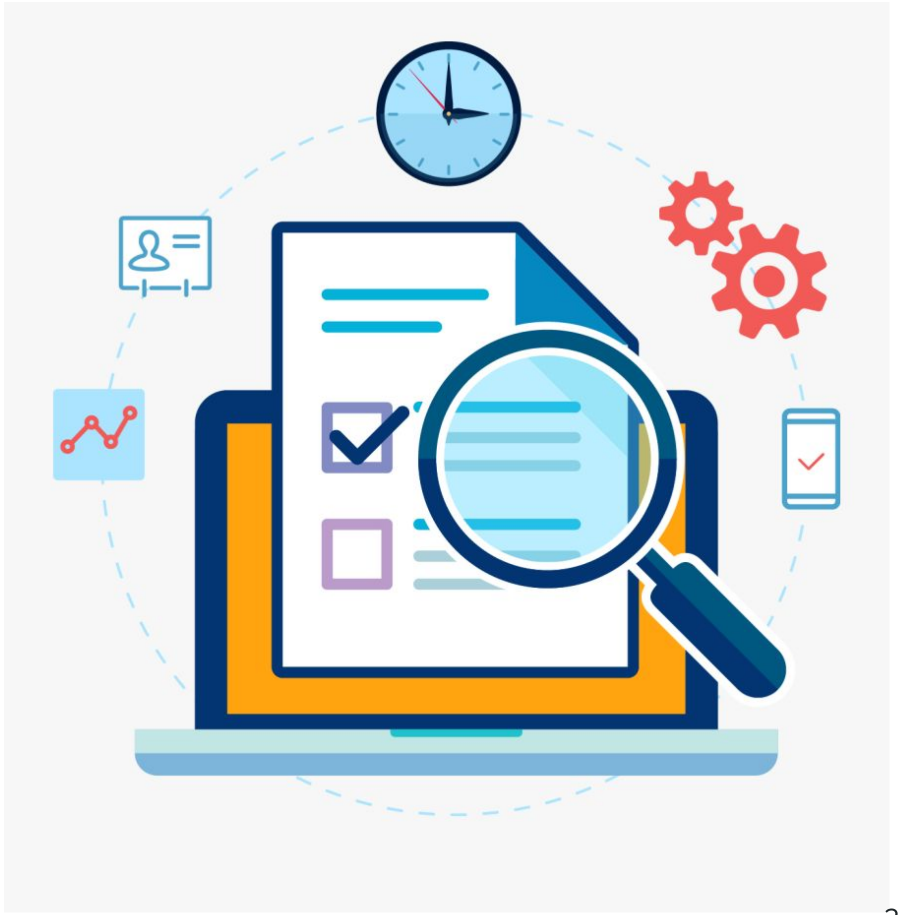
Automated Program Testing

Yue Duan

Program testing

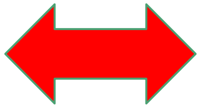
- Programs contain bugs
  - industry average
    - 10-50 bugs per 1K LOC
- Program testing
  - Manual testing
    - Testers manually identify any unexpected behavior or bug
  - Automated testing
    - Use automated techniques to perform the testing



Program testing

Program behaviors during execution

Expected behaviors



Program testing

- Manual testing
    - predefined testing cases
        - Deep domain knowledge required
        - Specific for each individual program
    - Manual checking
        - if the execution matches the expected behavior
    - Limitations
        - Extremely inefficient
        - Poor coverage

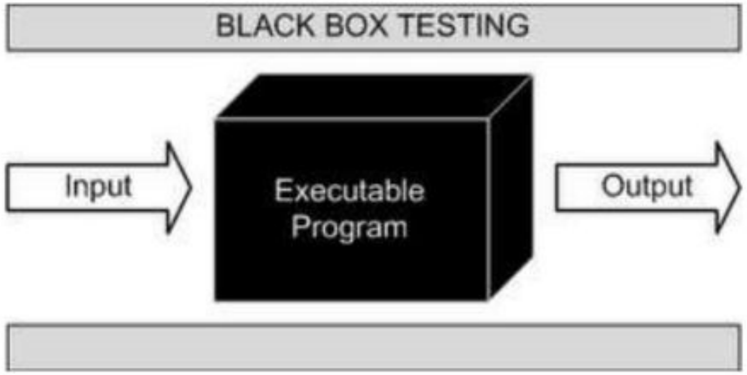
Program testing

- Automated testing
  - Run tested programs automatically
  - Detect unexpected behaviors during execution
  - Produce the discovered bugs easily
  - Three categories
    - Black-box testing
    - Grey-box testing
    - White-box testing

BLACK -BOX TESTING
ZERO KNOWLEDGE
GRAY -BOX TESTING
SOME KNOWLEDGE
WHITE -BOX TESTING
FULL KNOWLEDGE

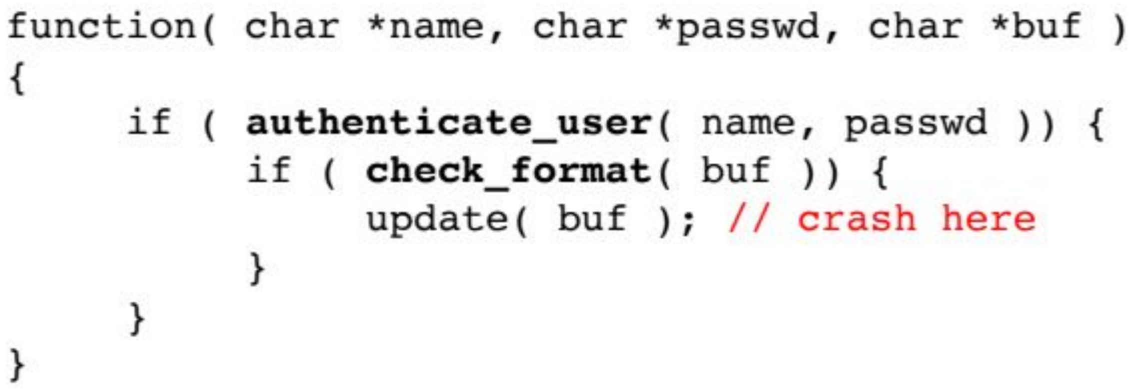
Black-box testing

- View tested programs as Black Box
- Randomly fuzz an existing input
  - Keep mutating existing input to create test data
  - Hope to find test data that triggers bugs
  - ‘Dumb fuzzing’
  - Sometimes still effective



Black-box testing

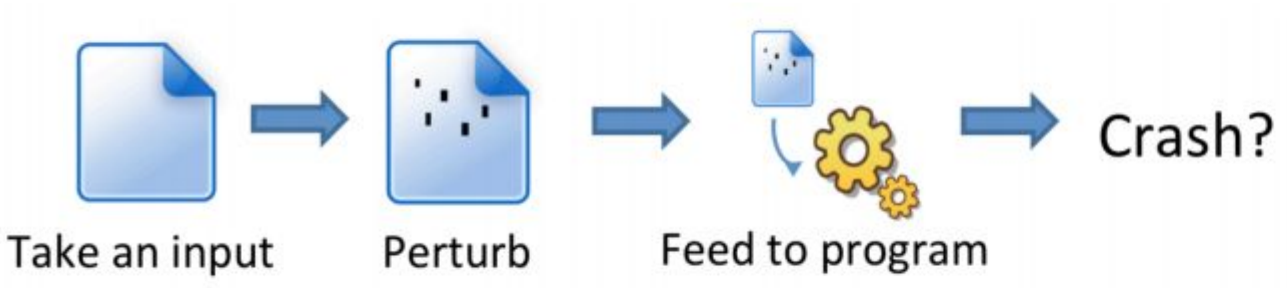
- Can get stuck very easily



```
function( char *name, char *passwd, char *buf )
{
        if ( authenticate_user( name, passwd )) {
                if ( check_format( buf )) {
                        update( buf ); // crash here
                }
        }
}
```

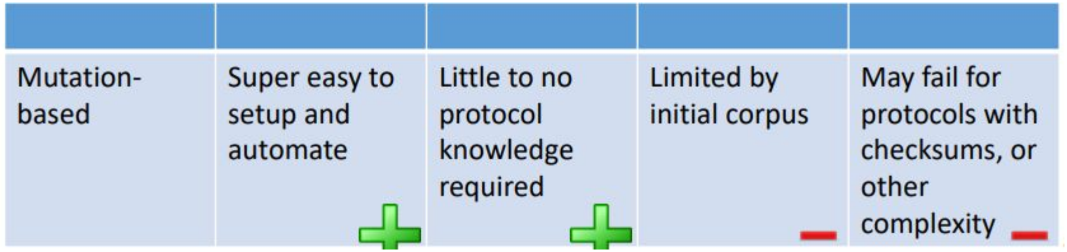
Black-box testing

- Mutation-based fuzzing
  - Idea: take a well-formed input as the initial input
  - Keep mutating the input (flipping random bits, etc)
  - Useful for passing some format checks



Black-box testing

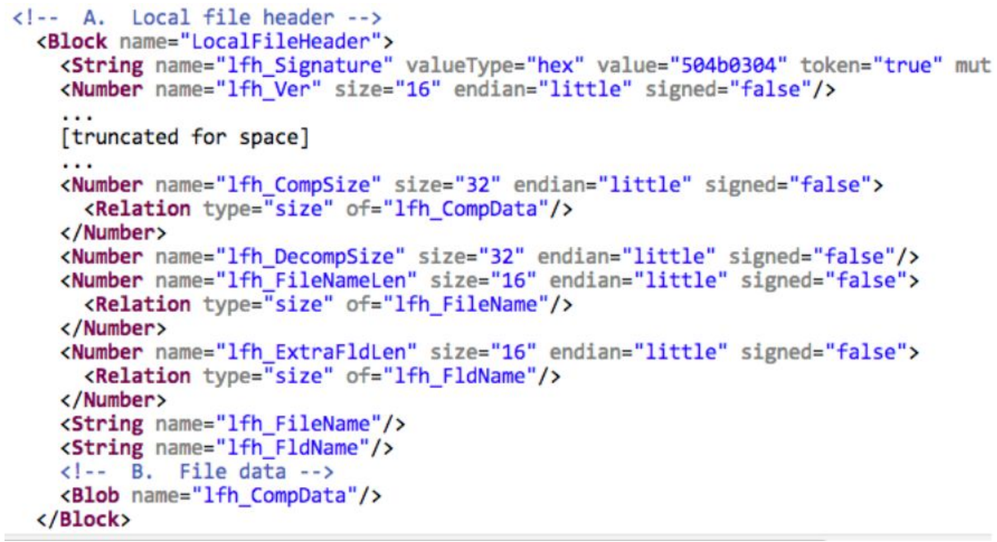
- Mutation-based fuzzing Example: PDF fuzzing
  - Search for PDF files
  - Crawl and download the results
  - Use a mutation-based fuzzer to:
    - Grab a PDF file
    - Mutate the file
    - Send the file to a PDF viewer
    - Record any crashes



Mutation-based	Super easy to setup and automate ➕	Little to no protocol knowledge required ➕	Limited by initial corpus ➖	May fail for protocols with checksums, or other complexity ➖

Black-box testing

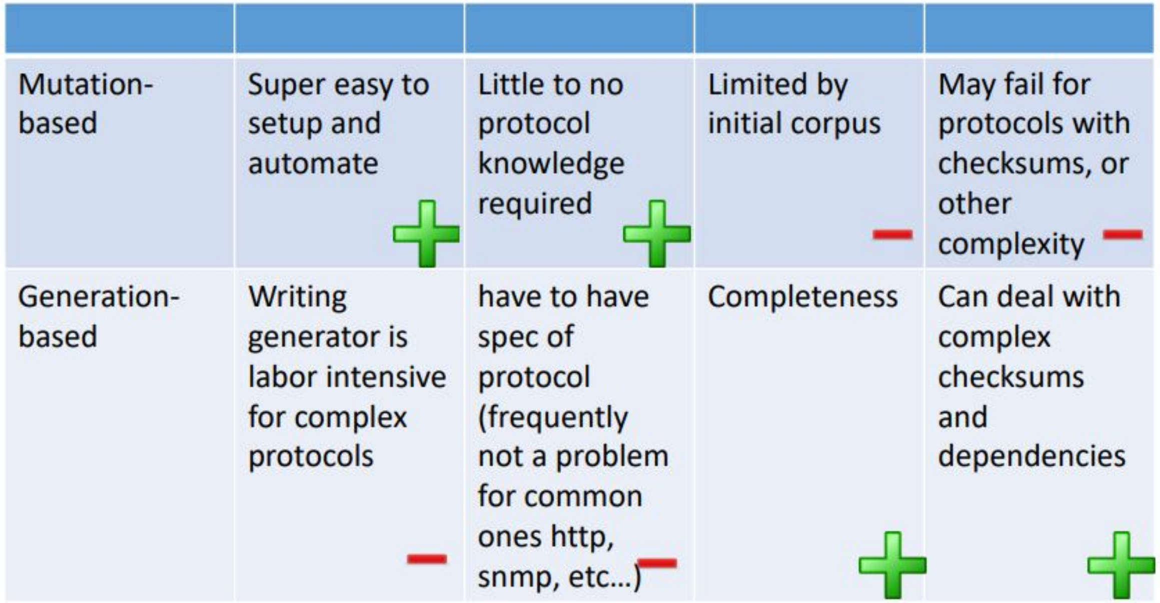
- Generation-based fuzzing
  - Generate test cases from certain well-documented formats (e.g., HTML spec)
  - Can generated well-formed inputs automatically
  - Take significant efforts to set up



```
<!--  A.  Local file header -->
 <Block name="LocalFileHeader">
   <String name="lfh_Signature" valueType="hex" value="504b0304" token="true" mut
   <Number name="lfh_Ver" size="16" endian="little" signed="false"/>
   ...
   [truncated for space]
   ...
   <Number name="lfh_CompSize" size="32" endian="little" signed="false">
     <Relation type="size" of="lfh_CompData"/>
   </Number>
   <Number name="lfh_DecompSize" size="32" endian="little" signed="false"/>
   <Number name="lfh_FileNameLen" size="16" endian="little" signed="false">
     <Relation type="size" of="lfh_FileName"/>
   </Number>
   <Number name="lfh_ExtraFldLen" size="16" endian="little" signed="false">
     <Relation type="size" of="lfh_FldName"/>
   </Number>
   <String name="lfh_FileName"/>
   <String name="lfh_FldName"/>
   <!--  B.  File data -->
   <Blob name="lfh_CompData"/>
 </Block>
```

Black-box testing

- Generation-based fuzzing V.S Mutation-based fuzzing



Mutation-based	Super easy to setup and automate +	Little to no protocol knowledge required +	Limited by initial corpus −	May fail for protocols with checksums, or other complexity −
Generation-based	Writing generator is labor intensive for complex protocols −	have to have spec of protocol (frequently not a problem for common ones http, snmp, etc…) −	Completeness +	Can deal with complex checksums and dependencies +

Grey-box testing

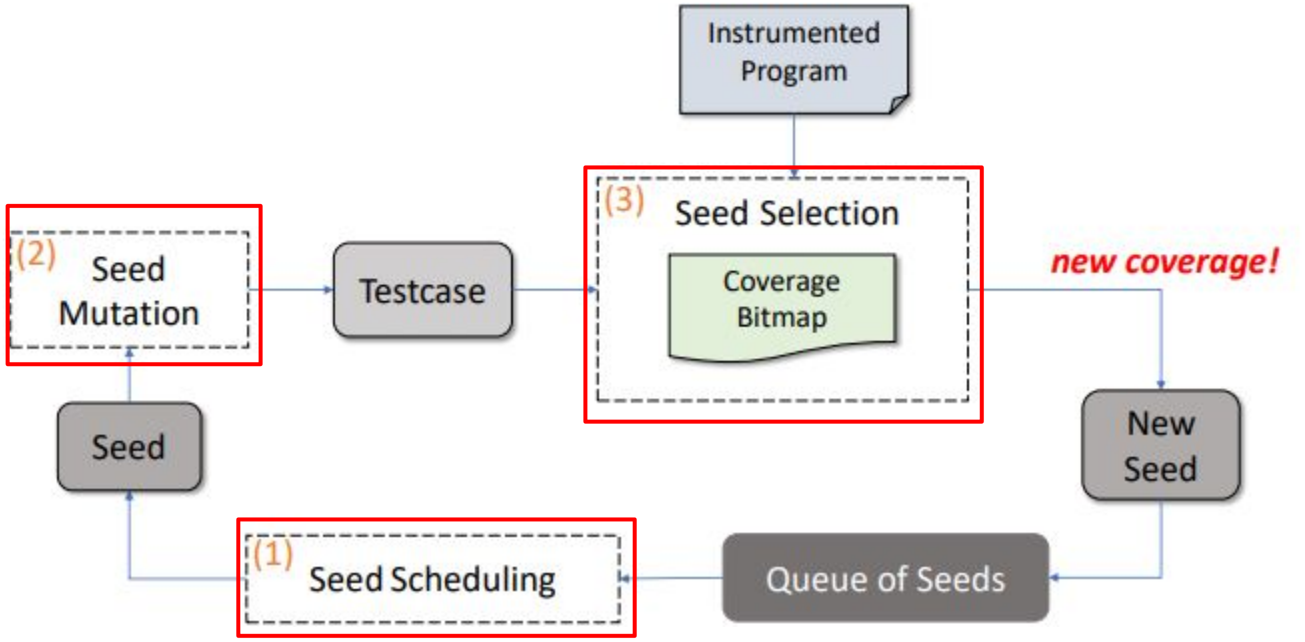
- Some knowledge is acquired during testing
- Generate inputs based on the response of the tested program
- Generated inputs can be preserved only when:
  - Considered as ‘interesting’ by fuzzer
    - How to define?
    - Inputs that can identify something new
  - Can contribute significantly
    - How to define?
- Other inputs will be discarded

Grey-box testing

- Coverage-guided fuzzing
  - ‘Interesting’ standard: new code coverage
    - Statement coverage
    - Branch coverage
    - Path coverage
    - And more
  - Try to maximize code coverage during testing
  - Hopefully bugs can be executed and discovered
  - Limitations?

Grey-box testing

- Example: coverage-guided fuzzing



Grey-box testing

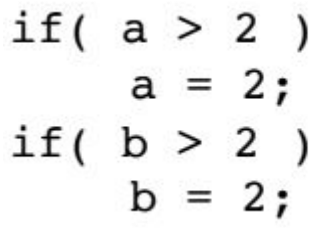
- Coverage-guided fuzzing
  - Seed scheduling
    - Pick the next seed for testing from a set of seed inputs
  - Seed mutation
    - More test cases can be generated based on scheduled seeds through mutation
  - Seed selection
    - Define the ‘interesting’ standard: metrics
    - Preserve only the interesting inputs for next round

Grey-box testing

- Coverage-guided fuzzing
  - Statement coverage
    - Measure how many lines of code have been executed
  - Branch coverage
    - Measure how many branches (conditional jumps) have been executed
  - Path coverage
    - Measure how many paths have been executed

Grey-box testing

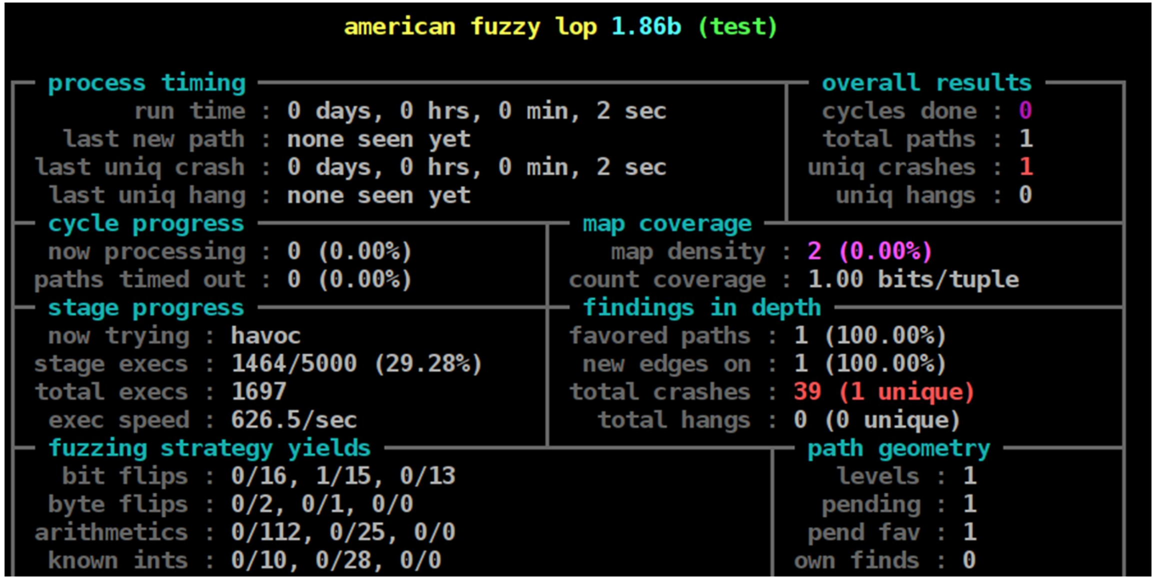
- Exercise
    - Are these inputs ‘interesting’ under the three coverage metrics?
        - Input 1: a = 1, b = 1
        - Input 2: a = 3, b = 1
        - Input 3: a =3, b = 3
        - Input 4: a = 1, b = 3



```
if( a > 2 )
    a = 2;
if( b > 2 )
    b = 2;
```

Grey-box testing

- fuzzing tool: American Fuzzy Lop
  - https://lcamtuf.coredump.cx/afl/
  - Monitor execution during testing

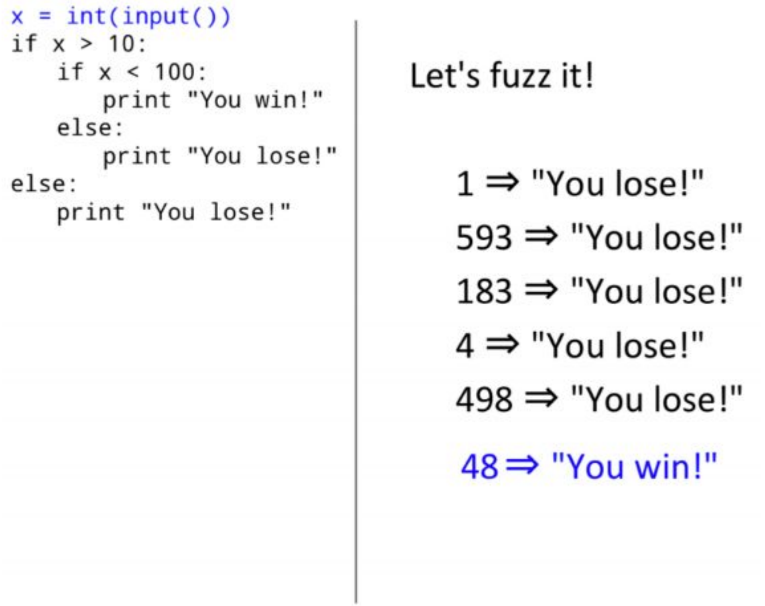


```
                    american fuzzy lop 1.86b (test)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 2 sec        │  cycles done : 0      │
│   last new path : none seen yet                      │  total paths : 1      │
│ last uniq crash : 0 days, 0 hrs, 0 min, 2 sec        │ uniq crashes : 1      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────┬─ map coverage ─────┴───────────────────────┤
│  now processing : 0 (0.00%)     │    map density : 2 (0.00%)                 │
│ paths timed out : 0 (0.00%)     │ count coverage : 1.00 bits/tuple           │
├─ stage progress ────────────────┼─ findings in depth ────────────────────────┤
│  now trying : havoc             │ favored paths : 1 (100.00%)                │
│ stage execs : 1464/5000 (29.28%)│  new edges on : 1 (100.00%)                │
│ total execs : 1697              │ total crashes : 39 (1 unique)              │
│  exec speed : 626.5/sec         │   total hangs : 0 (0 unique)               │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ────────────┤
│   bit flips : 0/16, 1/15, 0/13                  │    levels : 1              │
│  byte flips : 0/2, 0/1, 0/0                     │   pending : 1              │
│ arithmetics : 0/112, 0/25, 0/0                  │  pend fav : 1              │
│  known ints : 0/10, 0/28, 0/0                   │ own finds : 0              │
```

Grey-box testing

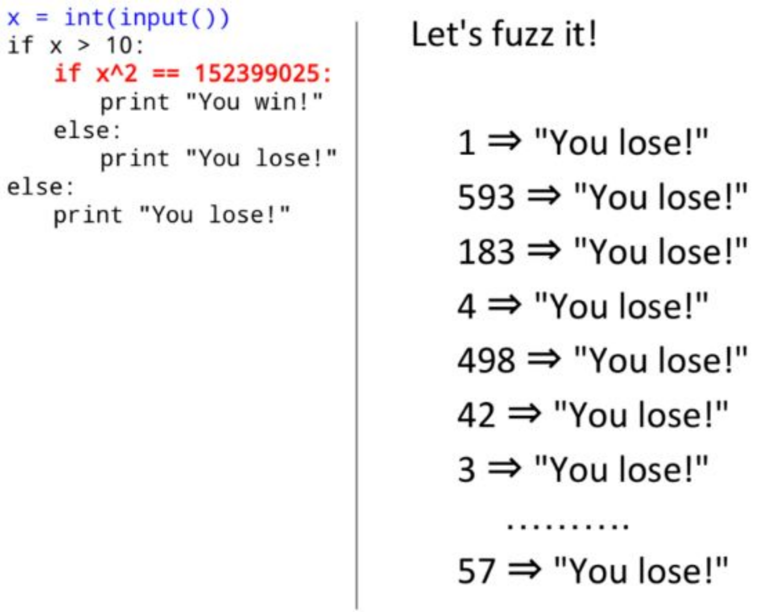
- Limitation



```
x = int(input())
if x > 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Let's fuzz it!

```
1 ⇒ "You lose!"
593 ⇒ "You lose!"
183 ⇒ "You lose!"
4 ⇒ "You lose!"
498 ⇒ "You lose!"
48 ⇒ "You win!"
```



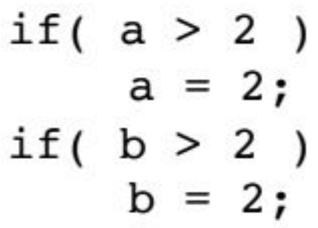
```
x = int(input())
if x > 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Let's fuzz it!

```
1 ⇒ "You lose!"
593 ⇒ "You lose!"
183 ⇒ "You lose!"
4 ⇒ "You lose!"
498 ⇒ "You lose!"
42 ⇒ "You lose!"
3 ⇒ "You lose!"
..........
57 ⇒ "You lose!"
```

White-box testing

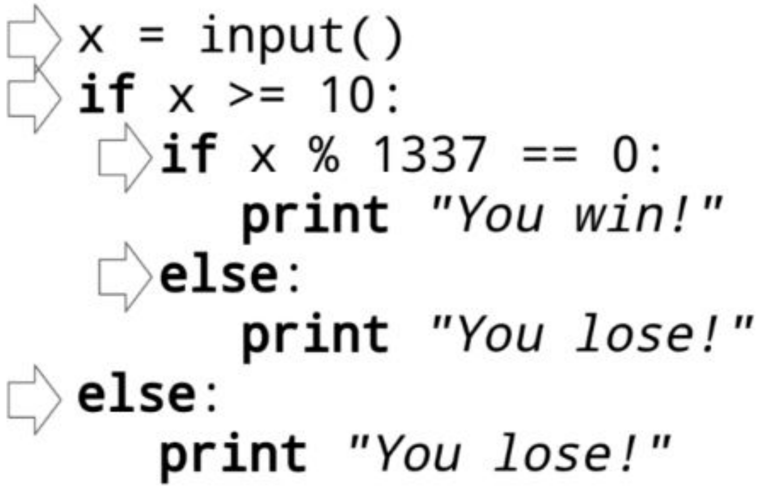
- Full knowledge about tested programs is collected during testing
- Also known as
  - Dynamic symbolic execution
  - or Concolic execution
- Key idea:
  - Evaluate the tested program on *symbolic* input values
    - Symbolic input: input that can take any value
  - Collect path constraints during testing
  - Use an automated theorem prover to generate concrete inputs



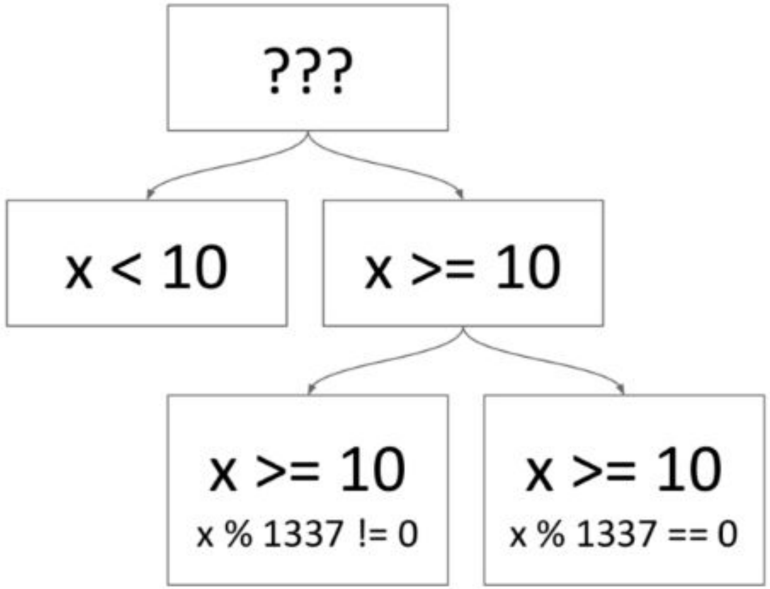
```
if( a > 2 )
    a = 2;
if( b > 2 )
    b = 2;
```

Path constraint:
a > 2 && b > 2
Solved input:
a = 3, b = 3

White-box testing

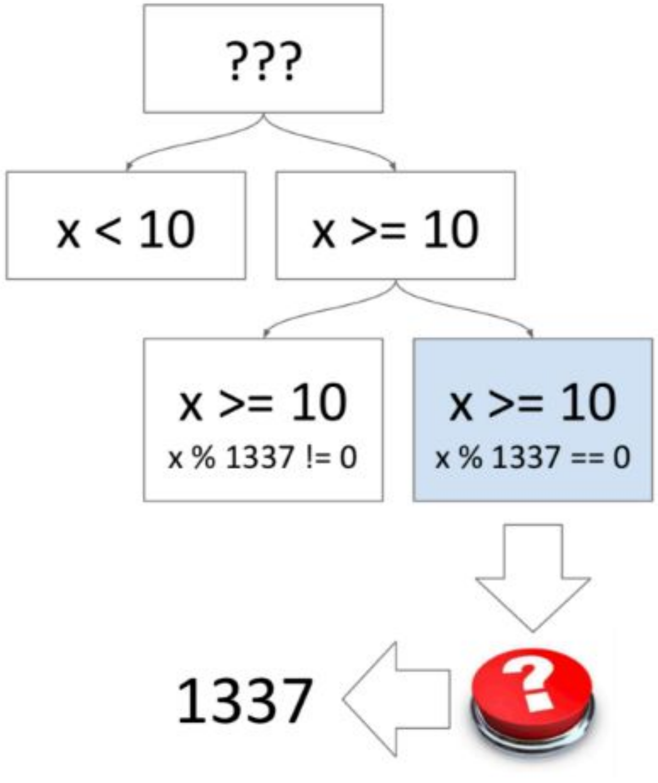


```
x = input()
if x >= 10:
    if x % 1337 == 0:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



White-box testing

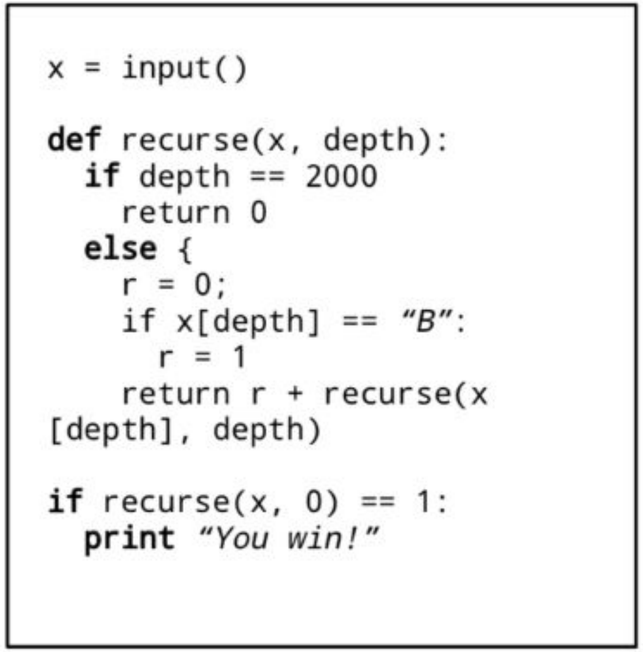
```
x = input()
if x >= 10:
    if x % 1337 == 0:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



White-box testing

- Limitations
  - Low inefficiency
    - Throughput comparison
      - Fuzzing: thousands per second
      - Symbolic execution: 1 per multiple minutes
  - Path explosion
    - Too many paths to explore: exponential
  - Unsolvable path constraints
    - Time-consuming
    - May never get an answer

White-box testing

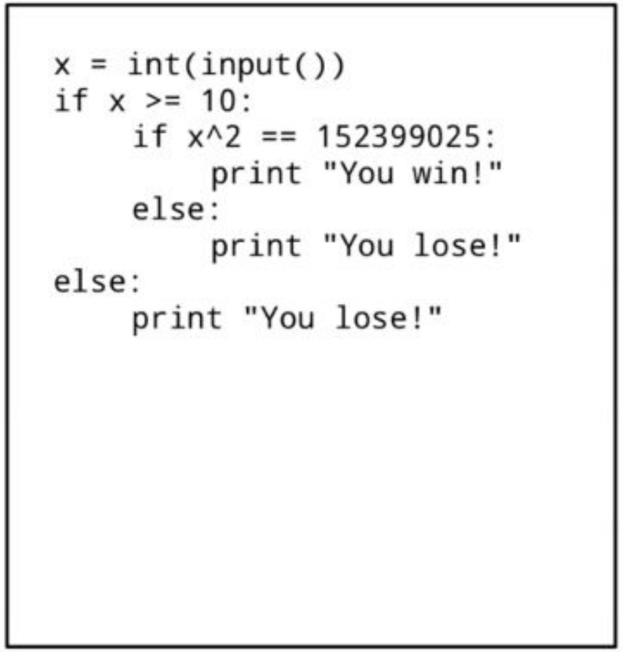


```
x = input()

def recurse(x, depth):
  if depth == 2000
    return 0
  else {
    r = 0;
    if x[depth] == “B”:
      r = 1
    return r + recurse(x
[depth], depth)

if recurse(x, 0) == 1:
  print “You win!”
```

Fuzzing Wins



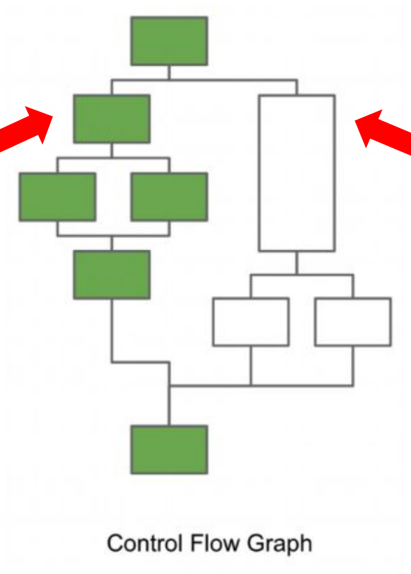
```
x = int(input())
if x >= 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Symbolic Execution Wins

Fuzzing + Symbolic execution?

- Hybrid fuzzing
  - Key idea:
    - Let fuzzer take major responsibility
      - Take advantage of its high throughput
    - Let symbolic executor solve hard problems
      - Utilize its capability of solving specific conditional checks

Fuzzing + Symbolic execution?



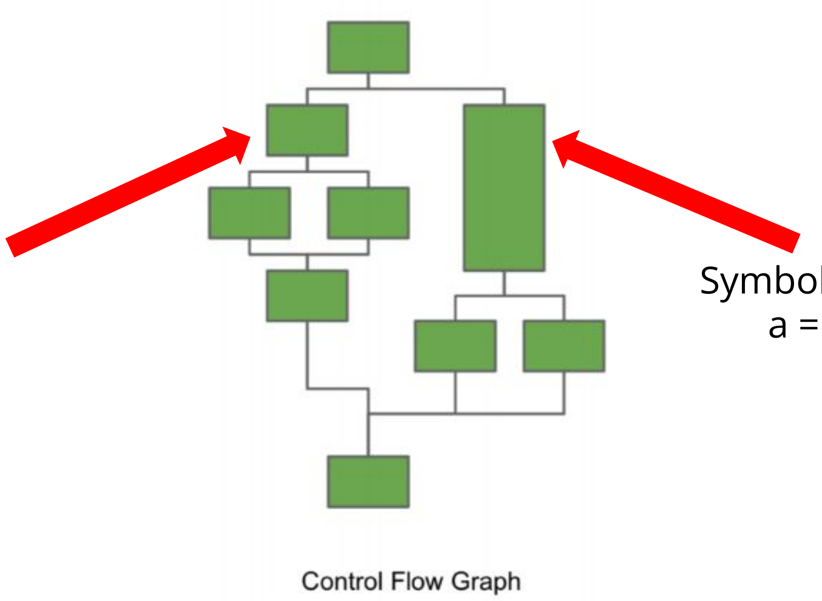
Condition: if (a > 5)

Condition: if (a = 0x43135)

Unreachable by fuzzing

Fuzzing + Symbolic execution?

Fuzzer:
a = 6



Symbolic execution:
a = 0x43135

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

Lei Zhao, Yue Duan, Heng Yin, Jifeng Xuan

NDSS 2019

Motivation

- State-of-the-art hybrid fuzzing approaches
  - **Demand launch**
    - Driller NDSS’16, Hybrid Concolic Testing ICSE’07
    - launch concolic execution when fuzzer gets stuck (blocked by condition checks
    - Assumptions
      - fuzzer in non-stuck state ⇒ concolic execution is not needed
      - stuck state ⇒ fuzzer cannot make progress
      - concolic execution is able to find and solve the hard-to-solve condition problems that block the fuzzer

Motivation

- State-of-the-art hybrid fuzzing approaches
    - **Optimal Strategy**
        - Markov Decision Processes with Costs ICSE’18
        - estimates the costs and always selects the best one
            - cost of fuzzing based on coverage statistics
            - cost of concolic execution based on constraints complexities
        - Assumptions
            - estimation is accurate and fast
            - decision making is lightweight enough

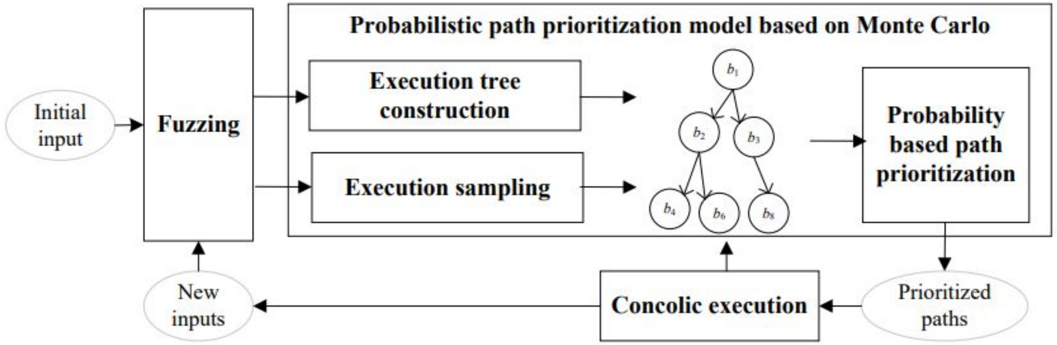
Motivation

- Systematic study is conducted to evaluate the strategies
  - **Demand launch**
    - the stuck state of a fuzzer is not a good indicator
    - not every missed branch requires concolic execution
    - cannot differentiate branches that block fuzzing from others
  - **Optimal strategy**
    - MDPC decision making is heavyweight
    - Throughput is significantly reduced
    - MDPC discovers fewer vulnerabilities

Probabilistic Path Prioritization

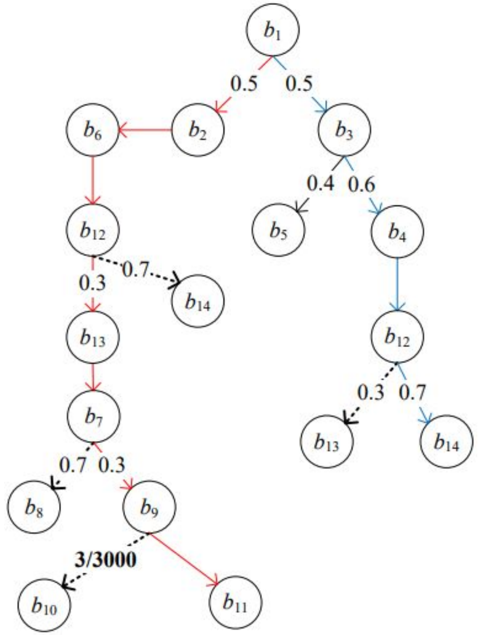
- Aim:
  - let concolic execution only solve the hardest problems
- Challenge:
  - how to **quantify** the difficulty of traversing a path for a fuzzer in a lightweight fashion
- Key idea:
  - Treat fuzzing as a sampling process
  - Estimate branch probabilities based on Monte-Carlo Method
  - Estimate path probabilities as Markov Chain of successive branches

Probabilistic Path Prioritization



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



$P_1 = <b_1, b_2, b_6, b_{12}, b_{13}, b_7, b_9, b_{10}>$, $P(P_1)=0.000045$

$P_2 = <b_1, b_2, b_6, b_{12}, b_{13}, b_7, b_8>$, $P(P_2)=0.063$

$P_3 = <b_1, b_2, b_6, b_{12}, b_{14}>)$, $P(P_3)=0.09$

$P_4 = <b_1, b_3, b_4, b_{12}, b_{13}>$, $P(P_4)=0.105$

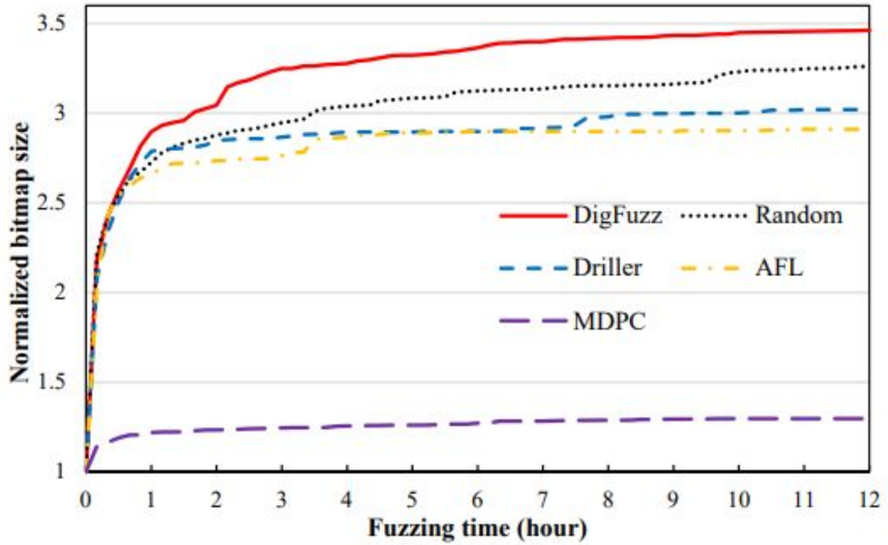
Fig. 5: The execution tree with probabilities

Evaluation

- Dataset
  - Darpa CGC binaries
    - 126 binaries
  - LAVA-M
    - 4 real-world binaries
- Baseline
  - AFL: pure fuzzing
  - MDPC: optimal strategy
  - Driller: demand launch
  - Random: concolic execution launched from the beginning (no path prioritization)

Evaluation

- 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)



Evaluation

- Discovered 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)

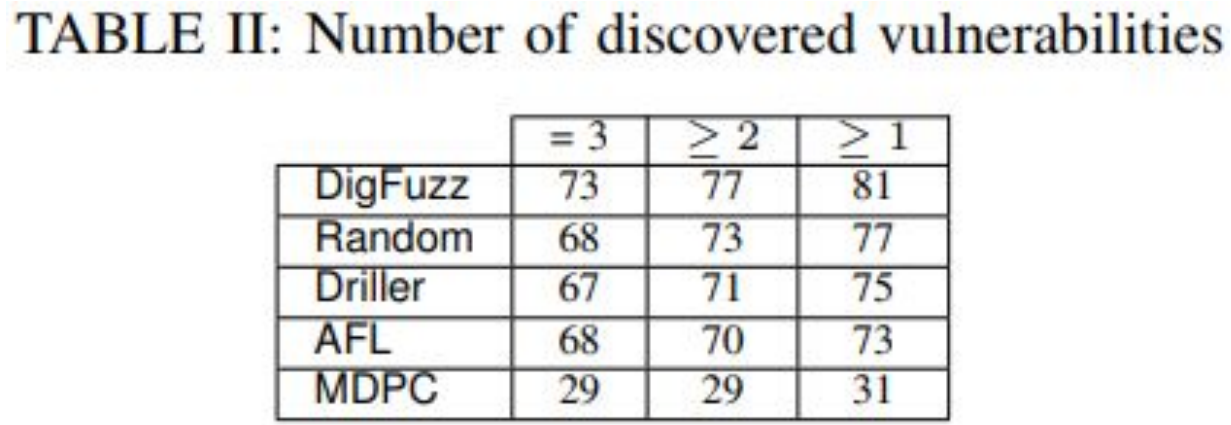
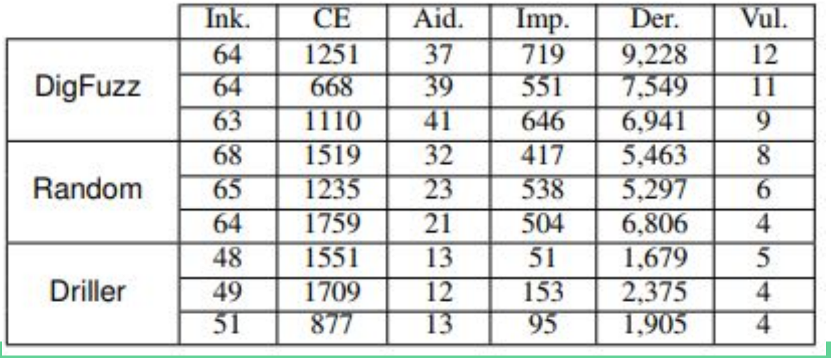


TABLE II: Number of discovered vulnerabilities

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

Evaluation

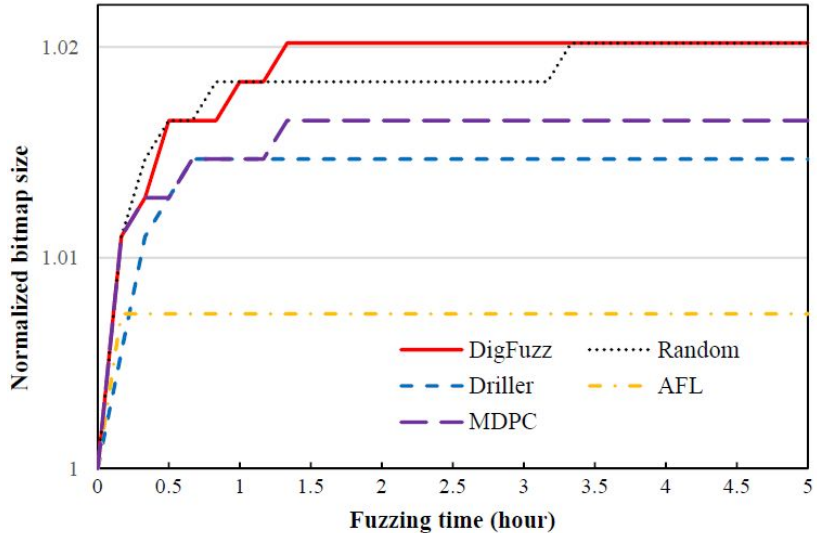
- Contribution of concolic execution
  - More binaries aided by concolic execution (Aid.) ⇒ CE launched in more binaries
  - More imported and derived inputs from concolic execution (Imp. and Der.) ⇒ better quality for generated inputs
  - More crashes are triggered by inputs from concolic execution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

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



Thank you!

Question?