Program Analysis – Lecture 7
Symbolic and Concolic Execution
(Part 1)
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
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

Some JS engines

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);
```

Arrays are objects

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

Options:
- 112233
- 0012
- 66
- Something else

Some JS engines
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++) ...
```

112233 0012 66

Something else

Some JS engines
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
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);
}
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
t = 2
-2 + 0 + 2 ≠ 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);
}
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);
    }
}

\[
\begin{align*}
    x &= x_0 \\
    y &= y_0 \\
    s &= \"foo\"
\end{align*}
\]

\[
\begin{tikzpicture}
    \node (x) {\(x_0 < y_0\)};
    \node (s) [below left of=x] {s = "bar"};
    \node (y) [below right of=x] {y_0 = 23};
    \node (t) [below right of=s] {t};
    \node (f) [below right of=y] {f};
    \node (cons) [below right of=t] {cons...};
    \node (cons2) [below right of=f] {cons...};
    \coordinate (node2) at (node1.south); 
    \node (node3) at (node2 |- node1) {$\rightarrow 3$ nodes, 7 edges}; 
\end{tikzpicture}
\]
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

```javascript
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, 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
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
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
function f(a) {
    var x = a;
    while (x > 0) {
        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