Program Analysis – Lecture 8
Symbolic and Concolic Execution (Part 2)

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2019/2020
Warm-up Quiz

What does the following code print?

```javascript
var x = 23;
function f() { console.log(this.x); }
var obj = Object.create({ f: f });
obj.x = 42;
f();
obj.f();
```

23 42 23 42
23 42 42 23

https://ilias3.uni-stuttgart.de/vote/KN2I
Warm-up Quiz

What does the following code print?

```javascript
var x = 23;
function f() { console.log(this.x); }
var obj = Object.create({ f: f });
obj.x = 42;
f();
obj.f();
```

obj’s prototype has method `f`

23  42  23  42
23  42  23  42
Warm-up Quiz

What does the following code print?

```javascript
var x = 23;
function f() { console.log(this.x); }
var obj = Object.create({ f: f });
obj.x = 42;
f();
obj.f();
```

dthis is determined at call site:

- Simple call: `global`
- Object method: `base object`

23 42 23 42
23 42 42 23

https://ilias3.uni-stuttgart.de/vote/KN2I
Outline

1. Classical **Symbolic Execution**
2. **Challenges** of Symbolic Execution
3. **Concolic Testing**
4. Large-Scale Application in **Practice**

Mostly based on these papers:

- **DART**: directed automated random testing, Godefroid et al., PLDI’05
- **KLEE**: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, Cadar et al., OSDI’08
- **Automated Whitebox Fuzz Testing**, Godefroid et al., NDSS’08
Concolic Testing

Mix *concrete* and *symbolic* execution = "concolic"

- Perform concrete and symbolic execution side-by-side
- Gather path constraints while program executes
- After one execution, negate one decision, and re-execute with new input that triggers another path
function double(n) {
    return 2 * n;
}

function testMe(x, y) {
    var z = double(y);
    if (z === x) {
        if (x > y + 10) {
            throw "Error";
        }
    }
}

- Symbolic execution

\[
\begin{align*}
    x &= x_0 \\
    y &= y_0 \\
    z &= 2 \cdot y_0 \\
    2 \cdot y_0 &= x_0
\end{align*}
\]

- Decision tree

- "Error"
Execution 1:

```
function double(n) {
    return 2 * n;
}

function testMe(x, y) {
    var z = double(y);
    if (z === x) {
        if (x > y + 10) {
            throw "Error";
        }
    }
}
```

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

Symbolic execution:
- \( x = x_0, \ y = y_0 \)
- \( z = 2 \cdot y_0 \)

Path conditions:
- \( 2 \cdot y_0 \neq x_0 \)

Solve: \( 2 \cdot y_0 = x_0 \)

Solution: \( x_0 = 2, \ y_0 = 7 \)
Execution 2:

```
function double(n) {
    return 2 * n;
}

function testMe(x, y) {
    var z = double(y);
    if (z === x) {
        if (x > y + 10) {
            throw "Error";
        }
    }
}
```

Concrete execution:
- \( x = 2, \ y = 1 \)
- \( x = x_0, \ y = y_0 \)
- \( z = 2 \)

Symbolic execution:
- \( x = x_0, \ y = y_0 \)
- \( z = 2 \cdot y_0 \)

Path conditions:
- \( 2 \cdot y_0 = x_0 \)
- \( x_0 \geq y_0 + 10 \)

Solve: \( 2 \cdot y_0 = x_0 \wedge x_0 > y_0 + 10 \)

Solution: \( x_0 = 30, \ y_0 = 15 \)

Hits "Error"
Exploring the execution tree
Algorithm

Repeat until all paths are covered

- Execute program with concrete input $i$ and collect symbolic constraints at branch points: $C$
- Negate one constraint to force taking an alternative branch $b'$: Constraints $C'$
- Call constraint solver to find solution for $C'$: New concrete input $i'$
- Execute with $i'$ to take branch $b'$
- Check at runtime that $b'$ is indeed taken

Otherwise: “divergent execution”
Divergent execution: Example

```javascript
function f(a) {
    if (Math.random() < 0.5) {
        if (a > 1) {
            console.log("took it");
        }
    }
}
```

First execution

- \(a = 0\)
- Branch taken
- Branch not taken
- Path constraint: \(a_0 \leq 1\)
- Negate & solve:
  - \(a_0 = 2\)

Second execution

- \(a = 2\)
- Branch not taken
- \(\rightarrow\) Divergent execution
After how many executions and how many queries to the solver does concolic testing find the error?

Initial input: $a=0, b=0$

```javascript
function concolicQuiz(a, b) {
    if (a === 5) {
        var x = b - 1;
        if (x > 0) {
            console.log("Error");
        }
    }
}
```

https://ilias3.uni-stuttgart.de/vote/KN2I
Benefits of Concolic Approach

When symbolic reasoning is impossible or impractical, **fall back to concrete values**

- Native/system/API functions
- Operations not handled by solver (e.g., floating point operations)
Outline

1. Classical Symbolic Execution
2. Challenges of Symbolic Execution
3. Concolic Testing
4. Large-Scale Application in Practice

Mostly based on these papers:

- DART: directed automated random testing, Godefroid et al., PLDI’05
- KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, Cadar et al., OSDI’08
- Automated Whitebox Fuzz Testing, Godefroid et al., NDSS’08
Large-Scale Concolic Testing

- **SAGE**: Concolic testing tool developed at Microsoft Research
- Test robustness against unexpected inputs read from files, e.g.,
  - Audio files read by media player
  - Office documents read by MS Office
- Start with known input files and handle bytes read from files as symbolic input
- Use concolic execution to compute variants of these files
■ Applied to hundreds of applications

■ Over 400 machine years of computation from 2007 to 2012

■ Found hundreds of bugs, including many security vulnerabilities
  □ One third of all the bugs discovered by file fuzzing during the development of Microsoft’s Windows 7

Details: Bounimova et al., ICSE 2013
Summary: Symbolic & Concolic Testing

Solver-supported, whitebox testing

- Reason symbolically about (parts of) inputs
- Create new inputs that cover not yet explored paths
- More systematic but also more expensive than random and fuzz testing
- Open challenges
  - Effective exploration of huge search space
  - Other applications of constraint-based program analysis, e.g., debugging and automated program repair