Introduction to 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

Specifications
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

- 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

```c
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

<table>
<thead>
<tr>
<th></th>
<th>Super easy to setup and automate</th>
<th>Little to no protocol knowledge required</th>
<th>Limited by initial corpus</th>
<th>May fail for protocols with checksums, or other complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation-based</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
**Black-box testing**

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

<table>
<thead>
<tr>
<th></th>
<th>Mutation-based</th>
<th>Generation-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Super easy to setup and automate</td>
<td>Writing generator is labor intensive for complex protocols</td>
</tr>
<tr>
<td>Knowledge</td>
<td>Little to no protocol knowledge required</td>
<td>have to have spec of protocol (frequently not a problem for common ones http, snmp, etc...)</td>
</tr>
<tr>
<td>Initial Corpus</td>
<td>Limited by initial corpus</td>
<td>Completeness</td>
</tr>
<tr>
<td>Failures</td>
<td>May fail for protocols with checksums, or other complexity</td>
<td>Can deal with complex checksums and dependencies</td>
</tr>
</tbody>
</table>
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

- De facto fuzzing tool: American Fuzzy Lop
  - [https://lcamtuf.coredump.cx/afl/](https://lcamtuf.coredump.cx/afl/)
  - Monitor execution during testing
Grey-box testing

- Limitation

```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!"

```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!"
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

Path constraint: $a > 2 \land b > 2$
Solved input: $a = 3, b = 3$
White-box testing

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

```python
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!"
```

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

Fuzzing Wins

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
Summary

● Program testing  
  ○ Manual testing  
  ○ Automated testing
● Black-box testing  
  ○ Mutation-based  
  ○ Generation-based
● Grey-box testing  
  ○ Coverage-based
● White-box testing  
  ○ Symbolic execution
● Hybrid approach
Thank you!

Question?