Program Testing
Symbolic Execution

Yue Duan
Outline

- Introduction to symbolic execution
- Research paper:
  - KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs
  - Automated Whitebox Fuzz Testing
Introduction to Symbolic Execution

- Symbolic Execution
  - assumes **symbolic values** for inputs rather than obtaining **actual inputs** as normal execution of the program would
  - arrives **expressions** in terms of those symbols for expressions and variables in the program
  - collects **constraints** in terms of those symbols for the possible outcomes of each conditional branch
  - aims to **explore all paths** in the program by generating specific inputs
Introduction to Symbolic Execution

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x) {
        if (x > y+10)
            ERROR;
    }
}
```
Problems

- Problem 1: Path explosion
  - states grow exponentially
  - unbounded loop iterations

```c
void testme_inf () {
    int sum = 0;
    int N = sym_input();
    while (N > 0) {
        sum = sum + N;
        N = sym_input();
    }
}
```
Problems

- Problem 2: Unsolvable formulas
  - non-linear computation is general hard to solve
  - solver could take too long

```c
int twice (int v) {
    return (v*v) % 50;
}
```
Problems

- Problem 3: Environment interactions
  - External function calls and system calls are hard to model
  - For efficiency, symbolic execution system usually model
    - file system related calls
    - string operations

```c
int main()
{
    FILE *fp = fopen("doc.txt");
    ...
    if (condition) {
        fputs("some data", fp);
    } else {
        fputs("some other data", fp);
    }
    ...
    data = fgets(..., fp);
}
```
Concolic Testing

- Symbolic + concrete execution
  - run symbolic execution dynamically
  - execute the program on some concrete input values

- Example
  - generate random input: $x=22$, $y=7$
  - execute the program both concretely and symbolically

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x) {
        if (x > y+10)
            ERROR;
    }
```
Concolic Testing

- Example (Cont.)
  - concrete execution takes the ‘else’ branch on Ln.7
  - symbolic execution generates the path constraint: \( x \neq 2*y \)
  - a negation will make the path constraint: \( x = 2*y \)
  - solve the path constraint and get a new test input: \( x=2, y=1 \)
  - test the program with the new input

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x) {
        if (x > y+10)
            ERROR;
    }
}
```
Concolic Testing

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution
- Concrete state:
  - $x = 22$
  - $y = 7$
- Symbolic state:
  - $x = x_0$
  - $y = y_0$

Path condition

Symbolic Execution
Concolic Testing

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution

- Concrete state:
  - \( x = 22 \)
  - \( y = 7 \)
  - \( z = 14 \)

Symbolic Execution

- Symbolic state:
  - \( x = x_\theta \)
  - \( y = y_\theta \)
  - \( z = 2y_\theta \)

Path condition: 

- Updated edge conditions:
Concolic Testing

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

**Concrete Execution**
- `x = 22`
- `y = 7`
- `z = 14`

**Symbolic Execution**
- `x = x_0`
- `y = y_0`
- `z = 2*y_0`

**Path Condition**
- `2*y_0 != x_0`
Concolic Testing

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution

- concrete state
  - $x = 22$
  - $y = 7$
  - $z = 14$

Symbolic Execution

- symbolic state
  - $x = x_0$
  - $y = y_0$
  - $z = 2*y_0$

Path Condition

- $2*y_0 != x_0$

Solve: $2*y_0 == x_0$

Solution: $x_0 = 2$, $y_0 = 1$
Concolic Testing

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>concrete state</td>
<td>symbolic state</td>
<td>path condition</td>
</tr>
<tr>
<td>x = 2</td>
<td>x = xθ</td>
<td></td>
</tr>
<tr>
<td>y = 1</td>
<td>y = yθ</td>
<td></td>
</tr>
</tbody>
</table>
Concolic Testing

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Concrete state</strong></td>
<td><strong>Symbolic state</strong></td>
</tr>
<tr>
<td>x = 2</td>
<td>x = $x_0$</td>
</tr>
<tr>
<td>y = 1</td>
<td>y = $y_0$</td>
</tr>
<tr>
<td>z = 2</td>
<td>z = 2*$y_0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>path condition</th>
</tr>
</thead>
</table>

Concolic Testing

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
    
    x = 2
    y = 1
    z = 2
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>concrete state</td>
<td>symbolic state</td>
</tr>
<tr>
<td>x = 2</td>
<td>x = x₀</td>
</tr>
<tr>
<td>y = 1</td>
<td>y = y₀</td>
</tr>
<tr>
<td>z = 2</td>
<td>z = 2 * y₀</td>
</tr>
</tbody>
</table>

path condition

\[2 * y₀ \leq x₀\]
Concolic Testing

```cpp
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
    
    y = 1
    z = 2
}
```

Concrete Execution | Symbolic Execution | Path condition
---|---|---
Concrete state | Symbolic state | path condition
x = 2 | x = x₀ | 2*y₀ = x₀
y = 1 | y = y₀ | x₀ <= y₀+10
z = 2 | z = 2*y₀ |

Solve: (2*y₀ = x₀) and (x₀ > y₀+10)

Solution: x₀ = 30, y₀ = 15
Concolic Testing

- Benefits:
  - solve complex formulas
    - $x = (y^2) \mod 50$, unsolvable if $x$ and $y$ are both symbolic
    - if some value is concretized, then it becomes solvable
  - External library call and system call
    - e.g., $fd = \text{open}(\text{filename})$
    - set filename to its concrete value “/tmp/abc.txt”
    - execute the system call concretely
    - set $fd$ to be concrete after the system call return
Online V.S Offline Approaches

● Online
  ○ encounter a new symbolic branch
  ○ solve path constraints for both ‘true’ and ‘false’
  ○ if both feasible, fork the execution states

● Offline
  ○ trace-based approach
  ○ choose an input, execute the program and collect execution trace
  ○ compute path contracts from the trace
  ○ negate each conjunct, solve the new path constraint and generate new inputs
  ○ start again
## Online V.S Offline Approaches

- **Pros and Cons**

<table>
<thead>
<tr>
<th></th>
<th>Online</th>
<th>Offline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficiency</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Implementation difficulty</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Symbolic State</td>
<td>Quickly exploded</td>
<td>No state management</td>
</tr>
</tbody>
</table>
Implementations

- Dynamic Instrumentation
  - source code needed:
    - compile C/C++ into LLVM bytecode
    - add instrumentation during compilation
  - binary:
    - run in QEMU with two machines (concrete and symbolic)
    - convert TCG IR to LLVM bytecode

- Trace-based
  - collect execution trace using tools such as Pintrace and tracecap
  - convert trace into IR
  - perform analysis on IR
Implementations

- Pure Interpretation or Simulation
  - interpret binary execution and add symbolic execution logic
  - Pros:
    - full control
    - easy to implement
  - Cons:
    - low efficiency (all instructions must be interpreted)
KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler

OSDI 2008
Introduction

- Operates on LLVM bytecode
- A **symbolic process** (or **state**) is the state of a symbolically executing process
  - register file, stack, heap, program counter, path condition
  - storage locations (stack, heap, registers) contain **symbolic expressions**
- when symbolic execution counters a branch
  - state is cloned
  - update instruction pointer and path condition accordingly
KLEE Architecture
State Exploration

- Problem:
  - The number of states grow exponentially

- Solution
  - Use compact state representation:
    - Copy-on-write at the object level
    - Heap as an immutable map can be shared among states
    - Heap can be cloned in constant time
Multiple concurrent states, representing different program executions

Aim:
  - good code coverage

Problem:
  - which state to run at each step?

Solution
  - two strategies
    - random path selection
    - coverage-optimized search
Environment Modeling

- Problem:
  - interactions with the environment are complex

- Solution:
  - model semantics and redirects library calls to these models

- Example: symbolic file system
  - single directory with N symbolic files
  - co-exist with real file system
    - when called with concrete file name, will open real file
      - int fd = open("/etc/fstab", O_RDONLY);
    - when called with symbolic file name, will form and match each of the N symbolic files
      - int fd = open(argv[1], O_RDONLY);
Evaluation

- Metrics: Line coverage
- Dataset:
  - Coreutils, Busybox, HiStar
- Results:
  - in much shorter time, better coverage than tests manually developed for over 15 years
Evaluation

- Found 10 unique bugs in Coreutils
- Found 21 bugs in Busybox
- Found 21 bugs in Minix
- All memory errors

Figure 7: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.
Automated Whitebox Fuzz Testing

Patrice Godefroid, Michael Y. Levin, David Molnar

NDSS 2008
Motivation

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 >= 3) crash();
}

input = "good"

I₀ ≠ 'b'
I₁ ≠ 'a'
I₂ ≠ 'd'
I₃ ≠ '!'  

- Traditional trace-based approach
  - Collect constraints from trace
  - create new constraints by negating
  - solve → new input
  - start again with new input
Motivation

good goo!

```c
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 >= 3) crash();
}
```
Motivation

```c
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 >= 3) crash();
}
```
Key Idea: One Trace Many Tests

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 >= 3) crash();
}

“Generation 1” test cases
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 >= 3) crash();
}
SAGE Architecture
Case Study

ANI Parsing - MS07-017

Critical, out-of-band security patch; affected Vista

RIFF...ACONLIST
B...INFOINAM....
3D Blue Alternat
e v1.1..IART....

........................
1996..anih$...$.
........................
..rate........
...........seq ..
.LIST...framic
on...........

Seed file

RIFF...ACONB
B...INFOINAM....
3D Blue Alternat
e v1.1..IART....

........................
1996..anih$...$.
........................
..rate........
...........seq ..
.anih..framic
on...........

SAGE-generated crashing test case

Only 1 in $2^{32}$ chance at random!
Evaluation

- Very large dataset

<table>
<thead>
<tr>
<th>App Tested</th>
<th>#Tests</th>
<th>Mean Depth</th>
<th>Mean #Instr.</th>
<th>Mean Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANI</td>
<td>11468</td>
<td>178</td>
<td>2,066,087</td>
<td>5,400</td>
</tr>
<tr>
<td>Media 1</td>
<td>6890</td>
<td>73</td>
<td>3,409,376</td>
<td>65,536</td>
</tr>
<tr>
<td>Media 2</td>
<td>1045</td>
<td>1100</td>
<td>271,432,489</td>
<td>27,335</td>
</tr>
<tr>
<td>Media 3</td>
<td>2266</td>
<td>608</td>
<td>54,644,652</td>
<td>30,833</td>
</tr>
<tr>
<td>Media 4</td>
<td>909</td>
<td>883</td>
<td>133,685,240</td>
<td>22,209</td>
</tr>
<tr>
<td>Compression</td>
<td>1527</td>
<td>65</td>
<td>480,435</td>
<td>634</td>
</tr>
<tr>
<td>Office 2007</td>
<td>3008</td>
<td>6502</td>
<td>923,731,248</td>
<td>45,064</td>
</tr>
</tbody>
</table>
Evaluation

- Most bugs found are ‘shallow’
Thank you!

Questions?