Program Analysis

Symbolic and Concolic Execution

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2021/2022
What does the following code print?

```javascript
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
    sum += x;
}
console.log(sum);
```

112233  0012  66  Something else
Warm-up Quiz

What does the following code print?

```javascript
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
    sum += x;
}
console.log(sum);
```

- 112233
- 0012
- 66
- Something else

Some JS engines
What does the following code print?

```javascript
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
  sum += x;
}
console.log(sum);
```

Arrays are objects

For-in iterates over object property names (not property values)

Some JS engines

112233 0012 66 Something else
Warm-up Quiz

What does the following code print?

```javascript
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
    sum += x;
}
console.log(sum);  
```

For arrays, use traditional for loop:

```javascript
for (var i=0; i < array.length; i++) ...
```

Some JS engines

112233  0012  66

Something else
Overview

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
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)
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 execute

\[ a = b = c = 1 \]
\[ x = y = z = 0 \]
true
\[ x = -2 \]
false
\[ -2 + 0 + 0 \neq 3 \] √
Symbolic execution

\[ a = a_0, \quad b = b_0, \quad c = c_0 \]
\[ x = y = z = 0 \]

\[ x = -2 \]
\[ z = 2 \]
\[ y = 1 \]

\[ a_0 \land b_0 > 5 \]
\[ a_0 \land b_0 \leq 5 \]
\[ a_0 \land \neg (a_0 \land c_0) \]

\[ \neg a_0 \land \neg c_0 \]

\[ \text{infeasible} \]

\[ 0 + 1 + 2 = 3 \]
\[ L \text{ assertion violated} \]
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);
  }
}
```
Quiz

\[ x = x_0, \ y = y_0 \]
\[ s = "foo" \]

\[ x_0 < y_0 \]
\[ s += "bar" \]
\[ \text{console.log(s)} \]

\[ y_0 = 23 \]
\[ \text{console.log(s)} \]

\[ y_0 = 23 \]
\[ \text{console.log(s)} \]

\[ \Rightarrow 3 \text{ nodes} \]
\[ 7 \text{ edges} \]
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

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, one 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
Overview

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
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
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
Path explosion

function f(a) {
    var x = a;
    while (x > 0) {
        x = x - 1;
    }
    ...
}
Dealing with Large Execution Trees

**Heuristically select which branch to explore next**

- Select at random
- Select based on **coverage**
- Prioritize based on distance to "interesting" program locations
- Interleaving symbolic execution with **random testing**
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
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
Modeling the Environment

- Program behavior may depend on parts of system not analyzed by symbolic execution
- E.g., native APIs, interaction with network, file system accesses

```javascript
var fs = require("fs");
var content = fs.readFileSync("/tmp/foo.txt");
if (content === "bar") {
  ...
}
```
Solution implemented by **KLEE**

- If all arguments are concrete, forward to OS
- Otherwise, provide **models that can handle symbolic files**
  - Goal: Explore all possible legal interactions with the environment

```javascript
var fs = {
  readFileSync: function(file) {
    // doesn’t read actual file system, but
    // models its effects for symbolic file names
  }
}
```
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
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

One approach: Mix symbolic with concrete execution
Overview

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
Example

```javascript
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: Execution tree

\[ x = x_0 \]
\[ y = y_0 \]
\[ z = 2 \cdot y_0 \]

\[ 2 \cdot y_0 = x_0 \]

\[ x_0 > y_0 + 10 \]

"Error"
<table>
<thead>
<tr>
<th>Concrete exec.</th>
<th>Symbolic exec.</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 22, \ y = 7, \ t = 14$</td>
<td>$x = x_0, \ y = y_0$</td>
<td></td>
</tr>
<tr>
<td>$x = 22, \ y = 7, \ t = 14$</td>
<td>$x = x_0, \ y = 2 \cdot y_0 \neq x_0$</td>
<td></td>
</tr>
<tr>
<td>After entering fet.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>After call to double ()</td>
<td>$2 \cdot y_0 = x_0$</td>
<td></td>
</tr>
<tr>
<td>after outer &quot;if&quot;</td>
<td>Solutions: $x_0 = 2, y_0 = 1$</td>
<td></td>
</tr>
</tbody>
</table>
Execution 2

Concrete exec.

- \( x = 2 \), \( y = 1 \)
- \( x = 2 \), \( y = 1 \)
- \( z = 2 \)

Symb. exec.

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

Path cond.

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

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

Solution: \( x_0 = 30 \), \( y_0 = 15 \)  \( \triangleright \) Hits “Error”
Exploring the Execution Tree
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 (1 < 0.5)) {
        if (a > 1) {
            console.log ("yes")
        }
    }
}
```

**Exec. 1**
- a = 0
- true
- path constraint: \(a_0 \leq 1\)
- negate & solve: \(a_0 = 2\)

**Exec. 2**
- a = 2
- false
- Divergent execution
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)
Overview

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
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’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