CS 250 Software Security

Fuzzing

What is Fuzzing?

- A form of vulnerability analysis
- Process:
  - Many slightly anomalous test cases are input into the application
  - Application is monitored for any sign of error

Example

Standard HTTP GET request

- § GET /index.html HTTP/1.1

Anomalous requests

- § AAAAAA...AAAA /index.html HTTP/1.1
- § GET ///////index.html HTTP/1.1
- § GET %n%n%n%n%n%n.html HTTP/1.1
- § GET /AAAAAAAAAAAAA.html HTTP/1.1
- § GET /index.html HTTTTTTTTTTTTTP/1.1
- § GET /index.html HTTP/1.1.1.1.1.1.1.1
- § etc...

Types of Fuzzers

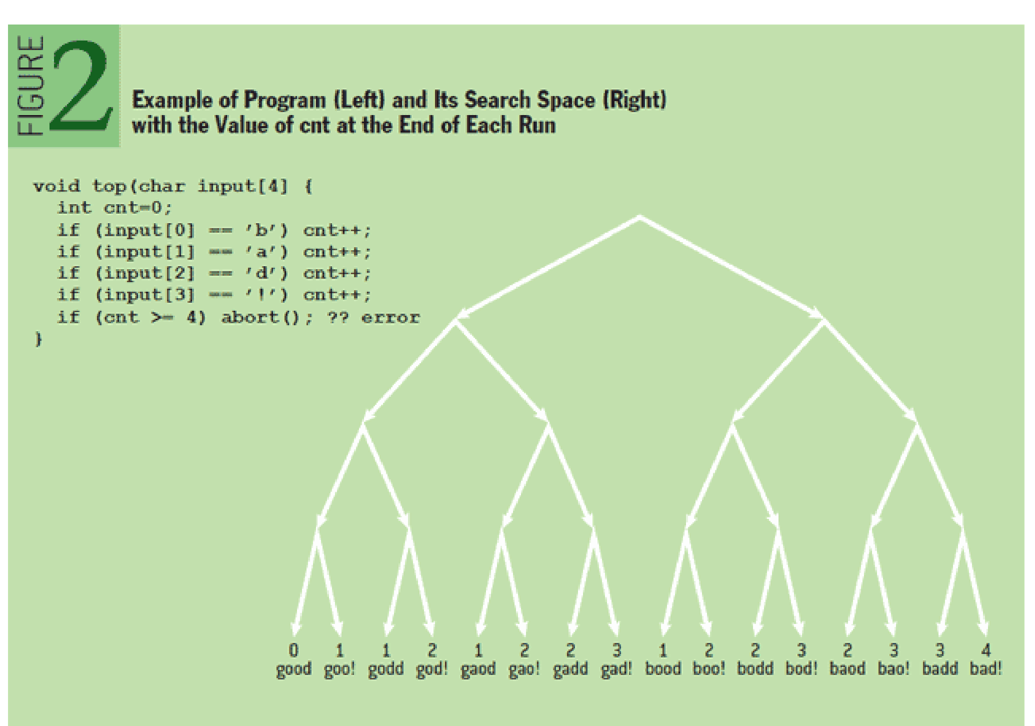
- In terms of input generation
  - Generational:
    - Define new tests based on a model or grammar
    - CSmith, LangFuzz, IFuzzer, Skyfire, Nautilus
  - Mutational:
    - Mutate existing data samples to create test data
    - Bit flips, additions, substitution, havoc, crossover
    - Custom mutators: https://github.com/AFLplusplus/AFLplusplus/tree/stable/custom_mutators

Types of Fuzzers

- In terms of program awareness
  - Blackbox: No awareness
  - Whitebox: Symbolic Execution
  - Greybox: API calls, Logs, Code Coverage, etc.

- With program awareness, fuzzing becomes evolutionary or genetic
  - Interesting inputs are kept as new seeds
  - More mutations are developed based on the new seeds to discover more new seeds…

Whitebox Fuzzing (2012)



- For a given input:
  - Perform symbolic execution,
  - When encountering a symbolic branch "deep" enough, generate a new testcase
- For each new testcase:
  - Execute it concretely
  - If it covers any new basic blocks, keep it in the first-level queue
  - If it covers a new path, keep it in the second level queue
- Fetch an input from first-level and then second-level

Greybox Fuzzing

Instrumented Program

(3) Seed Selection

Coverage Bitmap

new coverage!

(2) Seed Mutation

Testcase

Seed

New Seed

(1) Seed Scheduling

Queue of Seeds

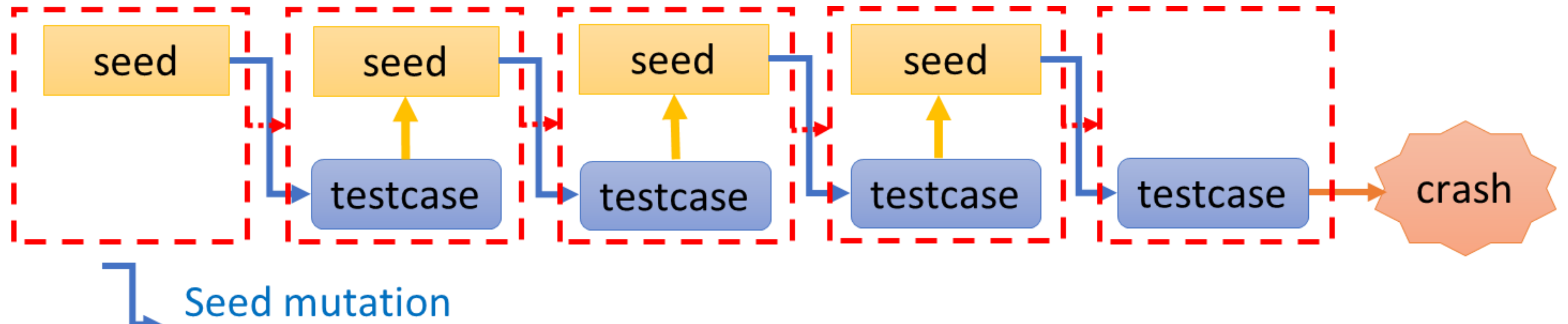
Coverage Metric

- Coverage metric is utilized to **measure the quality of testcases** during seed selection
  - HonggFuzz and Vuzzer: basic block coverage
  - AFL: improved branch coverage
  - LibFuzzer: block coverage or branch coverage
  - Angora: branch coverage extended with a calling context

Open Research Questions

- RQ1:
  - How to define the differences among different coverage metrics regarding their impact on greybox fuzzing?
- RQ2:
  - Is there an optimal coverage metric that outperforms all the others in greybox fuzzing?
- RQ3:
  - Is it a good idea to combine different metrics during fuzzing?

Coverage Metric Sensitivity



Seed mutation

Coverage metric decides whether a testcase will be ***selected*** as a new seed

Sensitivity is defined as the coverage metric's ability to preserve ***the chain of mutated testcases*** that finally reaches the bug triggering testcase

Coverage Metrics

Coverage Metric	Sensitivity Measurement
branch coverage	branch
n-gram branch coverage	n consecutive branches
context-sensitive branch coverage	branch + calling context
memory-access aware branch coverage	branch + memory access (r&w) pattern
memory-write access branch coverage	branch + memory write pattern

Implementation

- Based on AFL
  - Instrumentation via user-mode QEMU
    - Instrument *conditional jump* to get branch information
    - Instrument *call* and *ret* to get calling context information
    - Instrument *memory load* and *store* to get memory access information
  - Adopt the seed scheduling of AFLFast
- Available at https://github.com/bitsecurerlab/afl-sensitive

Comparison of Unique Crashes

Number of CGC binaries crashed by different coverage metrics

Comparison of Time to First Crash

number of crashed binaries

100
90
80
70
60
50
40

0 5000 10000 15000 20000

time to first crash (sec)

bc
ct
ma
mw
n2
n4
n8

Number of CGC binaries crashed overtime during fuzzing

Comparison of Seed Count

200
175
150
125
100
75
50
25
number of binaries
0
2000
4000
6000
8000
10000
12000
number of seeds
bc
ct
ma
mw
n2
n4
n8

Partial CDFs of seeds generated by different coverage metrics on the CGC dataset. A curve closer to the top left indicates fewer generated seeds.

Answer to RQ2:

- There is no grand slam coverage metric that can beat others
- Many of these more sensitive coverage metrics indeed lead to finding more bugs as well as finding them significantly fast
- Different coverage metrics often result in finding different sets of bugs.
- At different times of the whole fuzzing process, the best performer may vary.

Combination of Coverage Metrics

Number of CGC binaries crashed by combining different coverage metrics

Answer to RQ3

- A combination of these different metrics can help find more bugs and find them faster.

It is helpful to combine different coverage metrics.

But how?

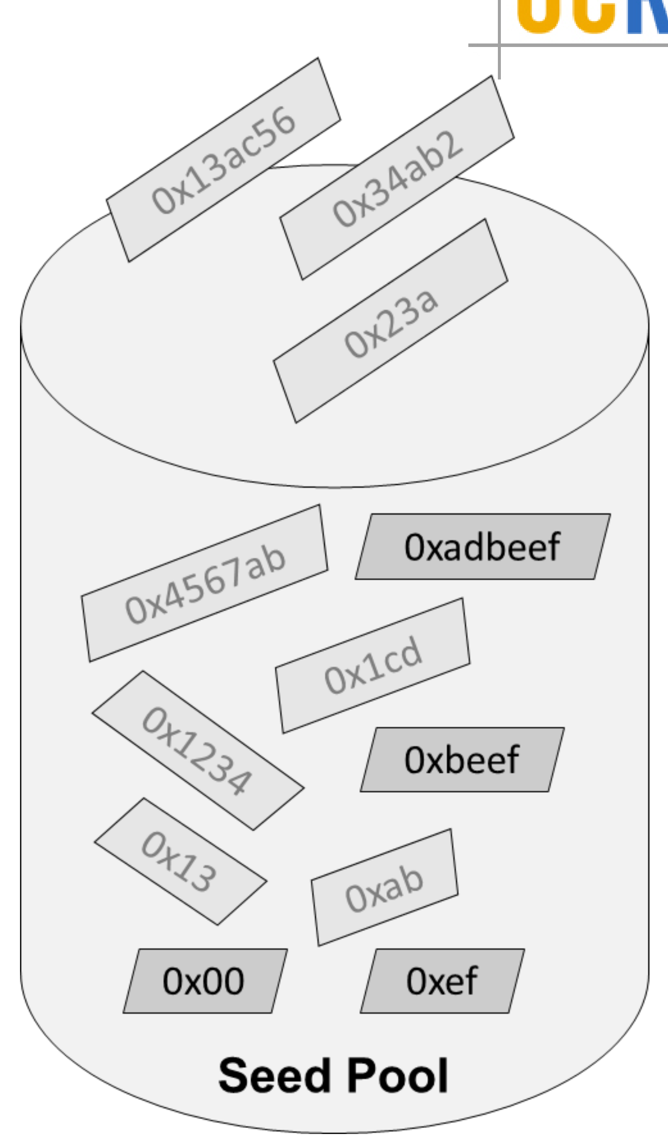
Our Solution:

Reinforcement Learning-based Hierarchical Seed Scheduling

The more sensitive, the better?

Seed Explosion

- Many more seeds that exceed the fuzzer's ability to schedule
- Given a fixed fuzzing campaign time
  - Many fresh but useful seeds may never be fuzzed
  - Important seeds may be not fuzzed enough time



A Multi-level Coverage Metric

- Seed pool is organized into a hierarchical tree
  - Internal nodes are coverage measurements and leaf nodes are seeds
  - An internal node represents a cluster of seeds with the same coverage

examined node/seed

scheduled node/seed

- M_F: function coverage
- M_E: edge coverage
- M_D: distance coverage

Seed scheduling is to seek a path from the root to a leaf node

Seed Exploitation & Exploration

- Exploration: try out other fresh nodes
  - Fresh nodes that have rarely been fuzzed may lead to surprisingly new coverage
- Exploitation: keep fuzzing interesting nodes to trigger a breakthrough
  - A few valuable nodes that have led to significantly more new coverage than others in recent rounds encourage to focus on fuzzing them

Fuzzing & MAB Model

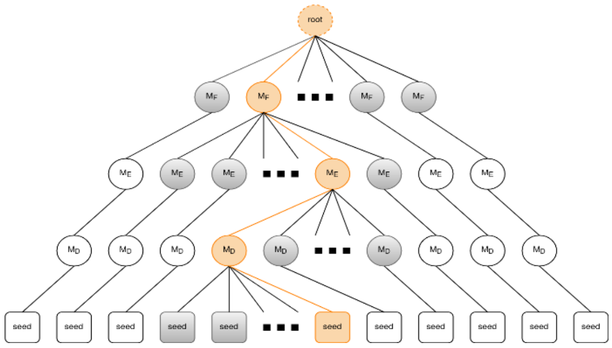
- We model the fuzzing process as a multi-armed bandit (MAB) problem
- We adopt the UCB1 algorithm to schedule seeds within levels to manage the balance between seed exploration and exploitation.

A reinforcement learning-based hierarchical seed scheduler

RL-based Hierarchical Seed Scheduling

UCR

- Scheduling
  - Internal level:
    - For each node, a **score** is calculated following the MAB model
    - Starting from the root node, select the child node with the highest score
  - Leaf level:
    - Select a seed with round-robin
- Rewarding
  - At the end of each fuzzing round, nodes along the scheduled path will be rewarded based on how much progress the current seed has made in this round.
    - Whether there is new coverage exercised by the generated test cases



Seed Scoring

UCB1

Score = (Reward + Uncertainty) * Rareness

We focus on nodes that have generated test cases covering rare features recently

We periodically try nodes that have been rarely fuzzed or contain many seeds

We prioritize nodes that exercise rare coverage features

Seed Rewarding

Score = (Reward + Uncertainty) * Rareness

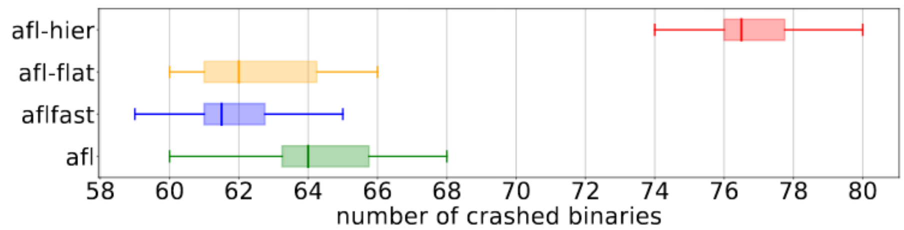
- We favor newer rewards than old ones
- We propagate rewards from lower to upper levels

Evaluation

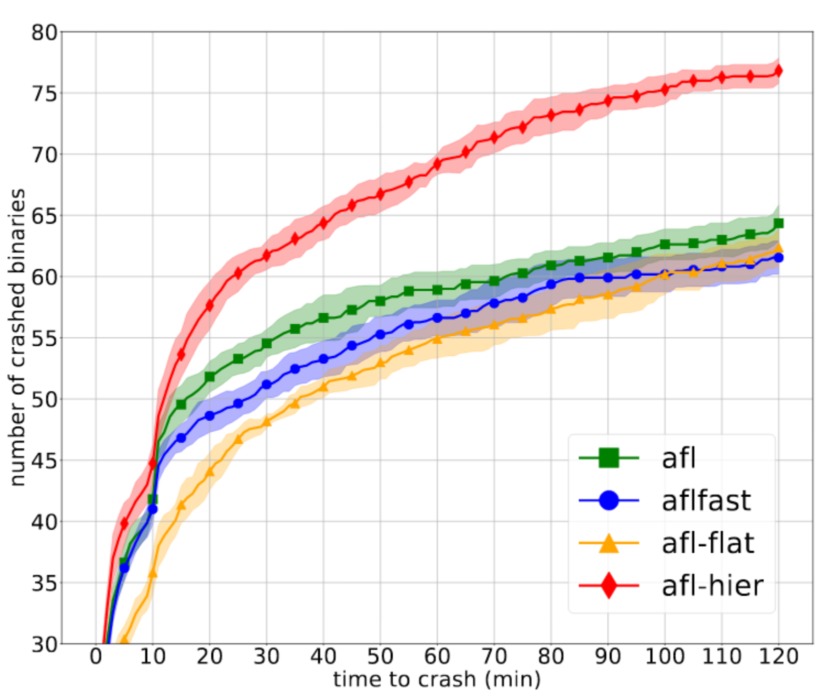
- Evaluation setup
  - Benchmarks
    - CGC (Darpa Cyber Grand Challenge), 180 binaries
    - Google FuzzBench, 20 real-world programs
  - Baseline fuzzers
    - CGC (vs AFL-Hier: $M_F + M_E + M_D$)
      - AFL
      - AFLFast
      - AFL-Flat (the same coverage metrics, but with the fast scheduler from AFLFast)
    - FuzzBench (vs AFL++-Hier)
      - AFL++
      - AFL++-Flat

Evaluation

- Bug detection



(a) Number of crashed CGC binaries.



(b) Number of CGC binaries crashed over time.

AFL-Hier crashes more CGC binaries and faster. Especially, it crashes the same number of binaries in 30 minutes, which AFLFast crashes in 2 hours

Evaluation

- Edge coverage

On FuzzBench, AFL++-Hier achieves higher coverage on 10 out of 20 programs

Fig. 5: Mean coverage in a 6 hour fuzzing campaign on FuzzBech benchmarks.

But still: it is hard to determine which metric to use

- Feedback/metric is important for fuzzing
- Humans have good insight
- Let’s add annotations to guide fuzzing process
- IJON: Exploring Deep State Spaces via Fuzzing, IEEE Security and Privacy 2020

An Example: Maze

- https://raw.githubusercontent.com/grese/klee-maze/master/maze.c
  - Klee can solve this version
  - AFL cannot

- A harder version
  - Neither can solve
  - AFL with Memory-Access and Memory-Write Metrics can
  - Why?

Add an IJON annotation

```
while(true) {
    ox=x; oy=y;

    IJON_SET(hash_int(x,y));
    switch (input[i]) {
        case 'w': y--; break;
//....
```

Listing 6: Annotated version of the maze.

Another Example: Protocol Fuzzing

```
msg = parse_msg();
switch(msg.type) {
    case Hello: eval_hello(msg); break;
    case Login: eval_login(msg); break;
    case Msg_A: eval_msg_a(msg); break;
}
```

Listing 2: A common problem in protocol fuzzing.

Annotations for Protocol Fuzzing

```
//abbreviated libtpms parsing code in ExecCommand.c
msg = parse(msg);
err = handle(msg);
if(err != 0){goto Cleanup;}

state_log=(state_log<<8)+command.index;
IJON_SET(state_log);
```

Listing 7: Annotated version of libtpms.

```
IJON_STATE(has_hello + has_login);
msg = parse_msg();
//...
```

Listing 8: Annotated version of the protocol fuzzing example (using IJON-STATE).

Another Example: Super Mario Bros

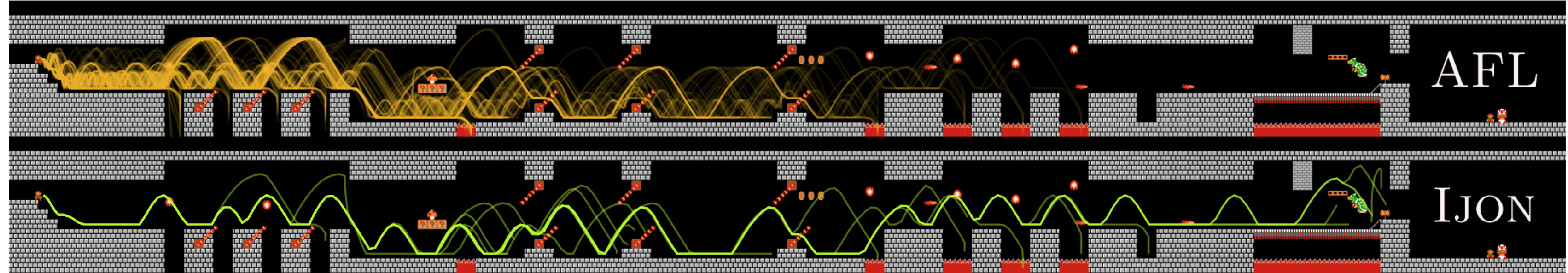


Fig. 1: AFL and AFL + IJON trying to defeat Bowser in Super Mario Bros. (Level 3-4). The lines are the traces of all runs found by the fuzzer.

```
//inside main loop, after calculating positions
IJON_MAX(player_y, player_x);
```

Listing 9: Annotated version of the game Super Mario Bros.

An Interesting Question

- Can LLM help annotate the program for fuzzing?

Much More about Fuzzing

- Mutation Strategies
  - Schedule the most effective mutations
  - Grammar/structure aware mutations
  - LLMs
- Hybrid Fuzzing: Combining Fuzzing and SE
  - AFL is dominant; What can SE do?
- Directed Fuzzing
  - Drive executions to a target code location

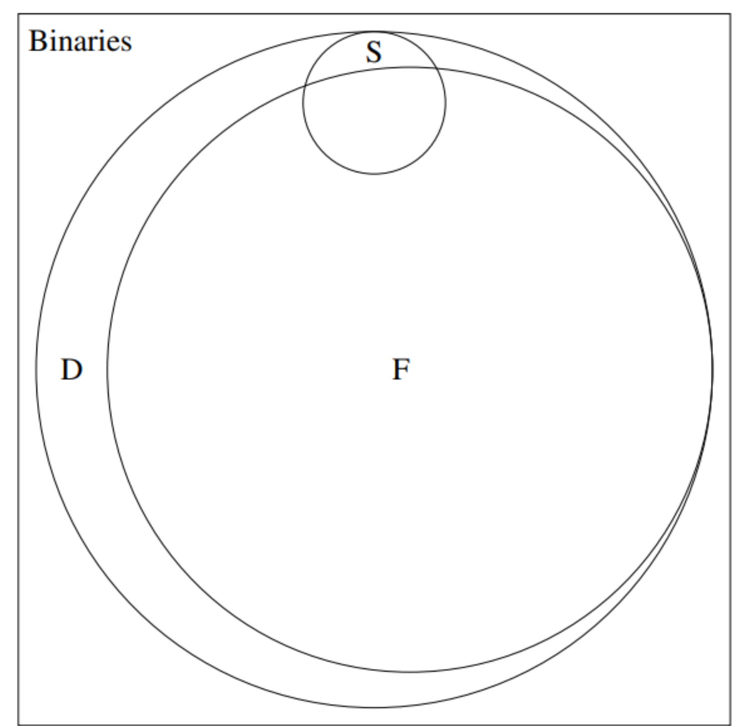
Hybrid Fuzzing

- Fuzzing
  - Fast: can explore large program space quickly
  - Dumb: cannot penetrate narrow conditions easily
- Symbolic Execution
  - Slow: take long time to process one input
  - Smart: can penetrate narrow conditions easily

- Question: how to combine them?

Driller (NDSS 2017)

- When AFL gets stuck, invoke Anger
- For each seed, conduct concolic execution
- For each encountered symbolic branch, flip this branch if the unvisited direction is not in the AFL bitmap



Method	Crashes Found
Fuzzing	68
Fuzzing ∩ Driller	68
Fuzzing ∩ Symbolic	13
Symbolic	16
Symbolic ∩ Driller	16
Driller	77

Limitations of Driller

- Fuzzer getting stuck is not a good indicator
  - 49 out of 118 binaries ever got stuck
  - 85% of stuck time periods are under 100s
- There are significantly more seeds than SE can handle
  - Angr takes 1654 seconds to process one input
  - Only 7.1% of seeds are processed by Angr

DigFuzz (NDSS 2019)

Probabilistic path prioritization model based on Monte Carlo

Initial input

Fuzzing

Execution tree construction

Execution sampling

b_1 b_2 b_3 b_4 b_6 b_8

Probability based path prioritization

Prioritized paths

Concolic execution

New inputs

$$P\left(br_i\right) = \begin{cases} \frac{cov(br_i)}{cov(br_i)+cov(br_j)}, & cov\left(br_i\right) \neq 0 \\ \frac{3}{cov(br_j)}, & cov\left(br_i\right) = 0 \end{cases} \qquad (1)$$

$$P\left(path_j\right) = \prod \left\{ P\left(br_i\right) \middle| br_i \in path_j \right\} \qquad (2)$$

DigFuzz Evaluation

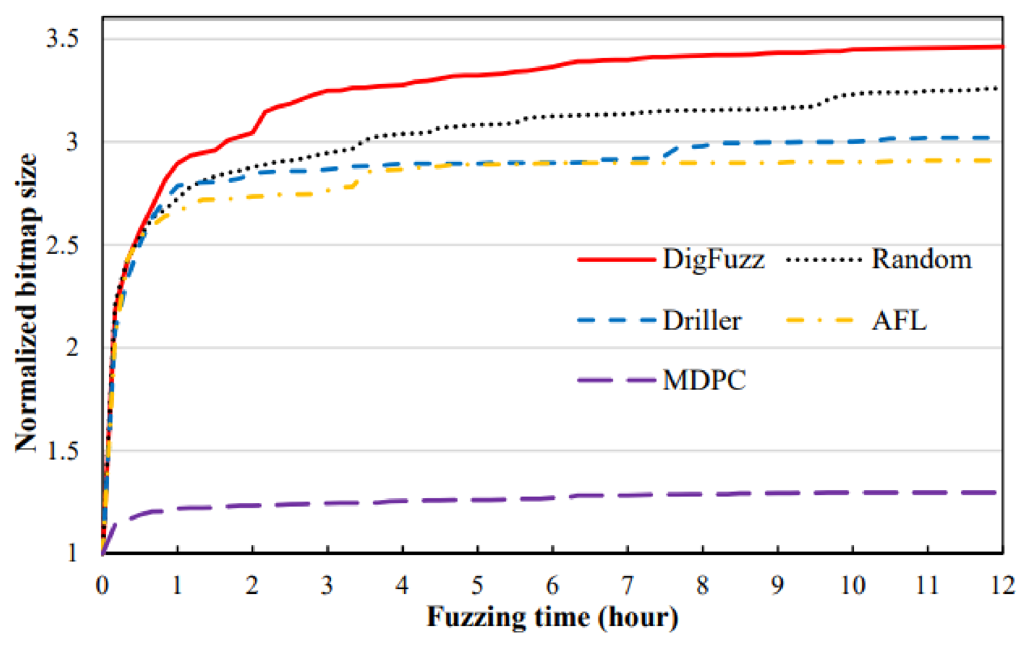


Fig. 6: Normalized bitmap size on CQE dataset

TABLE II: Number of discovered vulnerabilities

	$= 3$	≥ 2	≥ 1
DigFuzz	73	77	81
Random	68	73	77
Driller	67	71	75
AFL	68	70	73
MDPC	29	29	31

Looking Ahead

- Fuzzers become smarter
  - Branch-distance-guided search (e.g., Angora)
  - CmpLog for comparison-aware input mutation
  - LAF-Intel (“Split Compare”) to simplify hard comparisons
  - CompCov (CompareCoverage in QEMU/Unicorn) for comparison feedback
- Symbolic execution becomes faster and smarter
  - SymSan and SymFit
  - Marco: better path exploration
- How can AI/LLMs help?