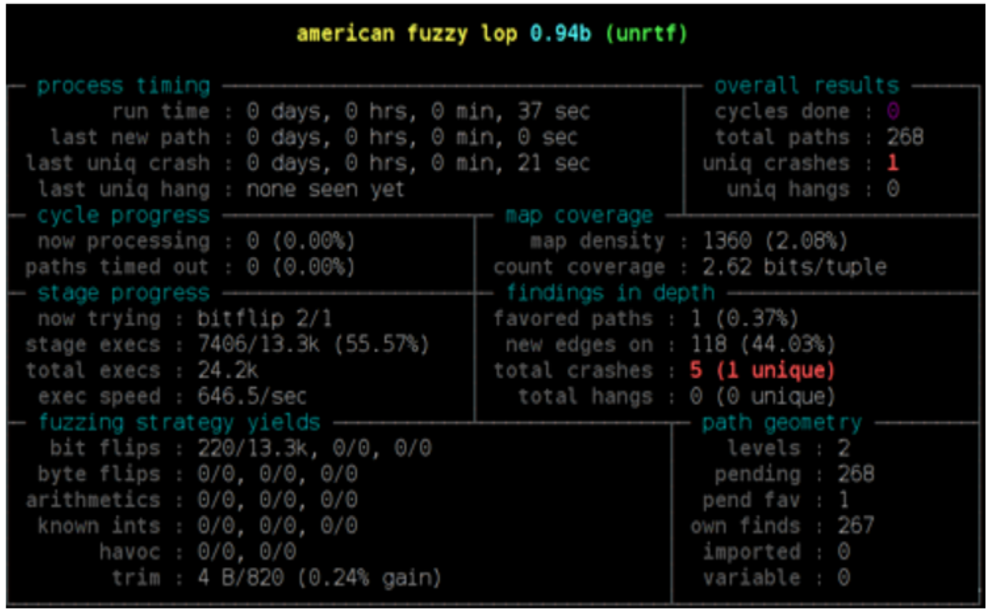
Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing

Lei Zhao, Yue Duan, Heng Yin, Jifeng Xuan

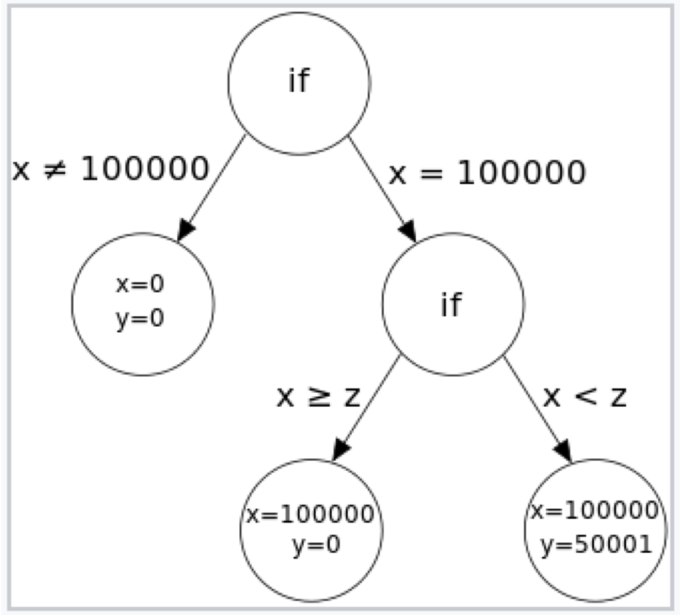
Wuhan University, China
University of California Riverside

Motivation

- Automatic vulnerability detection techniques



```
                     american fuzzy lop 0.94b (unrtf)

- process timing -------------------------------------- overall results ---
        run time : 0 days, 0 hrs, 0 min, 37 sec        cycles done : 0
   last new path : 0 days, 0 hrs, 0 min, 0 sec         total paths : 268
 last uniq crash : 0 days, 0 hrs, 0 min, 21 sec       uniq crashes : 1
  last uniq hang : none seen yet                        uniq hangs : 0
- cycle progress ------------------------ map coverage ---------------------
  now processing : 0 (0.00%)               map density : 1360 (2.08%)
 paths timed out : 0 (0.00%)            count coverage : 2.62 bits/tuple
- stage progress ------------------------ findings in depth ----------------
  now trying : bitflip 2/1              favored paths : 1 (0.37%)
 stage execs : 7406/13.3k (55.57%)       new edges on : 118 (44.03%)
 total execs : 24.2k                    total crashes : 5 (1 unique)
  exec speed : 646.5/sec                  total hangs : 0 (0 unique)
- fuzzing strategy yields ------------------------------ path geometry -----
   bit flips : 220/13.3k, 0/0, 0/0                        levels : 2
  byte flips : 0/0, 0/0, 0/0                             pending : 268
 arithmetics : 0/0, 0/0, 0/0                            pend fav : 1
  known ints : 0/0, 0/0, 0/0                           own finds : 267
       havoc : 0/0, 0/0                                 imported : 0
        trim : 4 B/820 (0.24% gain)                     variable : 0
```



Fuzzing

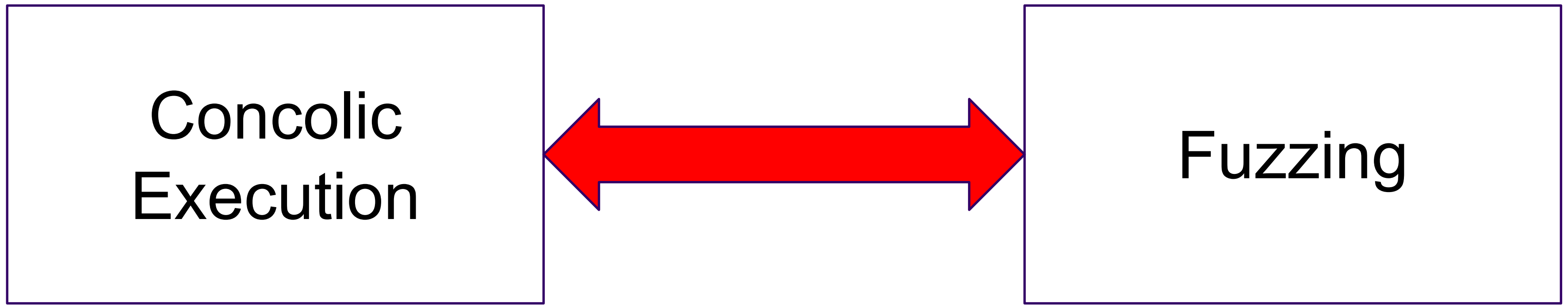
- Pros: scalable and efficient
- Cons: hard to generate satisfying inputs for specific conditions

Concolic Execution

- Pros: generate concrete inputs for a specific path
- Cons: path explosion, heavyweight

Motivation

- Hybrid fuzzing
  - Fuzzing and concolic execution are complementary in nature



  - High throughput by making fuzzing take the majority task of path exploration
  - Alleviate path explosion as concolic execution is directed for specific branches

State-of-the-art Hybrid Fuzzing

- Demand Launch: Driller (NDSS'16), Hybrid Concolic Testing(ICSE'07)
  - General idea: launch concolic execution when fuzzer gets stuck (blocked by condition checks)

  - Assumptions:
    1. fuzzer in non-stuck state ⇒ concolic execution is not needed.
    2. stuck state ⇒ fuzzer cannot make progress
    3. concolic execution is able to find and solve the hard-to-solve condition problems that block the fuzzer

  - Question:
    - Do these assumptions hold?

State-of-the-art Hybrid Fuzzing

- Optimal Strategy Markov Decision Processes with Costs (ICSE'18)
  - Insight: estimating the costs and always selects the best one
    - cost of fuzzing based on coverage statistics
    - cost of concolic execution based on constraints complexities
  - Assumptions:
    1. estimation is accurate and fast
    2. decision making is lightweight
  - Question:
    - How practical is the MDPC technique?

State-of-the-art Hybrid Fuzzing

- First systematic evaluation on hybrid fuzzing strategies
  - 118 binaries from DARPA Cyber Grand Challenge with 12 hours testing
- Findings for Demand Launch
  1. Concolic execution launched on only 49 out of 118 binaries
  2. 85% of the stuck time periods are under 100 seconds.

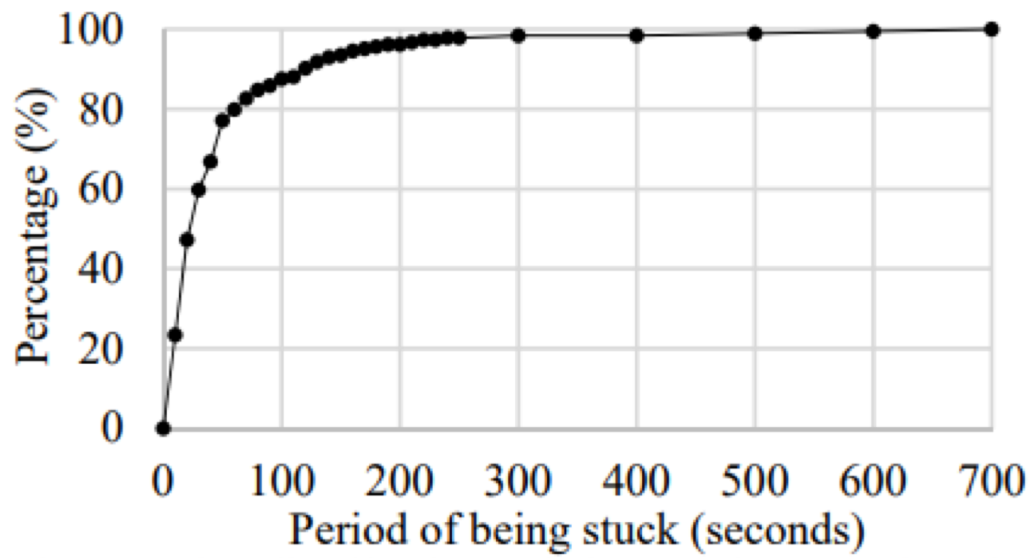


Fig. 1: The distribution of the stuck state duration

The stuck state is not a good indicator to decide whether the concolic execution is needed

State-of-the-art Hybrid Fuzzing

- Findings (cont.)

  3. Concolic execution on one input takes 1654 seconds on average

  4. Only 7.1% (1709 out of 23915) of the inputs retained by fuzzing are processed by concolic execution within the testing time.

  5. Fuzzer imports only a totally of 51 inputs on 13 binaries with 1709 runs of concolic execution

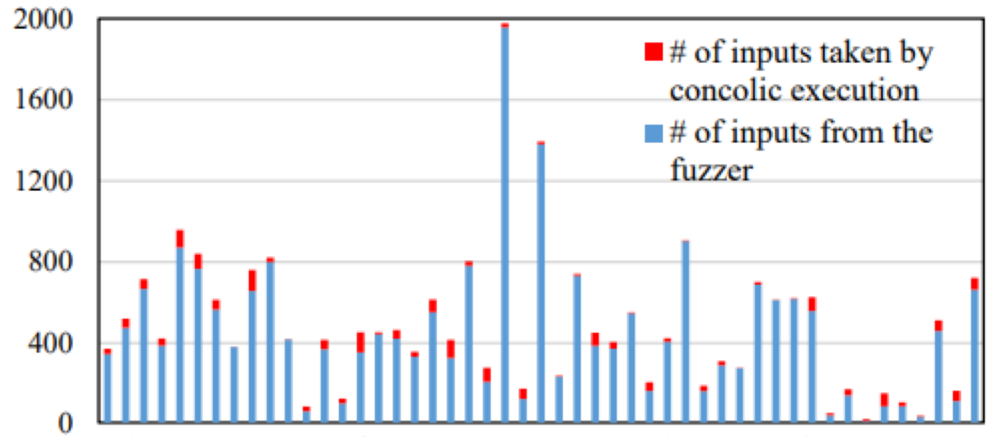


Fig. 2: The number of inputs retained by the fuzzer and the number of inputs taken by concolic execution.

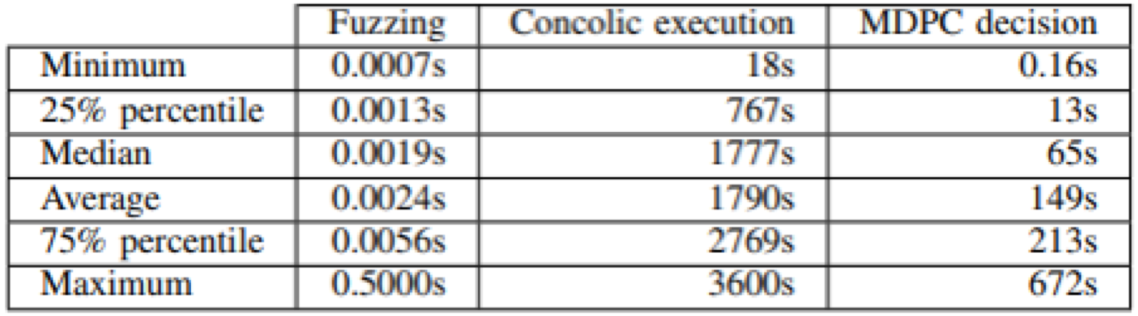
CE is too slow to process all inputs

After CE generates a good input, fuzzer may have already found one

State-of-the-art Hybrid Fuzzing

- Findings for Optimal Strategy
  1. MDPC decision making is heavyweight: several thousand times larger than fuzzing

TABLE I: Execution Time Comparison



	Fuzzing	Concolic execution	MDPC decision
Minimum	0.0007s	18s	0.16s
25% percentile	0.0013s	767s	13s
Median	0.0019s	1777s	65s
Average	0.0024s	1790s	149s
75% percentile	0.0056s	2769s	213s
Maximum	0.5000s	3600s	672s

  1. Throughput is significantly reduced:
     - from 417 eps (execution per second) in pure fuzzing to 2.6 eps
  2. MDPC discovers fewer vulnerabilities:
     - only in 29 binaries, whereas the pure fuzzing can discover vulnerabilities in 67 binaries.

Our Proposed Approach: Discriminative Dispatch

- Design principles:
  - Let fuzzing take the majority task of path exploration
  - Concolic execution only solves the hardest problems

- Key challenge:
  - quantify the difficulty of traversing a path for a fuzzer in a lightweight fashion. Any extra analysis must be lightweight to avoid negative impact on the performance of fuzzing

Probabilistic Path Prioritization

- Monte-Carlo Based Probabilistic Path Prioritization Model (*MCP*^{3})
  - Treat fuzzing as a sampling process
    - random sampling to the whole program space
    - large number of samples
  - Estimate branch probabilities based on Monte-Carlo Method

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

  - Estimate path probabilities as Markov Chain of successive branches

$$P\left(path_j\right)=\prod\left\{P\left(br_i\right)\left|br_i\in path_j\right.\right\}$$

Our Approach - Overview

- Iterative process:
  - *MCP*^{3} performs sampling, updates execution tree, calculates the probabilities of each path and prioritizes them.
  - Concolic executor generates new inputs along the path
  - Fuzzer takes the new inputs and further explores the program

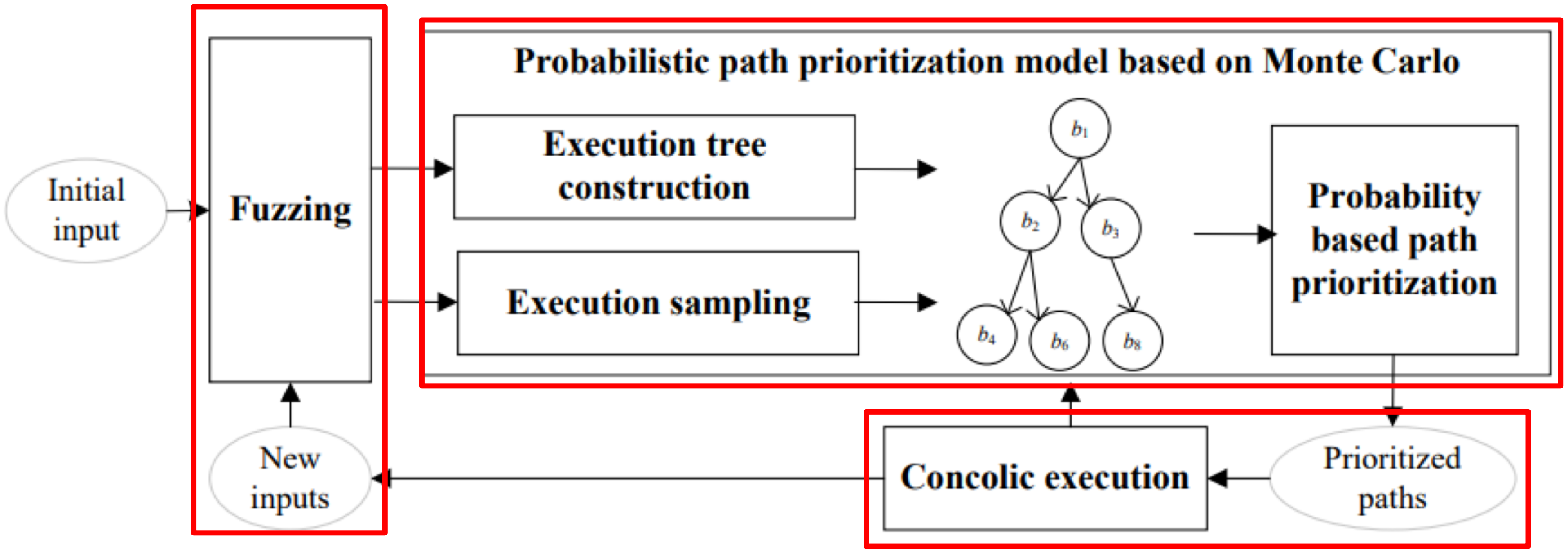

Initial input
Fuzzing
Probabilistic path prioritization model based on Monte Carlo
Execution tree construction
Execution sampling
b_1
b_2
b_3
b_4
b_6
b_8
Probability based path prioritization
Prioritized paths
Concolic execution
New inputs

Fig. 3: Overview of DigFuzz

Our Approach - Implementation Details

- A Fuzzing Component: AFL
  - Modify AFL to record the coverage statistics for every branch

- *MCP*^{3} model
  - Construct the execution tree based on execution traces
  - Calculate probabilities for missed branches and paths
  - Prioritize missed paths

- A concolic executor based on Angr

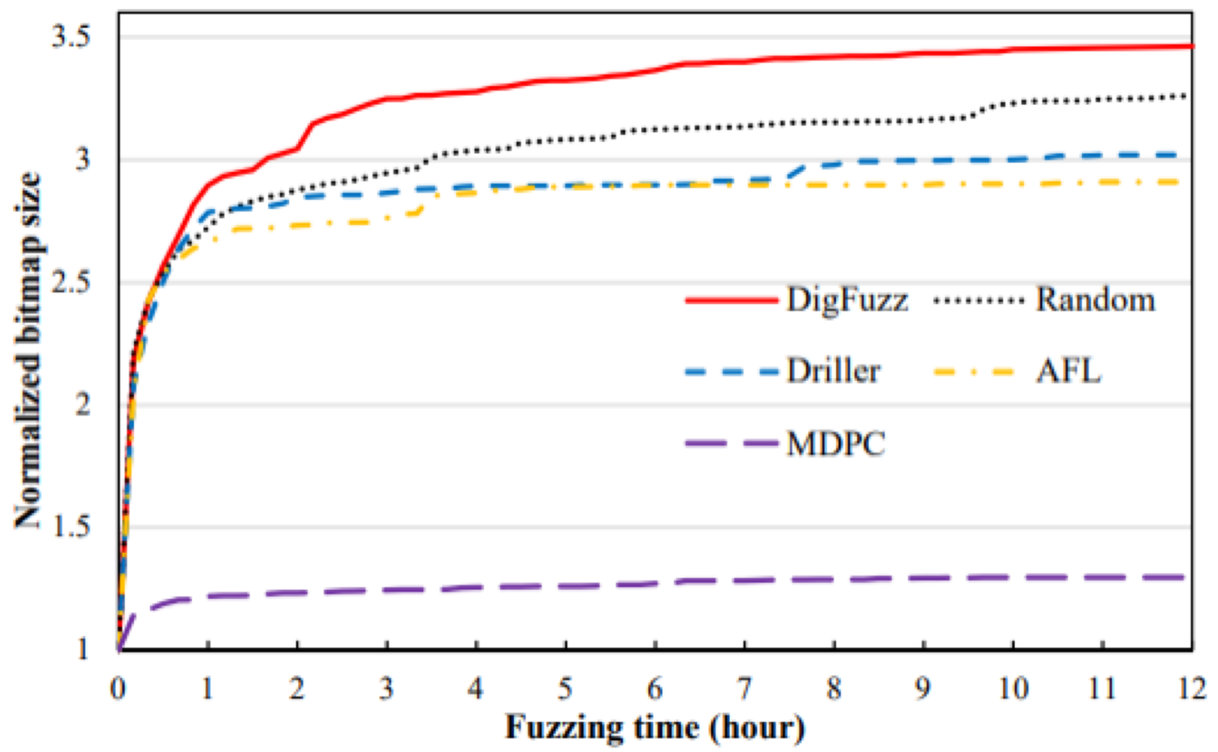
Evaluation

- Dataset
  - CQE challenges (126 binaries)
  - LAVA-M (4 real-world binaries)

- Baseline techniques
  - AFL: pure fuzzing
  - MDPC: Optimal Strategy
  - Driller* : allocate resources evenly other than a shared pool
  - Random: concolic execution launched from the beginning (no path prioritization)

Evaluation on the CQE dataset

- Code coverage
  - DigFuzz, Random, Driller, and AFL are 3.46 times, 3.25 times, 3.02 times and 2.91 times larger than the base (code coverage of the initial inputs)

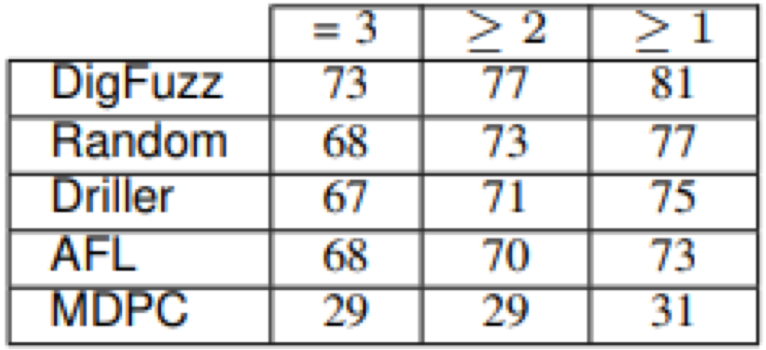


- Concolic execution can indeed help fuzzing
- Random outperforms Driller (demand launch doesn't work well)
- Path prioritization in DigFuzz is effective
- The contribution of concolic execution to bitmap size in DigFuzz is much larger than those in Driller (18.9% vs. 3.8%) and Random (18.9% vs. 11.7%)

Evaluation on the CQE dataset

- Discovered vulnerabilities
  - Tested 12 hours with 3 runs for each binary
  - Our approach steadily discovers more vulnerabilities
  - Per Driller paper report, DigFuzz can achieve similarly with only half of the running time (12 hours vs. 24 hours) and much less hardware resources (2 fuzzing instances per binary vs. 4 fuzzing instances per binary)

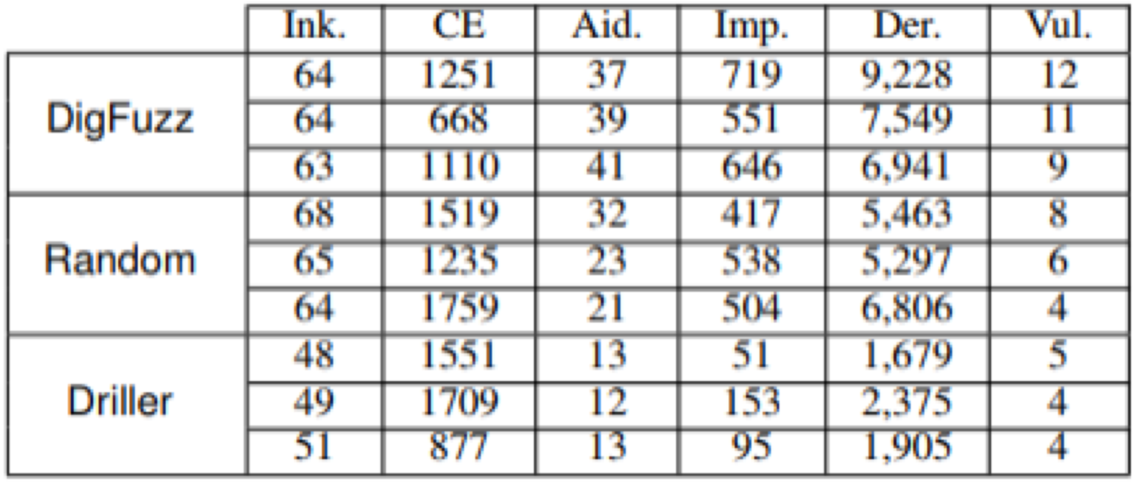
TABLE II: Number of discovered vulnerabilities



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

Evaluation on the CQE dataset

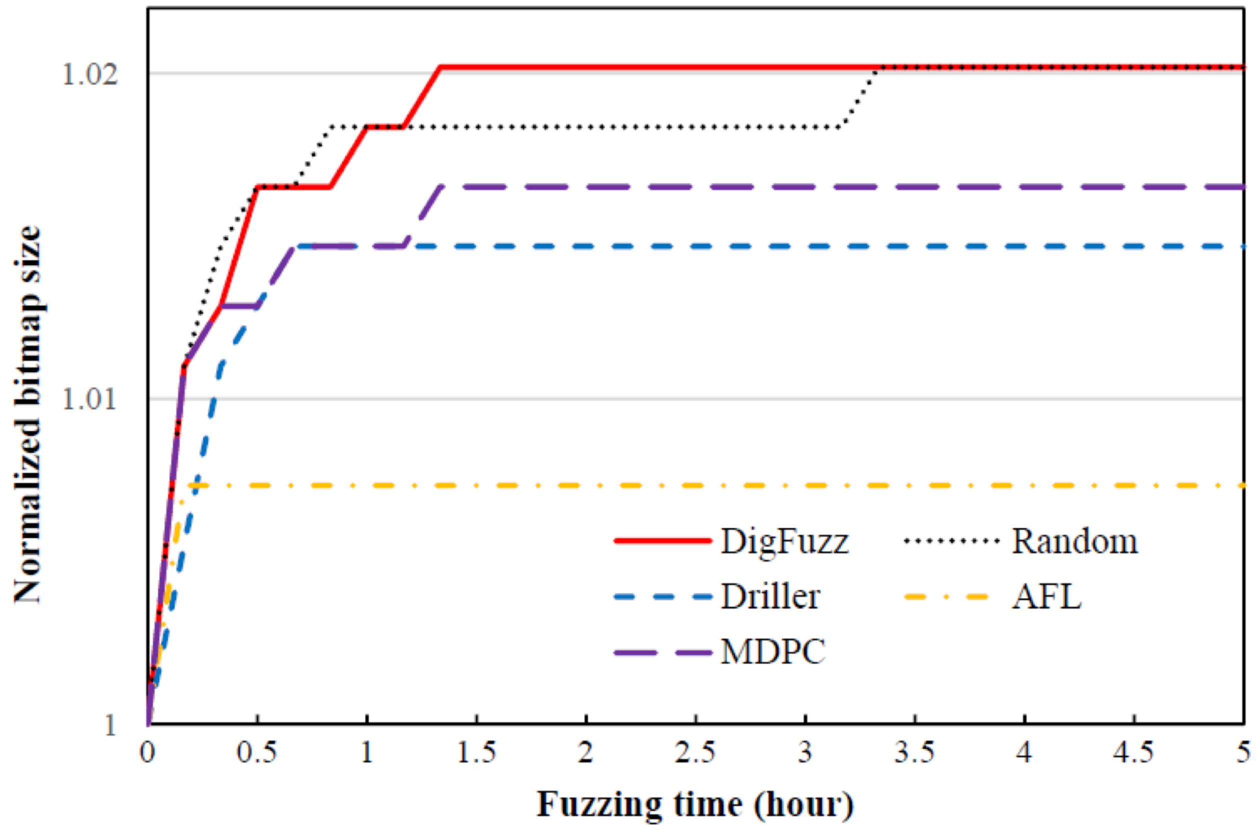
- Contribution of concolic execution
  - More binaries aided by concolic execution (Aid.) ⇒ CE launched in more binaries
  - More imported and derived inputs from concolic execution (Imp. and Der.) ⇒ better quality for generated inputs
  - More crashes are triggered by inputs from concolic execution ⇒ more effective in finding vulnerabilities



	Ink.	CE	Aid.	Imp.	Der.	Vul.
DigFuzz	64	1251	37	719	9,228	12
	64	668	39	551	7,549	11
	63	1110	41	646	6,941	9
Random	68	1519	32	417	5,463	8
	65	1235	23	538	5,297	6
	64	1759	21	504	6,806	4
Driller	48	1551	13	51	1,679	5
	49	1709	12	153	2,375	4
	51	877	13	95	1,905	4

Evaluation on the LAVA dataset

- LAVA-M consists of 4 small applications
  - DigFuzz achieved better code coverage
  - Random caught up because the programs are small



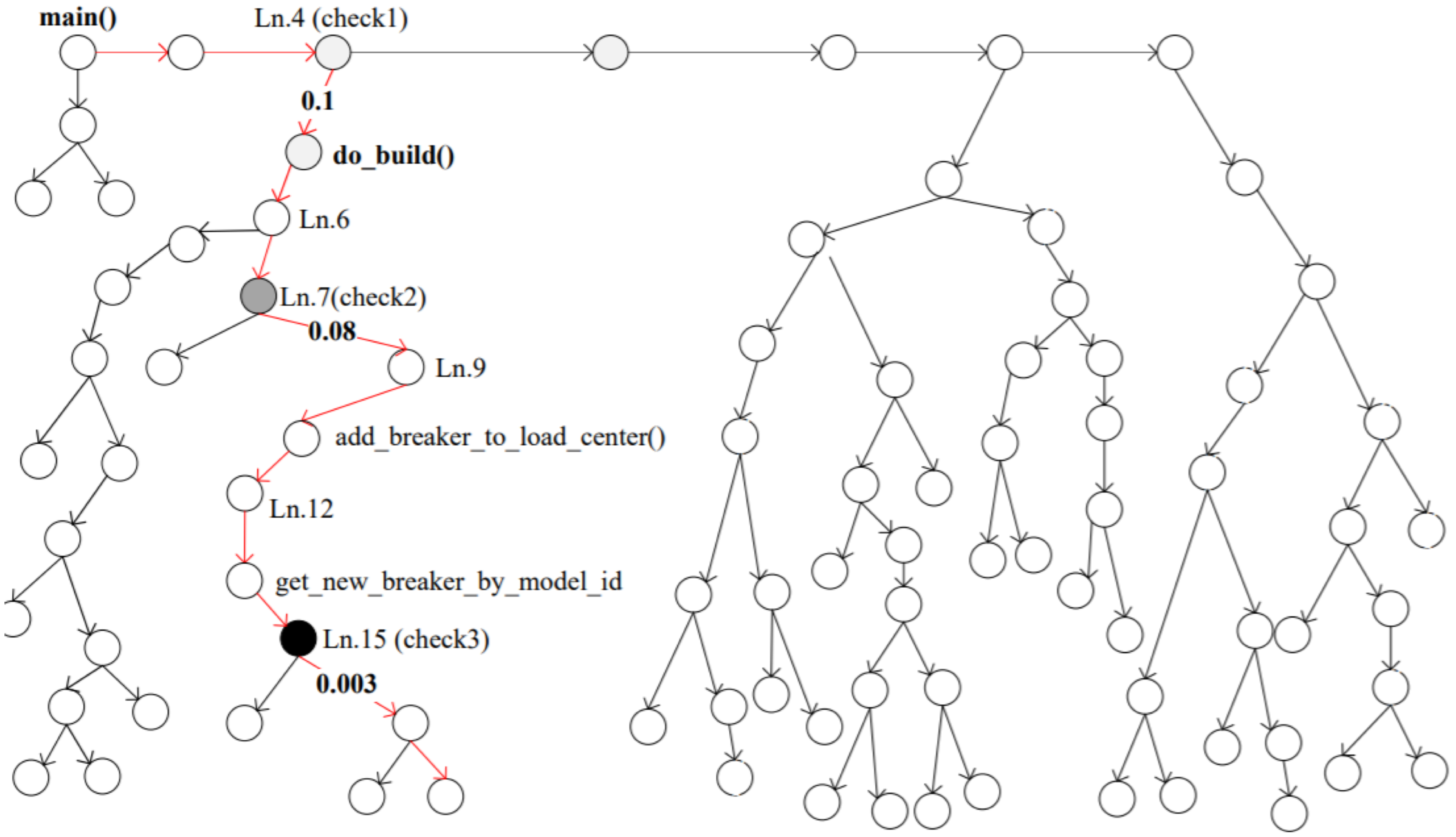
Case Study

- Performance
  - AFL: failed to trigger the vulnerability
  - Driller: took 2590s
  - Random: took 1438s
  - DigFuzz: took 691s

```
int main(void) {
  ...
  RECV(mode, sizeof(uint32_t));
  switch (mode[0]) {
    case MODE_BUILD: ret = do_build(); break;
  ... }

int do_build() {
   ...
   switch(command) {
    case B_CMD_ADD_BREAKER:
    model_id = recv_uint32();
    add_breaker_to_load_center(model_id, &result);
    break;
  ...}}

int8_t add_breaker_to_load_center() {
      get_new_breaker_by_model_id(...);}

int8_t get_new_breaker_by_model_id(...) {
      switch(model_id){
         case FIFTEEN_AMP:
            //vulnerability
            break;
```

Case Study

- Further Investigation on how DigFuzz performs



Discussion

- Evaluation on real-world programs
  - We tried to evaluate our approach on real world programs. However, the concolic execution engine (angr) fails to scale on real programs due to unsupported system calls.

- Only estimates the probability of fuzzing, but does not consider the cost of concolic execution
  - Collecting path constraints and estimating the complexity of constraint solving are challenging

Conclusion

- A thorough investigation
  - Report several fundamental limitations on the “demand launch” and “optimal switch” strategies.
- A “discriminative dispatch” strategy
  - to better utilize the capability of concolic execution
  - design a Monte Carlo based probabilistic path prioritization model to quantify each path’s difficulty.
- A prototype system DigFuzz
  - Evaluation results show that the concolic execution in DigFuzz is more effective with respect to code coverage and vulnerability discovery.

THANK YOU!