Program Testing and Analysis:
Random and Fuzz Testing (Part 2)

Dr. Michael Pradel
Software Lab, TU Darmstadt
What does the following code print?

```javascript
function f(a,b) {
   var x;
   for (var i = 0; i < arguments.length; i++) {
      x += arguments[i];
   }
   console.log(x);
}

f(1,2,3);
```

3  6  NaN  Nothing
Warm-up Quiz

What does the following code print?

```
function f(a,b) {
    var x;
    for (var i = 0; i < arguments.length; i++) {
        x += arguments[i];
    }    
    console.log(x);
}

f(1,2,3);
```

3 6 NaN Nothing
What does the following code print?

```javascript
function f(a,b) {
    var x;
    for (var i = 0; i < arguments.length; i++) {
        x += arguments[i];
    }
    console.log(x);
}
f(1,2,3);
```

Array-like object that contains all three arguments

3 6 NaN

Nothing
Warm-up Quiz

What does the following code print?

```javascript
function f(a, b) {
    var x = undefined;  // Initialized to undefined
    for (var i = 0; i < arguments.length; i++) {
        x += arguments[i];  // undefined + some number yields NaN
    }
    console.log(x);
}

f(1, 2, 3);
```

3 6 NaN  Nothing
Outline

- Feedback-directed random test generation
  Based on *Feedback-Directed Random Test Generation*, Pacheco et al., ICSE 2007

- Adaptive random testing
  Based on *ARTOO: Adaptive Random Testing for Object-oriented Software*, Ciupa et al., ICSE 2008

- Fuzz testing
  Based on *Fuzzing with Code Fragments*, Holler et al., USENIX Security 2012
Fuzz Testing

Generate random inputs

- **Generative**: Create new random input, possibly based on constraints and rules
- **Mutative**: Derive new inputs from existing input by randomly modifying it
Grammar-based Language Fuzzing

Idea: Combine generative and mutative approach to test JavaScript interpreter

- Create random JavaScript programs based on language grammar
- Use and re-combine fragments of code from existing corpus of programs
  - Corpus: Programs that have exposed bugs before
Overview of LangFuzz

Phase I
Learning code fragments from sample code and test suite

Phase II
LangFuzz generated (mutated) test cases

Phase III
Feed test case into interpreter, check for crashes and assertions
Learning Code Fragments

- Parse existing programs into ASTs
- Extract code fragments
  - Examples for non-terminals of grammar
Learning Code Fragments: Example

\( \langle P \rangle := \langle C \rangle | \langle E \rangle | \langle B \rangle \)

\( \langle C \rangle := 1 := \langle E \rangle \mid \langle C \rangle ; \langle C \rangle \mid \text{if } \langle B \rangle \text{ then } \langle C \rangle \text{ else } \langle C \rangle \mid \text{while } \langle B \rangle \text{ do } \langle C \rangle \mid \text{skip} \)

\( \langle E \rangle := \text{true} | \text{false} | \langle E \rangle \langle bop \rangle \langle E \rangle | \neg \langle B \rangle | \langle B \rangle \land \langle B \rangle \)

\( \langle bop \rangle := < | > | = \)

Corpus:

\[
\begin{align*}
\text{x := !} & \text{y := !} \\
\text{C} & \text{P}
\end{align*}
\]
Mutation of Code

- Randomly pick and parse an existing program
- Randomly pick some fragments and replace with learned fragments of same type
Mutation of Code: Example

if \( !x > 4 \) then \( y := !z \) else

\[ \]

\[ \]

if \( !x > 4 \) then \( y := !z \) else \( x := !y \)

command from corpus
Generation of Code

Breadth-first application of grammar rules

- Set current expansion $e_{\text{cur}}$ to start symbol $P$
- Loop $k$ iterations:
  - Choose a random non-terminal $n$ in $e_{\text{cur}}$
  - Pick one of the rules, $r$, that can be applied to $n$
  - Replace occurrence of $n$ in $e_{\text{cur}}$ by $r(n)$

After $k$ iterations: Replace remaining non-terminals with fragments
Code Generation: Example

$k = 3$

e_{cur} = P \quad \Rightarrow \quad p ::= C | ...$

e_{cur} = C \quad \Rightarrow \quad c ::= \ldots \mid \text{while } B \text{ do } C | ...$

e_{cur} = \text{while } B \text{ do } C \quad \Rightarrow \quad c ::= \ldots \mid C ; C | ...$

e_{cur} = \text{while } B \text{ do } C ; C

Replace with fragments learned from corpus
Quiz

Which of the following SIMP programs could have been generated by LangFuzz?

```plaintext
if B then C; C

if !x > 3 then skip else skip; y := 1

if !x > 3 then while; while
```
Quiz

Which of the following SIMP programs could have been generated by LangFuzz?

- \texttt{if B then C; C}\hspace{2cm}(has unexpanded non-terminals)
- \texttt{if !x > 3 then skip else skip; y := 1}\hspace{2cm}(syntactically incorrect)
- \texttt{if !x > 3 then while; while}
Results

- Used to test Mozilla’s and Chrome’s JavaScript engines
- Found various bugs
  Mostly crashes of engine due to memory issues
- Rewarded with bug bounties summing up to $50,000
- First author now works at Mozilla
Summary

Random and fuzz testing

- Fully automated and unbiased
- Non-naive approaches can be very effective
- Trade-off: Cost of generating inputs vs. effectiveness in exposing bugs
  - Quickly generated, less effective tests may be better than slowly generated, more effective tests
Program Testing and Analysis: Symbolic and Concolic Testing

Dr. Michael Pradel
Software Lab, TU Darmstadt
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);
}
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);
    }
}
```
Quiz:

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

3 nodes, 7 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

```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
Examples

hand-written notes
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
Dealing with Large Execution Trees

```
function f(a) {
    var x = a;
    while (x > 0) {
        x --;
    }
    ...
}
```

\[ a = a_0, \quad x = a_0 \]
\[ a_0 > 0 \]
\[ x -- \]
\[ a_0 - 1 > 0 \]
\[ t \]
\[ a_0 - 2 > 0 \]
\[ \rightarrow \text{ infinitely large execution tree} \]
\[ \rightarrow "\text{fork bomb}" \]
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") {
    ...
}
```
Modeling the Environment (2)

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
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
Concolic Execution: 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";
        }
    }
}
```

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

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

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

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

Execution 1:

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

Solution: \( x_0 = 2, y_0 = 1 \)
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: \( x_0 = 30 \land y_0 = 15 \) → Will hit error!
Exploring the execution tree

etc.