Program Testing
Hybrid Fuzzing

Yue Duan
Outline

- Hybrid fuzzing recap
- Research paper:
  - Driller: Augmenting Fuzzing through Symbolic Execution
  - Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing
Grey-box Fuzzing

```python
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!"
Grey-box Fuzzing

```python
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!"
Symbolic Execution

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

- Grey-box fuzzing
  - fast: throughput > 1000
  - good at exploring general branches
    - (e.g., \(x > 1\)) branches that have large satisfying value spaces
  - bad at exploring specific branches
    - (e.g., \(x \times 2 = 343212\)) branches that have small satisfying value spaces

- Symbolic Execution
  - slow: multiple mins for one execution
    - path explosion
  - can generate concrete inputs that lead to specific paths
Hybrid Fuzzing

Key idea:
- Let fuzzer take major responsibility for exploration
  - Take advantage of its high efficiency
- Let symbolic execution solve hard problems
  - Utilize its capability of solving specific conditions
  - Avoid path explosion
Hybrid Fuzzing

Condition: if (a > 5)

Condition: if ( a == 0x43135)

Unreachable by fuzzing
Hybrid Fuzzing

Fuzzer: a = 6

Symbolic execution: a = 0x43135
Major Challenge

How to distribute the workload?
Fuzzing vs. Symbolic Execution

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!"

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

Fuzzing

good at finding solutions for general input

Symbolic Execution

good at find solutions for specific input
Introduction

- Fuzzing + Symbolic Execution
  - AFL
    - state-of-the-art grey-box fuzzer
    - random mutation
    - basic block coverage tracking
  - Angr
    - symbolic execution engine
    - works on binary code
Combining the Two

Control Flow Graph

Test Cases
Combining the Two

“Cheap” fuzzing coverage

Test Cases

“χ”

“γ”
Combining the Two

“Cheap” fuzzing coverage

Tracing via Symbolic Execution

Reachable?

Test Cases

“X”

“Y”

Control Flow Graph
Combining the Two

“Cheap” fuzzing coverage

Tracing via Symbolic Execution

New test cases generated

Control Flow Graph

Entirely Synthesized!

Test Cases

“X”

“Y”

“MAGIC”
Combining the Two

Towards better code coverage

“Cheap” fuzzing coverage

Tracing via Symbolic Execution

New test cases generated

Test Cases

“X”

“Y”

“MAGIC”

“MAGICY”

Control Flow Graph
Design and Implementation

- ‘interesting’ inputs sharing
- launch SE when getting stuck

- generated inputs given back to fuzzer

Fuzzing

good at finding solutions for general input

Symbolic Execution

good at finding solutions for specific input
Hybrid Testing

Test Cases

```
strncpy(input, "MAGIC")
```

```
input[0] == 'X'
```

...
Hybrid Testing

Test Cases

“X”
Hybrid Testing

Test Cases

"X"

"Y"

\[ \text{strcmp(input, "MAGIC")} \]

\[ \text{input[0] == 'X'} \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]
Hybrid Testing

Test Cases

- "X"
- "Y"
- X

AFL

```
strcmp(input, "MAGIC")

input[0] == 'X'

...  ...

...  ...
```
Hybrid Testing

Test Cases

"X"

"Y"

```
strcmp(input, "MAGIC")
```

```
input[0] == 'X'
```

...
Hybrid Testing

Test Cases

“X”

“Y”

```
strcmp(input, "MAGIC")
```

```
input[0] == 'X'
```

```
...   
```

```
...   
```

```
...   
```

```
...   
```

```
...   
```

```
...   
```

```
...   
```

```
...   
```

```
...   
```

```
(angr)
```
Hybrid Testing

Test Cases

“X”

“Y”

“MAGIC”

```
strcmp(input, "MAGIC")
```

input[0] == 'X'

... ... ...

New state transition, synthesize!

```
Hybrid Testing
Evaluation

- Dataset
  - Darpa CGC binaries 128
- Symbolic Execution
  - crashed 16
- Fuzzing
  - crashed 68
- S & F shared
  - 13
Evaluation

- Driller
  - crashed 77/128 binaries
- Can cover all crashes
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

Fig. 3: Overview of DigFuzz

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**

<table>
<thead>
<tr>
<th></th>
<th>= 3</th>
<th>≥ 2</th>
<th>≥ 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>DigFuzz</td>
<td>73</td>
<td>77</td>
<td>81</td>
</tr>
<tr>
<td>Random</td>
<td>68</td>
<td>73</td>
<td>77</td>
</tr>
<tr>
<td>Driller</td>
<td>67</td>
<td>71</td>
<td>75</td>
</tr>
<tr>
<td>AFL</td>
<td>68</td>
<td>70</td>
<td>73</td>
</tr>
<tr>
<td>MDPC</td>
<td>29</td>
<td>29</td>
<td>31</td>
</tr>
</tbody>
</table>
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
Evaluation

● LAVA-M consists 4 small applications
  ○ DigFuzz achieved better code coverage
  ○ Random caught up because the programs are small
Thank you!

Questions?