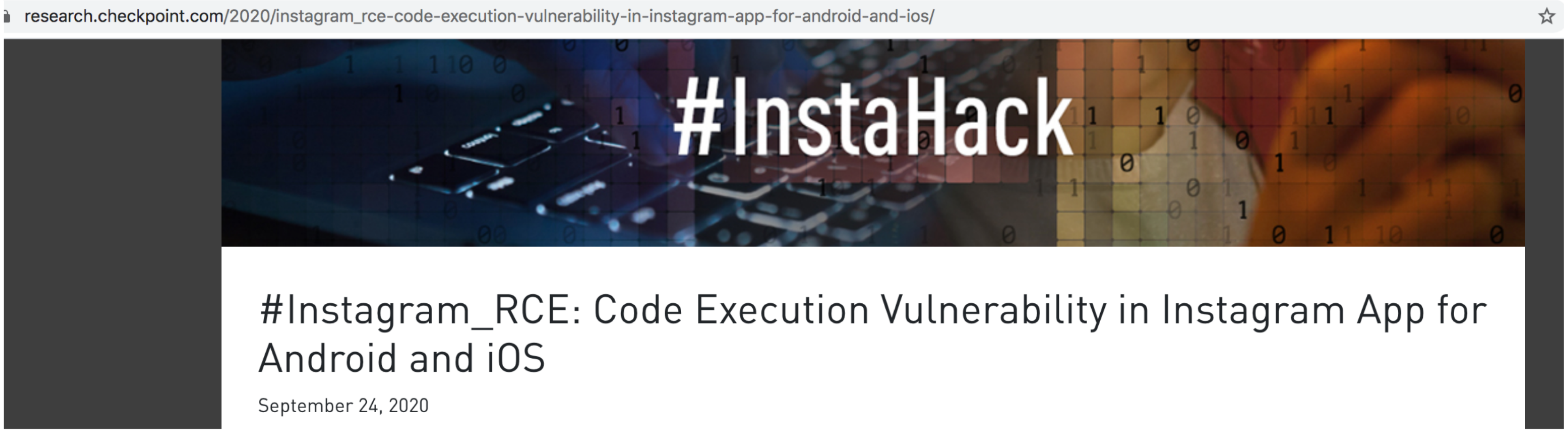
A journey in software fuzzing

Mohammadreza Hazhirpasand, Arash Ale Ebrahim

University of Bern, Switzerland

Fuzzing?

Fuzzing is a way of discovering bugs in software by providing randomized/pattern-based inputs to programs to find test cases that cause a crash.



Smart or dumb?

- A fuzzer that generates completely random input is known as a “dumb” fuzzer
- A fuzzer with knowledge of the input format is known as a “smart” fuzzer

Types of fuzzers

- Mutation → A valid input is mutated randomly to produce malformed input
  Dumb fuzzing / Smart fuzzing
- Replay → Place the fuzzer in the middle of a client and server
  Intercepting and modifying messages
- Generation → Generate input from scratch
  Only mutates randomly a chunk of an input
- Evolutionary → Use feedback from each test case to learn the format of the input
  Code coverage

Vulnerable friends!

- Protocols → TCP, DNS, FTP, SSL, Wireless protocols, …
- File formats → MP3, JPEG, PNG, TTF, …
- User inputs → Names, addresses, file names, ….
- Programming languages → JavaScript, PHP, …

A fuzzer's skeleton

- Test case generation → Completely blank or long strings, null character, max and min values for integers
- Reproducibility → Record test cases and associated information
- Crash detection → Attach a debugger, process disappears, timeouts

AFL – American Fuzzy Lop

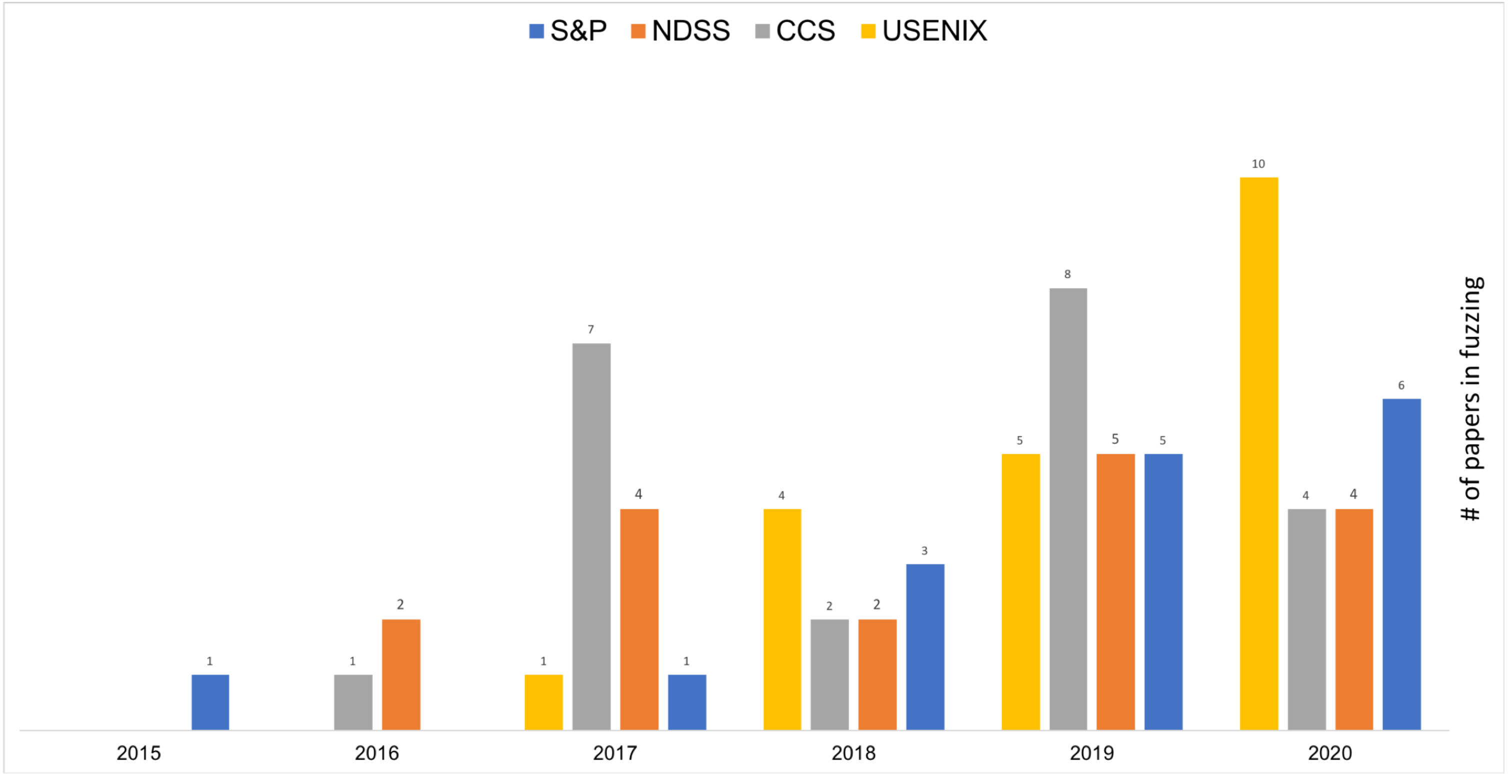
```
                    american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec       │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec       │  total paths : 195    │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec       │   uniq hangs : 1      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 38 (19.49%)       │    map density : 1217 (7.43%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.55 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 32/8         │ favored paths : 128 (65.64%)           │
│ stage execs : 0/9990 (0.00%)        │  new edges on : 85 (43.59%)            │
│ total execs : 654k                  │ total crashes : 0 (0 unique)           │
│  exec speed : 2306/sec              │   total hangs : 1 (1 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k            │    levels : 3          │
│  byte flips : 0/1804, 0/1786, 1/1750                │   pending : 178        │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k             │  pend fav : 114        │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k             │  imported : 0          │
│       havoc : 34/254k, 0/0                          │  variable : 0          │
│        trim : 2876 B/931 (61.45% gain)              │    latent : 0          │
└─────────────────────────────────────────────────────┴────────────────────────┘
```

AFL – American Fuzzy Lop

- Michal Zalewski, 2013
- First practical high performance guided fuzzer
- Compile-time instrumentation and genetic algorithms
- Many bugs!

DEMO

Fuzzing in conferences



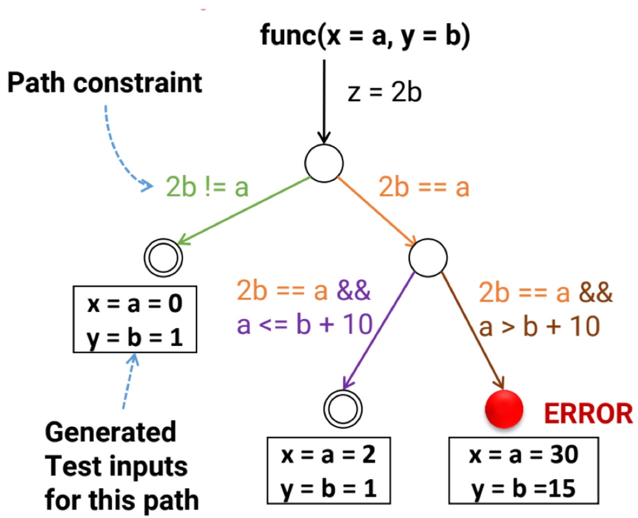
Our journey

1. A benchmark for existing concolic engine-based fuzzers
2. Optimize the AFL fuzzer
3. How the current fuzzers explore crypto libraries

#1 ~~Concolic Execution Engines~~ – Symbolic execution

- Traditional fuzzers fail to exercise all the possible behaviors that a program can have
- Execute the program with symbolic valued
- Generate new inputs at each branch to cover all parts of code

```
Void func(int x, int y){
   int z = 2 * y;
   if(z == x){
         if (x > y + 10)
            ERROR
   }
}
int main(){
   int x = sym_input();
   int y = sym_input();
   func(x, y);
   return 0;
}
```



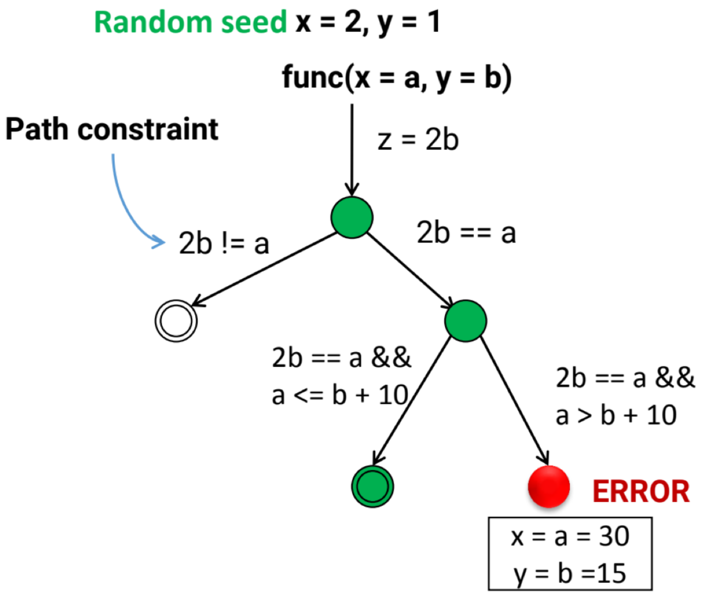
#1 ~~Concolic Execution Engines –~~ Symbolic execution!!!!

- **Path explosion**: symbolically executing all feasible program paths does not scale to large programs
- **Loops and recursions**: infinite execution tree
- **SMT solver limitations**: dealing with complex path constraints

#1 Concolic Execution Engines – ~~Symbolic execution~~

- **Concolic** = **Conc**rete + Symb**olic** *(dynamic symbolic execution)*
- A Program is executed with concrete (random inputs) and symbolic inputs

```
Void func(int x, int y){
   int z = 2 * y;
   if(z == x){
         if (x > y + 10)
            ERROR
   }
}
int main(){
   int x = input();
   int y = input();
   func(x, y);
   return 0;
}
```



#1 Concolic Execution Engines

- QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing - USENIX 2018
- Symbolic execution with SymCC: Don't interpret, compile! - USENIX 2020
- Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing - CCS 2019
- Eclipser : Grey-box Concolic Testing on Binary Code - ICSE 2019
- Driller: Augmenting Fuzzing Through Selective Symbolic Execution- NDSS 2016
- SAVIOR: Towards Bug-Driven Hybrid Testing - S&P 2019

#1 Concolic Execution Engines - challenges

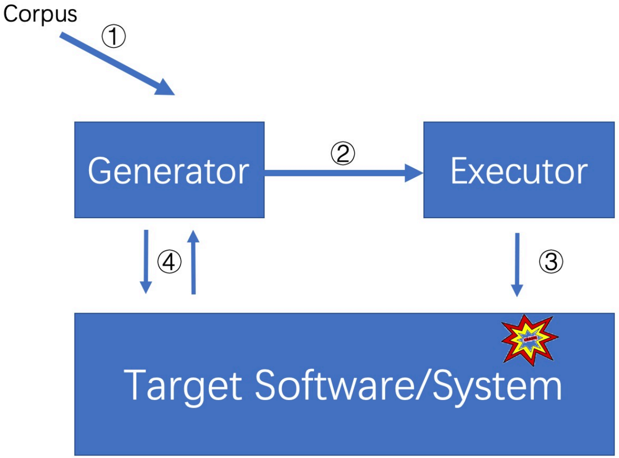
- QSYM: limited to Linux kernel 2.x
- SymCC: -
- Intriguer: buggy version
- Eclipser : complexity in running a fuzzing job
- Driller: outdated, not maintained anymore
- SAVIOR: poor documentation, buggy, over 2 months of discussion

#1 Concolic Execution Engines - benchmark

- LAVA-M benchmark test suite (4 vulnerable binaries)
- Real world targets: libpng, ffmpeg, libjpeg, libexpat, curl, OpenSSL, php

#2 AFLQL – High performance static guided fuzzing system

- **AFLQL** = **AFL** + Code**QL**
- Extract valuable information from the target program
- Optimize the generated corpus



+

#2 AFLQL – motivation

A good fuzzer should overcome:

1. Checksums
2. Magic numbers
3. Complex path constraints

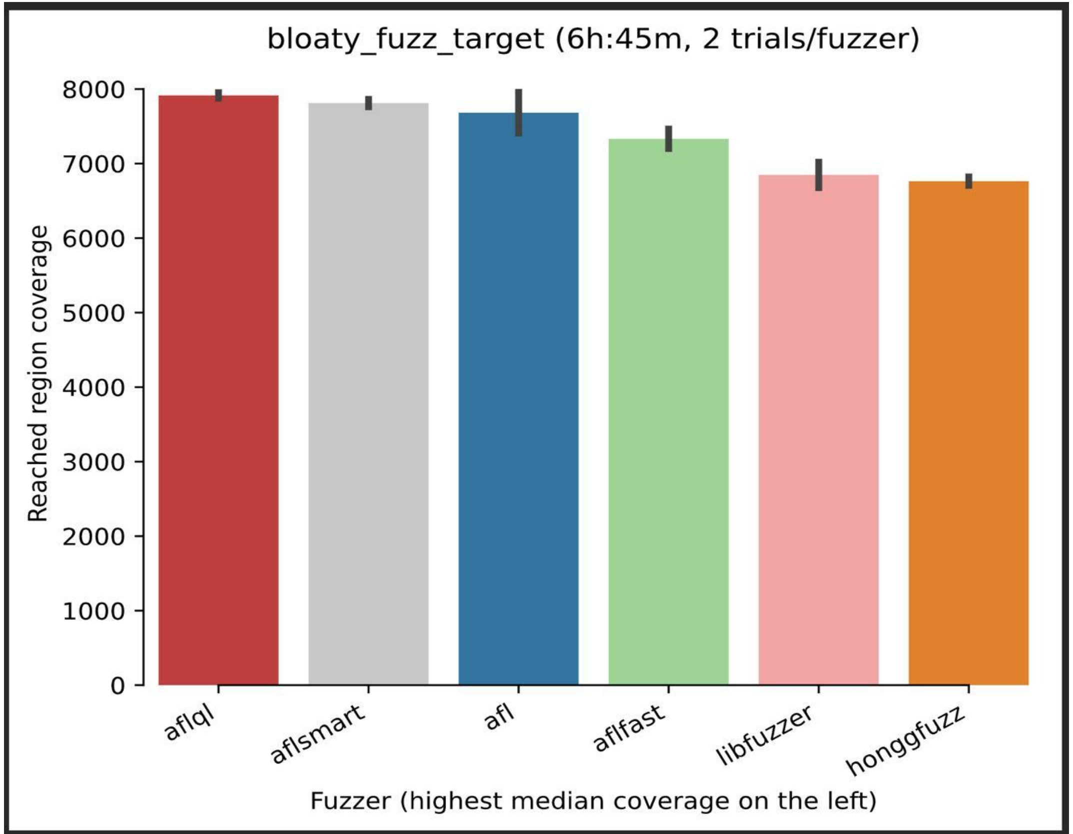
```
    //num[12] = 0x0681b201; num[13] = 0x0629a9d9;
    if (num[12] > 0x067fd111 && num[12] < 0x0691d629) {
        if (num[13] > 0x06209857 && num[13] < 0x06d93676) {
            if ((num[12] * num[13]) == 0x0681b201 * 0x0629a9d9) {
                flags[6] = 6;
            }
        }
    }
    //num[14] = 0x074fd355; num[15] = 0x075e1841;
    if (num[14] > 0x073f66a5 && num[14] < 0x07f04124) {
        if (num[15] > 0x07414558 && num[15] < 0x078e3e98) {
            if ((num[14] * num[15]) == 0x074fd355 * 0x075e1841) {
                flags[7] = 7;
            }
        }
    }
#if 0
#endif
    if (flags[0] == 0
            && flags[1] == 1
            && flags[2] == 2
            && flags[3] == 3
            && flags[4] == 4
            && flags[5] == 5
            && flags[6] == 6
            && flags[7] == 7
            /*
            */
            ) {
        *((volatile uint8_t *)0) = 0;
    }
    return 0;
}
```

#2 AFLQL – evaluation

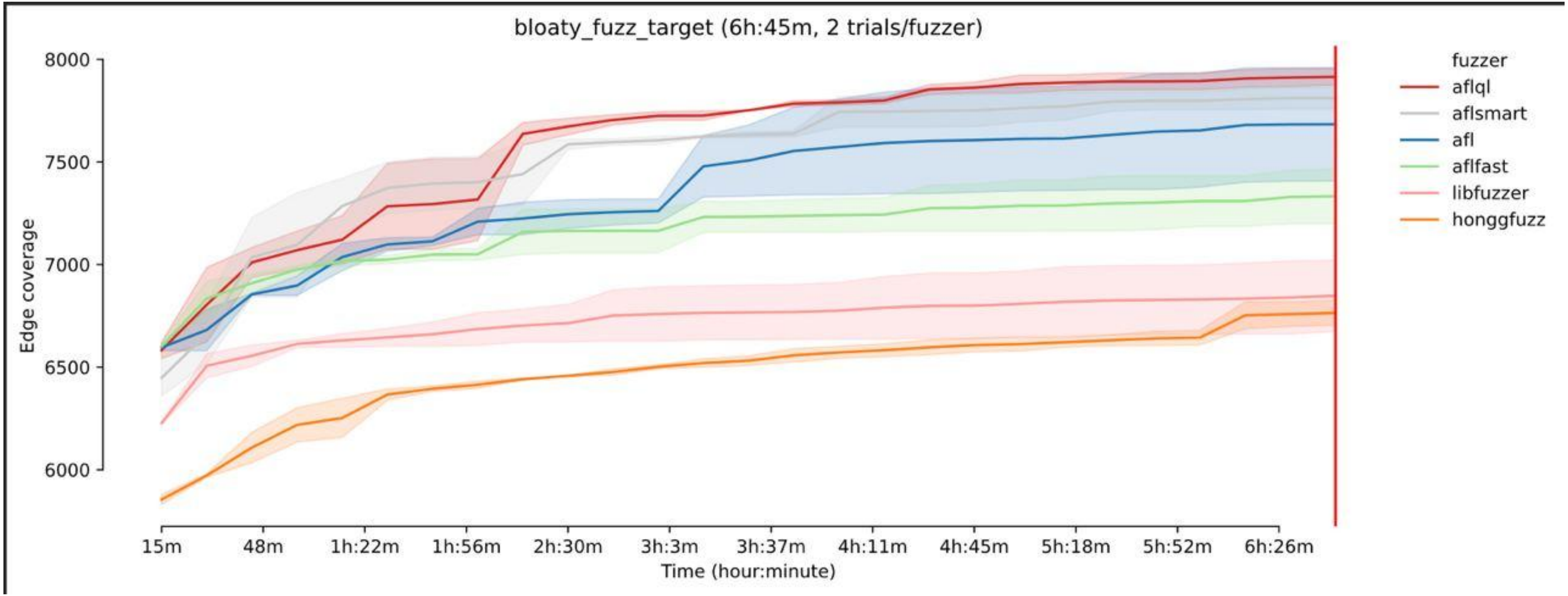
- Code coverage - Google FuzzBench (slow procedure for private research requests)
- Bug coverage - Magma benchmark suite (over a month of discussion)
- Bug and code coverage - LAVA-M benchmark suite

#2 AFLQL – code coverage (Google FuzzBench)

- Target program: Bloaty

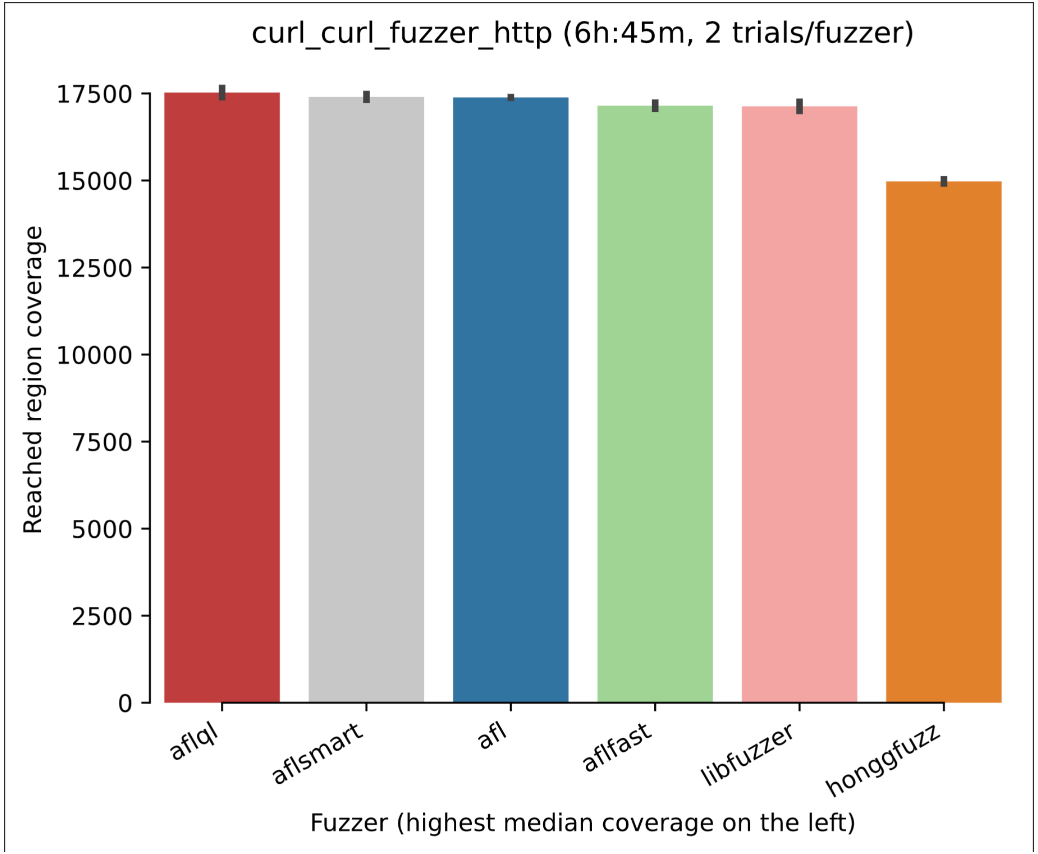


#2 AFLQL – code coverage (Google FuzzBench)

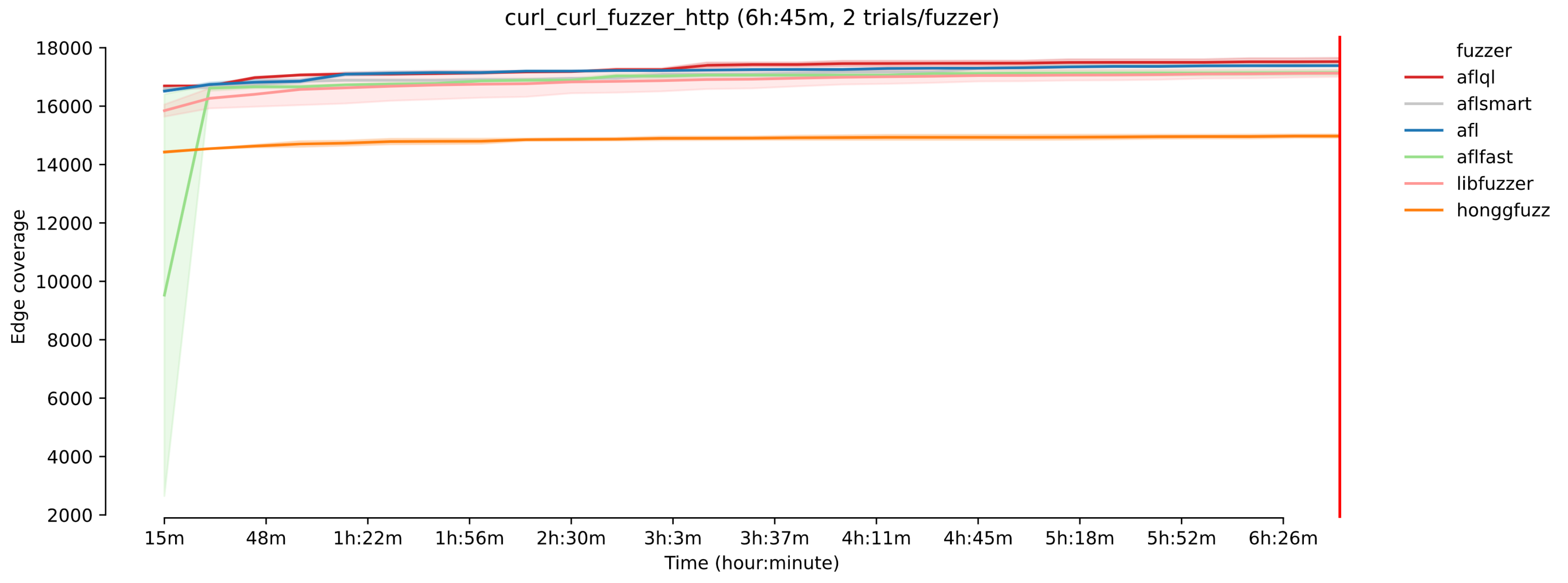


#2 AFLQL – code coverage (Google FuzzBench)

- Target program: cURL

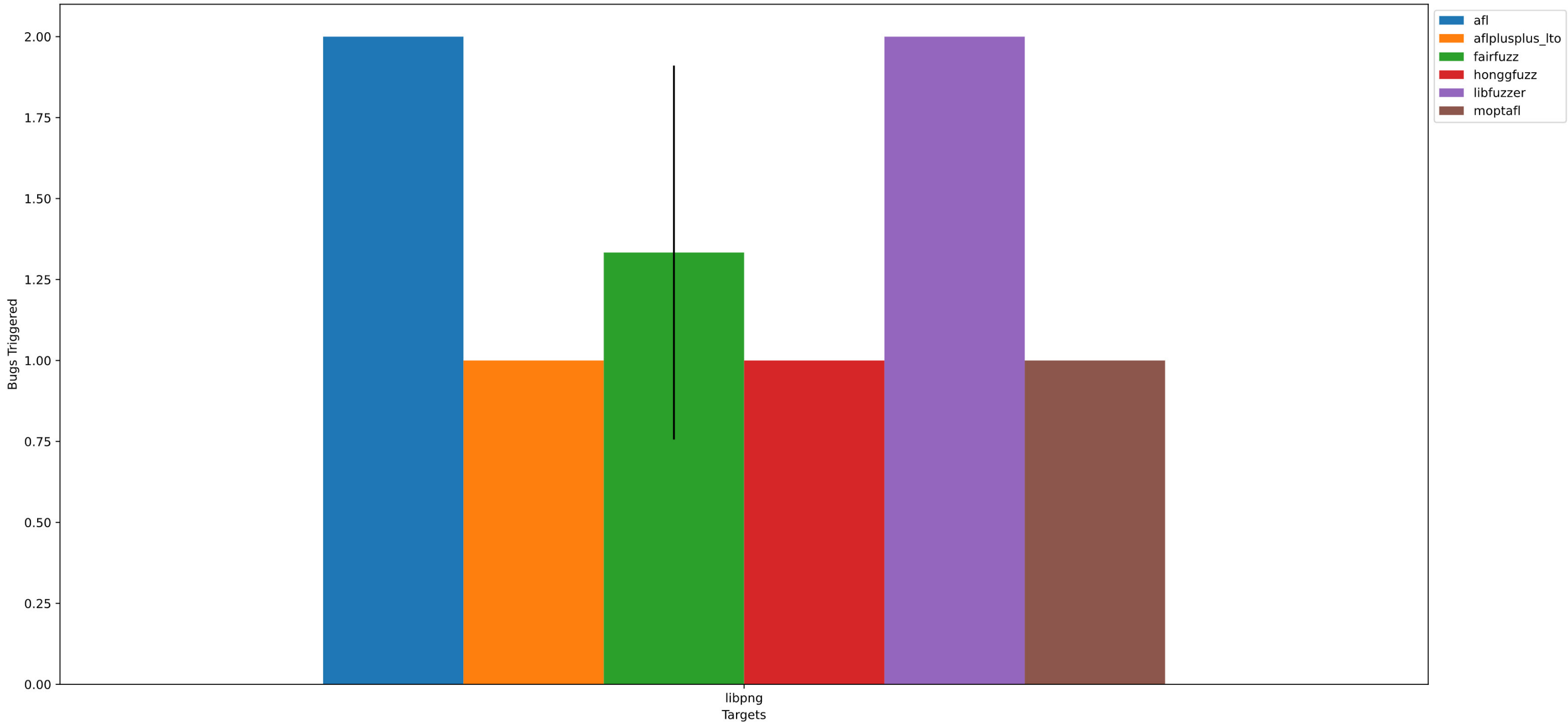


#2 AFLQL – code coverage (Google FuzzBench)



#2 AFLQL – bug Coverage - Magma

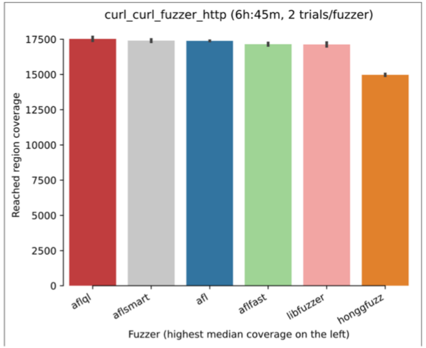
- Target program: Libpng



Summary

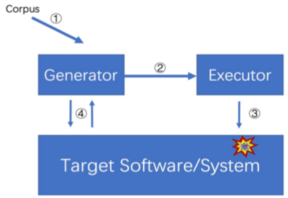
#2 AFLQL – code coverage (Google FuzzBench)

- Target program: cURL



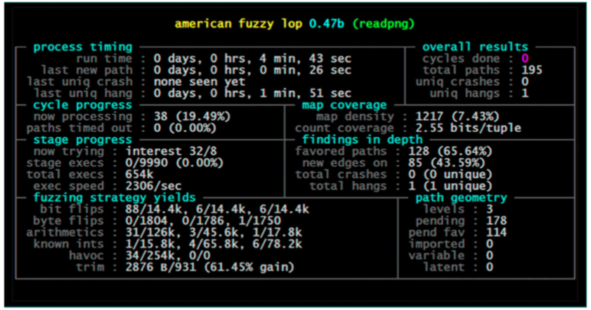
#2 AFLQL – High performance static guided fuzzing system

- **AFLQL** = **AFL** + Code**QL**
- Extract valuable information from the target program
- Optimize the generated corpus



+

AFL – American Fuzzy Lop



#1 Concolic Execution Engines

- QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing - USENIX 2018
- Symbolic execution with SymCC: Don't interpret, compile! - USENIX 2020
- Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing - CCS 2019
- Eclipser : Grey-box Concolic Testing on Binary Code - ICSE 2019
- Driller: Augmenting Fuzzing Through Selective Symbolic Execution- NDSS 2016
- SAVIOR: Towards Bug-Driven Hybrid Testing - S&P 2019