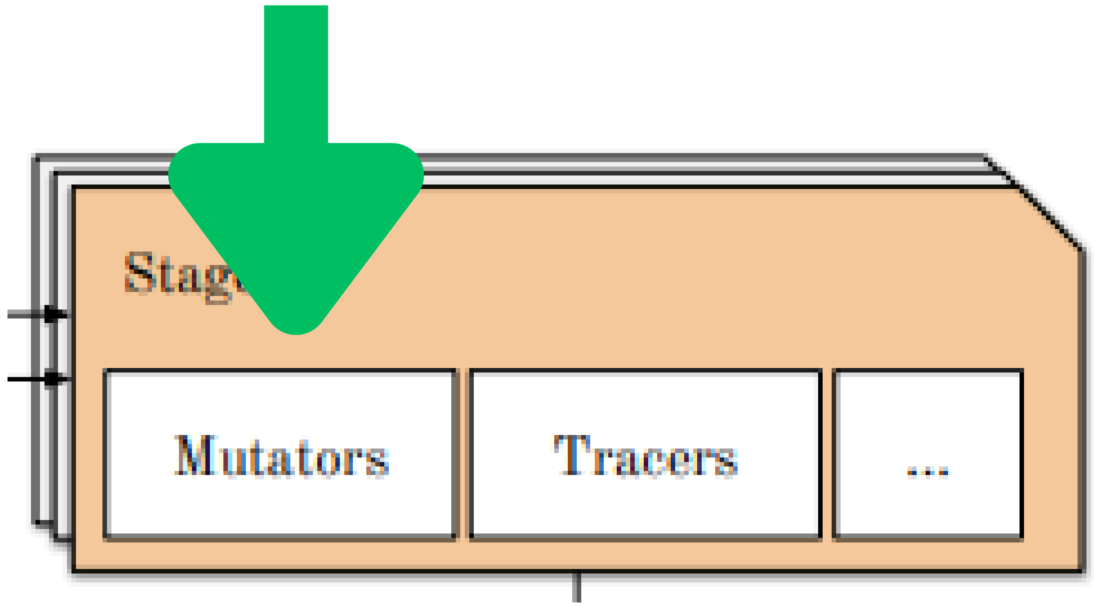
Fuzzer Architecture

Mutator

It is the component which performs the testcase mutation

One other important focus of fuzzing research, different mutation strategies could be theorized according to the input type and tartet.
Some emplyed techniques include:

- bit-flipping
- splicing
- block swapping
- truncating
- expanding

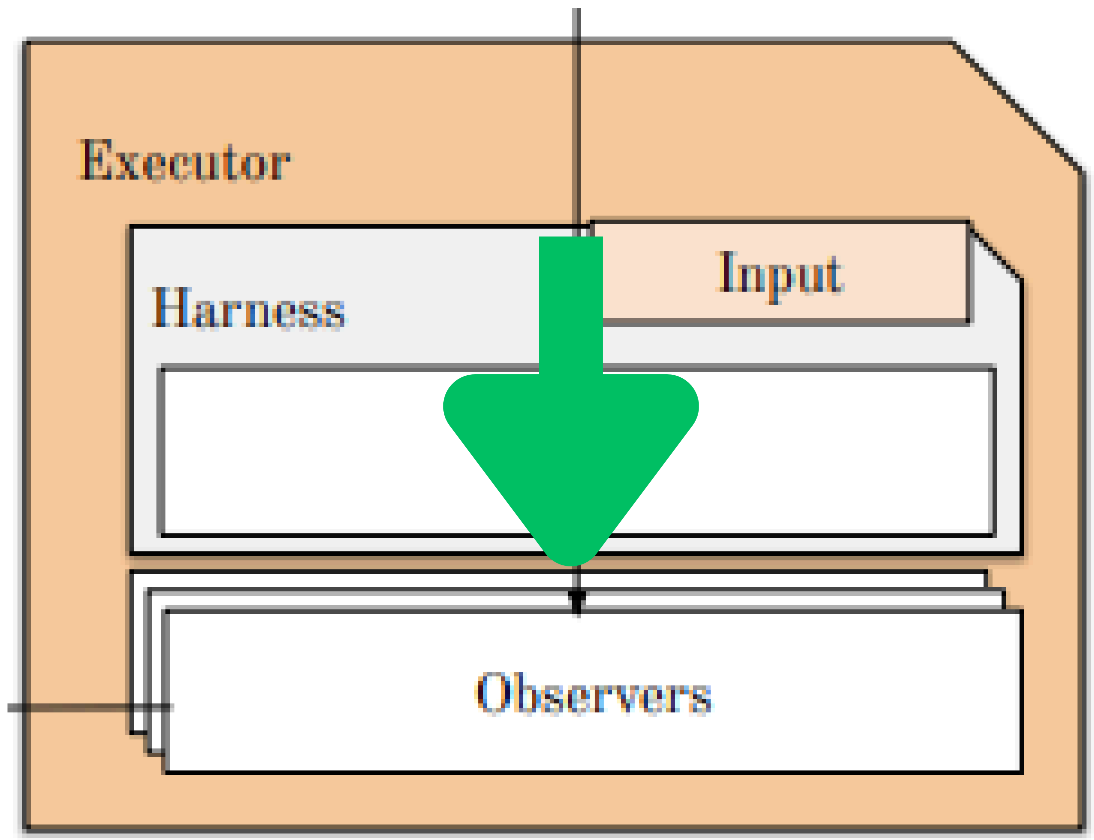


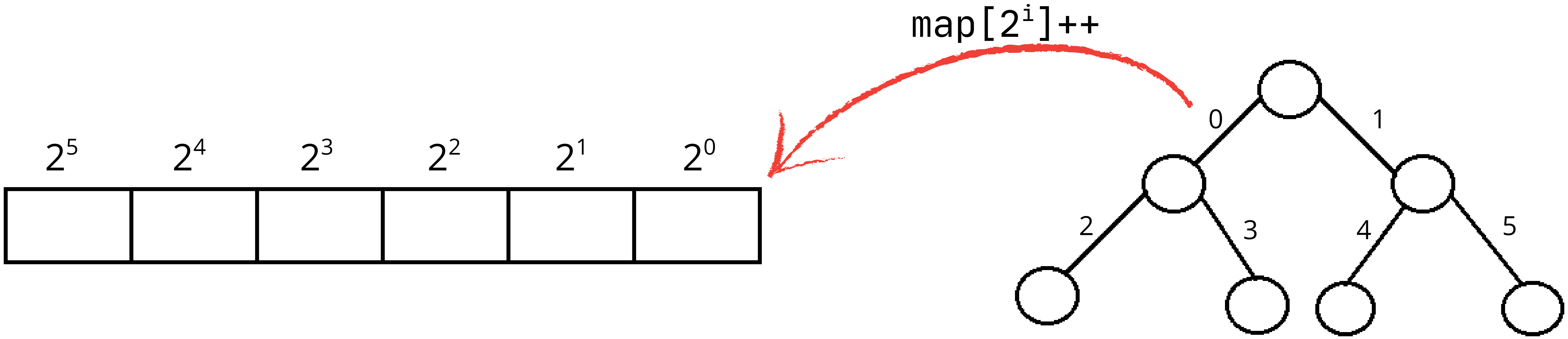
Fuzzer Architecture

Observer

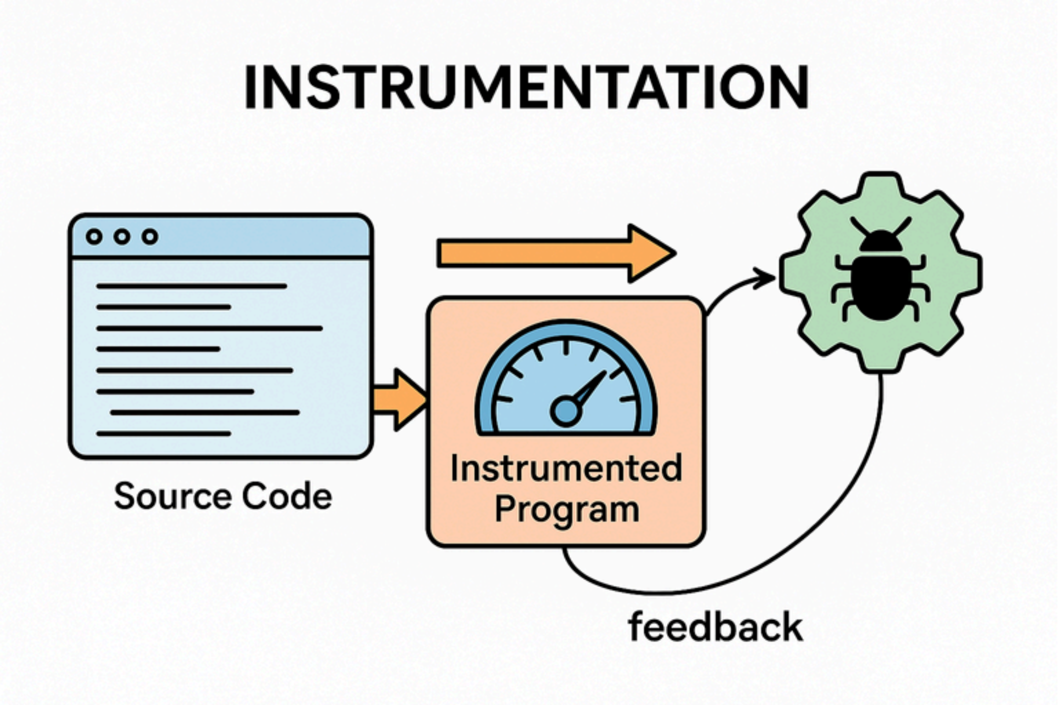
Provides information about a single execution

On important example is the coverage map used by AFL, which is a bytestring where each byte represent how many times the corresponding edge was executed.





Instrumentation



It refers to the process of modifying the binary with additional code that allow the observability and analysis during execution

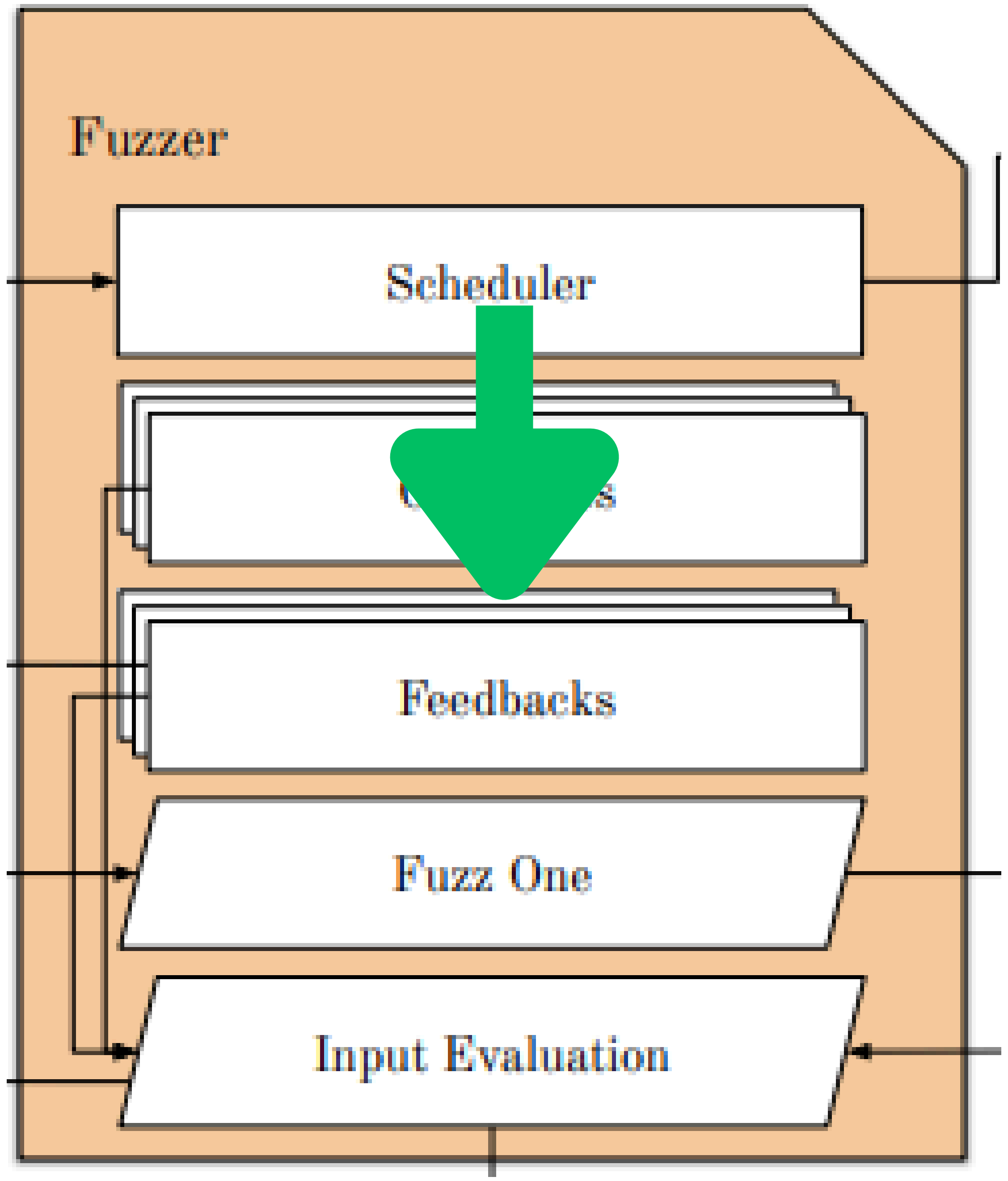
It is usually done as a custom compiler pass which allow injecting custom instructions in key points of the execution without modifying the source code

Fuzzer Architecture

Feedback

It classifies the outcome of the execution

It is deeply linked with the observer, the feedback analyses the information provided by the observer, for example to determine whether the testcase could be considered interesting and add it to the corpus.



Sanitizers

Sanitizers are runtime analysis tools that detect various classes of bugs and security vulnerabilities

Sanitizers operate by injecting checks into the program, either at compile time or dynamically, to catch problematic behavior as it occurs.
E.g.:

- `AddressSanitizer (ASan)`
- `UndefinedBehaviorSanitizer (UBSan)`
- `ThreadSanitizer (TSan)`

https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf

Sanitizer

AddressSanitizer

programing > cpp > asan > test.cc > ...

```cpp
int main(int argc, char **argv)
{
    int *array = new int[100];
    delete[] array;
    return array[argc];  // BOOM
}
```

问题 输出 端口 终端 GITLENS 搜索 调试控制台 源代码管理 3 GITLENS 源代码管理

终端

```
[gerryyang ~/github/mac-utils/programing/cpp/asan 09:13:31]$ ./test
=================================================================
==3914591==ERROR: AddressSanitizer: heap-use-after-free on address 0x614000000044 at pc 0x0000004f584f bp 0x7ffcc7e25010 sp 0x7ffcc7e25008
READ of size 4 at 0x614000000044 thread T0
    #0 0x4f584e in main /data/home/gerryyang/github/mac-utils/programing/cpp/asan/test.cc:6:12
    #1 0x7f0d859def92 in __libc_start_main (/lib64/libc.so.6+0x26f92)
    #2 0x41f31d in _start (/data/home/gerryyang/github/mac-utils/programing/cpp/asan/test+0x41f31d)

0x614000000044 is located 4 bytes inside of 400-byte region [0x614000000040,0x6140000001d0)
freed by thread T0 here:
    #0 0x4f35c0 in operator delete[](void*) /data/home/gerryyang/tools/clang/llvm-project-11.0.0/compiler-rt/lib/asan/asan_new_delete.cpp:163:3
    #1 0x4f581e in main /data/home/gerryyang/github/mac-utils/programing/cpp/asan/test.cc:5:5
    #2 0x7f0d859def92 in __libc_start_main (/lib64/libc.so.6+0x26f92)

previously allocated by thread T0 here:
    #0 0x4f2c28 in operator new[](unsigned long) /data/home/gerryyang/tools/clang/llvm-project-11.0.0/compiler-rt/lib/asan/asan_new_delete.cpp:102:3
    #1 0x4f5813 in main /data/home/gerryyang/github/mac-utils/programing/cpp/asan/test.cc:4:18
    #2 0x7f0d859def92 in __libc_start_main (/lib64/libc.so.6+0x26f92)

SUMMARY: AddressSanitizer: heap-use-after-free /data/home/gerryyang/github/mac-utils/programing/cpp/asan/test.cc:6:12 in main
Shadow bytes around the buggy address:
  0x0c287fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c287fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c287fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c287fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c287fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c287fff8000: fa fa fa fa fa fa fa fa[fd]fd fd fd fd fd fd fd
  0x0c287fff8010: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x0c287fff8020: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x0c287fff8030: fd fd fd fd fd fd fd fd fd fd fa fa fa fa fa fa
  0x0c287fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==3914591==ABORTING
```

State Of The Art: AFL++

Coverage-guided Graybox fuzzer

AFL++: Combining Incremental Steps of Fuzzing Research

Andrea Fioraldi[†], Dominik Maier[‡], Heiko Eißfeldt, Marc Heuse[§]

{andrea, dominik, heiko, marc}@aflplus.plus

†Sapienza University of Rome, ‡TU Berlin, §The Hacker's Choice

Abstract

In this paper, we present AFL++, a community-driven open-source tool that incorporates state-of-the-art fuzzing research, to make the research comparable, reproducible, combinable and — most importantly – useable. It offers a variety of novel features, for example its *Custom Mutator API*, able to extend the fuzzing process at many stages. With it, mutators for specific targets can also be written by experienced security testers. We hope for AFL++ to become a new baseline tool not only for current, but also for future research, as it allows to test new techniques quickly, and evaluate not only the effectiveness of the single technique versus the state-of-the-art, but also in combination with other techniques. The paper gives an evaluation of hand-picked fuzzing technologies — shining light on the fact that while each novel fuzzing method can increase performance in some targets — it decreases performance for other targets. This is an insight future fuzzing research should consider in their evaluations.

to combine functionality with the compatible techniques that address different, but related problems in fuzzing — for example picking a recent seed scheduling for their mutator. A new feedback concept may not live up to its full potential if it cannot be combined with existing techniques solving other problems — like overcoming hard comparison instructions — reducing the impact of the research on paper due to lackluster statistics.

In this paper, we try to solve these problems by raising the bar of broadly available, research-backed, fuzzing, and by giving researchers an extensible API to build upon. We propose a novel fuzzing framework, AFL++. Future research can use AFL++ as a new baseline. It gives researchers the possibility to evaluate combinations of their proposals with state-of-the-art orthogonal features already implemented in AFL++ — with a highly reduced implementation effort. At the same time, it offers industry professionals a large range of easy-to-use features adapted from cutting-edge research, that can greatly improve the outcome of a fuzzing campaign.

https://www.usenix.org/system/files/woot20-paper-fioraldi.pdf

Not Only Binary

Nyx-Net: Network Fuzzing with Incremental Snapshots

Sergej Schumilo[1], Cornelius Asch... z[3]

syzkaller

the next gen kernel fuzzer

Qualcomm Mobile Security Summit 2017

Dmitry Vyukov (dvyukov@), Google

...cilitating Non-Intrusive In-Vivo Firmware Testing with Stateless Instrumentation

...heng Shi	Wenqiang Li	Wenwen Wang	Le Guan
...ty of Georgia	Independent Researcher	University of Georgia	University of Georg...
...g@uga.edu	wenqiang-li@outlook.com	wenwen@cs.uga.edu	leguan@uga.edu

DroidFuzzer: Fuzzing the Android Apps wit... Intent-Filter Tag

Hui Ye[1], Shaoyin Cheng[1], Lanbo Zha...[2]

[1]University of Science and Techn...

[2]Univ...

SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses

Jaeseung Choi*	Doyeon Kim*†	Soomin Kim
KAIST	*LINE Plus Corporation*	*KAIST*
jschoi17@kaist.ac.kr	doyeon1017@linecorp.com	soomink@kaist.ac.kr
Gustavo Grieco	Alex Groce	Sang Kil Cha
Trail of Bits	*Northern Arizona University*	*KAIST*
gustavo.grieco@trailofbits.com	alex.groce@nau.edu	sangkilc@kaist.ac.kr