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 /AAAAAAAAAAAAAA.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…

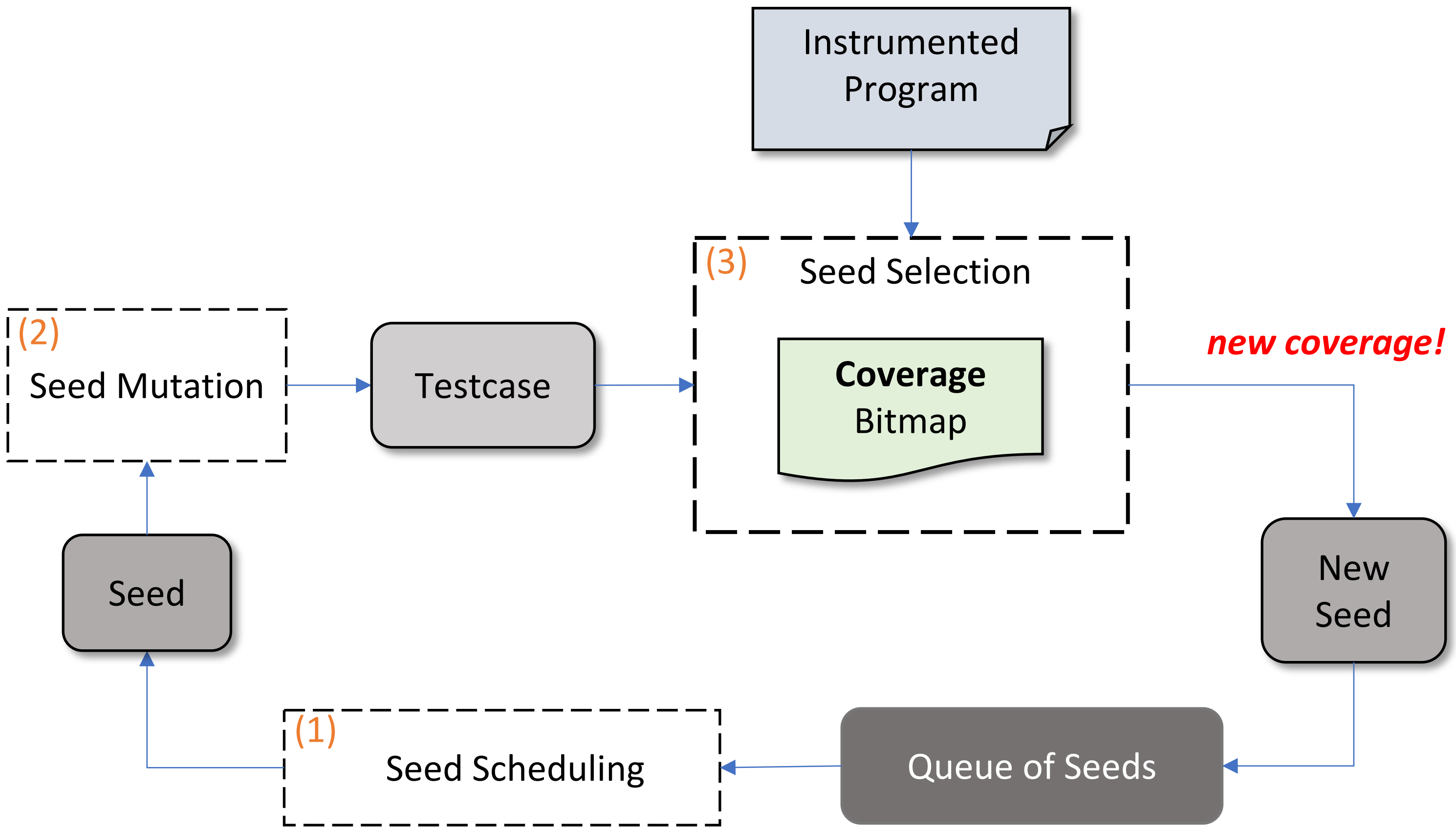
Whitebox Fuzzing (2012)

FIGURE 2 Example of Program (Left) and Its Search Space (Right) with the Value of cnt at the End of Each Run

```
void top(char input[4] {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 4) abort(); ?? error
}
```

- 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



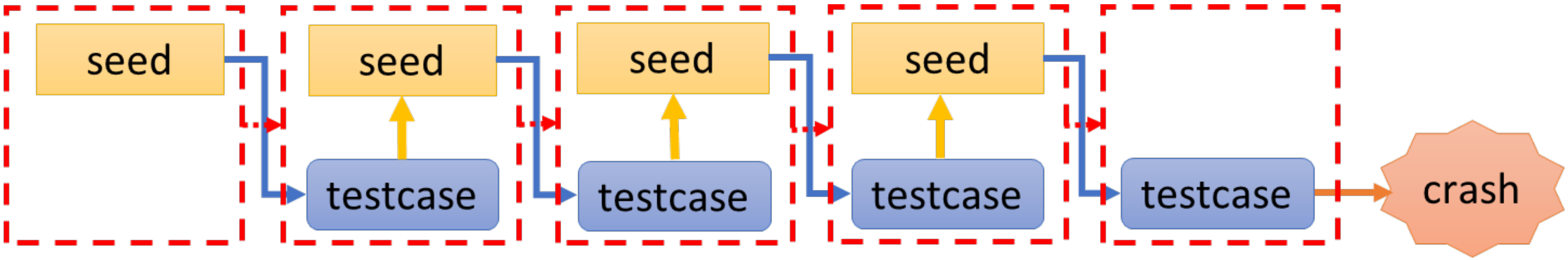
Coverage Metric

- 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

Formal Definition of Sensitivity

Given two coverage metrics C_i and C_j ,

C_i is *"more sensitive"* than C_j if

(1) $\forall P \in \mathcal{P},\ \forall I_1, I_2 \in \mathcal{I},\ C_i(P, I_1) = C_i(P, I_2) \rightarrow C_j(P, I_1) = C_j(P, I_2)$, and

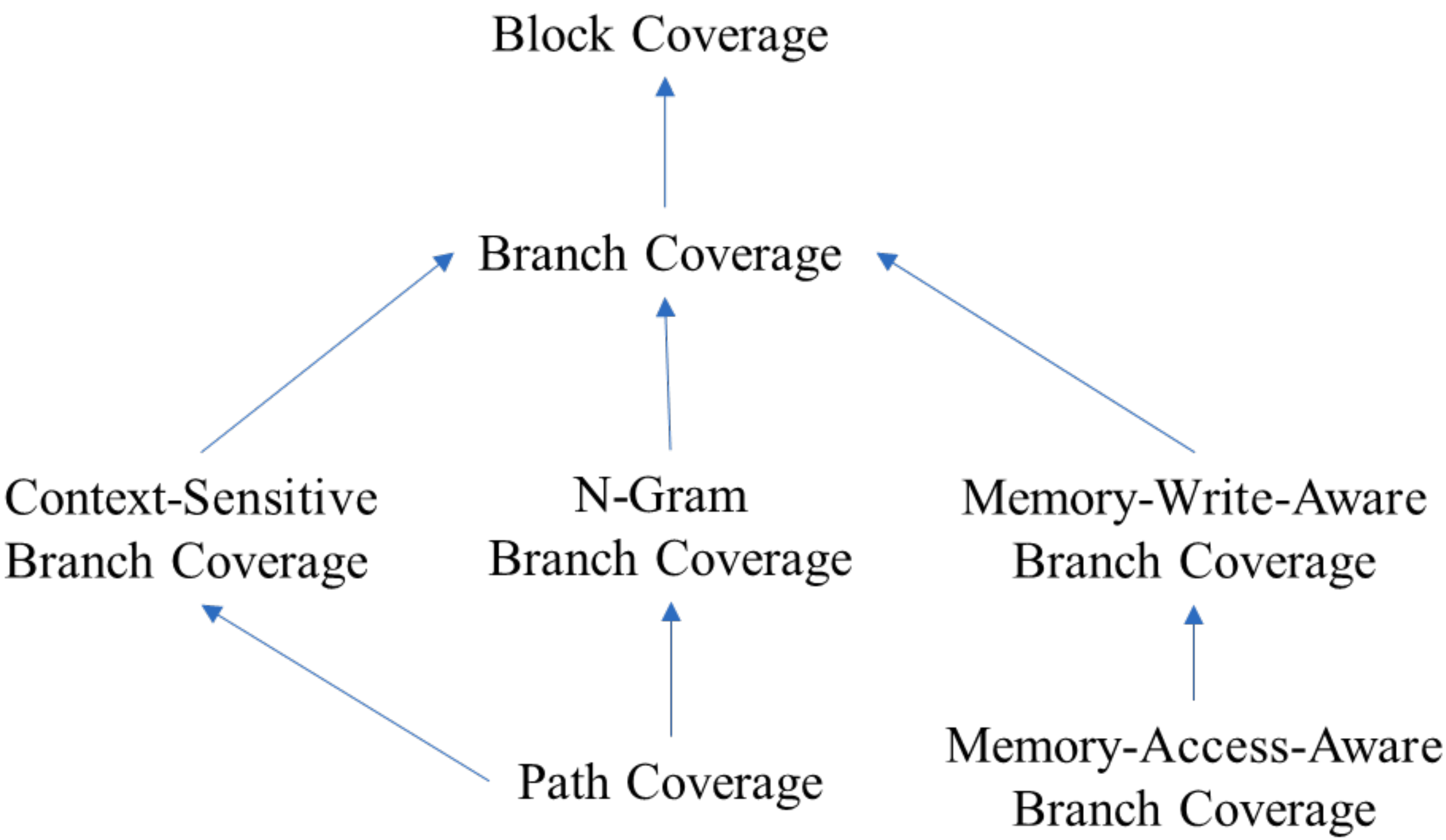
(2) $\exists P \in \mathcal{P},\ \exists I_1, I_2 \in \mathcal{I},\ C_j(P, I_1) = C_j(P, I_2) \wedge C_i(P, I_1) \neq C_i(P, I_2)$

where coverage metric C is defined as a function that takes a program *P, an input I and produces a measurement M = C(P, I)*

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

Sensitivity Lattice

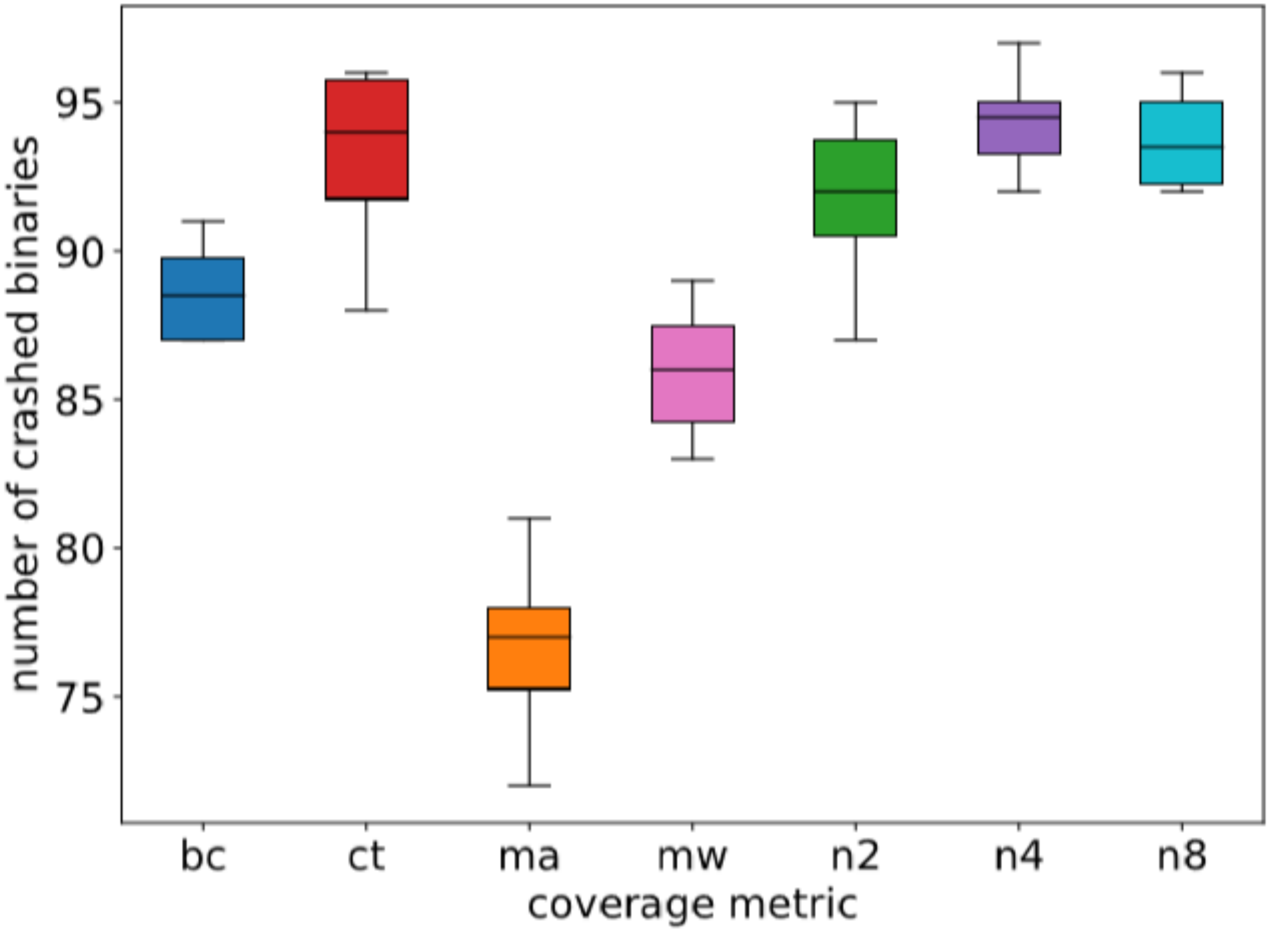


Implementation

- Based on AFL
  - Instrumentation via user-model 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

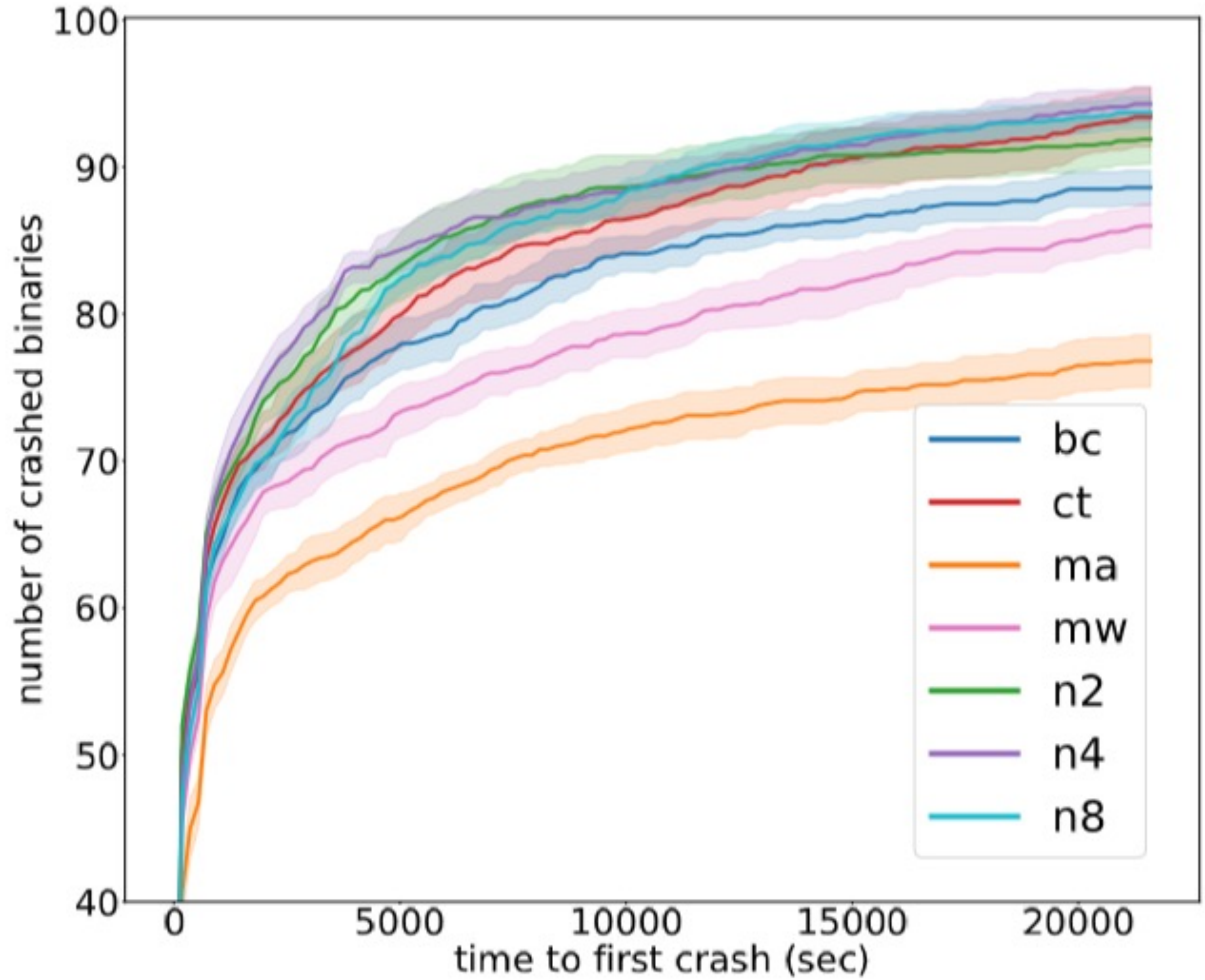
- Available at https://github.com/bitsecurerlab/afl-sensitive

Comparison of Unique Crashes



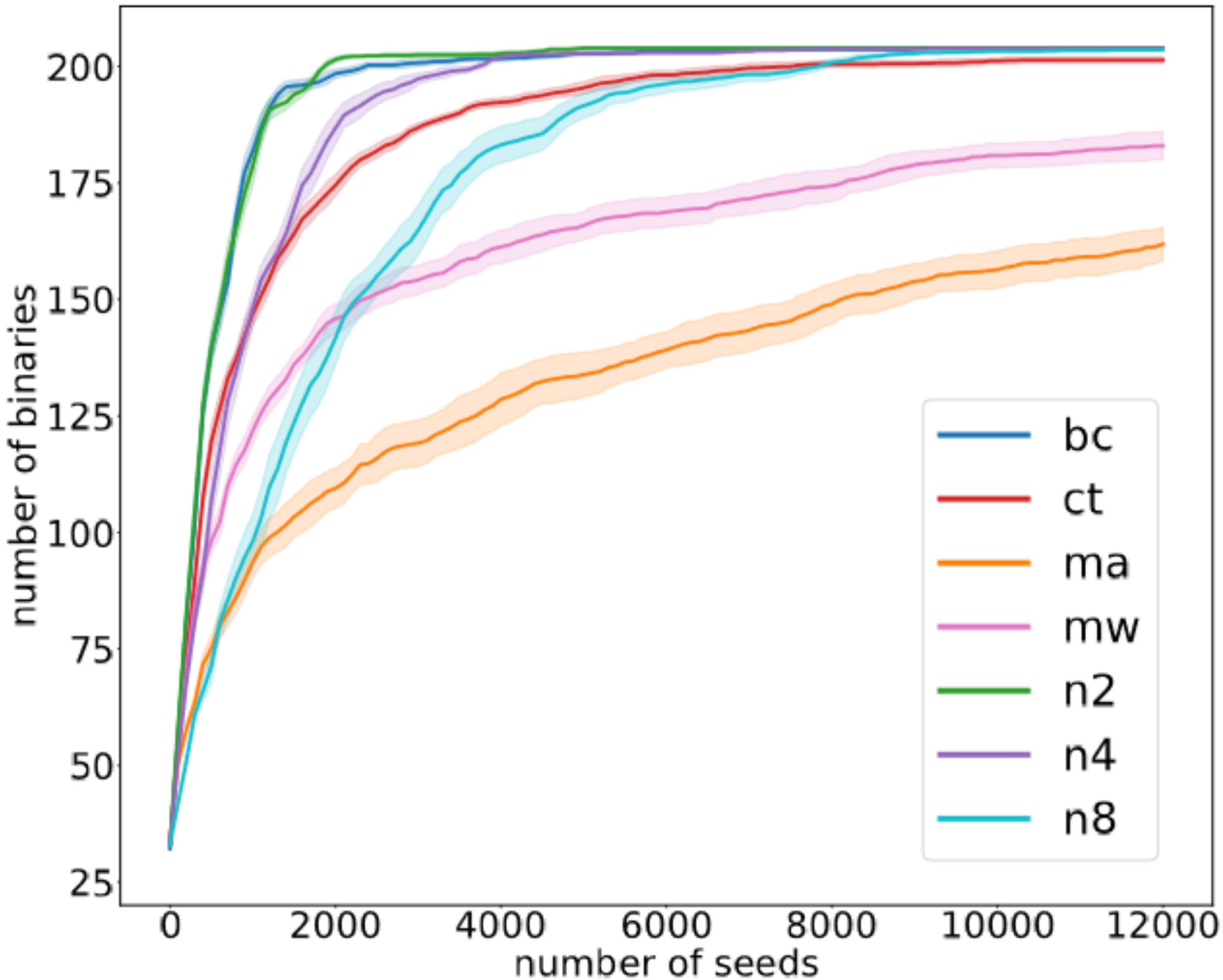
Number of CGC binaries crashed by different coverage metrics

Comparison of Time to First Crash



Number of CGC binaries crashed overtime during fuzzing

Comparison of Seed Count

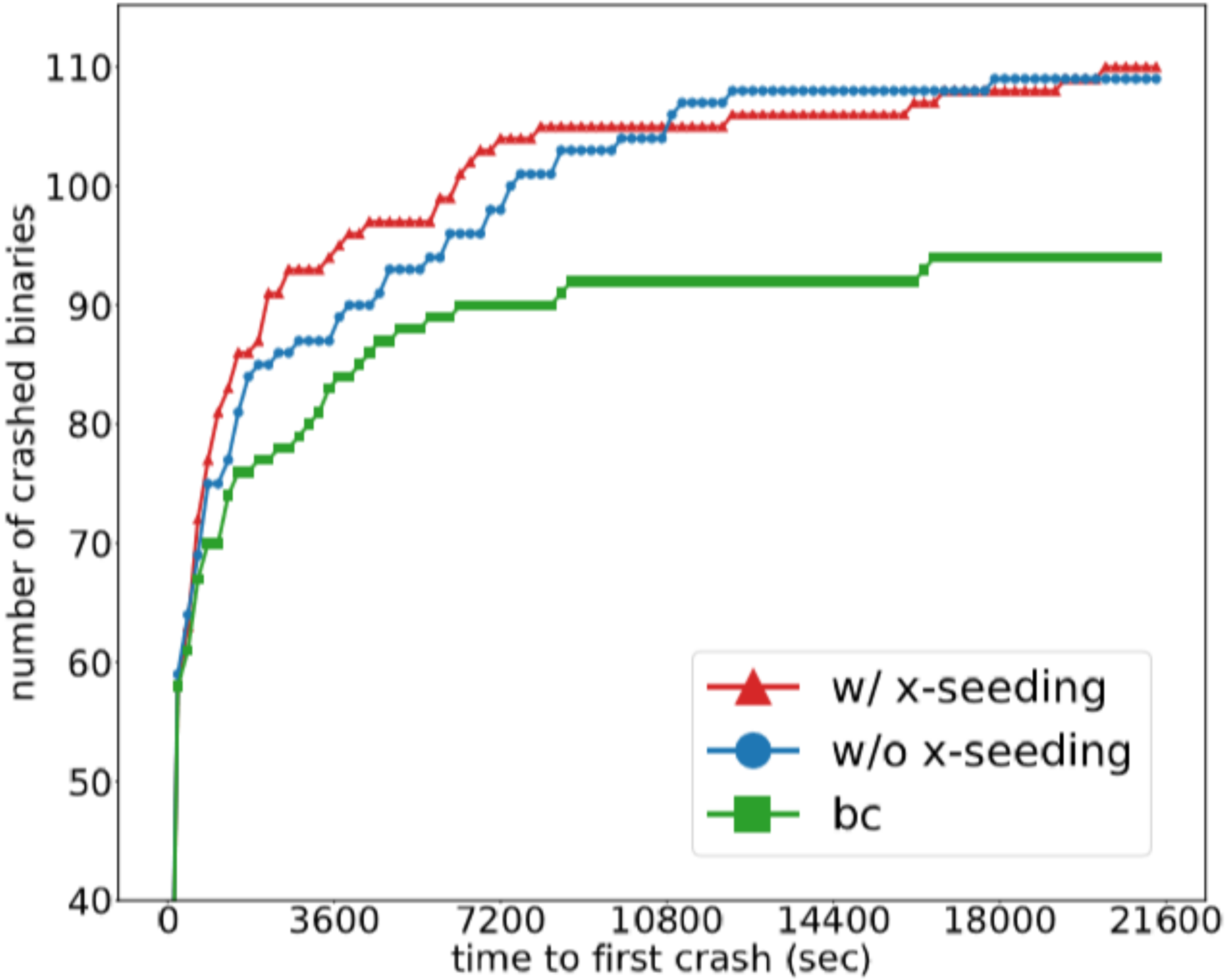


Partial CDFs of seeds generated by different coverage metrics on the CGC dataset. A curve closer to the top left indicates fewer generated seeds.

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?

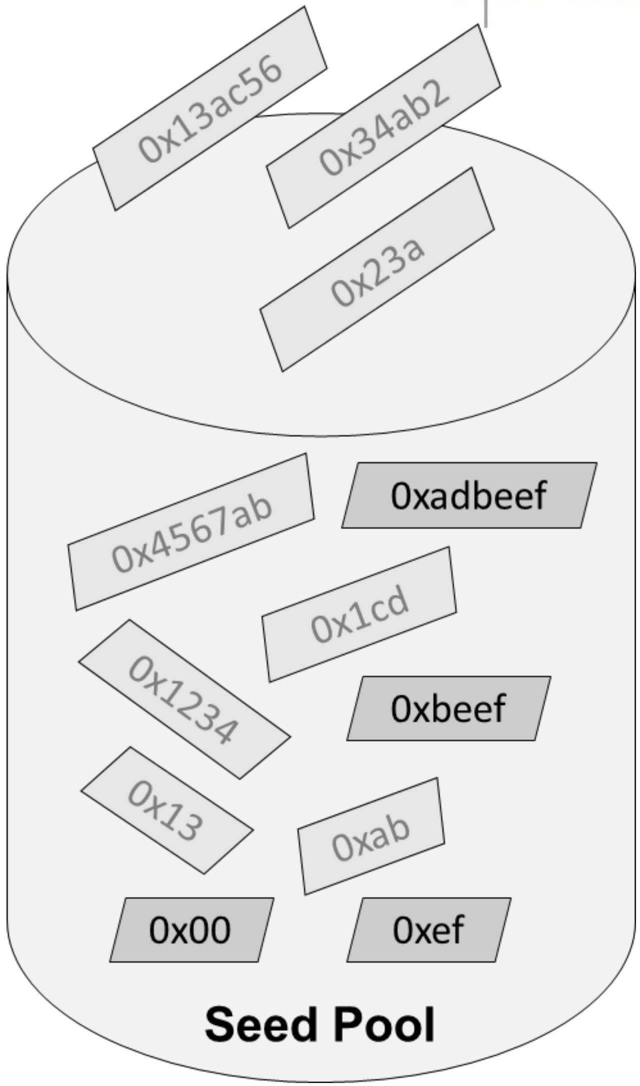
Our Solution:

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



The more sensitive, the better?

The coverage metric and the corresponding seed scheduler should be carefully crafted

Challenge 1: too many (similar) seeds to examine

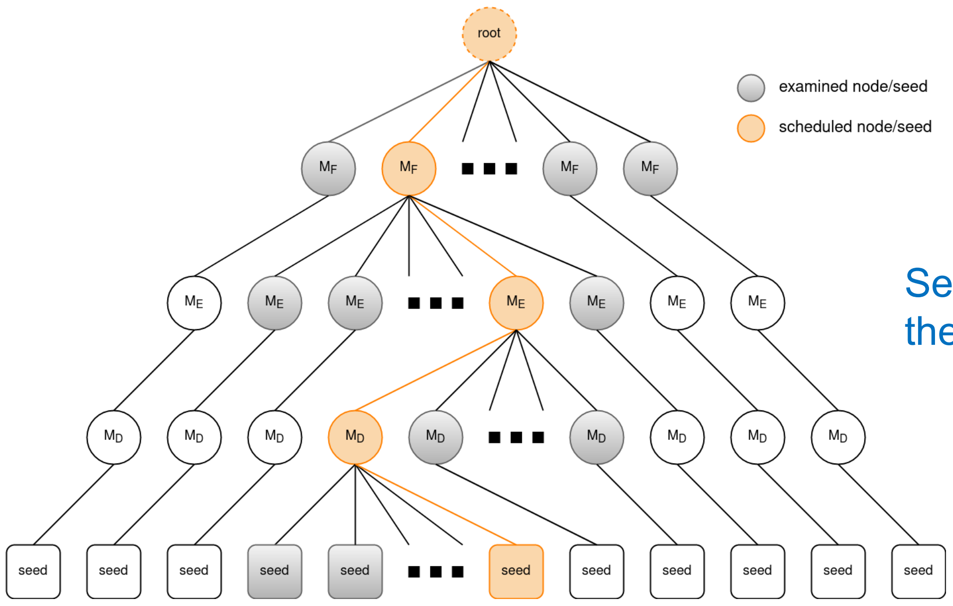
Seed Clustering

- We use a less sensitive metric to cluster seeds selected by a more sensitive metric
- We use more than one level of clustering

→ A multi-level coverage metric

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



- 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

Challenge 2: seed exploration vs exploitation

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

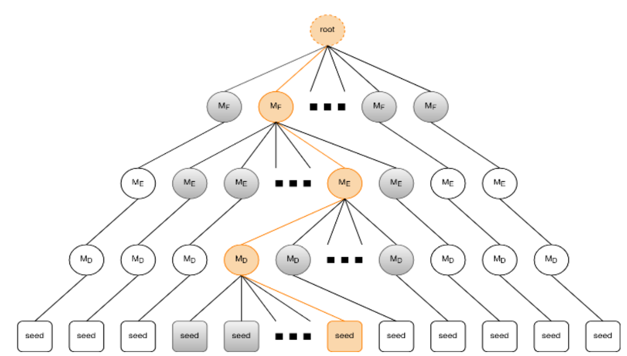
Fuzzing & MAB Model

- 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

- 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

- Reward: We focus on nodes that have generated test cases covering rare features recently
- Uncertainty: We periodically try nodes that have been rarely fuzzed or contain many seeds
- Rareness: We prioritize nodes that exercise rare coverage features

Seed Rewarding

Score = (Reward + Uncertainty) * Rareness

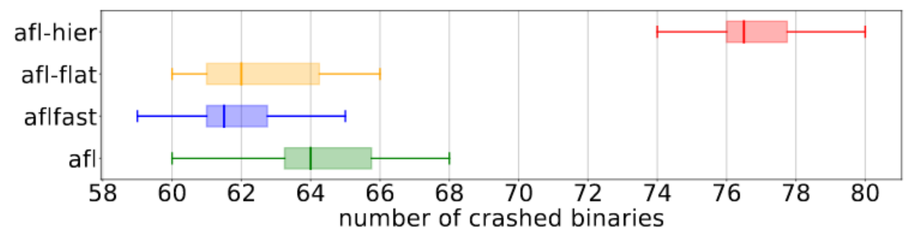
- We favor newer rewards than old ones
- We propagate rewards from lower to upper levels

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

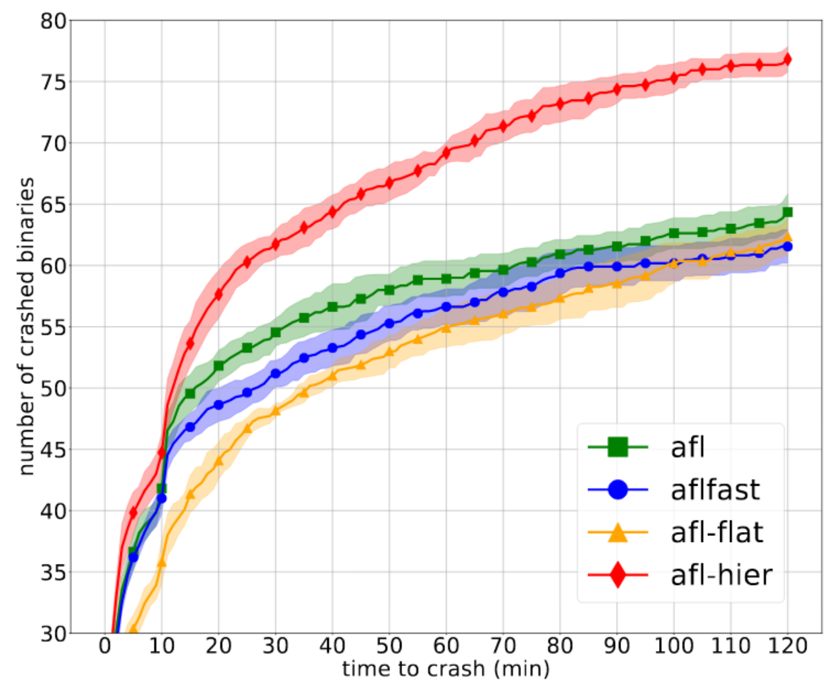
Evaluation

- Bug detection



(a) Number of crashed CGC binaries.

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



(b) Number of CGC binaries crashed over time.

Evaluation

- Edge coverage

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

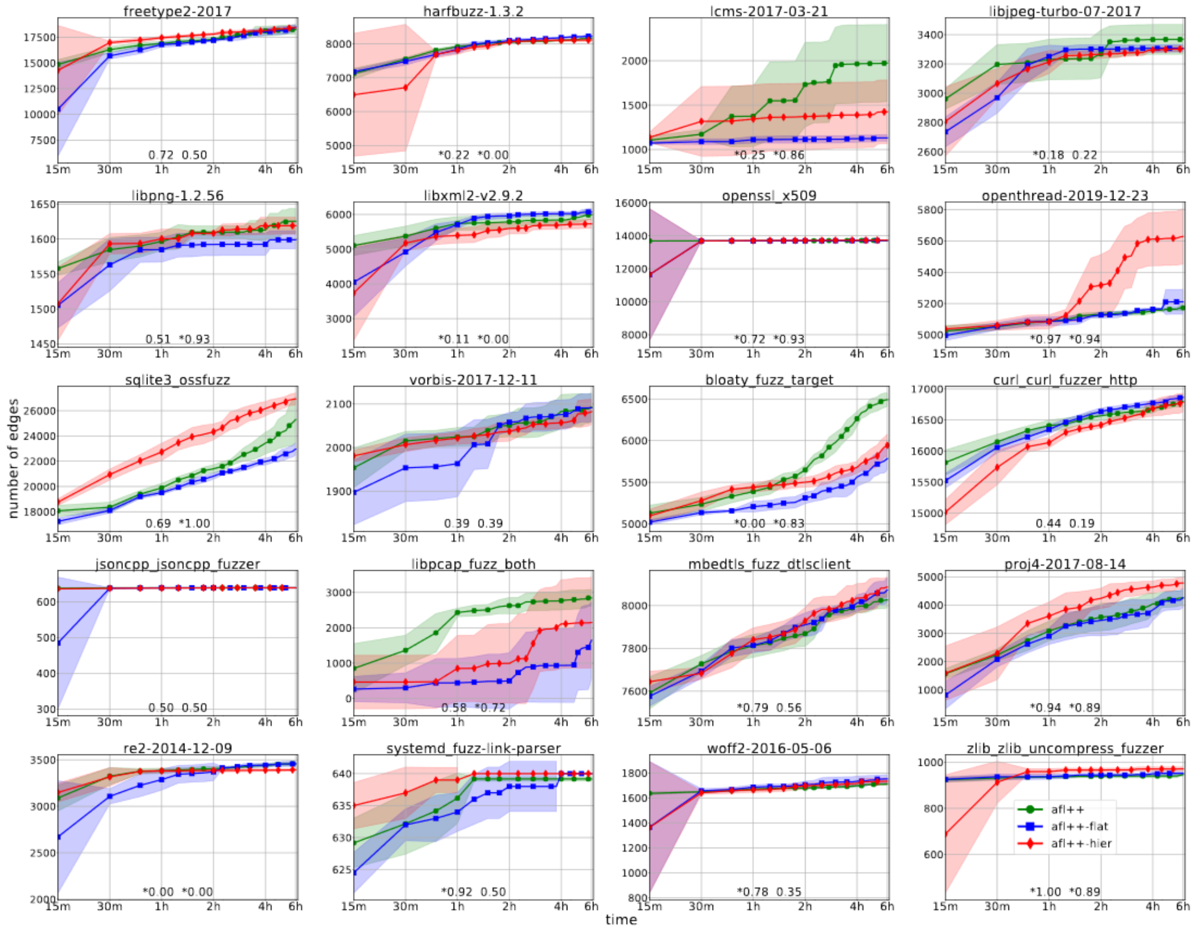


Fig. 5: Mean coverage in a 6 hour fuzzing campaign on FuzzBech benchmarks.

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

- A good resource: https://fuzzingbook.org

IJON: Exploring Deep State Spaces via Fuzzing

IEEE Security and Privacy 2020

Basic Idea

- Feedback is important for fuzzing
- Humans have good insight
- Let’s add annotation to guide fuzzing process

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
  - 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

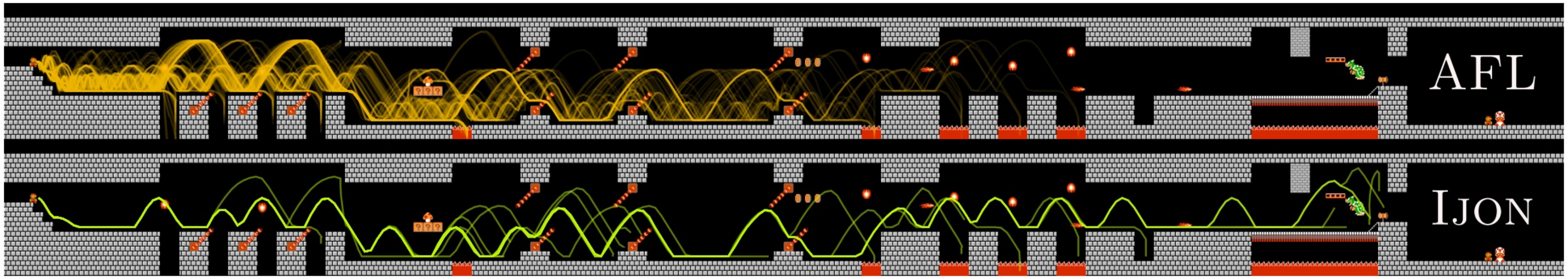


Fig. 1: AFL and AFL + IJON trying to defeat Bowser in Super Mario Bros. (Level 3-4). The lines are the traces of all runs found by the fuzzer.

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

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

Another Example: Hash Map Lookup

```
//shortened version of a hashmap lookup from binutils
entry* bfd_get_section_by_name(table *tbl, char *str) {
  entry *lp;
  uint32_t hash = bfd_hash_hash(str);
  uint32_t i = hash % tbl->size;

  //Every hash bucket contains a linked list of strings
  for (lp = tbl->table[i]; lp != NULL; lp = lp->next) {
    if (lp->hash == hash && strcmp( lp->string, str) == 0)
            return lp;
  }
  return NULL;
}

// used somewhere else
section = bfd_get_section_by_name (abfd, ".bootloader");
if (section != NULL){ ... }
```

Listing 3: A hash map lookup that is hard to solve (from binutils libbfd, available at bfd/section.c).

Annotated Version

```
// callback used to iterate the hash map
void ijon_hash_feedback(bfd_hash_entry* ent, char* data){
    IJON_SET(IJON_STRDIST(ent->string, data));
}

//shortened version of a hashmap lookup from binutils
entry* bfd_get_section_by_name(table *tbl, char *str) {
//perform a string feedback for each entry in the hashmap.
 bfd_hash_traverse(tab, ijon_hash_feedback, str);
//....  rest of the function as shown earlier.
}
```

Listing 11: Annotated version of the hash map example.

More details

- https://github.com/RUB-SysSec/ijon

```
void ijon_enable_feedback();
void ijon_disable_feedback();

#define _IJON_CONCAT(x, y) x##y
#define _IJON_UNIQ_NAME() IJON_CONCAT(temp,__LINE__)
#define _IJON_ABS_DIST(x,y) ((x)<(y) ? (y)-(x) : (x)-(y))

#define IJON_BITS(x) ((x==0)?{0}:__builtin_clz(x))
#define IJON_INC(x) ijon_map_inc(ijon_hashstr(__LINE__,__FILE__)^(x))
#define IJON_SET(x) ijon_map_set(ijon_hashstr(__LINE__,__FILE__)^(x))

#define IJON_CTX(x) ({ uint32_t hash = hashstr(__LINE__,__FILE__); ijon_xor_state(hash);
__typeof__(x) IJON_UNIQ_NAME() = (x); ijon_xor_state(hash); IJON_UNIQ_NAME(); })

#define IJON_MAX(x) ijon_max(ijon_hashstr(__LINE__,__FILE__),(x))
#define IJON_MIN(x) ijon_max(ijon_hashstr(__LINE__,__FILE__),0xffffffffffffffff-(x))
#define IJON_CMP(x,y) IJON_INC(__builtin_popcount((x)^(y)))
#define IJON_DIST(x,y) ijon_min(ijon_hashstr(__LINE__,__FILE__), _IJON_ABS_DIST(x,y))
#define IJON_STRDIST(x,y) IJON_SET(ijon_hashint(ijon_hashstack(), ijon_strdist(x,y)))
```

Lab 2 Assignment

- Experimenting with Symbolic Execution and Fuzzing
- Pick Some CGC Challenge Programs
  - https://github.com/hengyin/cb-multios/tree/master/challenges
- Try Klee and AFL (or AFL++)
  - Can they solve these challenges?
  - How much is the code coverage?
- Try to add IJON annotations
  - Can your added annotations improve code coverage and solve these challenges?