Lec 23: Symbolic Execution

CS492E: Introduction to Software Security

Sang Kil Cha

Motivation

```
if (input == 42) {
  /* ... */
} else {
  /* ... */
}
```

Program Execution

Simple Language (SLang)

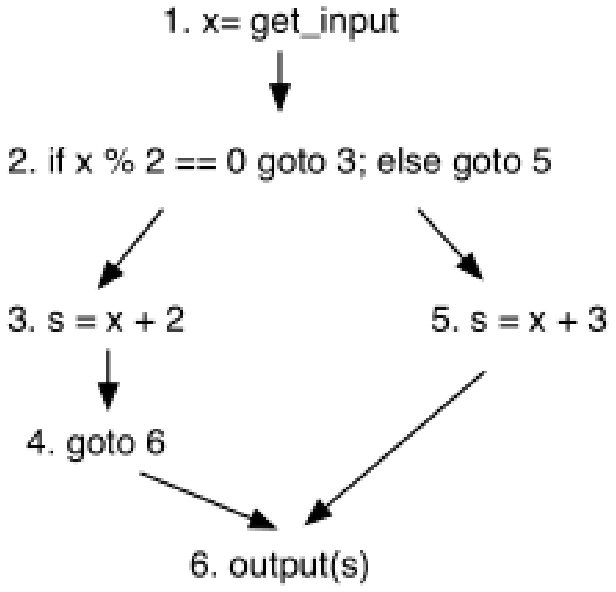
- Simple assembly-like language
- Assume that there is only one type: 32-bit integer
- □ denotes a binary operator (+, -, x, /, etc.)
- ◊ denotes a unary operator (minus)

SLang (in BNF)

$$
\begin{array}{lcl}
\text{program} & ::= & \text{stmt}^* \\
\\
\text{stmt} & ::= & \text{var} = \text{exp} \\
 & | & \text{goto exp} \\
 & | & \text{if exp then goto exp}_1\text{, else goto exp}_2 \\
 & | & \text{store(exp}_1\text{,exp}_2\text{)} \\
 & | & \text{output(exp)} \\
\\
\text{exp} & ::= & \text{exp} \; \Box \; \text{exp} \\
 & | & \Diamond \; \text{exp} \\
 & | & \text{load(exp)} \\
 & | & \text{get\_input()} \\
 & | & \text{var} \\
 & | & \text{integer}
\end{array}
$$

Example Program

```
1  x = 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)
```



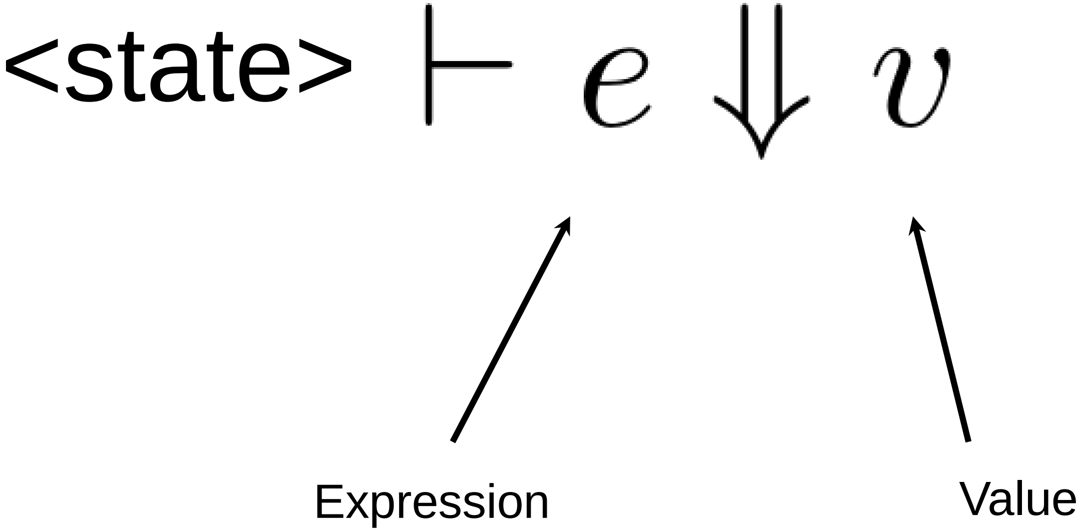
Defining Semantics

(2)

$$\frac{\text{Computations}}{\text{<Current state>}, \mathit{stmt} \;\rightarrow\; \text{<End state>}, \mathit{stmt'}}$$

(1) (3)

Evaluation Rule



$$\text{<state>} \vdash e \Downarrow v$$

Expression

Value

Execution Context (State)

- Δ : variables
- $\sum$: list of statements
- μ : current memory state
- *pc* : program counter

Operational Semantics

$$\frac{v \text{ is input from src}}{\mu, \Delta \vdash \texttt{get\_input}(src) \Downarrow v}\ \textsc{Input} \qquad \frac{\mu, \Delta \vdash e \Downarrow v_1 \quad v = \mu[v_1]}{\mu, \Delta \vdash \text{load } e \Downarrow v}\ \textsc{Load}$$

$$\frac{}{\mu, \Delta \vdash var \Downarrow \Delta[var]}\ \textsc{Var} \qquad \frac{\mu, \Delta \vdash e \Downarrow v \quad v' = \Diamond v}{\mu, \Delta \vdash \Diamond e \Downarrow v'}\ \textsc{Unary-Op}$$

$$\frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad v' = v_1 \Box v_2}{\mu, \Delta \vdash e_1 \Box e_2 \Downarrow v'}\ \textsc{Binary-Op} \qquad \frac{}{\mu, \Delta \vdash v \Downarrow v}\ \textsc{Const}$$

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad \Delta' = \Delta[var \leftarrow v] \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \Delta, pc, var := e \leadsto \Sigma, \mu, \Delta', pc + 1, \iota}\ \textsc{Assign} \qquad \frac{\mu, \Delta \vdash e \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \Delta, pc, \text{goto } e \leadsto \Sigma, \mu, \Delta, v_1, \iota}\ \textsc{Goto}$$

Operational Semantics (Cont'd)

$$\frac{\mu, \Delta \vdash e \Downarrow 1 \quad \Delta \vdash e_1 \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \leadsto \Sigma, \mu, \Delta, v_1, \iota} \ \text{\textsc{True-Cond}}$$

$$\frac{\mu, \Delta, \vdash e \Downarrow 0 \quad \Delta \vdash e_2 \Downarrow v_2 \quad \iota = \Sigma[v_2]}{\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \leadsto \Sigma, \mu, \Delta, v_2, \iota} \ \text{\textsc{False-Cond}}$$

$$\frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad \iota = \Sigma[pc + 1] \quad \mu' = \mu[v_1 \leftarrow v_2]}{\Sigma, \mu, \Delta, pc, \text{store}(e_1,\ e_2) \leadsto \Sigma, \mu', \Delta, pc + 1, \iota} \ \text{\textsc{Store}}$$

$$\frac{\mu, \Delta \vdash e \Downarrow 1 \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \Delta, pc, \text{assert}(e) \leadsto \Sigma, \mu, \Delta, pc + 1, \iota} \ \text{\textsc{Assert}}$$

Example

- Let μ = {}, Δ = {x ↦ 3, y ↦ 5, z ↦ 7}
- Evaluate x + y, given μ and Δ

$$\dfrac{\dfrac{\dfrac{}{\mu, \Delta \vdash x \Downarrow 3} \qquad \dfrac{}{\mu, \Delta \vdash y \Downarrow 5}}{\mu, \Delta \vdash x + y \Downarrow 8} \qquad \dfrac{}{\mu, \Delta \vdash y \Downarrow 5}}{\mu, \Delta \vdash (x + y)y \Downarrow 40}$$

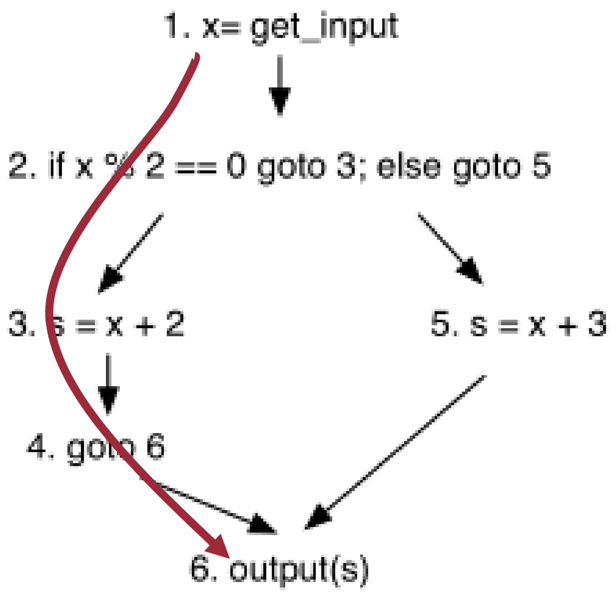
Example 2

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

Example Program (Revisited)

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

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



1. x= get_input

2. if x % 2 == 0 goto 3; else goto 5

3. s = x + 2

5. s = x + 3

4. goto 6

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

$$\frac{v \text{ is input from src}}{\mu, \Delta \vdash \texttt{get\_input}(src) \Downarrow v}\ \textsc{Input}$$

⬇

$$\frac{v \text{ is a fresh symbol}}{\mu, \Delta \vdash \texttt{get\_input}(\cdot) \Downarrow v}\ \textsc{Input}$$

A user input = a fresh new symbol

Symbolic Execution Semantics

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

$$\frac{\mu, \Delta \vdash e \Downarrow \boxed{1} \quad \Delta \vdash e_1 \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Sigma, \mu, \Delta, v_1, \iota}\ \textsc{True-Cond}$$

$$\frac{\mu, \Delta, \vdash e \Downarrow \boxed{0} \quad \Delta \vdash e_2 \Downarrow v_2 \quad \iota = \Sigma[v_2]}{\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Sigma, \mu, \Delta, v_2, \iota}\ \textsc{False-Cond}$$

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

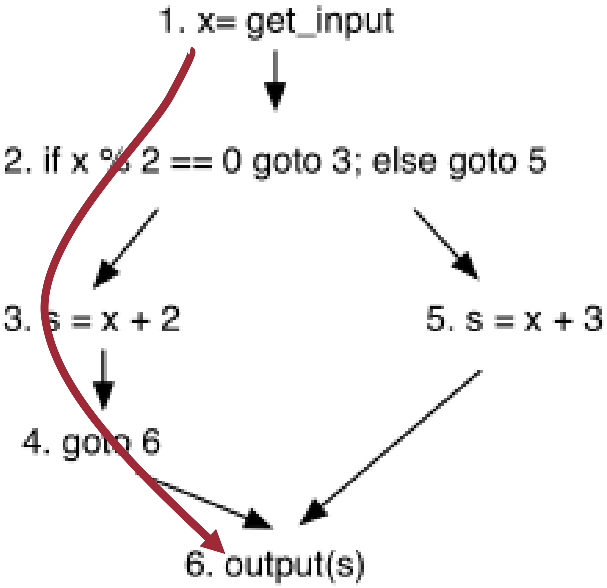
$$\frac{\mu, \Delta \vdash e \Downarrow e' \quad \Delta \vdash e_1 \Downarrow v_1 \quad \Pi' = \Pi \wedge (e' = 1) \quad \iota = \Sigma[v_1]}{\Pi, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \leadsto \Pi', \Sigma, \mu, \Delta, v_1, \iota} \ \textsc{True-Cond}$$

$$\frac{\mu, \Delta, \vdash e \Downarrow e' \quad \Delta \vdash e_2 \Downarrow v_2 \quad \Pi' = \Pi \wedge (e' = 0) \quad \iota = \Sigma[v_2]}{\Pi, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \leadsto \Pi', \Sigma, \mu, \Delta, v_2, \iota} \ \textsc{False-Cond}$$

Example Program (Revisited)

Can we symbolically evaluate this program now?

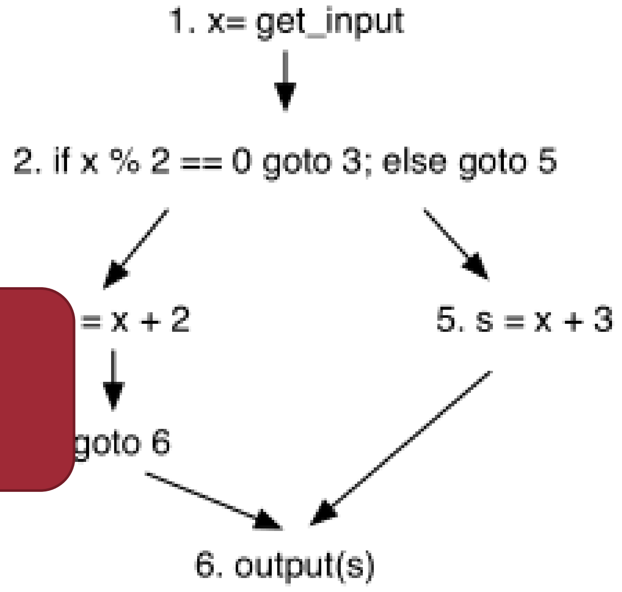
```
1  x = get_input()  // symbolic input!
2  if x % 2 == 0 goto 3 else goto 5
3  s = x + 2
4  goto 6
5  s = x + 3
6  output(s)
```



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	**Dynamic Symbolic Execution**
• No need to run the program	• Runtime analysis
• Environment handling difficult	• Easy to handle environments
• Complete (in theory)	• Incomplete
• Too complex formulas	• Simpler formulas
• No need to select paths	• Path selection problem

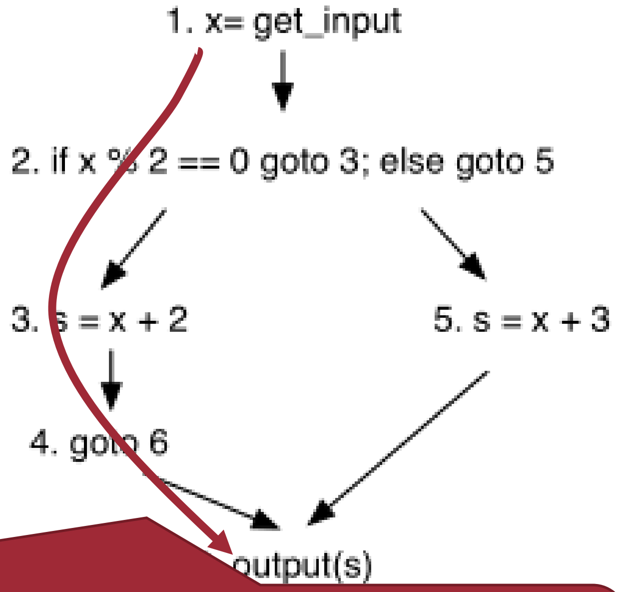
Soundness really matter in practice

Dynamic Symbolic Execution

- Concrete + Symbolic = Concolic
- CUTE: A Concolic Unit Testing Engine for C, ***FSE 2005***
- DART: Directed Automated Random Testing, ***PLDI 2005***
- EXE: Automatically Generating Inputs of Death, ***CCS 2006***

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)
```



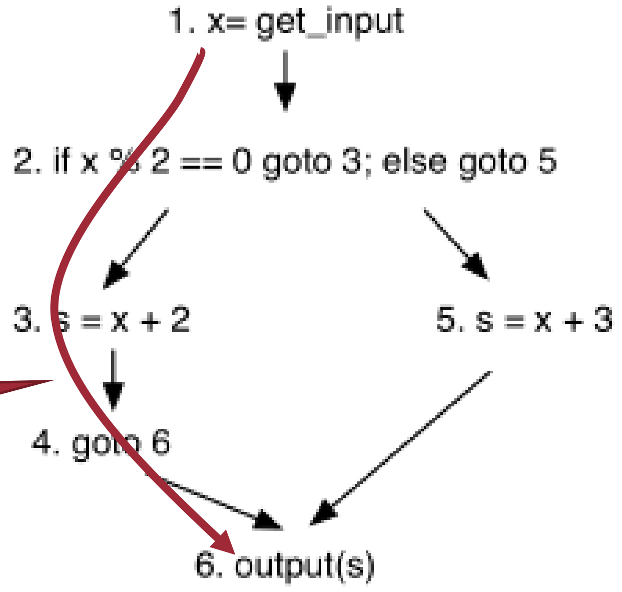
How to generate a concrete test case from a path formula?

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)

```
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?