Practical Program Analysis:
Symbolic and Concolic Testing

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2019/2020
Plan for Today

1. Classical **Symbolic Execution**
2. **Concolic Testing**
3. **Course Project**

Papers with more details:

- **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
Symbolic Execution

- Reason about behavior of program by "executing" it with symbolic values
- Originally proposed by James King (1976, CACM) and Lori Clarke (1976, IEEE TSE)
- Became practical around 2005 because of advances in constraint solving (SMT solvers)
Example

```javascript
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
        x = -2;
    }
    if (b > 5) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
        x = -2;
    }
    if (b < 5) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}

---

**Concrete execution**

a = b = c = 1
x = y = z = 0
true
x = -2
true
false
z = 2
✓
Execution Tree

All possible execution paths

- Binary tree
- Nodes: Conditional statements
- Edges: Execution of sequence on non-conditional statements
- Each path in the tree represents an equivalence class of inputs
Quiz

Draw the execution tree for this function. How many nodes and edges does it have?

```javascript
function f(x, y) {
  var s = "foo";
  if (x < y) {
    s += "bar";
    console.log(s);
  }
  if (y === 23) {
    console.log(s);
  }
}
```
function f(x, y) {
    var s = "foo";
    if (x < y) {
        s += "bar";
        console.log(s);
    }
    if (y === 23) {
        console.log(s);
    }
}
Symbolic Values and Symbolic State

- Unknown values, e.g., user inputs, are kept symbolically.
- Symbolic state maps variables to symbolic values.

```javascript
function f(x, y) {
    var z = x + y;
    if (z > 0) {
        ...
    }
}
```
Symbolic Values and Symbolic State

- Unknown values, e.g., user inputs, are kept symbolically
- Symbolic state maps variables to symbolic values

```javascript
function f(x, y) {
  var z = x + y;
  if (z > 0) {
    ...
  }
}
```

Symbolic input
values: \(x_0, y_0\)

Symbolic state:
\[z = x_0 + y_0\]
Path Conditions

Quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far

function f(x, y) {
    var z = x + y;
    if (z > 0) {
        ...
    }
}
Path Conditions

Quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far

```javascript
function f(x, y) {
    var z = x + y;
    if (z > 0) {
        ...
    }
}
```

Path condition:

\[ x_0 + y_0 > 0 \]
Satisfiability of Formulas

Determine whether a path is feasible:
Check if its path condition is satisfiable

■ Done by powerful SMT/SAT solvers
  □ SAT = satisfiability,
  SMT = satisfiability modulo theory
  □ E.g., Z3, Yices, STP

■ For a satisfiable formula, solvers also provide a concrete solution

■ Examples:
  □ \( a_0 + b_0 > 1 \): Satisfiable, e.g., solution: \( a_0 = 1, b_0 = 1 \)
  □ \( (a_0 + b_0 < 0) \land (a_0 - 1 > 5) \land (b_0 > 0) \): Unsatisfiable
Applications of Symbolic Execution

- **General goal:** Reason about behavior of program

- **Basic applications**
  - Detect infeasible paths
  - Generate test inputs
  - Find bugs and vulnerabilities

- **Advanced applications**
  - Generating program invariants
  - Prove that two pieces of code are equivalent
  - Debugging
  - Automated program repair
Problems of Symbolic Execution

- **Loops and recursion**: Infinite execution trees
- **Path explosion**: Number of paths is exponential in the number of conditionals
- **Environment modeling**: Dealing with native/system/library calls
- **Solver limitations**: Dealing with complex path conditions
- **Heap modeling**: Symbolic representation of data structures and pointers
Plan for Today

1. Classical Symbolic Execution

2. Concolic Testing

3. Course Project

Papers with more details:

- *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
Symbolic execution (again)

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

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

\[
x = x_0 \\
y = y_0 \\
z = 2 \cdot y_0 \\
2 \cdot y_0 = x_0 \\
x_0 > y_0 + 10 \\
\]

"Error"
Concolic execution

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

Symbolic execution

Path conditions

```
x = 22, y = 7
x = x₀, y = y₀
z = 14
x = 22, y = 7, x = x₀, y = y₀, z = 2 * y₀
```

Solve: \(2 \cdot y₀ = x₀\)

Solution: \(x₀ = 2, y₀ = 7\)
Concolic execution

Execution 2:

Concrete execution

<table>
<thead>
<tr>
<th>Concrete execution</th>
<th>Symbolic execution</th>
<th>Path conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 2, \ y = 1 )</td>
<td>( x = x_0, \ y = y_0 )</td>
<td>( x = 2, \ y = 1, \ x_0 = x_0, \ y_0 = y_0, \ z = 2, \ y_0 = x_0, 2 \cdot y_0 = x_0, x_0 \leq y_0 + 10 )</td>
</tr>
<tr>
<td>( z = 2 )</td>
<td>( z = 2 \cdot y_0 )</td>
<td>( z = 2 \cdot y_0 )</td>
</tr>
</tbody>
</table>

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

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

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

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

Will hit "Error"
Exploring the execution tree

1
2
3

negate

negate

negate
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"
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");
        }
    }
}
```
Quit

\[ a = a_0 \]
\[ b = b_0 \]
\[ a_0 = 5 \]
\[ \kappa = b_0 - 1 \]
\[ b_0 - 1 > 0 \]
\[ \text{Error} \]

**Exec. 1**

\[ a_0 = 0, \ b_0 = 0 \]

Solve: \( a_0 = 5 \) \( \rightarrow a_0 = 5 \)

**Exec. 2**

\[ a_0 = 5, \ b_0 = 0 \]

Solve: \( a_0 = 5 \land b_0 - 1 > 0 \)

\( \rightarrow a_0 = 5, \ b_0 = 2 \)

**Exec. 3**

\( \rightarrow \text{reach error} \)

\( \Rightarrow 3 \text{ executions, 2 queries} \)
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)
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
Large-Scale Concolic Testing (2)

- 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 Windows 7

Details: Bounimova et al., ICSE 2013
Plan for Today

1. Classical **Symbolic Execution**
2. **Concolic Testing**
3. **Course Project**

Papers with more details:

- **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