L15: Fuzzing: Black-box & Grey-box Fuzzing

Software Analysis

IIT Guwahati · September 2026

The Original Fuzz Experiment

Miller, Fredriksen, **1990**: feed *random characters* to UNIX utilities and see what happens.

Miller, Fredriksen & So, *An Empirical Study of the Reliability of UNIX Utilities*, CACM 33(12), 1990

The Original Fuzz Experiment

Miller, Fredriksen, **1990**: feed *random characters* to UNIX utilities and see what happens.

- the input is random: no model of the program, or of the input format, or of the system;

Miller, Fredriksen & So, *An Empirical Study of the Reliability of UNIX Utilities*, CACM 33(12), 1990

The Original Fuzz Experiment

Miller, Fredriksen, **1990**: feed *random characters* to UNIX utilities and see what happens.

- the input is random: no model of the program, or of the input format, or of the system;
- If the program **crashes or hangs** it fails, otherwise it passes.

Miller, Fredriksen & So, *An Empirical Study of the Reliability of UNIX Utilities*, CACM 33(12), 1990

The Original Fuzz Experiment

Miller, Fredriksen, **1990**: feed *random characters* to UNIX utilities and see what happens.

- the input is random: no model of the program, or of the input format, or of the system;
- If the program **crashes or hangs** it fails, otherwise it passes.
- the result: between **25% and 33%** of the utilities tested crashed or hung.

Miller, Fredriksen & So, *An Empirical Study of the Reliability of UNIX Utilities*, CACM 33(12), 1990

The Original Fuzz Experiment

Miller, Fredriksen, **1990**: feed *random characters* to UNIX utilities and see what happens.

- the input is random: no model of the program, or of the input format, or of the system;
- If the program **crashes or hangs** it fails, otherwise it passes.
- the result: between **25% and 33%** of the utilities tested crashed or hung.

It is the classic empirical study of random testing and the origin of the term *fuzzing*.

Miller, Fredriksen & So, *An Empirical Study of the Reliability of UNIX Utilities*, CACM 33(12), 1990

Where It Came From: a Class Assignment

CS 736, Advanced Operating Systems, Wisconsin, Fall 1988

```
(1) Operating System Utility Program Reliability - The Fuzz Generator:
    The goal of this project is to evaluate the robustness of various
    UNIX utility programs, given an unpredictable input stream. [...]
    First, you will build a fuzz generator. This is a program that will
    output a random character stream. Second, you will take the fuzz
    generator and use it to attack as many UNIX utilities as possible,
    with the goal of trying to break them. For the utilities that break,
    you will try to determine what type of input cause the break.
```

Miller, *CS 736 Project List*, Wisconsin, Fall 1988

A Black-box Fuzzing Example

Toy parser

```
int parse(const uint8_t *d, size_t n) {
    if (n < 16)       return 0;
    if (d[0] != 'F') return 1;
    if (d[1] != 'U') return 2;
    if (d[2] != 'Z') return 3;
    if (d[3] != 'Z') return 4;
    if (d[4] != '!') return 5;

    char buf[8];
    memcpy(buf, d, 12);

    return 6;
}
```

What does the black-box fuzzer do?

A Black-box Fuzzing Example

Toy parser

```
int parse(const uint8_t *d, size_t n) {
    if (n < 16)       return 0;
    if (d[0] != 'F') return 1;
    if (d[1] != 'U') return 2;
    if (d[2] != 'Z') return 3;
    if (d[3] != 'Z') return 4;
    if (d[4] != '!') return 5;

    char buf[8];
    memcpy(buf, d, 12);

    return 6;
}
```

Target:

```
FUZZ!  →  memcpy(buf, d, 12)
```

What does the black-box fuzzer do?

1. Generate 16 random bytes.
2. Run the program.
3. Observe the outcome.
4. Keep the input if it crashes.

A Black-box Fuzzing Example

Toy parser

```
int parse(const uint8_t *d, size_t n) {
    if (n < 16)       return 0;
    if (d[0] != 'F') return 1;
    if (d[1] != 'U') return 2;
    if (d[2] != 'Z') return 3;
    if (d[3] != 'Z') return 4;
    if (d[4] != '!') return 5;

    char buf[8];
    memcpy(buf, d, 12);

    return 6;
}
```

Target:

`FUZZ!` $\rightarrow$ `memcpy(buf, d, 12)`

What does the black-box fuzzer do?

1. Generate 16 random bytes.
2. Run the program.
3. Observe the outcome.
4. Keep the input if it crashes.

The problem
To reach the bug:

$$d[0] = \texttt{'F'}$$
$$d[1] = \texttt{'U'}$$
$$d[2] = \texttt{'Z'}$$
$$d[3] = \texttt{'Z'}$$
$$d[4] = \texttt{'!'}$$

Random Inputs

```
$ ./blackbox 1000000
1000000 random inputs
  passed  'F':              3845
  passed  'FU':             16
  passed  'FUZ':            0
  passed  'FUZZ':           0
  reached the bug:          0
```

Random Inputs

```
$ ./blackbox 1000000
1000000 random inputs
  passed  'F':              3845
  passed  'FU':             16
  passed  'FUZ':            0
  passed  'FUZZ':           0
  reached the bug:          0
```

Exercise: Each byte of the header costs a factor of 256. If the header is 5 bytes long, calculate the probability of randomly generating the correct header.

Random Inputs

```
$ ./blackbox 1000000
1000000 random inputs
  passed  'F':              3845
  passed  'FU':             16
  passed  'FUZ':            0
  passed  'FUZZ':           0
  reached the bug:          0
```

Exercise: Each byte of the header costs a factor of 256. If the header is 5 bytes long, calculate the probability of randomly generating the correct header.

Question

The fuzzer runs an input and the program does *not* crash.

Was that execution useless?

Question

The fuzzer runs an input and the program does *not* crash.

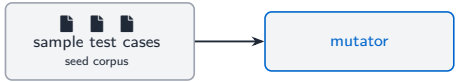
Was that execution useless?

Not necessarily: it may have executed code that no earlier input reached, which is information the fuzzer can record and use.

Grey-box Fuzzing

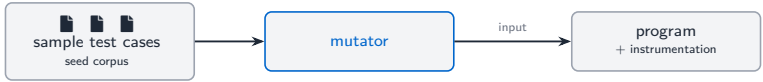
Coverage-Guided Grey-box Fuzzing

Coverage-Guided Grey-box Fuzzing



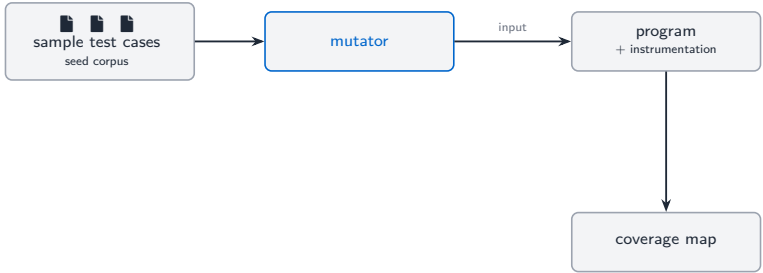
Technical whitepaper for afl-fuzz · Böhme, Pham & Roychoudhury, *Coverage-based Greybox Fuzzing as Markov Chain*, TSE 45(5), 2019

Coverage-Guided Grey-box Fuzzing



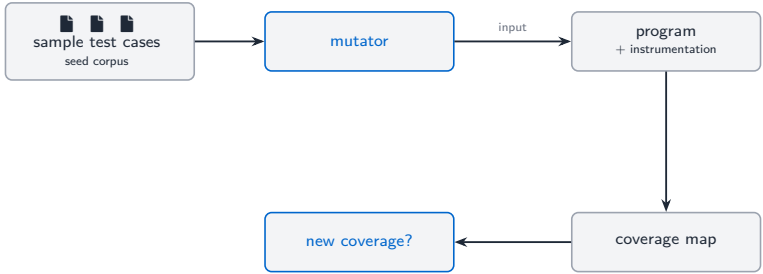
Technical whitepaper for afl-fuzz · Böhme, Pham & Roychoudhury, *Coverage-based Greybox Fuzzing as Markov Chain*, TSE 45(5), 2019

Coverage-Guided Grey-box Fuzzing



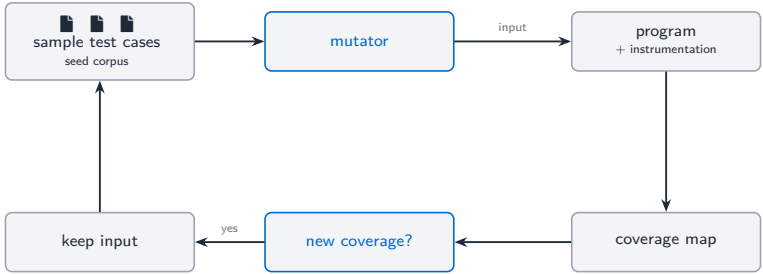
Technical whitepaper for afl-fuzz · Böhme, Pham & Roychoudhury, *Coverage-based Greybox Fuzzing as Markov Chain*, TSE 45(5), 2019

Coverage-Guided Grey-box Fuzzing



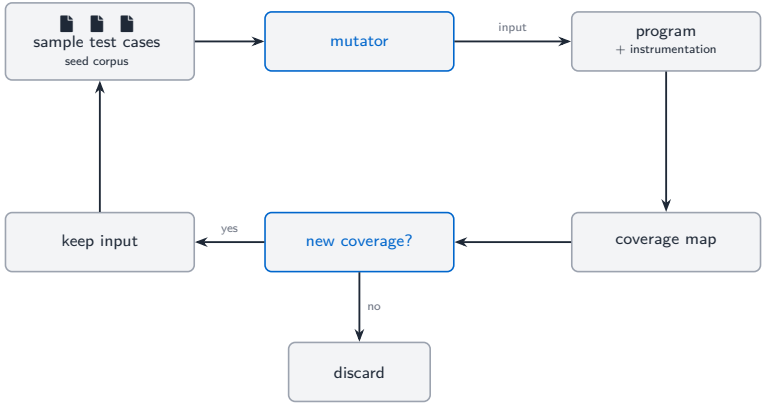
Technical whitepaper for afl-fuzz · Böhme, Pham & Roychoudhury, *Coverage-based Greybox Fuzzing as Markov Chain*, TSE 45(5), 2019

Coverage-Guided Grey-box Fuzzing



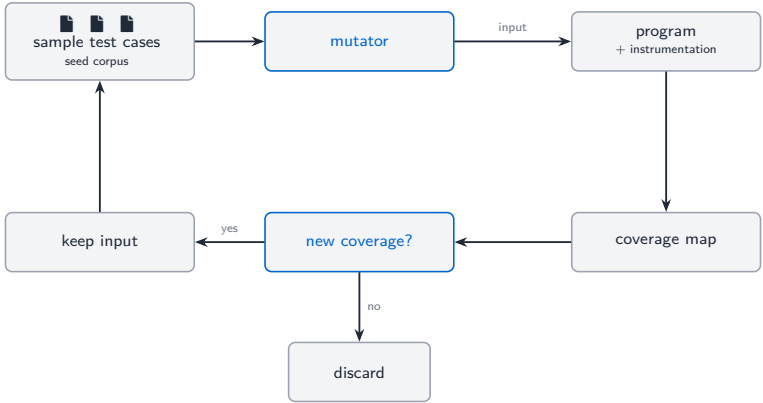
Technical whitepaper for afl-fuzz · Böhme, Pham & Roychoudhury, *Coverage-based Greybox Fuzzing as Markov Chain*, TSE 45(5), 2019

Coverage-Guided Grey-box Fuzzing



Technical whitepaper for afl-fuzz · Böhme, Pham & Roychoudhury, *Coverage-based Greybox Fuzzing as Markov Chain*, TSE 45(5), 2019

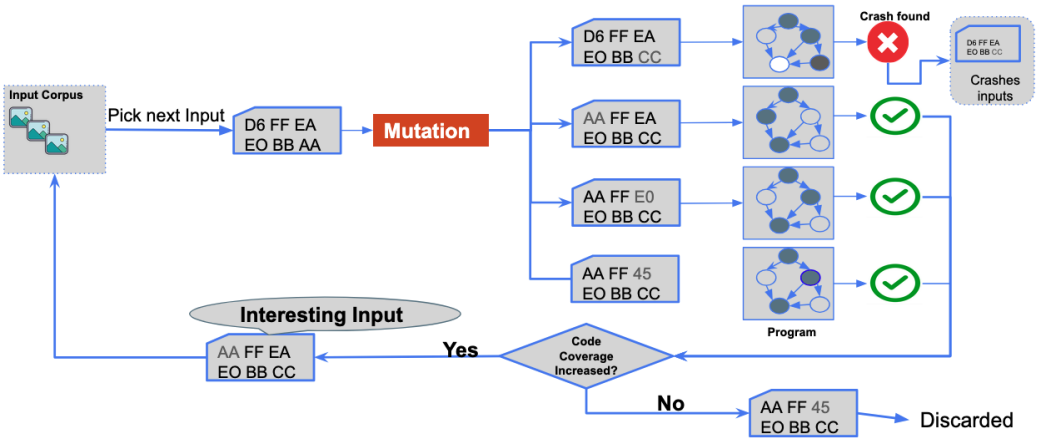
Coverage-Guided Grey-box Fuzzing



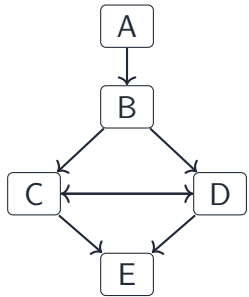
Selection rule: keep an input when its execution discovers new coverage.

Technical whitepaper for afl-fuzz · Böhme, Pham & Roychoudhury, *Coverage-based Greybox Fuzzing as Markov Chain*, TSE 45(5), 2019

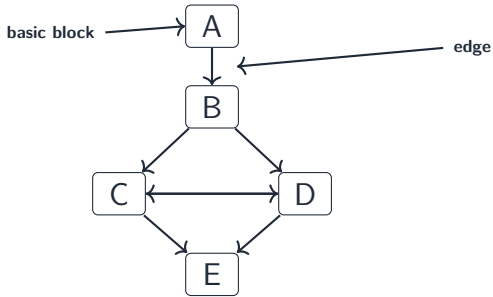
Coverage-Guided Grey-Box Fuzzing



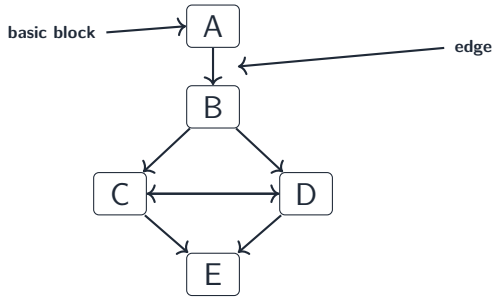
Block vs. Edge Coverage



Block vs. Edge Coverage

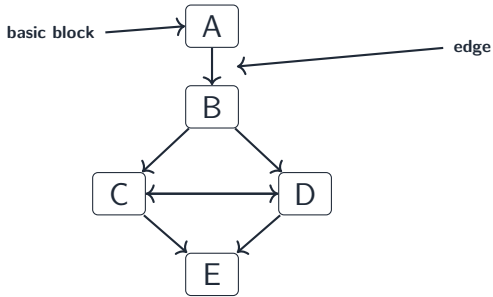


Block vs. Edge Coverage



An execution follows a **path** through the CFG.

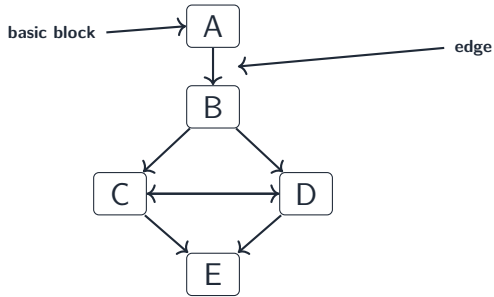
Block vs. Edge Coverage



An execution follows a **path** through the CFG.

Path 1: $A \to B \to C \to D \to E$

Block vs. Edge Coverage

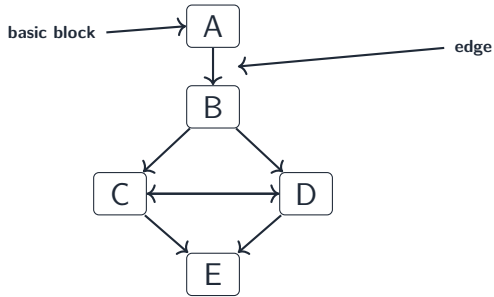


An execution follows a **path** through the CFG.

Path 1: $A \to B \to C \to D \to E$

Path 2: $A \to B \to D \to C \to E$

Block vs. Edge Coverage



An execution follows a **path** through the CFG.

Path 1: $A \to B \to C \to D \to E$

Path 2: $A \to B \to D \to C \to E$

Same blocks, different edges

Why Block Coverage Is Not Enough

Two executions can visit the same blocks.

Execution 1

Execution 2

Blocks visited
A, B, C, D, E

Blocks visited
A, B, C, D, E

Why Block Coverage Is Not Enough

Two executions can visit the same blocks.

Execution 1

Execution 2

Blocks visited
A, B, C, D, E

Blocks visited
A, B, C, D, E

Block coverage says: same.

Why Block Coverage Is Not Enough

Two executions can visit the same blocks.

Execution 1

Execution 2

Blocks visited
A, B, C, D, E

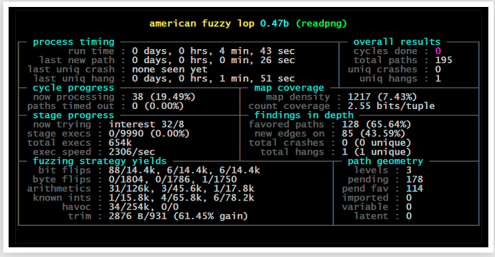
Blocks visited
A, B, C, D, E

Block coverage says: same.

But the transitions are different.

american fuzzy lop (2.53b)

American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. This substantially improves the functional coverage for the fuzzed code. The compact synthesized corpora produced by the tool are also useful for seeding other, more labor- or resource-intensive testing regimes down the road.



American Fuzzy Lop

- **Mutation-based:** new inputs are byte-level edits of inputs already in the queue;

Technical whitepaper for afl-fuzz

American Fuzzy Lop

- **Mutation-based:** new inputs are byte-level edits of inputs already in the queue;
- **Coverage-guided:** an input is queued when it produces an edge tuple not seen before;

Technical whitepaper for afl-fuzz

American Fuzzy Lop

- **Mutation-based:** new inputs are byte-level edits of inputs already in the queue;
- **Coverage-guided:** an input is queued when it produces an edge tuple not seen before;
- **Lightweight instrumentation:** a few instructions per branch point, a 64 kB shared map;

Technical whitepaper for afl-fuzz

American Fuzzy Lop

- **Mutation-based:** new inputs are byte-level edits of inputs already in the queue;
- **Coverage-guided:** an input is queued when it produces an edge tuple not seen before;
- **Lightweight instrumentation:** a few instructions per branch point, a 64 kB shared map;
- **Fast:** the design is organised around executions per second.

Technical whitepaper for afl-fuzz

How AFL Obtains Coverage

The instrumentation injected at every branch point is equivalent to

```
cur_location = <COMPILE_TIME_RANDOM>;          /* one id per basic block */
shared_mem[cur_location ^ prev_location]++;    /* index = an edge        */
prev_location = cur_location >> 1;             /* keep A->B != B->A      */
```

Technical whitepaper for afl-fuzz

How AFL Obtains Coverage

The instrumentation injected at every branch point is equivalent to

```
cur_location = <COMPILE_TIME_RANDOM>;          /* one id per basic block */
shared_mem[cur_location ^ prev_location]++;    /* index = an edge        */
prev_location = cur_location >> 1;             /* keep A->B != B->A      */
```

`shared_mem` is a **64 kB** array shared with the fuzzer: one byte per edge tuple.

How AFL Obtains Coverage

The instrumentation injected at every branch point is equivalent to

```
cur_location = <COMPILE_TIME_RANDOM>;          /* one id per basic block */
shared_mem[cur_location ^ prev_location]++;    /* index = an edge        */
prev_location = cur_location >> 1;             /* keep A->B != B->A      */
```

`shared_mem` is a **64 kB** array shared with the fuzzer: one byte per edge tuple.

The shift by one keeps the encoding asymmetric — without it A→B and B→A would collide, since XOR is commutative.

Technical whitepaper for afl-fuzz

How AFL Obtains Coverage

The instrumentation injected at every branch point is equivalent to

```
cur_location = <COMPILE_TIME_RANDOM>;          /* one id per basic block */
shared_mem[cur_location ^ prev_location]++;    /* index = an edge        */
prev_location = cur_location >> 1;             /* keep A->B != B->A      */
```

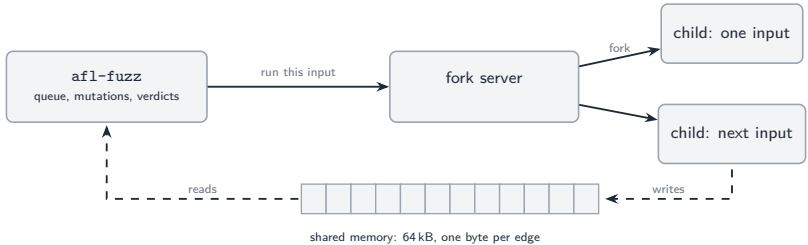
`shared_mem` is a **64 kB** array shared with the fuzzer: one byte per edge tuple.

The shift by one keeps the encoding asymmetric — without it `A`→`B` and `B`→`A` would collide, since XOR is commutative.

Collisions still occur: with 10 000 branch points about 7% of tuples collide.

Technical whitepaper for afl-fuzz

How AFL Runs the Target



Dynamic linking and libc start-up **once**

Technical whitepaper for afl-fuzz

AFL Instrumentation Example

```
int classify(int x) {           A   id 0x24
    if (x < 0)  return -1;      B   id 0x7b
    if (x == 0) return  0;      C   id 0x11   D  id 0x5c
    return 1;                   E   id 0x33
}
```

At the start of each basic block, the compiler inserts three instructions containing the block's identifier:

```
map[prev ^ 0x24]++;  prev = 0x24 >> 1;      /* block A */
```

Technical whitepaper for afl-fuzz

AFL Instrumentation Example

```
int classify(int x) {           A   id 0x24
    if (x < 0)  return -1;      B   id 0x7b
    if (x == 0) return  0;      C   id 0x11   D  id 0x5c
    return 1;                   E   id 0x33
}
```

At the start of each basic block, the compiler inserts three instructions containing the block's identifier:

```
map[prev ^ 0x24]++;  prev = 0x24 >> 1;      /* block A */
```

The identifiers are chosen at random at compile time, so no global numbering across translation units is needed.

Technical whitepaper for afl-fuzz

AFL Coverage Example

Input `x = -5` takes `A` → `B`. The instrumentation runs twice:

block	prev	cur	prev ^ cur	effect
A	0x00	0x24	0x24	map[0x24]++, then prev = 0x12
B	0x12	0x7b	0x69	map[0x69]++, then prev = 0x3d

6 of 65 536 bytes shown

AFL Coverage Example

Input `x = -5` takes `A` $\rightarrow$ `B`. The instrumentation runs twice:

block	prev	cur	prev ^ cur	effect
A	0x00	0x24	0x24	map[0x24]++, then prev = 0x12
B	0x12	0x7b	0x69	map[0x69]++, then prev = 0x3d

A run produces an array of counters. Two runs are compared by comparing these arrays.

Technical whitepaper for afl-fuzz

AFL Coverage Example: A Second Input

Input `x = 0` takes `A` → `C` → `D`:

block	prev	cur	index	
A	0x00	0x24	0x24	already seen
C	0x12	0x11	0x03	new
D	0x08	0x5c	0x54	new

two bytes were 0 before

⇒ queue the input

AFL Coverage Example: A Second Input

Input `x = 0` takes `A` → `C` → `D`:

block	prev	cur	index	
A	0x00	0x24	0x24	already seen
C	0x12	0x11	0x03	new
D	0x08	0x5c	0x54	new

two bytes were 0 before

⇒ queue the input

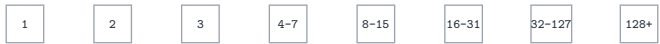
The fuzzer keeps a **global map** of every index seen so far. An input is retained when its trace sets an index the global map has not recorded.

Hit Counts

A byte per edge holds a count. The counts are read through eight buckets:

Hit Counts

A byte per edge holds a count. The counts are read through eight buckets:



- a move **from one bucket to another** counts as new behaviour.

Technical whitepaper for afl-fuzz

Hit Counts

A byte per edge holds a count. The counts are read through eight buckets:

- a move **from one bucket to another** counts as new behaviour.
- a change **within** a bucket is ignored.

Collisions

Identifiers are random, the map is finite, so two different edges can land on the same byte:

```
edge A -> B :  prev = 0x24 >> 1 = 0x12    0x12 ^ 0x7b = 0x69
edge P -> Q :  prev = 0x40 >> 1 = 0x20    0x20 ^ 0x49 = 0x69
```

- With a 64 kB map one collision is expected at about **256** instrumented blocks;
- a real target has **10 000–50 000**, giving on the order of **750–18 000** colliding pairs. A colliding edge is an edge the fuzzer cannot see.

AFL++ LLVM instrumentation documentation

Collisions

Identifiers are random, the map is finite, so two different edges can land on the same byte:

```
edge A -> B :  prev = 0x24 >> 1 = 0x12    0x12 ^ 0x7b = 0x69
edge P -> Q :  prev = 0x40 >> 1 = 0x20    0x20 ^ 0x49 = 0x69
```

- With a 64 kB map one collision is expected at about **256** instrumented blocks;
- a real target has **10 000–50 000**, giving on the order of **750–18 000** colliding pairs. A colliding edge is an edge the fuzzer cannot see.

AFL++ LLVM instrumentation documentation

Collisions

Identifiers are random, the map is finite, so two different edges can land on the same byte:

```
edge A -> B :  prev = 0x24 >> 1 = 0x12    0x12 ^ 0x7b = 0x69
edge P -> Q :  prev = 0x40 >> 1 = 0x20    0x20 ^ 0x49 = 0x69
```

- With a 64 kB map one collision is expected at about **256** instrumented blocks;
- a real target has **10 000–50 000**, giving on the order of **750–18 000** colliding pairs. A colliding edge is an edge the fuzzer cannot see.

AFL++ removes the collisions by numbering the blocks at **link time**, when every translation unit is present:

```
afl-clang-lto ...
[+] Instrumented 12071 locations with no collisions
    (on average 1046 collisions would be in afl-clang-fast CLASSIC)
```

American Fuzzy Lop plus plus (AFL++)

Release version: 5.02c

GitHub version: 5.03a

Repository: https://github.com/AFLplusplus/AFLplusplus

AFL++ is maintained by:

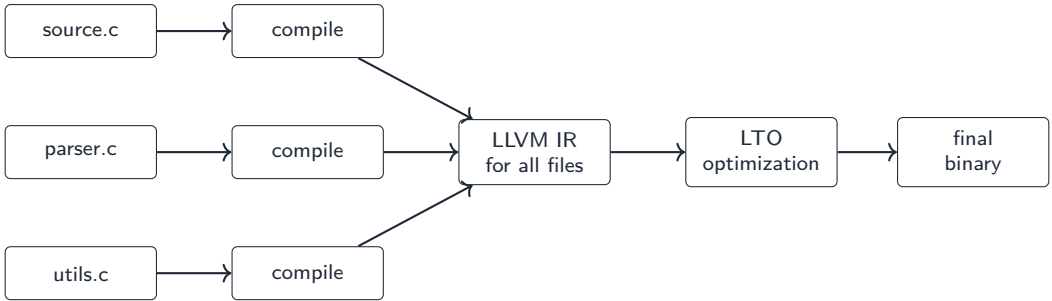
- Marc "van Hauser" Heuse mh@mh-sec.de
- Dominik Maier mail@dmnk.co
- Andrea Fioraldi andreafioraldi@gmail.com
- Heiko "hexcoder-" Eissfeldt heiko.eissfeldt@hexco.de
- frida_mode is maintained by @Worksbutnottested

Originally developed by Michal "lcamtuf" Zalewski.

AFL++ is a superior fork to Google's AFL - more speed, more and better mutations, more and better instrumentation, custom module support, etc.

AFL++ is licensed under the **AGPL-3.0-or-later**, and it also contains files under the Apache-2.0 License. **Everything compiled into a fuzzing harness is and will stay Apache 2.0 licensed.** Each file states its own license in its `SPDX-License-Identifier` header — that is the license you must follow for that file. An optional **commercial license** is available for organizations that cannot use the AGPL (obtained by donating to a good cause — the project and its maintainers receive no money). See LICENSING.md for a plain-language overview and the License section below for details.

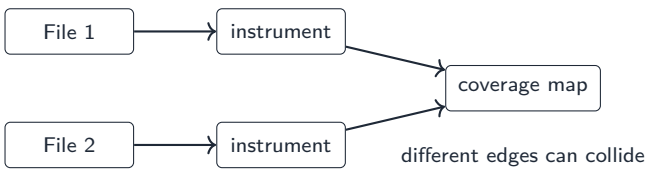
LLVM Link-Time Optimization (LTO)



https://llvm.org/docs/LinkTimeOptimization.html

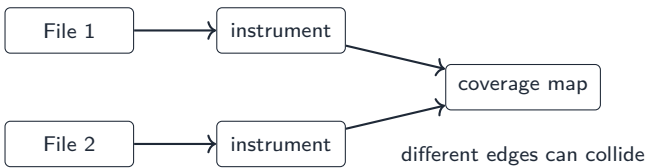
AFL++: Removing Coverage Collisions

Classic instrumentation: files are instrumented separately



AFL++: Removing Coverage Collisions

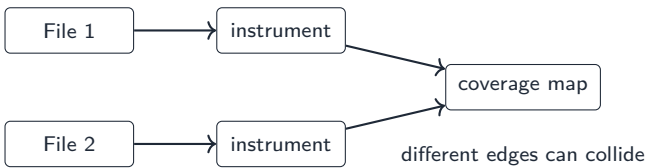
Classic instrumentation: files are instrumented separately



AFL++ LTO: instrument after all files are available

AFL++: Removing Coverage Collisions

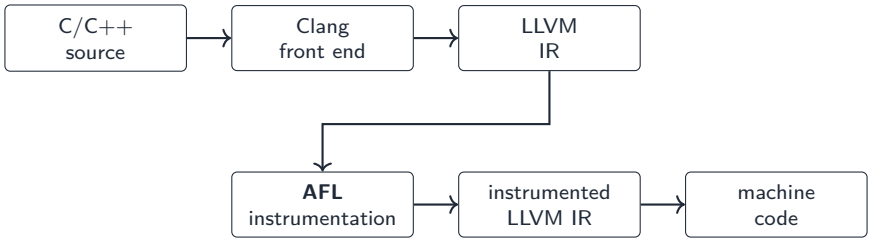
Classic instrumentation: files are instrumented separately



AFL++ LTO: instrument after all files are available

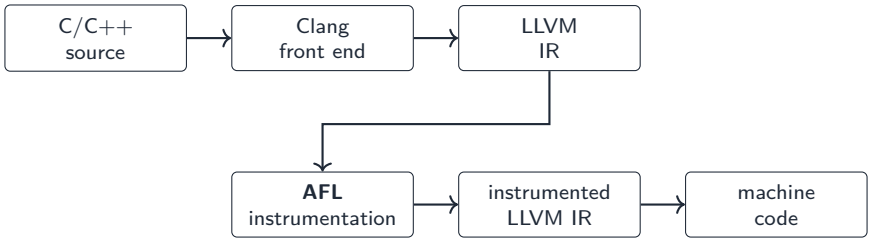
12,071 locations → no coverage collisions

AFL Instrumentation



AFL++ LLVM instrumentation documentation

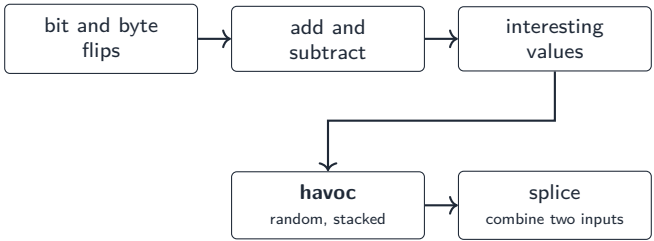
AFL Instrumentation



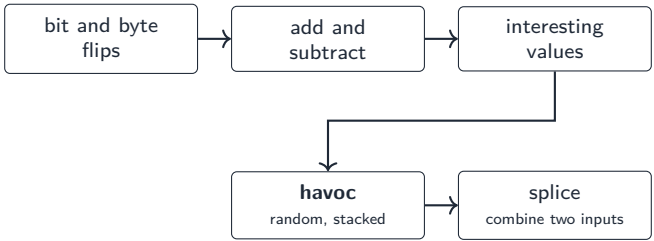
The instrumentation pass inserts coverage-tracking code into the IR.

AFL++ LLVM instrumentation documentation

The Mutation Pipeline

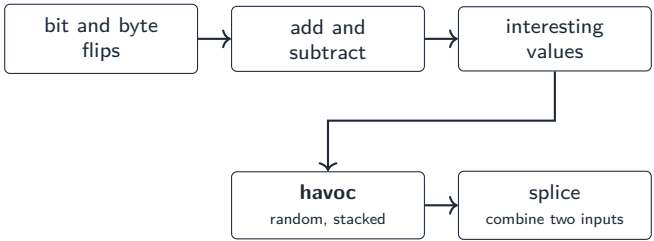


The Mutation Pipeline



Deterministic stages: systematically try predefined mutations.

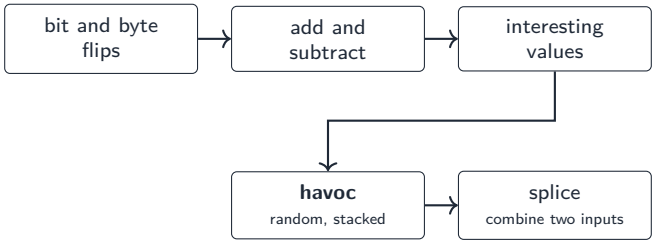
The Mutation Pipeline



Deterministic stages: systematically try predefined mutations.

Havoc: apply a random sequence of mutations.

The Mutation Pipeline



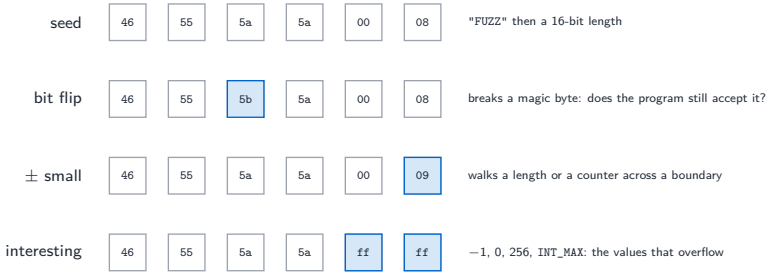
Deterministic stages: systematically try predefined mutations.

Havoc: apply a random sequence of mutations.

Splice: combine two queue entries and continue fuzzing.

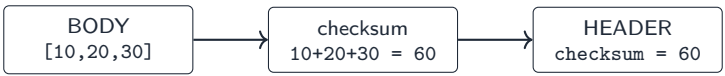
Mutation Example

A queue entry, six bytes of a header with a length field:

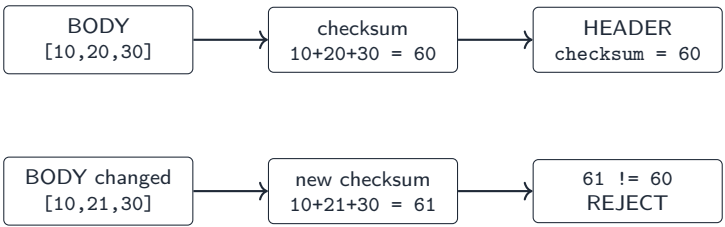


Technical whitepaper for afl-fuzz

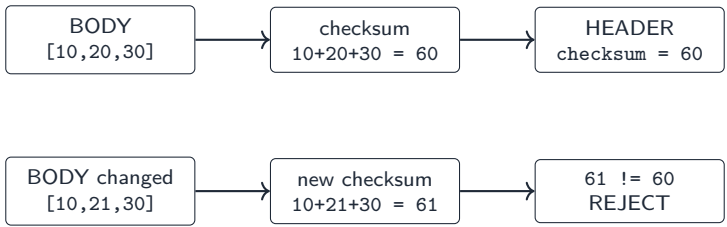
A Simple Checksum



A Simple Checksum



A Simple Checksum



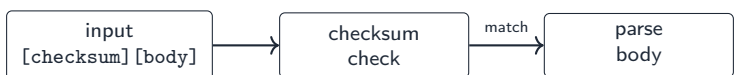
Change the body → checksum changes

Checksum Check in a Parser

```
int parse(const uint8_t *data, size_t len)
{
    uint8_t checksum = data[0];
    const uint8_t *body = &data[1];

    if (sum(body, len - 1) != checksum)
        return REJECT;

    return parse_body(body, len - 1);
}
```

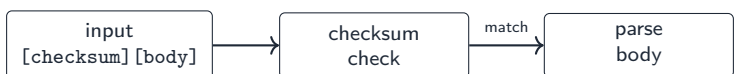


Checksum Check in a Parser

```
int parse(const uint8_t *data, size_t len)
{
    uint8_t checksum = data[0];
    const uint8_t *body = &data[1];

    if (sum(body, len - 1) != checksum)
        return REJECT;

    return parse_body(body, len - 1);
}
```



mismatch → **REJECT**

Where Mutation Fails

```
if (crc32(body, len) != header_checksum)
    return REJECT;

parse(body, len);
```

Where Mutation Fails

```
if (crc32(body, len) != header_checksum)
    return REJECT;

parse(body, len);
```

body changed → CRC changes
stored checksum remains unchanged

Where Mutation Fails

```
if (crc32(body, len) != header_checksum)
    return REJECT;

parse(body, len);
```

body changed → CRC changes
stored checksum remains unchanged

Most mutations therefore never reach `parse()`.

Grey-box on the Same Target

```
$ ./greybox_raw 200000            # seed: "aaaaaaaaaaaaaaaa"
     461 execs  new coverage, corpus=  2  input="Faaaaa"
    2117 execs  new coverage, corpus=  3  input="FUaaaa"
   20032 execs  new coverage, corpus=  4  input="FUZaaa"
   44305 execs  new coverage, corpus=  5  input="FUZZaa"
   53236 execs  new coverage, corpus=  6  input="FUZZ!a"
done: 200000 execs, corpus 6 inputs
```

Grey-box on the Same Target

```
$ ./greybox_raw 200000            # seed: "aaaaaaaaaaaaaaaa"
     461 execs  new coverage, corpus=  2  input="Faaaaa"
    2117 execs  new coverage, corpus=  3  input="FUaaaa"
   20032 execs  new coverage, corpus=  4  input="FUZaaa"
   44305 execs  new coverage, corpus=  5  input="FUZZaa"
   53236 execs  new coverage, corpus=  6  input="FUZZ!a"
done: 200000 execs, corpus 6 inputs
```

The header is discovered one byte at a time: each correct byte opens a new edge, the input is queued, and subsequent mutations start from it.

Grey-box on the Same Target

```
$ ./greybox_raw 200000            # seed: "aaaaaaaaaaaaaaaa"
     461 execs  new coverage, corpus=  2  input="Faaaaa"
    2117 execs  new coverage, corpus=  3  input="FUaaaa"
   20032 execs  new coverage, corpus=  4  input="FUZaaa"
   44305 execs  new coverage, corpus=  5  input="FUZZaa"
   53236 execs  new coverage, corpus=  6  input="FUZZ!a"
done: 200000 execs, corpus 6 inputs
```

The header is discovered one byte at a time: each correct byte opens a new edge, the input is queued, and subsequent mutations start from it.

The black-box probability 256^{-5} is replaced by five independent steps of about 256 each: **53 236 executions** in this run.

Quiz: Which Inputs Are Interesting?

1. An input executes an edge no previous input executed. **Keep it?**

Technical whitepaper for afl-fuzz

Quiz: Which Inputs Are Interesting?

1. An input executes an edge no previous input executed. **Keep it?**

   Usually yes: it enlarges the explored region and is a starting point for further mutation.

Technical whitepaper for afl-fuzz

Quiz: Which Inputs Are Interesting?

1. An input executes an edge no previous input executed. **Keep it?**

   Usually yes: it enlarges the explored region and is a starting point for further mutation.

2. An input executes exactly the edges an existing input already executed. **Keep it?**

Technical whitepaper for afl-fuzz

Quiz: Which Inputs Are Interesting?

1. An input executes an edge no previous input executed. **Keep it?**

   Usually yes: it enlarges the explored region and is a starting point for further mutation.

2. An input executes exactly the edges an existing input already executed. **Keep it?**

   By coverage alone, no it adds no new coverage.

Technical whitepaper for afl-fuzz

Quiz: Which Inputs Are Interesting?

1. An input executes an edge no previous input executed. **Keep it?**

   Usually yes: it enlarges the explored region and is a starting point for further mutation.

2. An input executes exactly the edges an existing input already executed. **Keep it?**

   By coverage alone, no it adds no new coverage.

3. An input discovers a new edge but the program does not crash. **Is this a bug?**

Technical whitepaper for afl-fuzz

Quiz: Which Inputs Are Interesting?

1. An input executes an edge no previous input executed. **Keep it?**

   Usually yes: it enlarges the explored region and is a starting point for further mutation.

2. An input executes exactly the edges an existing input already executed. **Keep it?**

   By coverage alone, no it adds no new coverage.

3. An input discovers a new edge but the program does not crash. **Is this a bug?**

   No. It is new coverage, which is not the same thing as a failure.

Technical whitepaper for afl-fuzz

Test Oracles

The Test Oracle Problem

Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

The Test Oracle Problem

The **test oracle** decides whether the observed behaviour is acceptable.

Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

The Test Oracle Problem

The **test oracle** decides whether the observed behaviour is acceptable.

coverage feedback	*did we explore something new?*
test oracle	*did something go wrong?*

Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

The Test Oracle Problem

The **test oracle** decides whether the observed behaviour is acceptable.

coverage feedback	*did we explore something new?*
test oracle	*did something go wrong?*

Two different questions. A fuzzer needs both.

Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

Four Kinds of Oracle

Explicit

The expected result is known

```
assert(add(2,3) == 5);
```

Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

Four Kinds of Oracle

Explicit

The expected result is known

```
assert(add(2,3) == 5);
```

Differential

Two implementations should agree

```
impl1(p) == impl2(p)
```

Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

Four Kinds of Oracle

Explicit

The expected result is known

```
assert(add(2,3) == 5);
```

Differential

Two implementations should agree

```
impl1(p) == impl2(p)
```

Metamorphic

A relation between executions must hold

```
sort(sort(x)) == sort(x)
```

Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

Four Kinds of Oracle

Explicit

The expected result is known

```
assert(add(2,3) == 5);
```

Differential

Two implementations should agree

```
impl1(p) == impl2(p)
```

Metamorphic

A relation between executions must hold

```
sort(sort(x)) == sort(x)
```

Implicit

Detect obviously bad behaviour

crash, assertion, timeout, sanitizer report

Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

Four Kinds of Oracle

Explicit

The expected result is known

```
assert(add(2,3) == 5);
```

Differential

Two implementations should agree

```
impl1(p) == impl2(p)
```

Metamorphic

A relation between executions must hold

```
sort(sort(x)) == sort(x)
```

Implicit

Detect obviously bad behaviour

crash, assertion, timeout, sanitizer report

In fuzzing, implicit oracles are especially useful because they require no expected output for each generated input.

Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

Sanitizers

Builds and Bug Detection With Sanitizer

One input, `"FUZZ!aaaaaaaaaaa"`, reaches `memcpy(buf, d, 12)` with `char buf[8]`.

```
$ clang -O0 -fno-stack-protector \
    target.c -o poc_raw
$ ./poc_raw
parse("FUZZ!aaaaaaaaaaa") = 6
no failure reported
```

Builds and Bug Detection With Sanitizer

One input, `"FUZZ!aaaaaaaaaaa"`, reaches `memcpy(buf, d, 12)` with `char buf[8]`.

```
$ clang -O0 -fno-stack-protector \
    target.c -o poc_raw
$ ./poc_raw
parse("FUZZ!aaaaaaaaaaa") = 6
no failure reported
```

```
$ clang -O2 target.c -o poc_hard
$ ./poc_hard
process terminated   # exit status 133
```

Builds and Bug Detection With Sanitizer

One input, `"FUZZ!aaaaaaaaaaa"`, reaches `memcpy(buf, d, 12)` with `char buf[8]`.

```
$ clang -O0 -fno-stack-protector \
    target.c -o poc_raw
$ ./poc_raw
parse("FUZZ!aaaaaaaaaaa") = 6
no failure reported
```

```
$ clang -O2 target.c -o poc_hard
$ ./poc_hard
process terminated   # exit status 133
```

```
$ clang -O1 -fsanitize=address target.c -o poc_asan
$ ./poc_asan
ERROR: AddressSanitizer:  stack-buffer-overflow
WRITE of size 12
    #1 copy_input target.c:12
    #2 parse       target.c:23
    [32, 40) 'buf' <== overflows this variable
```

Builds and Bug Detection With Sanitizer

One input, `"FUZZ!aaaaaaaaaaa"`, reaches `memcpy(buf, d, 12)` with `char buf[8]`.

```
$ clang -O0 -fno-stack-protector \
    target.c -o poc_raw
$ ./poc_raw
parse("FUZZ!aaaaaaaaaaa") = 6
no failure reported
```

```
$ clang -O2 target.c -o poc_hard
$ ./poc_hard
process terminated   # exit status 133
```

```
$ clang -O1 -fsanitize=address target.c -o poc_asan
$ ./poc_asan
ERROR: AddressSanitizer:  stack-buffer-overflow
WRITE of size 12
    #1 copy_input target.c:12
    #2 parse       target.c:23
    [32, 40) 'buf' <== overflows this variable
```

Same source + same input, but different build instrumentation.

Fuzzing Without and With a Sanitizer

The same fuzzer, the same seed, the same 200 000 executions.

```
$ ./greybox_raw 200000
   53236 execs  new coverage, corpus=  6  input="FUZZ!a"
done: 200000 execs, corpus 6 inputs
BUG EXECUTED - NO FAILURE REPORTED
```

Fuzzing Without and With a Sanitizer

The same fuzzer, the same seed, the same 200 000 executions.

```
$ ./greybox_raw 200000
   53236 execs  new coverage, corpus=  6  input="FUZZ!a"
done: 200000 execs, corpus 6 inputs
BUG EXECUTED - NO FAILURE REPORTED
```

```
$ ./greybox_asan 200000
   44305 execs  new coverage, corpus=  5  input="FUZZaa"

==76586==ERROR: AddressSanitizer:
stack-buffer-overflow
WRITE of size 12 ... in copy_input target.c:12
BUG FOUND
```

Fuzzing Without and With a Sanitizer

The same fuzzer, the same seed, the same 200 000 executions.

```
$ ./greybox_raw 200000
   53236 execs  new coverage, corpus=  6  input="FUZZ!a"
done: 200000 execs, corpus 6 inputs
BUG EXECUTED - NO FAILURE REPORTED
```

```
$ ./greybox_asan 200000
   44305 execs  new coverage, corpus=  5  input="FUZZaa"

==76586==ERROR: AddressSanitizer:
stack-buffer-overflow
WRITE of size 12 ... in copy_input target.c:12
BUG FOUND
```

Coverage finds the input; the sanitizer makes the failure observable.

The Sanitizer Family

Sanitizer	Detects	Flag
ASan	buffer overflows, use-after-free	`-fsanitize=address`
UBSan	undefined behaviour: overflow, bad shifts, ...	`-fsanitize=undefined`
MSan	reads of uninitialised memory	`-fsanitize=memory`
TSan	data races	`-fsanitize=thread`
LSan	memory leaks	`-fsanitize=leak`

Serebryany et al., *AddressSanitizer*, USENIX ATC 2012

The Sanitizer Family

Sanitizer	Detects	Flag
ASan	buffer overflows, use-after-free	`-fsanitize=address`
UBSan	undefined behaviour: overflow, bad shifts, ...	`-fsanitize=undefined`
MSan	reads of uninitialised memory	`-fsanitize=memory`
TSan	data races	`-fsanitize=thread`
LSan	memory leaks	`-fsanitize=leak`

Each sanitizer converts a class of otherwise unobservable defects into a runtime failure.

Serebryany et al., *AddressSanitizer*, USENIX ATC 2012

The Sanitizer Family

Sanitizer	Detects	Flag
ASan	buffer overflows, use-after-free	`-fsanitize=address`
UBSan	undefined behaviour: overflow, bad shifts, ...	`-fsanitize=undefined`
MSan	reads of uninitialised memory	`-fsanitize=memory`
TSan	data races	`-fsanitize=thread`
LSan	memory leaks	`-fsanitize=leak`

Each sanitizer converts a class of otherwise unobservable defects into a runtime failure.

ASan and **MSan** cannot be combined: They are not mutually compatible

Serebryany et al., *AddressSanitizer*, USENIX ATC 2012

What It Costs

An average slowdown of **73%** and about **3.4**× the memory.

```
$ make bench          # 40,000 executions of the same target
no sanitizer:       real 0.68 s        ~ 59,000 execs/s
AddressSanitizer:   real 1.12 s        ~ 36,000 execs/s      1.65x slower
```

Serebryany et al., *AddressSanitizer*, USENIX ATC 2012 · `fuzz-demo/` in the lecture repository

What It Costs

An average slowdown of **73%** and about **3.4**× the memory.

```
$ make bench          # 40,000 executions of the same target
no sanitizer:       real 0.68 s        ~ 59,000 execs/s
AddressSanitizer:   real 1.12 s        ~ 36,000 execs/s      1.65x slower
```

Throughput determines how many inputs are tried per unit time.

Serebryany et al., *AddressSanitizer*, USENIX ATC 2012 · `fuzz-demo/` in the lecture repository

What It Costs

An average slowdown of **73%** and about **3.4×** the memory.

```
$ make bench          # 40,000 executions of the same target
no sanitizer:       real 0.68 s        ~ 59,000 execs/s
AddressSanitizer:   real 1.12 s        ~ 36,000 execs/s      1.65x slower
```

Throughput determines how many inputs are tried per unit time.

The trade-off: more checks means better detection and fewer executions; fewer checks means more executions and more missed bugs.

Serebryany et al., *AddressSanitizer*, USENIX ATC 2012 · `fuzz-demo/` in the lecture repository

What It Costs

An average slowdown of **73%** and about **3.4×** the memory.

```
$ make bench          # 40,000 executions of the same target
no sanitizer:       real 0.68 s        ~ 59,000 execs/s
AddressSanitizer:   real 1.12 s        ~ 36,000 execs/s      1.65x slower
```

Throughput determines how many inputs are tried per unit time.

The trade-off: more checks means better detection and fewer executions; fewer checks means more executions and more missed bugs.

Common practice: fuzz a fast build to find inputs, then replay the corpus against sanitizer builds.

Serebryany et al., *AddressSanitizer*, USENIX ATC 2012 · `fuzz-demo/` in the lecture repository

Questions

1. **Is coverage a test oracle?**

Questions

1. **Is coverage a test oracle?**

   No. Coverage says what was explored, never whether the behaviour was correct.

Questions

1. **Is coverage a test oracle?**

   No. Coverage says what was explored, never whether the behaviour was correct.

2. **If ASan reports an error, do we still need to know the expected output?**

Questions

1. **Is coverage a test oracle?**

   No. Coverage says what was explored, never whether the behaviour was correct.

2. **If ASan reports an error, do we still need to know the expected output?**

   Not for that class of bug: the sanitizer is an implicit oracle. A wrong *result* with no memory error still needs one.

Questions

1. **Is coverage a test oracle?**

   No. Coverage says what was explored, never whether the behaviour was correct.

2. **If ASan reports an error, do we still need to know the expected output?**

   Not for that class of bug: the sanitizer is an implicit oracle. A wrong *result* with no memory error still needs one.

3. **Why not fuzz with every sanitizer enabled at once?**

Questions

1. **Is coverage a test oracle?**

   No. Coverage says what was explored, never whether the behaviour was correct.

2. **If ASan reports an error, do we still need to know the expected output?**

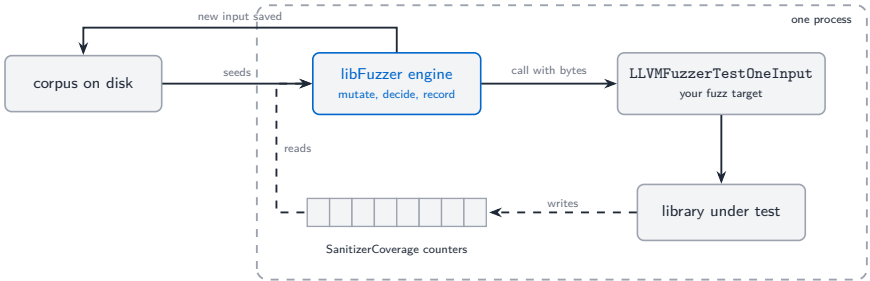
   Not for that class of bug: the sanitizer is an implicit oracle. A wrong *result* with no memory error still needs one.

3. **Why not fuzz with every sanitizer enabled at once?**

   Overhead reduces throughput, and several sanitizers cannot be combined in one binary.

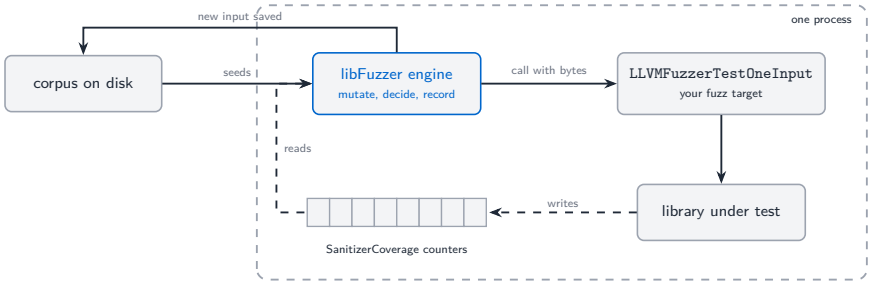
Engines in Practice

libFuzzer: Fuzzing Inside the Process



LLVM, *libFuzzer: a library for coverage-guided fuzz testing*

libFuzzer: Fuzzing Inside the Process



An **in-process, coverage-guided, evolutionary** engine linked into the target: no process is forked per input, so a single execution costs a function call.

LLVM, *libFuzzer: a library for coverage-guided fuzz testing*

AFL++ and libFuzzer Compared

	AFL++	**libFuzzer**
what you fuzz	a whole program	a function you write
process model	fork server, one child per input	in process, one call per input
coverage	64 kB hashed map (LTO: collision-free)	SanitizerCoverage counters
a crash	kills the child, the fuzzer continues	kills the process, restart from corpus
binary-only targets	QEMU, Frida, Unicorn modes	not supported
typical use	command-line tools, whole systems	libraries and APIs

LLVM, *libFuzzer: a library for coverage-guided fuzz testing* · AFL++ LLVM instrumentation documentation

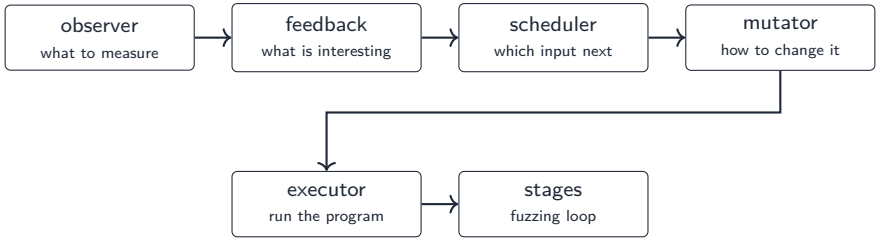
AFL++ and libFuzzer Compared

	AFL++	**libFuzzer**
what you fuzz	a whole program	a function you write
process model	fork server, one child per input	in process, one call per input
coverage	64 kB hashed map (LTO: collision-free)	SanitizerCoverage counters
a crash	kills the child, the fuzzer continues	kills the process, restart from corpus
binary-only targets	QEMU, Frida, Unicorn modes	not supported
typical use	command-line tools, whole systems	libraries and APIs

The fuzz target is the same function in both cases: a target written for libFuzzer can be run by AFL++.

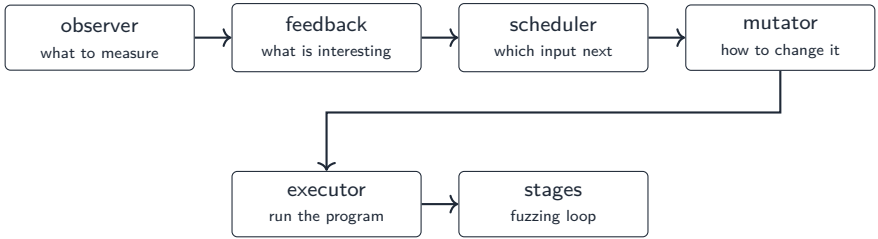
LLVM, *libFuzzer: a library for coverage-guided fuzz testing* · AFL++ LLVM instrumentation documentation

LibAFL: Build a Fuzzer from Parts



Fioraldi, Maier, Zhang & Balzarotti, *LibAFL: A Framework to Build Modular and Reusable Fuzzers*, CCS 2022

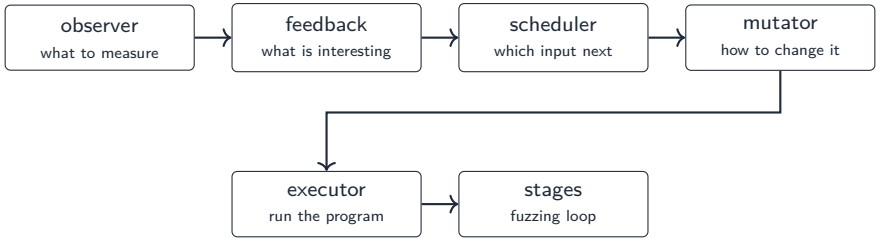
LibAFL: Build a Fuzzer from Parts



LibAFL is a library for assembling these components.

Fioraldi, Maier, Zhang & Balzarotti, *LibAFL: A Framework to Build Modular and Reusable Fuzzers*, CCS 2022

LibAFL: Build a Fuzzer from Parts

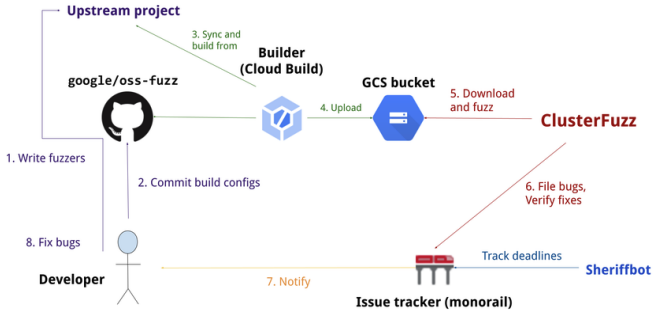


LibAFL is a library for assembling these components.

Change one component without rewriting the whole fuzzer.

Fioraldi, Maier, Zhang & Balzarotti, *LibAFL: A Framework to Build Modular and Reusable Fuzzers*, CCS 2022

OSS-Fuzz: Fuzzing as Infrastructure



OSS-Fuzz provides continuous fuzzing infrastructure for open-source projects.

https://github.com/google/oss-fuzz