AFL++: Combing Incremental Steps of Fuzzing Research

In 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association, Aug. 2020.

刘冯润

2021/04/08

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*

Background

Intro

Cutting Edge

AFL++

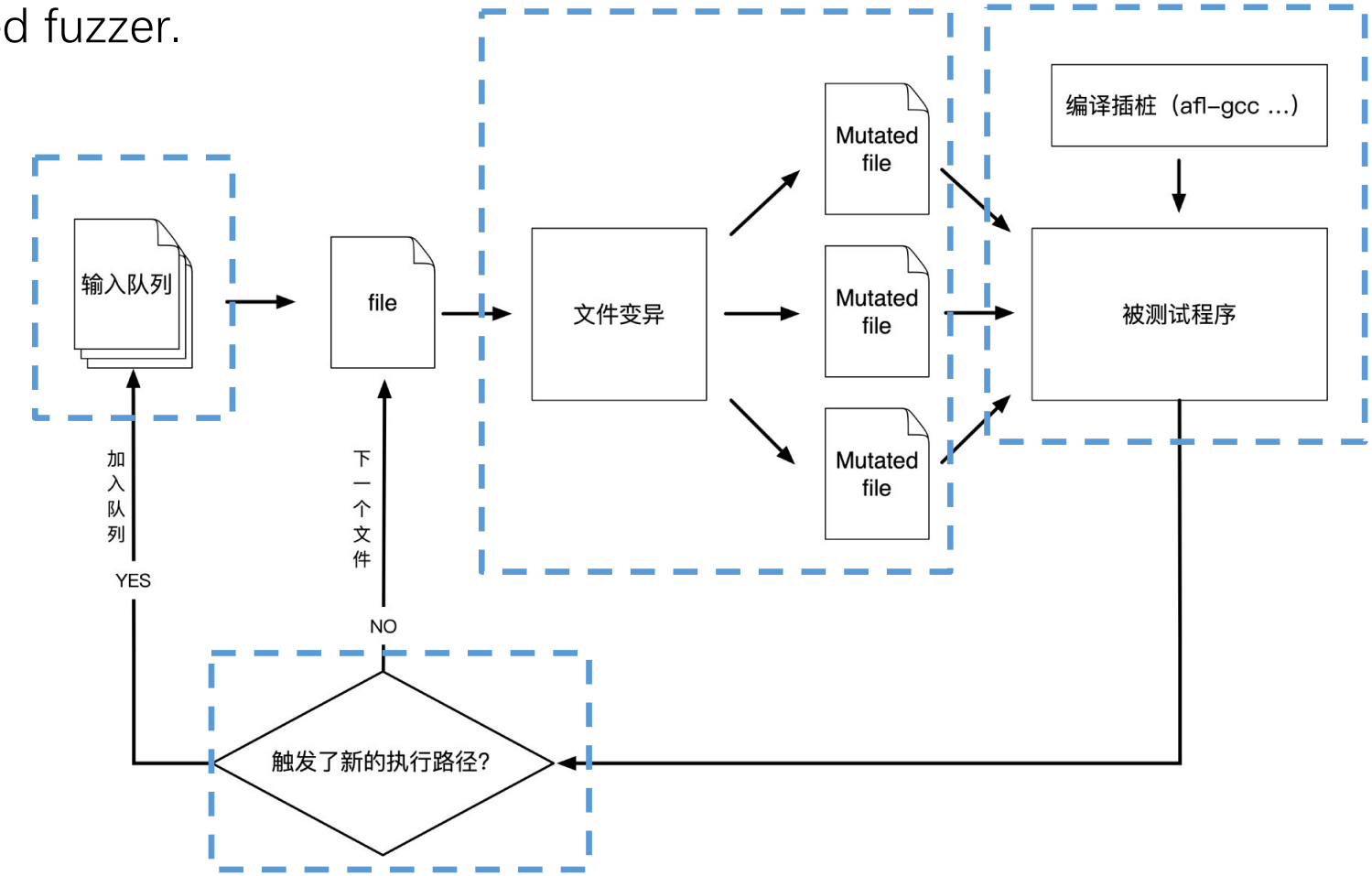
Evaluation

AFL (@lcamtuf):

A mutational, coverage-guided fuzzer.

1. Instrumentation
2. queue
3. mutations to file
4. trigger new path
5. go to 2

maximize code coverage



https://www.freebuf.com/articles/system/191536.html

Challenges && Contributions

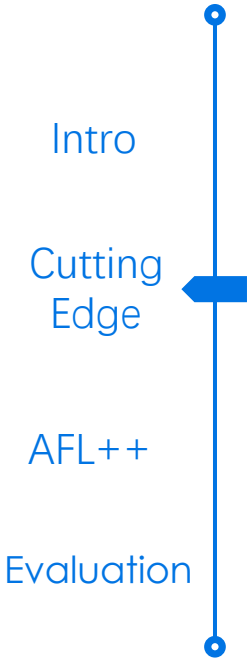
Challenges:

1. Combining state-of-the-art fuzzing techniques is hard.
2. Evaluating combinations is hard.

Contributions:

1. a usable tool, incorporating recent fuzzing research.
2. novel Custom Mutator API, easy to implement and combine.
3. evaluate incorporated technologies, show target-dependence.

State-of-the-Art



Overview:

- **American Fuzzy Lop**
- **Smart Scheduling**
  - AFLFast: Seed Scheduling
  - MOpt: Mutation Scheduling
- **Bypassing Roadblocks**
  - LAF-Intel
  - RedQueen
- **Mutate Structured Inputs**
  - AFLSmart

State-of-the-Art

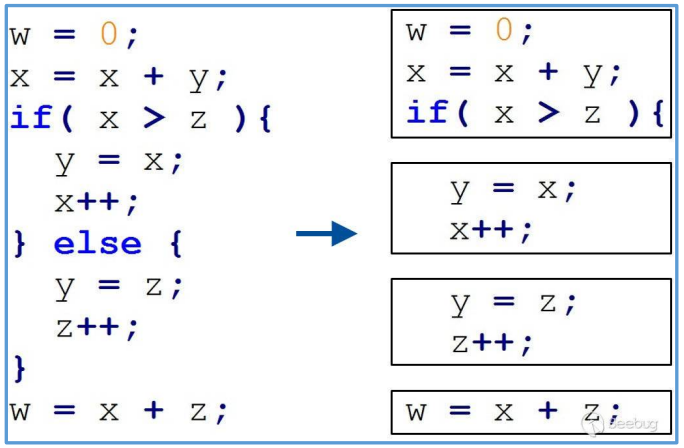
Cutting Edge

- AFL
- AFLFast
- MOpt
- LAF-Intel
- RedQueen
- AFLSmart

AFL:

- Coverage Guided Feedback
- Mutations
- Forkserver
- Persistent Mode

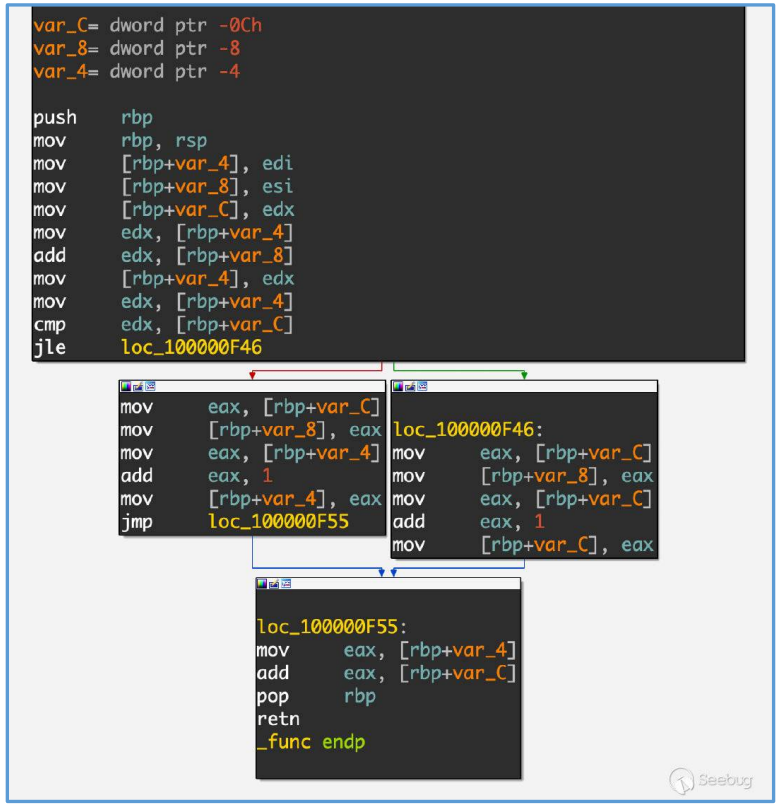
Block:



https://paper.seebug.org/842/

https://paper.seebug.org/842/

Edge:



Code Coverage:

A -> B -> C -> D -> E (tuples: AB, BC, CD, DE)

A -> B -> D -> C -> E (tuples: AB, BD, DC, CE)

State-of-the-Art

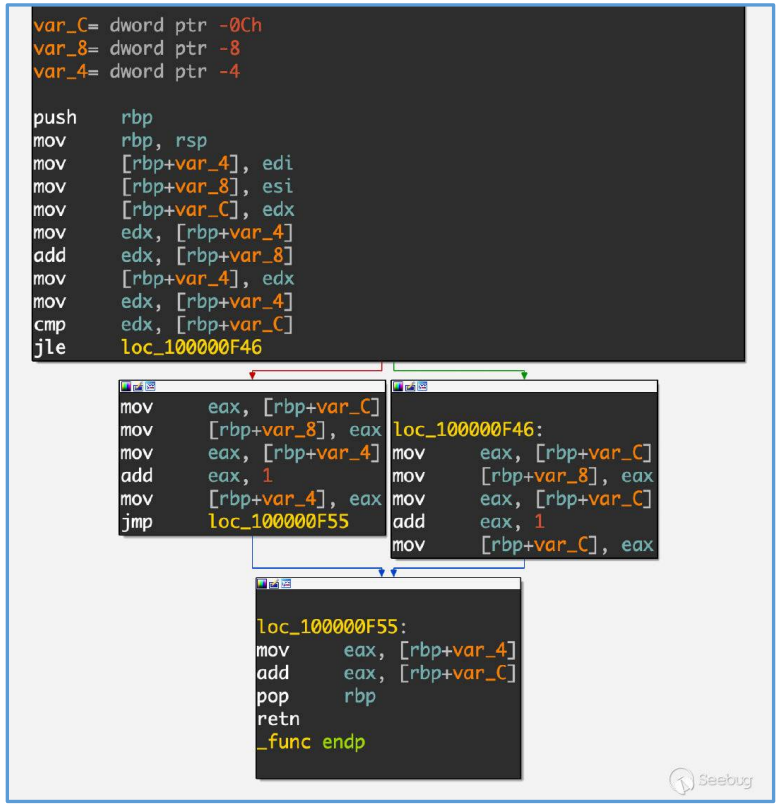
Cutting Edge

- AFL
- AFLFast
- MOpt
- LAF-Intel
- RedQueen
- AFLSmart

AFL:

- Coverage Guided Feedback
- Mutations
- Forkserver
- Persistent Mode

Edge:



Instrumentation:

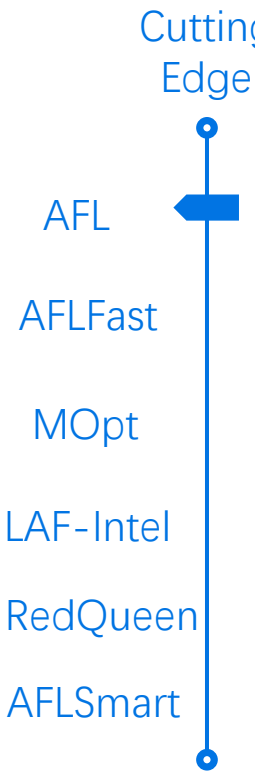
```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Code Coverage:

A -> B -> C -> D -> E (tuples: AB, BC, CD, DE)

A -> B -> D -> C -> E (tuples: AB, BD, DC, CE)

State-of-the-Art



AFL:

- Coverage Guided Feedback
- Mutations
- Forkserver
- Persistent Mode

deterministic stage:

bit flips

addition and subtraction

insertion ...

havoc stage: a stack of mutations

State-of-the-Art

Cutting Edge

AFL

AFLFast

MOpt

LAF-Intel

RedQueen

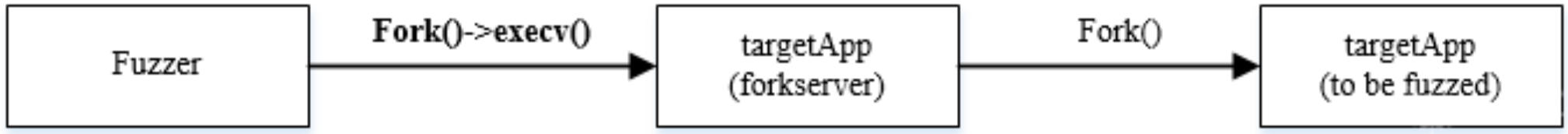
AFLSmart

AFL：

- Coverage Guided Feedback
- Mutations
- Forkserver
- Persistent Mode

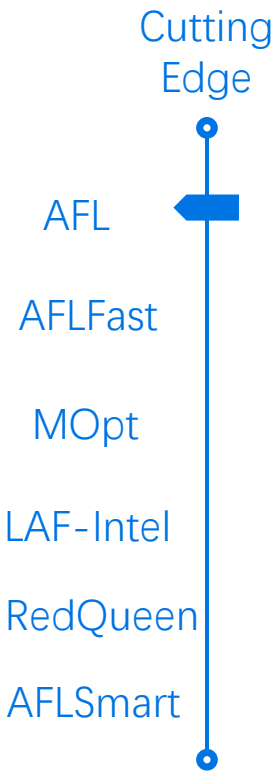
Advantage：The fuzzed process goes through execv(), linking, and libc initialization only once.

The **exec**() family of functions replaces the current process image with a new process image.



https://bbs.pediy.com/thread-254705.htm

State-of-the-Art



AFL:

- Coverage Guided Feedback
- Mutations
- Forkserver
- Persistent Mode

Patch a loop into the target:

```
int main(int argc, char** argv) {

  while (__AFL_LOOP(1000)) {

    /* Reset state. */
    memset(buf, 0, 100);

    /* Read input data. */
    read(0, buf, 100);

    /* Parse it in some vulnerable way. You'd normally call a library here. */
    if (buf[0] != 'p') puts("error 1"); else
    if (buf[1] != 'w') puts("error 2"); else
    if (buf[2] != 'n') puts("error 3"); else
      abort();

  }

}
```

https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html

State-of-the-Art: Seed Scheduling

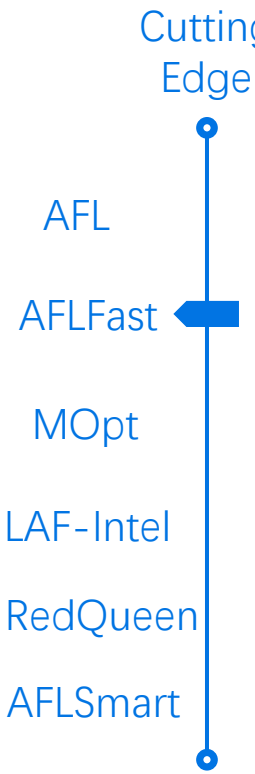
AFLFast：

Contributions:

- observed that most generated inputs exercise the same few "high-frequency" paths.
- developed strategies to stress low-frequency paths.

- **Search Strategy** decides the **order** of the fuzzer pick the seeds
- **Power Schedules** decides the amount of generated inputs from each seed (the seed’ s **energy**)

State-of-the-Art: Seed Scheduling



AFLFast：

- **Search Strategy** decides the order of the fuzzer pick the seeds
- **Power Schedules** decides the amount of generated inputs from each seed (the seed’ s **energy**)

AFL:

- order: update_bitmap_score: fav_factor ← exec_us*len
  - smaller fav_factor means more favored
- energy: calculate_score: perf_score ←
  - larger perf_score means more energy

exec_us: execution time
bitmap_size: number of bits set in bitmap
handicap: number of queue cycles behind
depth: path depth

State-of-the-Art: Seed Scheduling

AFLFast:

AFLFast:

- order: update_bitmap_score: s(i), f(i), fav_factor ← exec_us*len
  - smaller means more favored

seed i:

f(i): total number of being fuzzed (frequency)
s(i): number of pick

- energy: calculate_score: pref_score
  - larger pref_score means more energy

exec_us: execution time
bitmap_size: number of bits set in bitmap
handicap: number of queue cycles behind
depth: path depth

energy : the amount of generated inputs from each seed

State-of-the-Art: Seed Scheduling

Cutting Edge

- AFL
- AFLFast
- MOpt
- LAF-Intel
- RedQueen
- AFLSmart

AFLFast:

AFLFast Power Schedules: p(i) = energy

1. EXPLOIT: p(i) = AFL
2. EXPLORE: p(i) = AFL / const

$$p(i) = \alpha(i)$$

$$p(i) = \frac{\alpha(i)}{\beta}$$

```
1 void crashme (char* s) {
2   if (s[0] == 'b')
3     if (s[1] == 'a')
4       if (s[2] == 'd')
5         if (s[3] == '!')
6           abort();
7 }
```

#Total Tests	State	Explored States
1	****	****
$2^{16}+1$	b***	****, b***
$2\cdot 2^{16}+1$	ba**	****, b***, ba**
$3\cdot 2^{16}+1$	bad*	****, b***, ba**, bad*
$4\cdot 2^{16}+1$	bad!	****, b***, ba**, bad*, bad!

Figure 3: The crash is found after $2^{18} = 256k$ inputs were generated by fuzzing when $p = 2^{16}$ is constant.

energy : the amount of generated inputs from each seed

more energy than is required in expectation

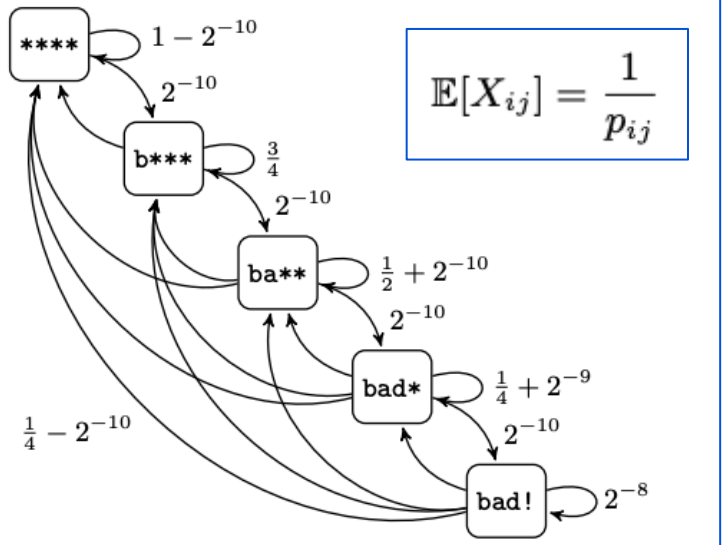


Figure 2: Markov chain for motivating example

$$\mathbb{E}[X_{01}]+\mathbb{E}[X_{12}]+\mathbb{E}[X_{23}]+\mathbb{E}[X_{34}] = 4\cdot 2^{10} = 4k$$

State-of-the-Art: Seed Scheduling

Cutting Edge

AFL

AFLFast

MOpt

LAF-Intel

RedQueen

AFLSmart

AFLFast:

AFLFast Power Schedules: p(i) = energy

f(i): total number of being fuzzed (frequency)
s(i): number of pick

1. EXPLOIT: p(i) = AFL

$$p(i) = \alpha(i)$$

2. EXPLORE: p(i) = AFL / const

$$p(i) = \frac{\alpha(i)}{\beta}$$

3. Cut-Off Exponential (COE)

$$p(i) = \begin{cases} 0 & \text{if } f(i) > \mu \\ \min\left(\frac{\alpha(i)}{\beta} \cdot 2^{s(i)}, M\right) & \text{otherwise.} \end{cases}$$

$$\mu = \frac{\sum_{i \in S^+} f(i)}{|S^+|}$$

mean number of fuzz exercising a discovered path

4. Exponential Schedule (FAST)

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{2^{s(i)}}{f(i)}, M\right)$$

5. Linear Schedule (LINEAR)

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)}{f(i)}, M\right)$$

6. Quadratic Schedule (QUAD)

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)^2}{f(i)}, M\right)$$

3-6: **prevent** high-frequency paths to be fuzzed until they become low-frequency path

State-of-the-Art: Mutation Scheduling

Cutting Edge

AFL

AFLFast

MOpt

LAF-Intel

RedQueen

AFLSmart

MOpt:

Contributions:

- observe that efficient mutations are selected with a small number
- utilize a customized PSO to find optimal selection probability distribution of operators

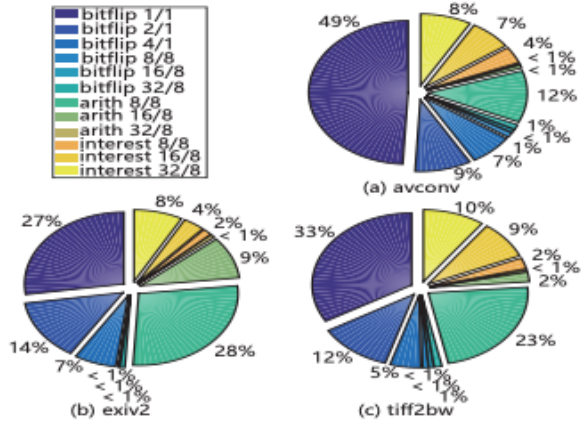


Figure 3: Percentages of interesting test cases produced by different operators in the deterministic stage of AFL.

https://www.usenix.org/system/files/sec19-lyu.pdf

spend more time on efficient mutations

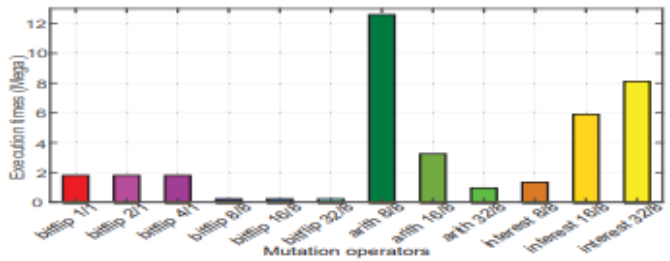


Figure 4: The times that mutation operators are selected when AFL fuzzes a target program `avconv`.

https://www.usenix.org/system/files/sec19-lyu.pdf

https://wiki.vul337.team:8888/doku.php?id=wiki:mopt_optimize_mutation_scheduling_for_fuzzers

State-of-the-Art: Mutation Scheduling

Cutting Edge

AFL

AFLFast

MOpt

LAF-Intel

RedQueen

AFLSmart

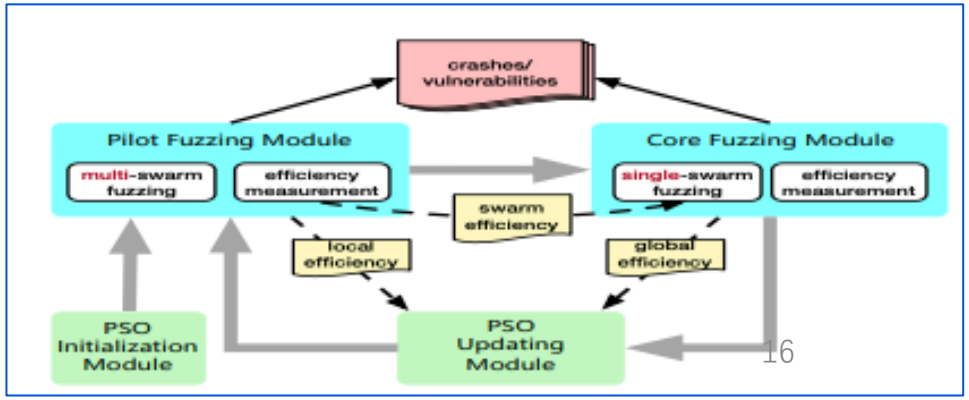
MOpt:

customized PSO

- (position of) a particle : (selection probability of) per operator
- swarm : selection probability distribution of operators

Multiple Swarms

- Pilot: evaluate each swarm fuzzing efficiency
- Core: perform fuzz with the best swarm selected by Pilot



https://www.usenix.org/system/files/sec19-lyu.pdf

State-of-the-Art :Bypassing Roadblocks

LAF-Intel：

Challenge: tricky conditional statements

- almost correct：0xabad1dee

```
if (input == 0xabad1dea) {
  /* terribly buggy code */
} else {
  /* secure code */
}
```

Idea:

- split up comparisons into multiple single-byte comparisons

LLVM Passes

- The split-compares-pass
- The compare-transform-pass
- The split-switches-pass

```
if (input >> 24 == 0xab){
  if ((input & 0xff0000) >> 16 == 0xad) {
    if ((input & 0xff00) >> 8 == 0x1d) {
      if ((input & 0xff) == 0xea) {
        /* terrible code */
        goto end;
      }
    }
  }
}

/* good code */

end:
```

State-of-the-Art :Bypassing Roadblocks

Cutting Edge

AFL

AFLFast

MOpt

LAF-Intel

RedQueen

AFLSmart

LAF-Intel:

LLVM Passes

deoptimize code to increase code coverage

- The split-compares-pass
  - only remain: <, >, ==, !=, all unsigned
- The compare-transform-pass
  - rewrite strcmp and memcmp calls
- The split-switches-pass
  - rewrite to lists of if
  - use split-compare-pass

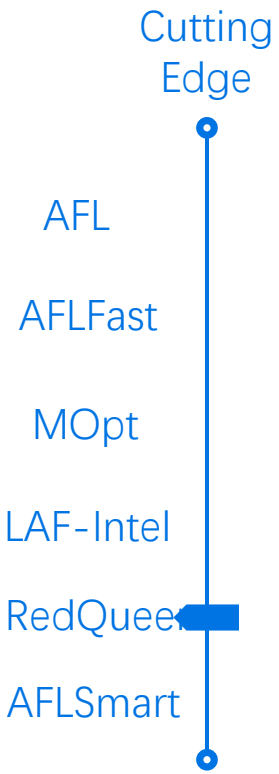
```
if(!strcmp(directive, "crash")) {
  programbug()
}
```

```
if(directive[0] == 'c') {
  if(directive[1] == 'r') {
    if(directive[2] == 'a') {
      if(directive[3] == 's') {
        if(directive[4] == 'h') {
          if(directive[5] == 0) {
            programbug()
}
```

```
int x = userinput();
switch(x) {
  case 0x11ff:
    /* handle case 0x11ff */
    break;
  case 0x22ff:
    /* handle case 0x22ff */
    break;
  default:
    /* handle default */
}
```

```
int x = userinput();
if(x >> 24 == 0) {
  if((x & 0xff0000) >> 16 == 0x00) {
    if((x & 0xff) == 0xff) {
      if((x & 0xff00) >> 8 == 0x11) {
        /* handle case 0x11ff */
        goto after_switch;
      } else if((x & 0xff00) >> 8 == 0x22) {
        /* handle case 0x22ff */
        goto after_switch;
      } else {
```

State-of-the-Art :Bypassing Roadblocks



RedQueen：

Roadblocks:

- magic number

```
if( u64(input) == u64("MAGICHDR"))
   bug(1);
```

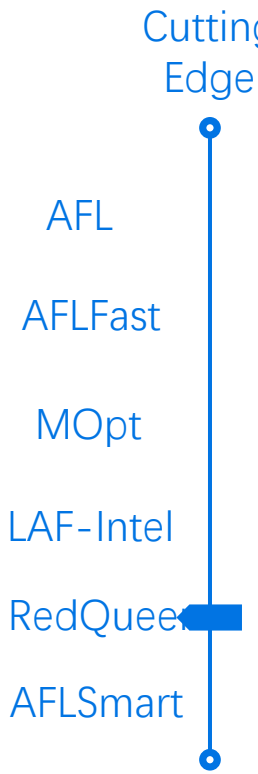
https://hexgolems.com/talks/redqueen.pdf

- nested checksum

```
if( u64(input) == hash(input[8..len]) )
   if( u64(input+8) == hash(input[16..len]) )
      if( input[16] == 'R' && input[17] == 'Q')
         bug(2);
```

https://hexgolems.com/talks/redqueen.pdf

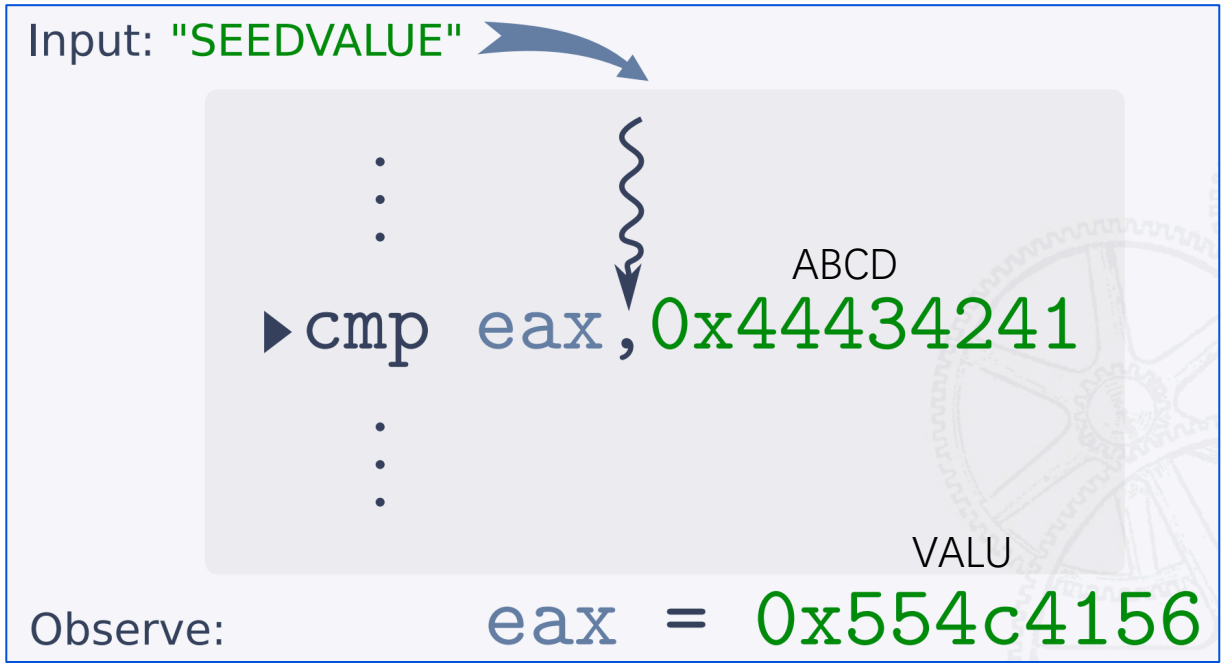
State-of-the-Art :Bypassing Roadblocks



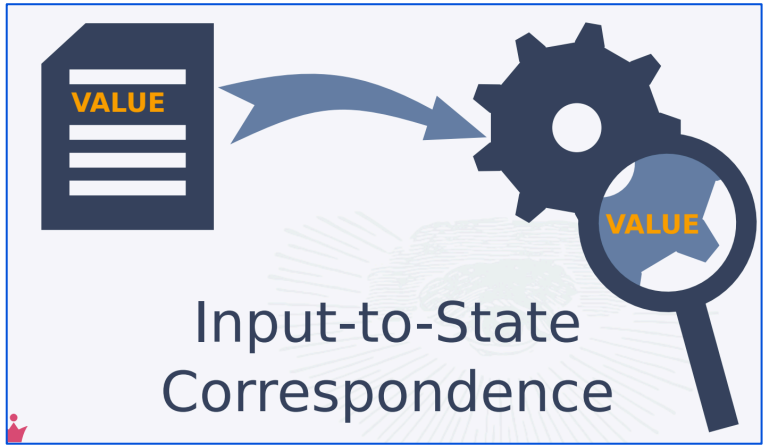
RedQueen：

Contributions:

- observe that values from the input are directly used at various states during execution
- exploit **input-to-state** relation to deal with roadblocks



https://hexgolems.com/talks/redqueen.pdf



https://hexgolems.com/talks/redqueen.pdf

State-of-the-Art :Bypassing Roadblocks

Cutting Edge

AFL

AFLFast

MOpt

LAF-Intel

RedQueen

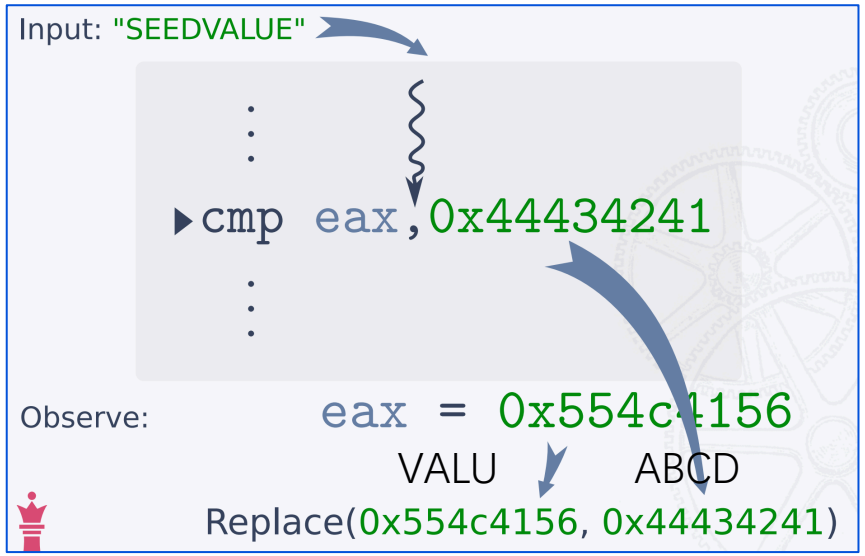
AFLSmart

RedQueen：

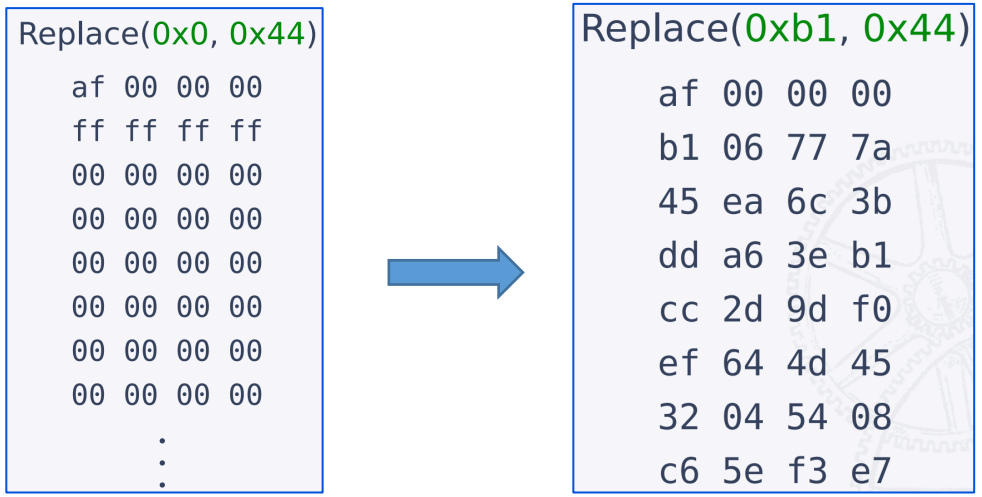
Magic Bytes

1. tracing: hook comparisons and extract args
2. variations: addition, subtraction ...
3. encoding: little-endian, hex, base-64, ...
4. application: **<** pattern -> repl **>**

https://hexgolems.com/talks/redqueen.pdf



- **Colorization** (remain bitmap): reduce the number of candidate positions



https://hexgolems.com/talks/redqueen.pdf

State-of-the-Art :Bypassing Roadblocks

Cutting Edge

AFL

AFLFast

MOpt

LAF-Intel

RedQueen

AFLSmart

RedQueen：

Nested Checksum

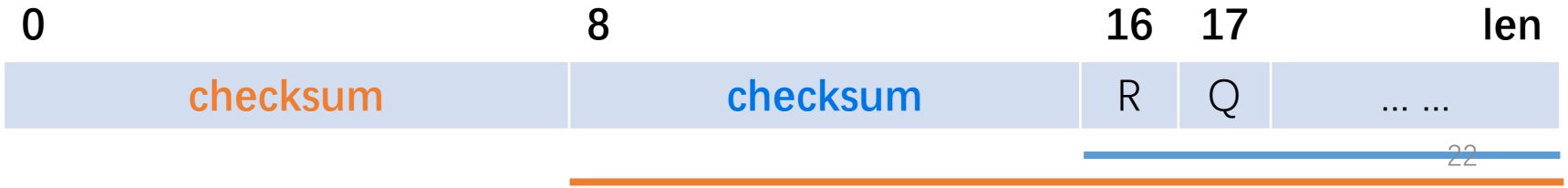
1. colorization
2. identification checksum cmp
3. patching yields true: False Positive
4. input validation and fixing

fixing

- magic bytes **<**pattern -> repl**>**
- nesting: Topological Sort

```
/* nested checksum */
if( u64(input) == hash(input[8..len]) )
   if( u64(input+8) == hash(input[16..len]) )
      if( input[16] == 'R' && input[17] == 'Q')
         bug(2);
```

Patch with `cmp al,al`



State-of-the-Art :Mutate Structure Inputs

Cutting Edge

AFL

AFLFast

MOpt

LAF-Intel

RedQueen

AFLSmart

AFLSmart:

A Common Issue: fuzzers generate mostly invalid inputs

Contributions:

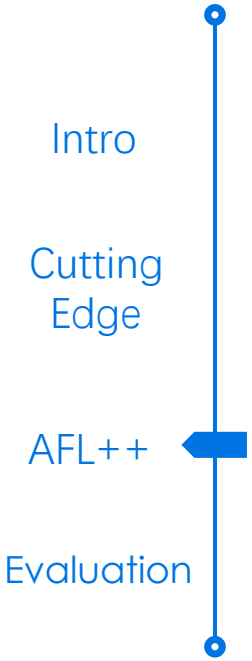
- a high-level **structural representation** of the seed
  - parses input into Peach pits
- define innovative mutation operators
  - **work on** virtual file structure
  - rather than on the bit level

Stored Bits	Information	Description
52 49 46 46	R I F F	RIFF.ckID
24 08 00 00	2084	RIFF.cksize
57 41 56 45	W A V E	RIFF.WAVEID
66 6d 74 20	f m t ␣	fmt.ckID
10 00 00 00	16	fmt.cksize
01 00 02 00	1 2	fmt.wFormatTag (1=PCM) & fmt.nChannels
22 56 00 00	22050	fmt.nSamplesPerSec
88 58 01 00	88200	fmt.nAvgBytesPerSec
04 00 10 00	4 16	fmt.nBlockAlign & fmt.wBitsPerSample
64 61 74 61	d a t a	data.ckID
00 08 00 00	2048	data.cksize
00 00 00 00	sound data 1	left and right channel
24 17 1e f3	sound data 2	left and right channel
3c 13 3c 14	sound data 3	left and right channel
16 f9 18 f9	sound data 4	left and right channel
34 e7 23 a6	sound data 5	left and right channel
3c f2 24 f2	sound data 6	left and right channel
11 ce 1a 0d	sound data 7	left and right channel

```
<DataModel name="Chunk">
  <String name="ckID" length="4"/>
  <Number name="cksize" size="32" >
    <Relation type="size" of="Data"/>
  </Number>
  <Blob name="Data"/>
  <Padding alignment="16"/>
</DataModel>
<DataModel name="ChunkFmt" ref="Chunk">
   <String name="ckID" value="fmt "/>
   <Block name="Data">
      <Number name="wFormatTag" size="16"/>
      <Number name="nChannels" size="16"/>
      <Number name="nSampleRate" size="32"/>
      <Number name="nAvgBytesPerSec" size="32"/>
      <Number name="nBlockAlign" size="16" />
      <Number name="nBitsPerSample" size="16"/>
   </Block>
</DataModel>
...
<DataModel name="Wav" ref="Chunk">
  <String name="ckID" value="RIFF"/>
  <String name="WAVE" value="WAVE"/>
  <Choice name="Chunks" maxOccurs="30000">
    <Block name="FmtChunk" ref="ChunkFmt"/>
    ...
    <Block name="DataChunk" ref="ChunkData"/>
  </Choice>
</DataModel>
```


https://thuanpv.github.io/publications/TSE19_aflsmart.pdf

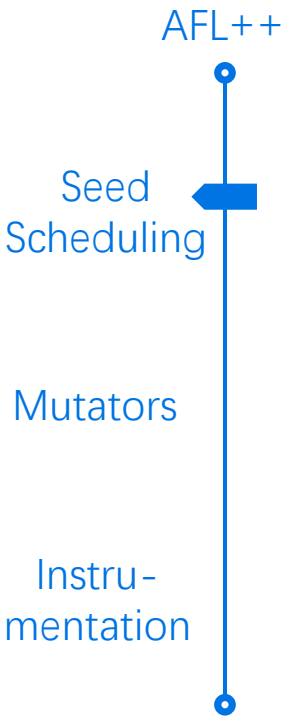
New Baseline: AFL++



Overview:

- **Seed Scheduling**
  - based on power schedules of AFLFast
- **Mutators**
  - Custom Mutator API
  - RedQueen: Input-To-State Mutator
  - MOpt Mutator
- **Instrumentation**

New Baseline: AFL++



Seed Scheduling：

AFLFAST Power Scheduling:

decides the amount of generated inputs from each seed (the seed’ s **energy**)

1. EXPLOIT: AFL
2. EXPLORE: AFL / const
3. Cut-Off Exponential (COE)
4. Exponential Schedule (FAST)
5. Linear Schedule (LINEAR)
6. Quadratic Schedule (QUAD)

more energy than is required in expetation

prevent high-frequency paths to be fuzzed

until they become low-frequency path

7. Mmopt

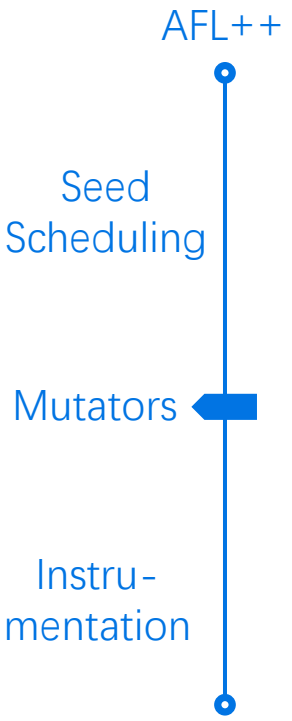
focus on newest seeds

8. Rare

ignore runtime of the seed and

focus on seeds with rarely edges

New Baseline: AFL++



Custom Mutator API：

AFL++ incorporates many mutators.

Framework

- can be easily extend
- can be adapted to specific targets

implement API:

- afl_custom_(de)init
- afl_custom_queue_get
- **afl_custom_fuzz: custom mutations**
- afl_custom_havoc mutation
- afl_custom_post_process
- afl_custom_queue_new_entry

Trimming Support: custom trim api

New Baseline: AFL++

AFL++

Seed Scheduling

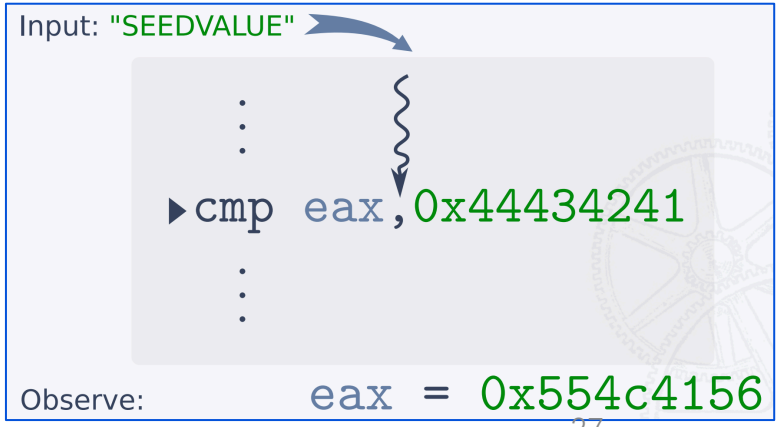
Mutators

Instru-mentation

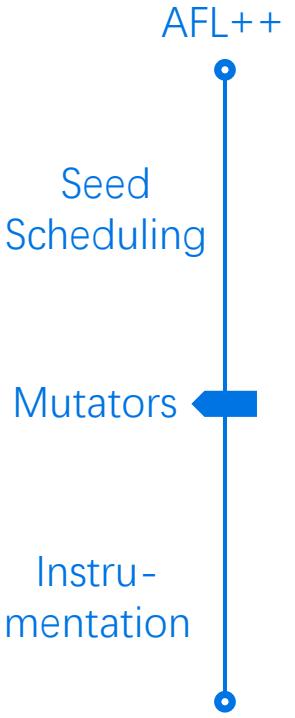
Input-To-State Mutator：

Based on REDQUEEN’ s Input-To-State

- **Colorization**
  - RedQueen: remain hash of bitmap
  - AFL++: but also **remain the execution speed** (bounds of a 2x slowdown)
- **Bypass Comparison**: probabilistic fuzzing
  - fail to bypass: fuzzed with low probability next time
  - RedQueen
    - cmp hooking: hardware-assisted VM breakpoints
    - hit a small numer times: remove breakpoint



New Baseline: AFL++

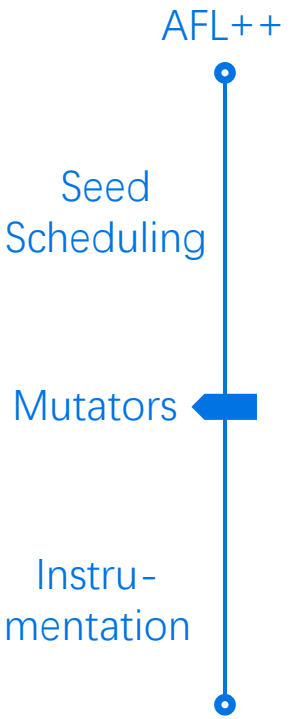


Input-To-State Mutator：

Based on RedQueen’ s Input-To-State

- Colorization
- Bypass Comparison: probabilistic fuzzing
- **CmpLog Instrumentation**
  - RedQueen:
    - comparisons are hooked by hardware-assisted VM breakpoints
    - arguments are extracted when hit
  - AFL++:
    - a shared table
    - each comparison logs the operands of its last 256 executions

New Baseline: AFL++



MOpt:

- implements Core and the Pilot mode
- can be combined with Input-To-State mutator

New Baseline: AFL++

AFL++

Seed Scheduling

Mutators

Instru-mentation

Instrumentation:

Problem: hit count can overflow to 0

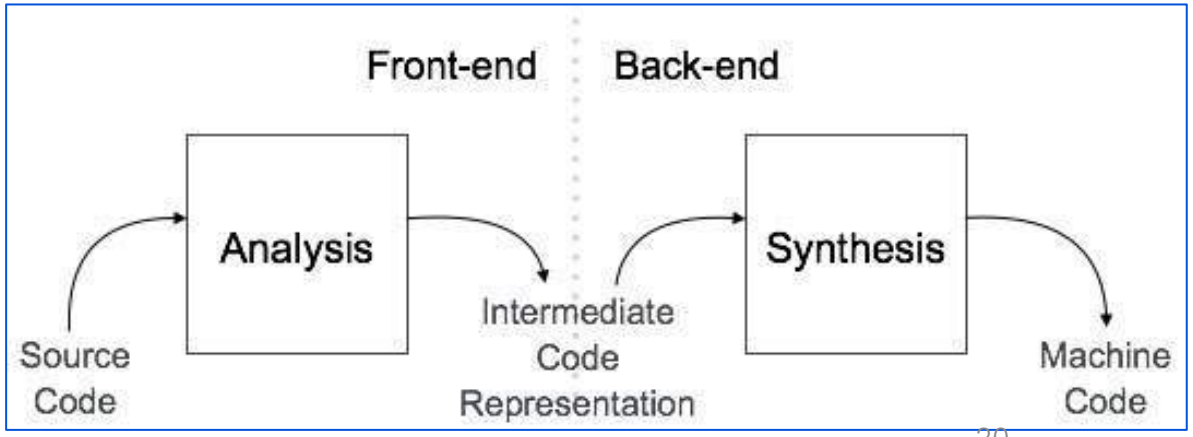
- **NeverZero**: add the carry flag 🔼
- Saturated Counters: freeze at 255 🔽

```
static const u8 count_class_lookup8[256] = {

  [0]           = 0,
  [1]           = 1,
  [2]           = 2,
  [3]           = 4,
  [4 ... 7]     = 8,
  [8 ... 15]    = 16,
  [16 ... 31]   = 32,
  [32 ... 127]  = 64,
  [128 ... 255] = 128

};
```

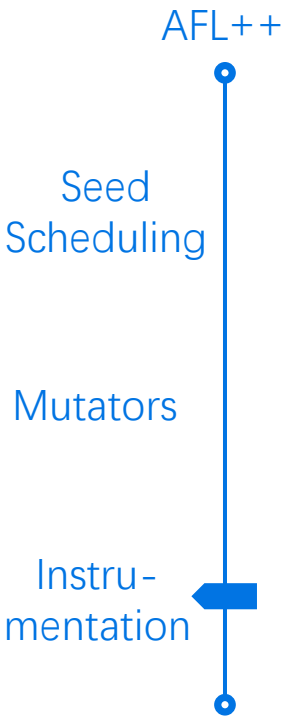
AFL++ supports several backends:

- **LLVM**
- **GCC**
- **QEMU**
- Unicornafl: support to Unicorn Engine
- QBDI: support to andriod libraries

https://www.tutorialspoint.com/compiler_design/compiler_design_architecture.htm



New Baseline: AFL++



LLVM:

Coverage Metrics

- Edge Coverage: consider prev and cur
  - more collisions and less speed
- Ngram: consider cur and N-1 prev blocks (N in 2-16)

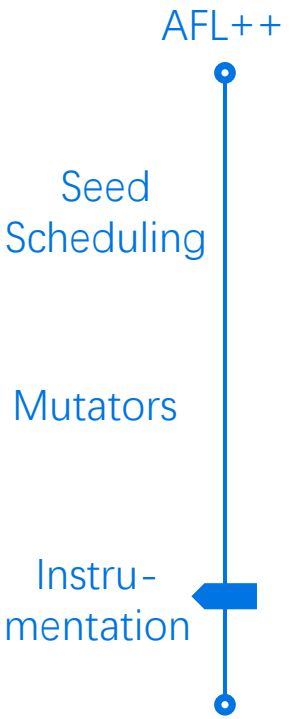
```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Pass:

- coverage feedback pass
- LAF-Intel Passes (improved)
- CmpLog Passes
- ...

Pass : A pass refers to the traversal of a compiler through the entire program.

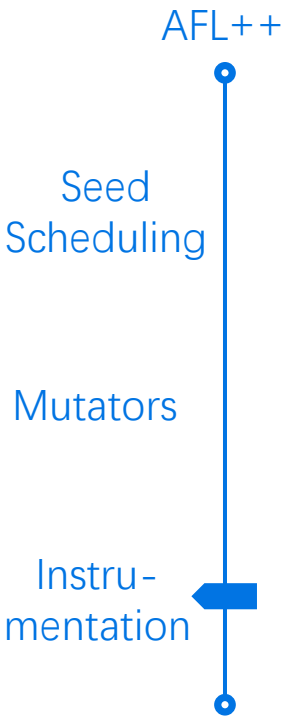
New Baseline: AFL++



GCC:

- AFL
  - afl-gcc, afl-g++
  - assembly-level rewriting instrumentation
- AFL++
  - afl-gcc-fast, afl-g++-fast: wrapper of afl-gcc, afl-g++
  - **GCC plugin**: true compiler-level instrumentation
  - not LLVM, **like** AFL LLVM mode (afl-clang-fast)

New Baseline: AFL++

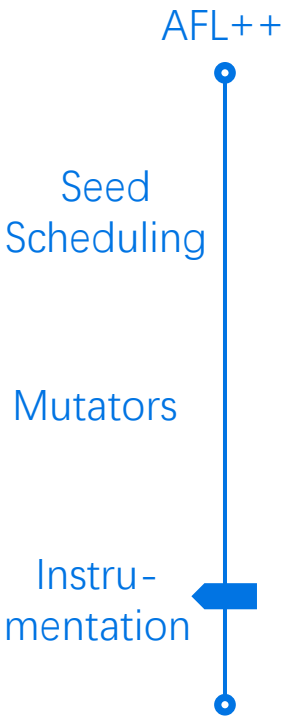


QEMU:

- Deferred initialization
- **Persistent mode**
- Snapshot mode
- Partial instrumentation
- **CompareCoverage**
- CmpLog mode
- Wine mode
- ...

https://github.com/AFLplusplus/AFLplusplus/tree/stable/qemu_mode

New Baseline: AFL++



QEMU：

- **CompareCoverage**
  - binary-level CompareCoverage = source-level LAF-Intel
  - can be configured by env variable AFL_COMPCOV_LEVEL
    - AFL_COMPCOV_LEVEL=1: split only immediate values
    - AFL_COMPCOV_LEVEL=2: instrument all comparison instructions
- **Persistent Mode**
  - AFL QEMU mode: don’ t support persistent mode
  - The START address
  - The RET address
  - …

New Baseline: AFL++

Instrumentation:

AFL++

Seed Scheduling

Mutators

Instru-mentation

Table with supported features for each instrumentation backend

	afl-gcc	LLVM mode	GCC plugin	QEMU mode	UNICORN mode	QBDI mode
NeverZero	✓	✓		✓	✓	
Persistent mode		✓	✓	✓	✓	✓
LAF-INTEL/ CompCov		✓		✓	✓	
CmpLog		✓		✓		
Instrument filelist		✓	✓	partial		
InsTrim		✓				
Ngram/Ctx coverage		✓				
Snapshot LKM		✓				

https://aflplus.plus//papers/aflpp-woot2020.pdf

Evaluation Use Cases

Evaluation：

Compare with FuzzBench

1. Default : AFL with some fixes
2. MOpt : Mutator
3. Ngram4 : Instrumentation
4. RedQueen : Additional cmp feedback
5. Ngram4, Rare: Instrumentation and Rare Scheduling
6. MOpt, RedQueen

FuzzBench: offer free service that evaluates fuzzers on a wide variety of real-world benchmarks.

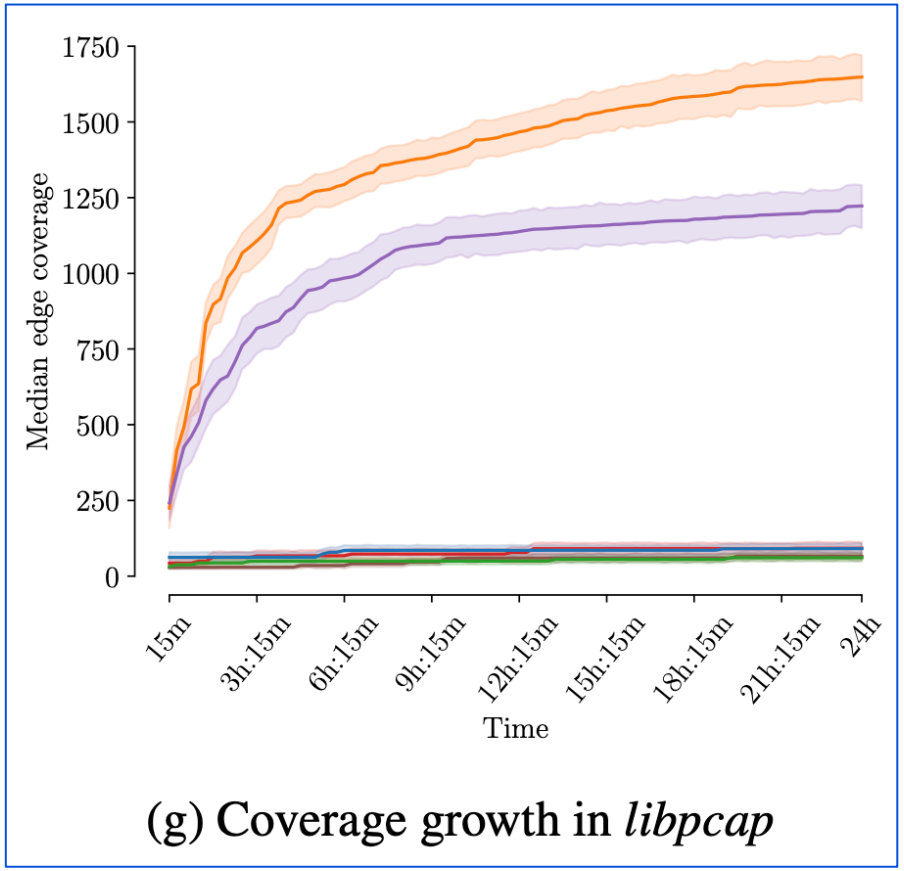
Evaluation Use Cases

Evaluate RedQueen：

[Default] RedQueen Ngram4, Rare Ngram4 MOpt, RedQueen MOpt

Median edge coverage growth with 66% confidence interval produced by each fuzzer over 20 trials at 24 hours.

- RedQueen can bypass roadblocks.
- MOpt increase the coverage.



(g) Coverage growth in *libpcap*

Evaluation Use Cases

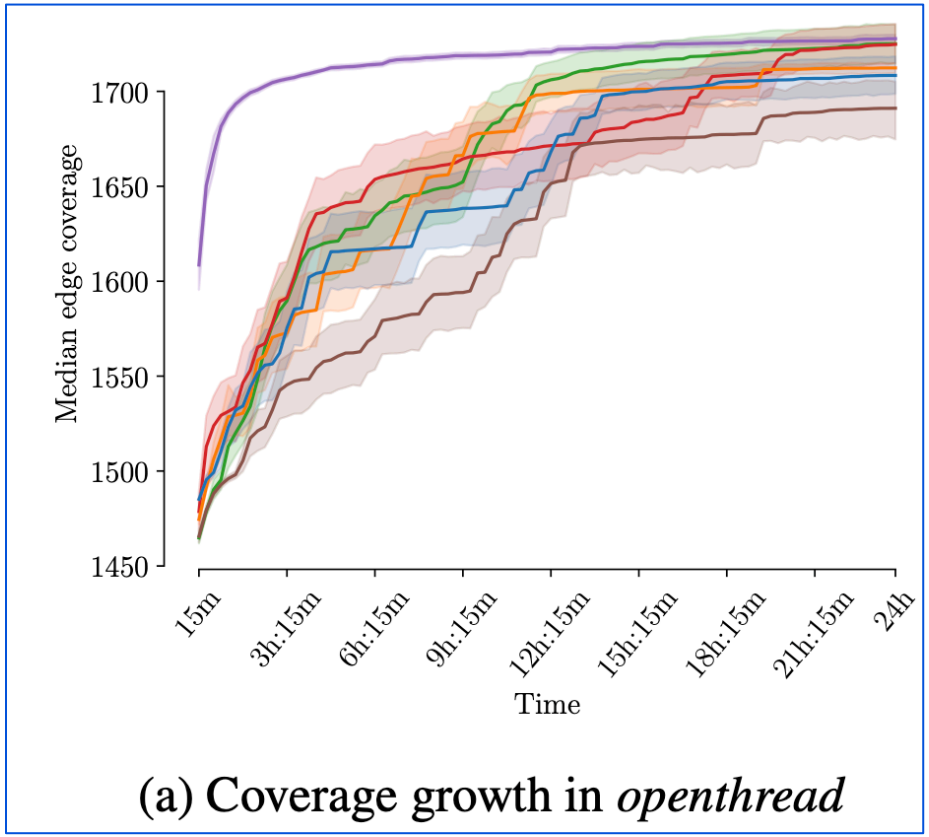
Evaluate RedQueen：

[Default] RedQueen Ngram4, Rare Ngram4 MOpt, RedQueen MOpt

Median edge coverage growth with 66% confidence interval produced by each fuzzer over 20 trials at 24 hours.

- RedQueen can bypass roadblocks.
- MOpt suppress performance.

target specific



(a) Coverage growth in *openthread*

Evaluation Use Cases

Intro

Cutting Edge

AFL++

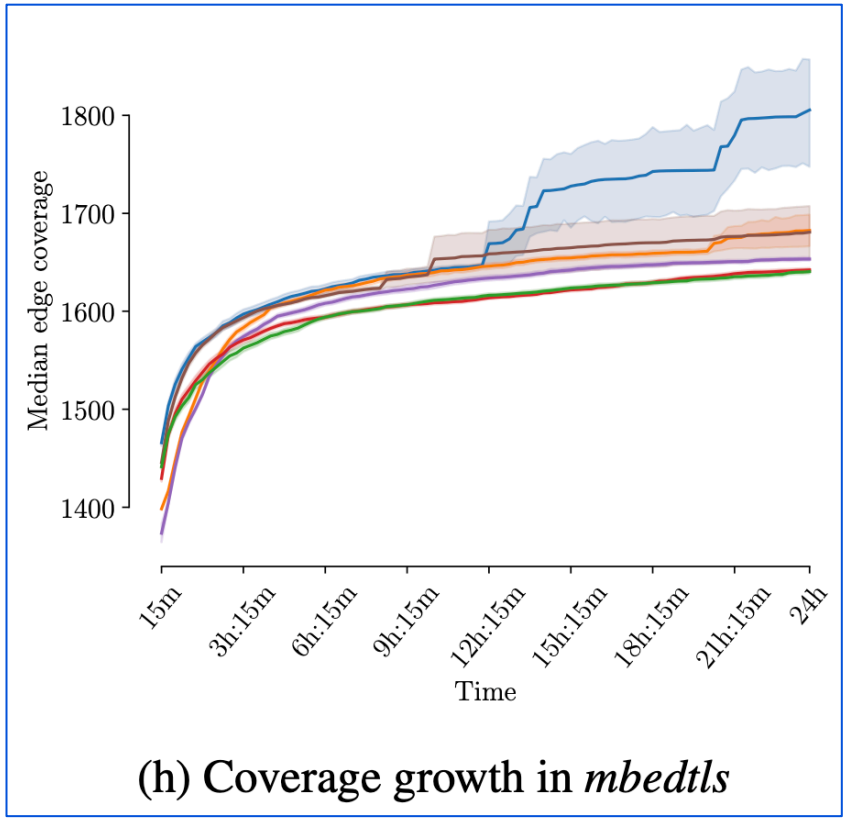
Evaluation

Evaluate MOpt:

[Default] RedQueen Ngram4, Rare Ngram4 MOpt, RedQueen MOpt

Median edge coverage growth with 66% confidence interval produced by each fuzzer over 20 trials at 24 hours.

- Mopt suddenly starts gaining a massive code coverage in the middle of the run.
- It happens for multiple runs.



(h) Coverage growth in *mbedtls*

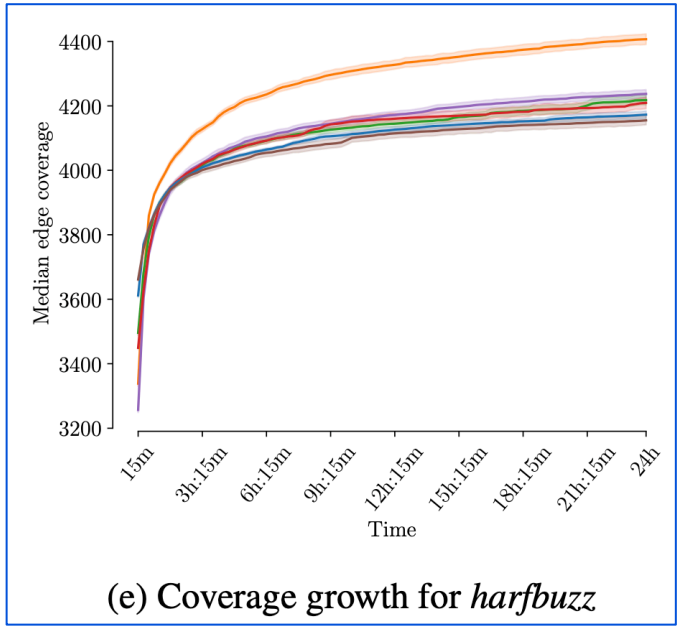
Evaluation Use Cases

Evaluate MOpt:

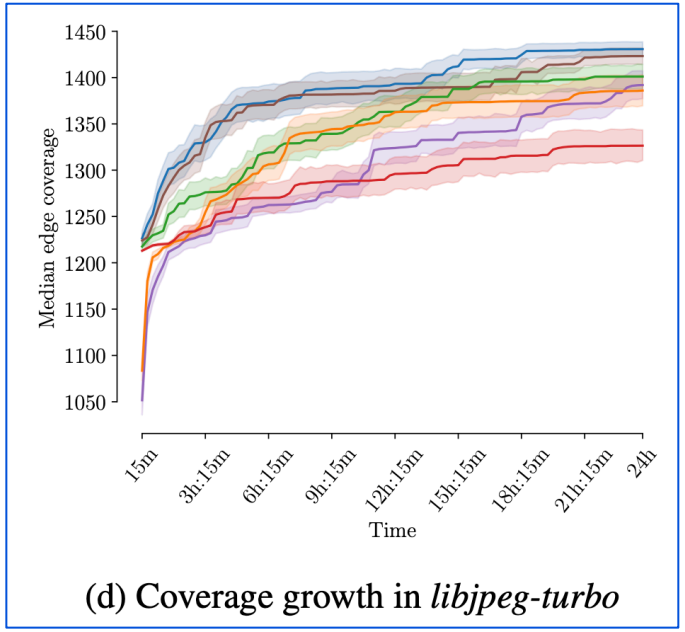
[Default] RedQueen Ngram4, Rare Ngram4 MOpt, RedQueen MOpt

Median edge coverage growth with 66% confidence interval produced by each fuzzer over 20 trials at 24 hours.

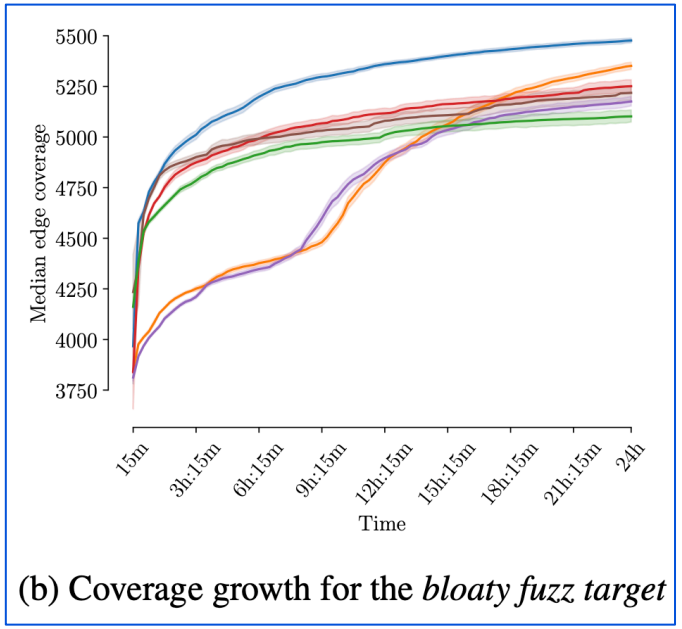
- MOpt helps RedQueen a lot.



(e) Coverage growth for *harfbuzz*



(d) Coverage growth in *libjpeg-turbo*



(b) Coverage growth for the *bloaty fuzz target*

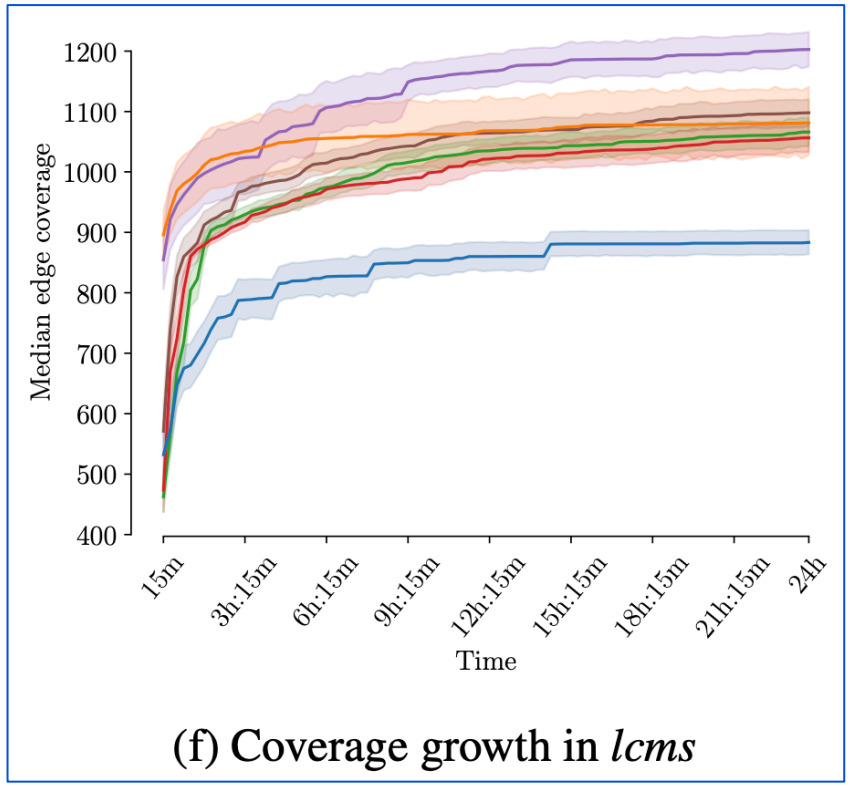
Evaluation Use Cases

Evaluate MOpt:

[Default] RedQueen Ngram4, Rare Ngram4 MOpt, RedQueen MOpt

Median edge coverage growth with 66% confidence interval produced by each fuzzer over 20 trials at 24 hours.

- MOpt has negative impact to RedQueen.



(f) Coverage growth in *lcms*

Evaluation Use Cases

Intro

Cutting Edge

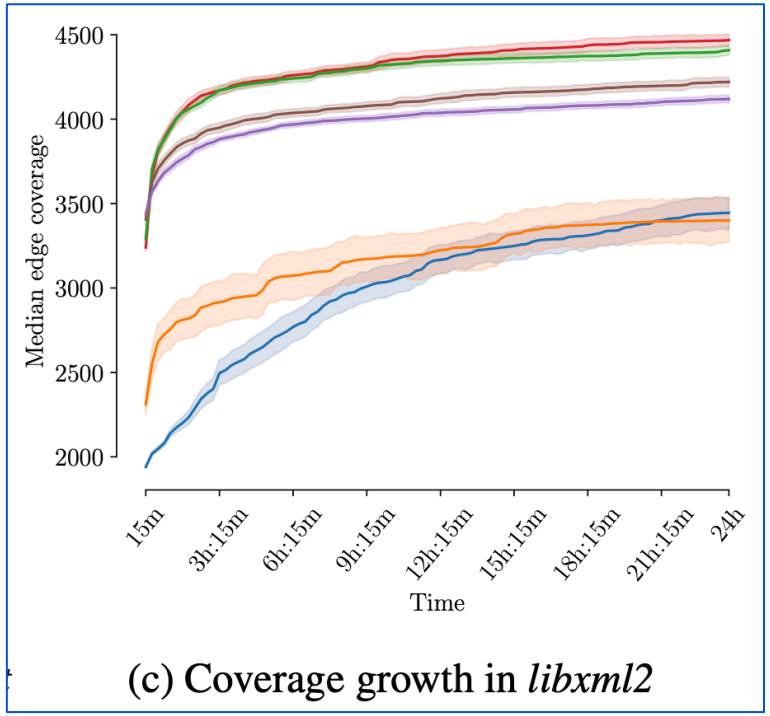
AFL++

Evaluation

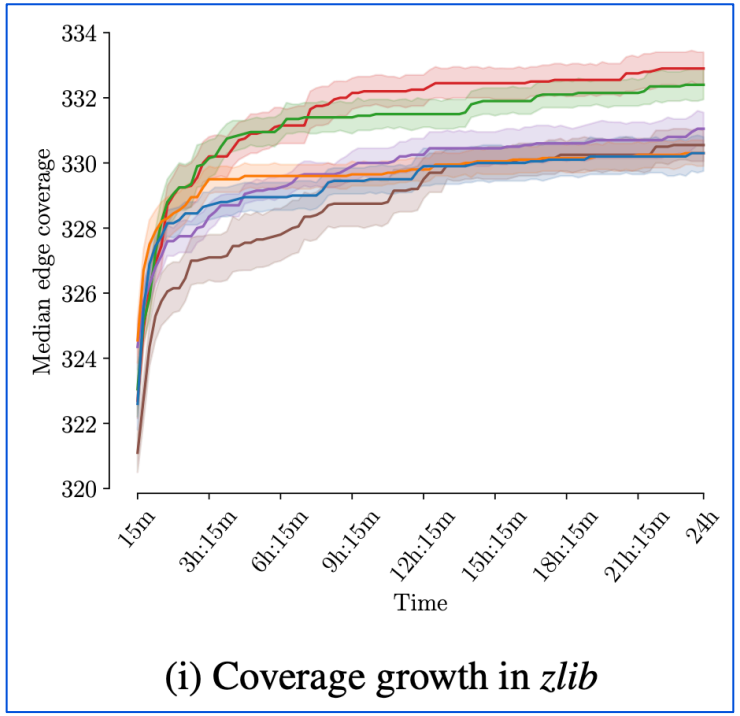
Evaluate Ngram：

[Default] RedQueen Ngram4, Rare Ngram4 MOpt, RedQueen MOpt

Median edge coverage growth with 66% confidence interval produced by each fuzzer over 20 trials at 24 hours.



(c) Coverage growth in *libxml2*



(i) Coverage growth in *zlib*

target specific

Ref

afl：

https://afl-1.readthedocs.io/en/latest/#

aflfast：

https://mboehme.github.io/paper/CCS16.pdf

https://github.com/mboehme/aflfast

RedQueen：

https://react-h2020.eu/m/filer_public/6d/86/6d869f98-f544-49cc-8221-b380c593888f/ndss19-redqueen.pdf

https://hexgolems.com/talks/redqueen.pdf

MOpt：

https://www.usenix.org/system/files/sec19-lyu.pdf

Ref

aflsmart：

https://thuanpv.github.io/publications/TSE19_aflsmart.pdf

aflplusplus：

https://aflplus.plus/papers/

Thanks for suggestions from Wang.

Q&A or Suggestions

Thanks for listening:)

刘冯润

2021/04/08