Lec 23: Symbolic Execution

CS492E: Introduction to Software Security

Sang Kil Cha
Motivation

```java
if (input == 42) {
    /* ... */
} else {
    /* ... */
}
```
Program Execution
Simple Language (SLang)

- Simple assembly-like language
- Assume that there is only one type: 32-bit integer
- denotes a binary operator (+, -, x, /, etc.)
- denotes a unary operator (minus)
SLang (in BNF)

```
program ::= stmt*

stmt ::= var = exp
      | goto exp
      | if exp then goto exp₁, else goto exp₂
      | store(exp₁,exp₂)
      | output(exp)

exp ::= exp ⊗ exp
     | ◇ exp
     | load(exp)
     | get_input()
     | var
     | integer
```
Example Program

1. \( x = \text{get\_input()} \)
2. if \( x \% 2 == 0 \) goto 3 else goto 5
3. \( s = x + 2 \)
4. goto 6
5. \( s = x + 3 \)
6. output(s)
Defining Semantics

(2)

\[
\begin{align*}
\text{Computations} \\
<\text{Current state}, \ stmt \quad \rightarrow \quad <\text{End state}, \ stmt'>
\end{align*}
\]
Evaluation Rule

\[
\langle \text{state} \rangle \vdash e \Downarrow v
\]

Expression \quad Value
Execution Context (State)

- $\Delta$: variables
- $\Sigma$: list of statements
- $\mu$: current memory state
- $pc$: program counter
Operational Semantics

\[ \frac{v \text{ is input from src}}{\mu, \Delta \vdash \text{get\_input}(\text{src}) \downarrow v} \quad \text{INPUT} \]

\[ \frac{\mu, \Delta \vdash \text{var} \downarrow \Delta[\text{var}]}{\mu, \Delta \vdash \text{var} \downarrow \Delta[\text{var}]} \quad \text{VAR} \]

\[ \frac{\mu, \Delta \vdash e_1 \downarrow v_1 \quad \mu, \Delta \vdash e_2 \downarrow v_2 \quad v' = v_1 \Box v_2}{\mu, \Delta \vdash e_1 \Box e_2 \downarrow v'} \quad \text{BINARY-OP} \]

\[ \frac{\mu, \Delta \vdash e \downarrow v \quad \Delta' = \Delta[\text{var} \leftarrow v] \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \Delta, pc, \text{var} := e \rightsquigarrow \Sigma, \mu, \Delta', pc + 1, \iota} \quad \text{ASSIGN} \]

\[ \frac{\mu, \Delta \vdash e \downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \Delta, pc, \text{goto} e \rightsquigarrow \Sigma, \mu, \Delta, v_1, \iota} \quad \text{GOTO} \]

\[ \frac{\mu, \Delta \vdash e \downarrow v \quad v' = \Diamond v}{\mu, \Delta \vdash \Diamond e \downarrow v'} \quad \text{UNARY-OP} \]

\[ \frac{\mu, \Delta \vdash e \downarrow v}{\mu, \Delta \vdash e \downarrow v} \quad \text{LOAD} \]
Operational Semantics (Cont’d)

\[
\begin{align*}
\mu, \Delta \vdash e \downarrow 1 & \quad \Delta \vdash e_1 \downarrow v_1 & \quad \iota = \Sigma[v_1] & \quad \text{TRUE-COND} \\
\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \leadsto \Sigma, \mu, \Delta, v_1, \iota \\
\mu, \Delta, \vdash e \downarrow 0 & \quad \Delta \vdash e_2 \downarrow v_2 & \quad \iota = \Sigma[v_2] & \quad \text{FALSE-COND} \\
\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \leadsto \Sigma, \mu, \Delta, v_2, \iota \\
\mu, \Delta \vdash e_1 \downarrow v_1 & \quad \mu, \Delta \vdash e_2 \downarrow v_2 & \quad \iota = \Sigma[pc + 1] & \quad \mu' = \mu[v_1 \leftarrow v_2] \quad \text{STORE} \\
\Sigma, \mu, \Delta, pc, \text{store}(e_1, e_2) \leadsto \Sigma, \mu', \Delta, pc + 1, \iota \\
\mu, \Delta \vdash e \downarrow 1 & \quad \iota = \Sigma[pc + 1] & \quad \text{ASSERT} \\
\Sigma, \mu, \Delta, pc, \text{assert}(e) \leadsto \Sigma, \mu, \Delta, pc + 1, \iota
\end{align*}
\]
Example

• Let $\mu = \{\}$, $\Delta = \{x \mapsto 3, y \mapsto 5, z \mapsto 7\}$
• Evaluate $x + y$, given $\mu$ and $\Delta$

\[
\frac{\mu, \Delta \vdash x \downarrow 3 \quad \mu, \Delta \vdash y \downarrow 5}{\mu, \Delta \vdash x + y \downarrow 8}
\]

\[
\frac{\mu, \Delta \vdash x + y \downarrow 8 \quad \mu, \Delta \vdash y \downarrow 5}{\mu, \Delta \vdash (x + y)y \downarrow 40}
\]
Example 2

• Let $\mu = \emptyset$, $\Delta = \{x \mapsto 3, y \mapsto 5, z \mapsto 7\}$
• Evaluate $x + y > z$, given $\mu$ and $\Delta$
Example Program (Revisited)

We can now evaluate this program formally based on the operational semantics.

1. \( x = \text{get\_input}() \) // returns 2
2. if \( x \mod 2 == 0 \) goto 3 else goto 5
3. \( s = x + 2 \)
4. goto 6
5. \( s = x + 3 \)
6. output(s)
Symbolic Execution
Concrete vs. Symbolic Execution

- Concrete execution = runs a program with a concrete input
- Symbolic execution = runs a program with a symbolic input
  - We mark user input as a symbol.
  - A symbol represents any possible value.
  - We cannot evaluate a symbol into a concrete value.

In terms of semantics, we can have two types of values: Either integer or symbolic variable
Symbolic Execution Semantics

Value can be either an integer or a symbol

\[
\frac{v \text{ is input from src}}{\mu, \Delta \vdash \text{get\_input}(src) \downarrow v} \quad \text{INPUT}
\]

\[
\frac{v \text{ is a fresh symbol}}{\mu, \Delta \vdash \text{get\_input}(\cdot) \downarrow v} \quad \text{INPUT}
\]

A user input = a fresh new symbol
Symbolic Execution Semantics

What if we encounter a conditional jump where the condition is symbolic?

\[
\begin{align*}
\text{TRUE-COND} & \\
\mu, \Delta \models e \Downarrow 1 & \quad \Delta \models e_1 \Downarrow v_1 & & i = \Sigma[v_1] \\
\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 & \rightsquigarrow \Sigma, \mu, \Delta, v_1, i
\end{align*}
\]

\[
\begin{align*}
\text{FALSE-COND} & \\
\mu, \Delta \models e \Downarrow 0 & \quad \Delta \models e_2 \Downarrow v_2 & & i = \Sigma[v_2] \\
\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 & \rightsquigarrow \Sigma, \mu, \Delta, v_2, i
\end{align*}
\]

The condition is symbolic now …
Introducing a New Execution Context

(Π)

Path formula (a.k.a. path constraints, path predicate) Π

- Π is true at the beginning of the program
- For every symbolic branch, we update the path formula

\[
\begin{align*}
\frac{\mu, \Delta \vdash e \downarrow e' \quad \Delta \vdash e_1 \downarrow v_1}{\Pi, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \leadsto \Pi', \Sigma, \mu, \Delta, v_1, \tau} \quad \text{TRUE-COND} \\
\frac{\mu, \Delta, \vdash e \downarrow e' \quad \Delta \vdash e_2 \downarrow v_2}{\Pi, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \leadsto \Pi', \Sigma, \mu, \Delta, v_2, \tau} \quad \text{FALSE-COND}
\end{align*}
\]
Example Program (Revisited)

Can we symbolically evaluate this program now?

```plaintext
1  x = get_input()  // symbolic input!
2  if x % 2 == 0 goto 3 else goto 5
3  s = x + 2
4  goto 6
5  s = x + 3
6  output(s)
```
Example Program (Revisited)

```
1 x = get_input() // symbolic input!
2 if x % 2 == 0 goto 3 else goto 5
3 s = x + 2
4 goto 6
5 s = x + 3
6 output(s)
```

Which branch to take?
Two Categories

• Static Symbolic Execution
  – Considers all branches
  – Symbolic Execution and Program Testing, *CACM 1976*

• Dynamic Symbolic Execution
  – Considers a single branch at a time
  – DART: Directed Automated Random Testing, *PLDI 2005*
  – EXE: Automatically Generating Inputs of Death, *CCS 2006*
Static vs. Dynamic Symbolic Execution

Static Symbolic Execution
- No need to run the program
- Environment handling difficult
- Complete (in theory)
- Too complex formulas
- No need to select paths

Dynamic Symbolic Execution
- Runtime analysis
- Easy to handle environments
- Incomplete
- Simpler formulas
- Path selection problem

Soundness really matter in practice
Dynamic Symbolic Execution

- Concrete + Symbolic = Concolic


- DART: Directed Automated Random Testing, *PLDI 2005*

- EXE: Automatically Generating Inputs of Death, *CCS 2006*
Example Program (Revisited)

1. $x = \text{get\_input}()$  // symbolic input!
2. if $x \% 2 == 0$ goto 3 else goto 5
3. $s = x + 2$
4. goto 6
5. $s = x + 3$
6. output(s)

How to generate a concrete test case from a path formula?
Constraint Solving

- Compute satisfying answers from a given formula
- SAT (Boolean Satisfiability Problem)
  - Given a Boolean formula, find satisfying assignments
- SMT (Satisfiability Modulo Theory)
  - SAT++ (SAT + first-order theories)
  - Nonlinear constraints are problematic (e.g., sin, cos, etc.)
Example Program (Revisited)

// symbolic input!

```
1 x = get_input()  // symbolic input!
2 if x % 2 == 0 goto 3 else goto 5
3 s = x + 2
4 goto 6
5 s = x + 3
6 output(s)
```

\[ \Pi : x \% 2 = 0 \]

SMT solver

Test Case, e.g., \( x = 42 \)
Exploring Path with Symbolic Execution

• (Dyamic) symbolic execution exercises each execution path systematically

• But how do we detect that we found a bug?

Safety Property
Safety Property in Symbolic Execution

- Memory out of bounds
- Null dereference
- Integer overflow
- Etc.
Dyanmic Symbolic Execution = White-box Fuzzing

• White-box fuzzing vs. grey-box fuzzing?

• White-box fuzzing vs. black-box fuzzing?
Key Challenges

• Path explosion

• SMT solving is hard
Conclusion

• White-box fuzzing (dynamic symbolic execution) is a systematic way to explore program execution paths.

• There are several key challenges in symbolic execution, and it is an active research area.
Questions?