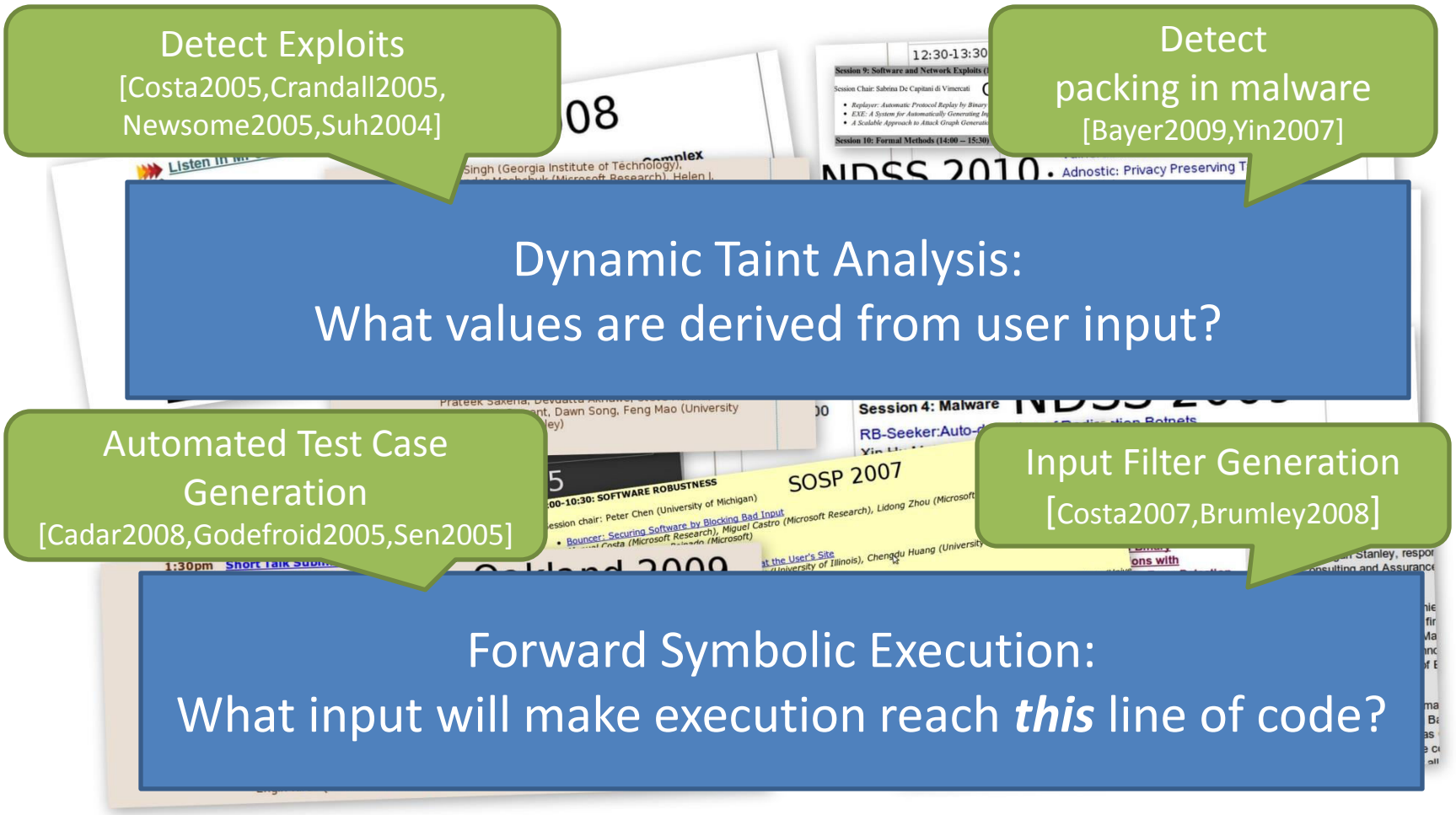
Taint Analysis

Paper

- Edward J. Schwartz, Thanassis Avgerinos, and David Brumley
  - All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask).
  - IEEE Symposium on Security and Privacy 2010

Two Essential Runtime Analyses



tainted untainted

```
x = get_input( )
y = x + 42
…
goto y
```

Input is tainted

Δ

Var	Val
x	7

Taint Introduction

$$\text{Input}\ \frac{t = \text{IsUntrusted}(src)}{\text{get\_input}(src)\downarrow t}$$

τ

Var	Tainted?
x	T

tainted untainted

```
x = get_input( )
y = x + 42
...
goto y
```

Data derived from user input is tainted

Δ

Var	Val
x	7
y	49

Taint Propagation

$$\text{BinOp}\ \frac{t_1 = \tau[x_1]\ ,\ t_2 = \tau[x_2]}{x_1 + x_2 \downarrow t_1 \vee t_2}$$

τ

Var	Tainted?
x	T
y	T

tainted untainted

x = get_input( )

y = x + 42

…

goto y

Policy Violation Detected

Taint Checking

$P_{goto}(t_a) = \neg t_a$

(Must be true to execute)

Δ

Var	Val
x	7
y	49

τ

Var	Tainted?
x	T
y	T

x = get_input()

y = …

…

goto y

Jumping to overwritten return address

Real Use: Exploit Detection

```
…
strcpy(buffer,argv[1]) ;
…
return ;
```

Memory Load

Variables

Δ

Var	Val
x	7

τ

Var	Tainted?
x	T

Memory

μ

Addr	Val
7	42

τ_μ

Addr	Tainted?
7	F

Problem: Memory Addresses

```
x = get_input( )
y = load( x )
…
goto y
```

All values derived from user input are tainted??

Δ

Var	Val
x	7

μ

Addr	Val
7	42

τ_μ

Addr	Tainted?
7	F

Policy 1: Taint depends only on the memory cell

x = get_ir

y = load(

…

goto y

Δ

Var	Val
x	7

Addr	Val
7	42

Undertainting

Failing to identify tainted values

- e.g., missing exploits

Taint Propagation

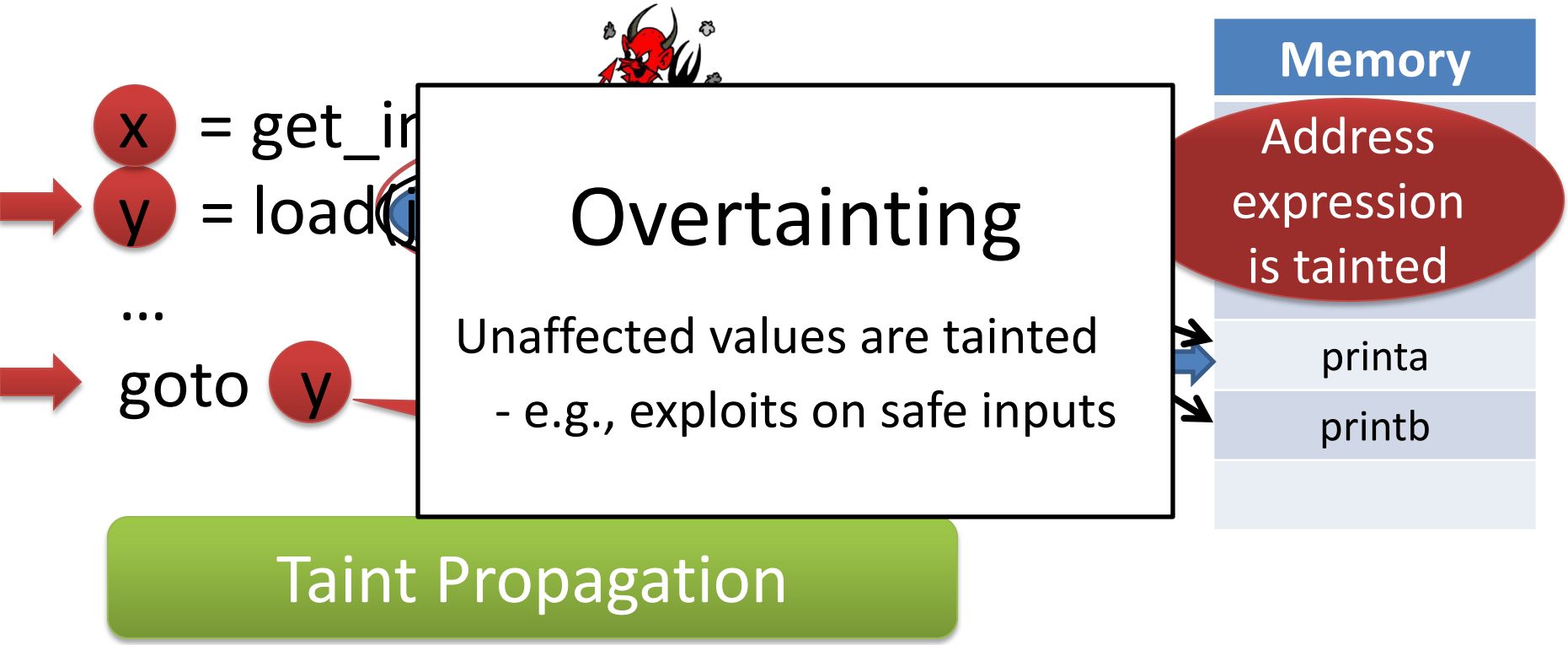
$$\text{Load}\ \frac{v = \Delta[x]\ ,\ t = \tau_\mu[v]}{\text{load}(x) \downarrow t}$$

τ_μ

Addr	Tainted?
7	F

Policy 2:

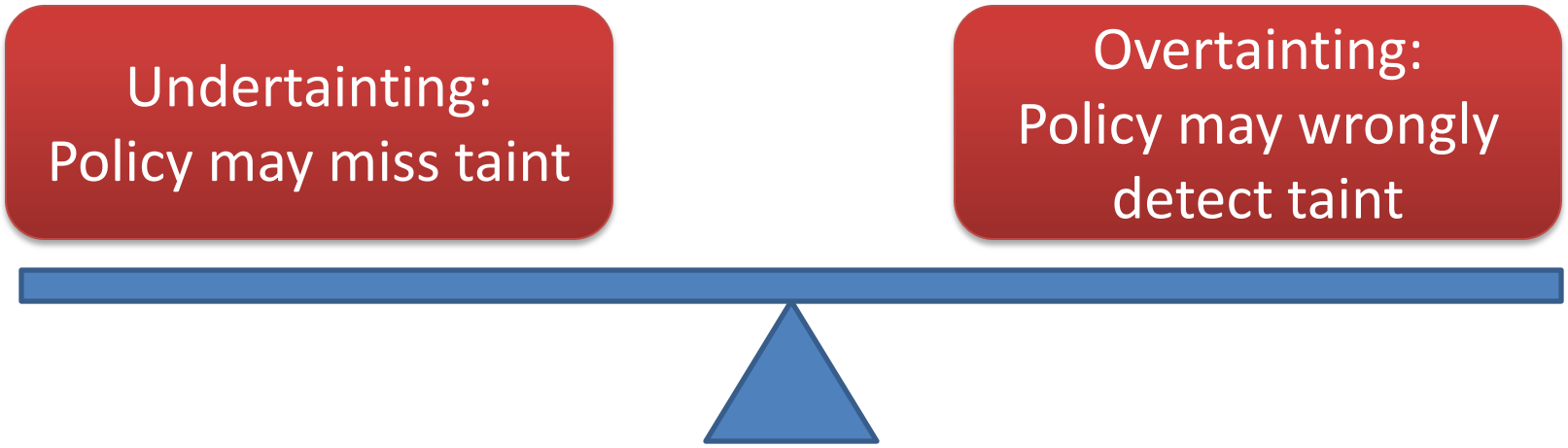
If either the address or the memory cell is tainted, then the value is tainted



Taint Propagation

$$\text{Load}\ \frac{v = \Delta[x]\ ,\ t = \tau_\mu[v],\ t_a = \tau[x]}{load(x) \downarrow t\ v\ t_a}$$

Research Challenge
State-of-the-Art is not perfect for all programs

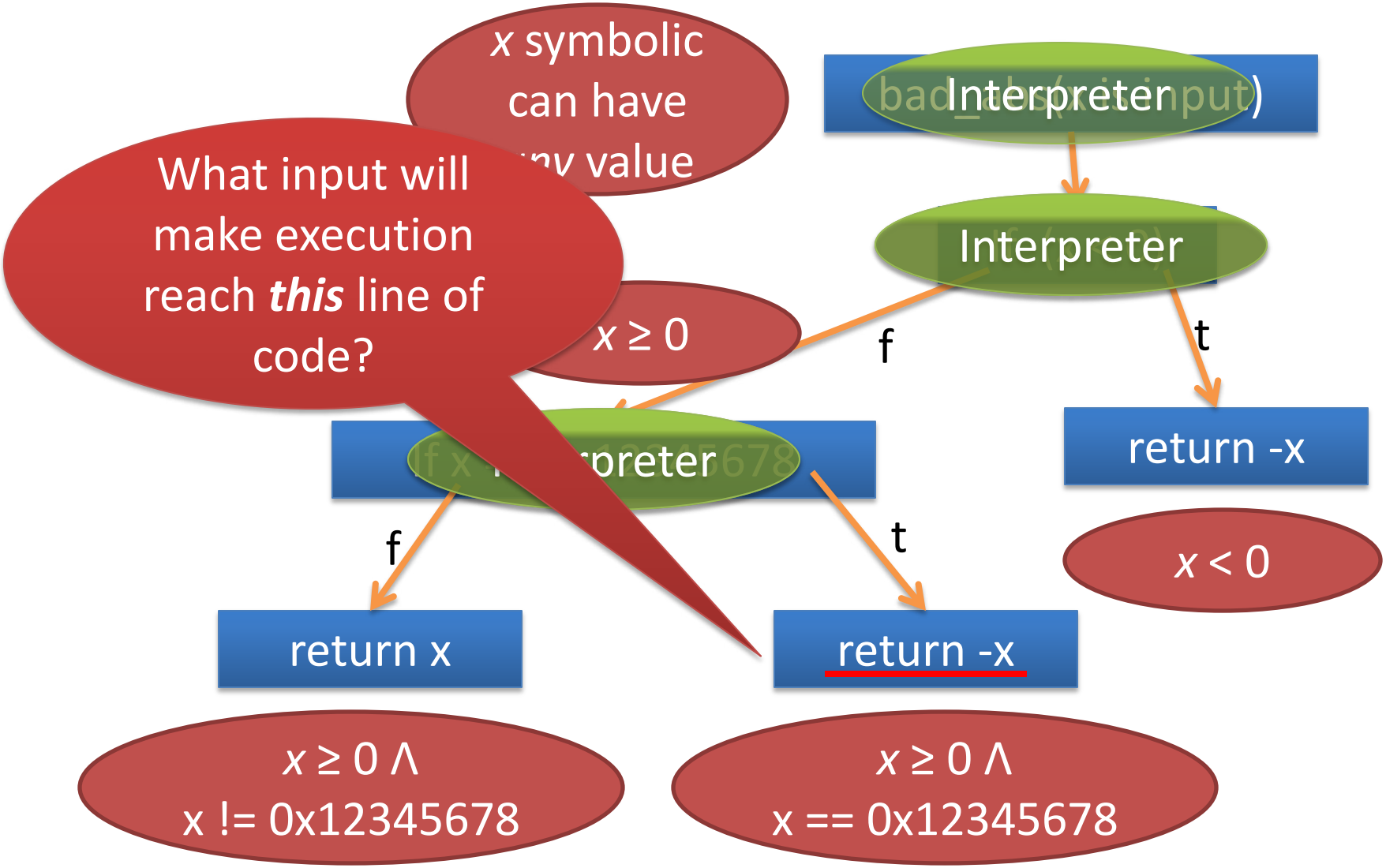


The Challenge

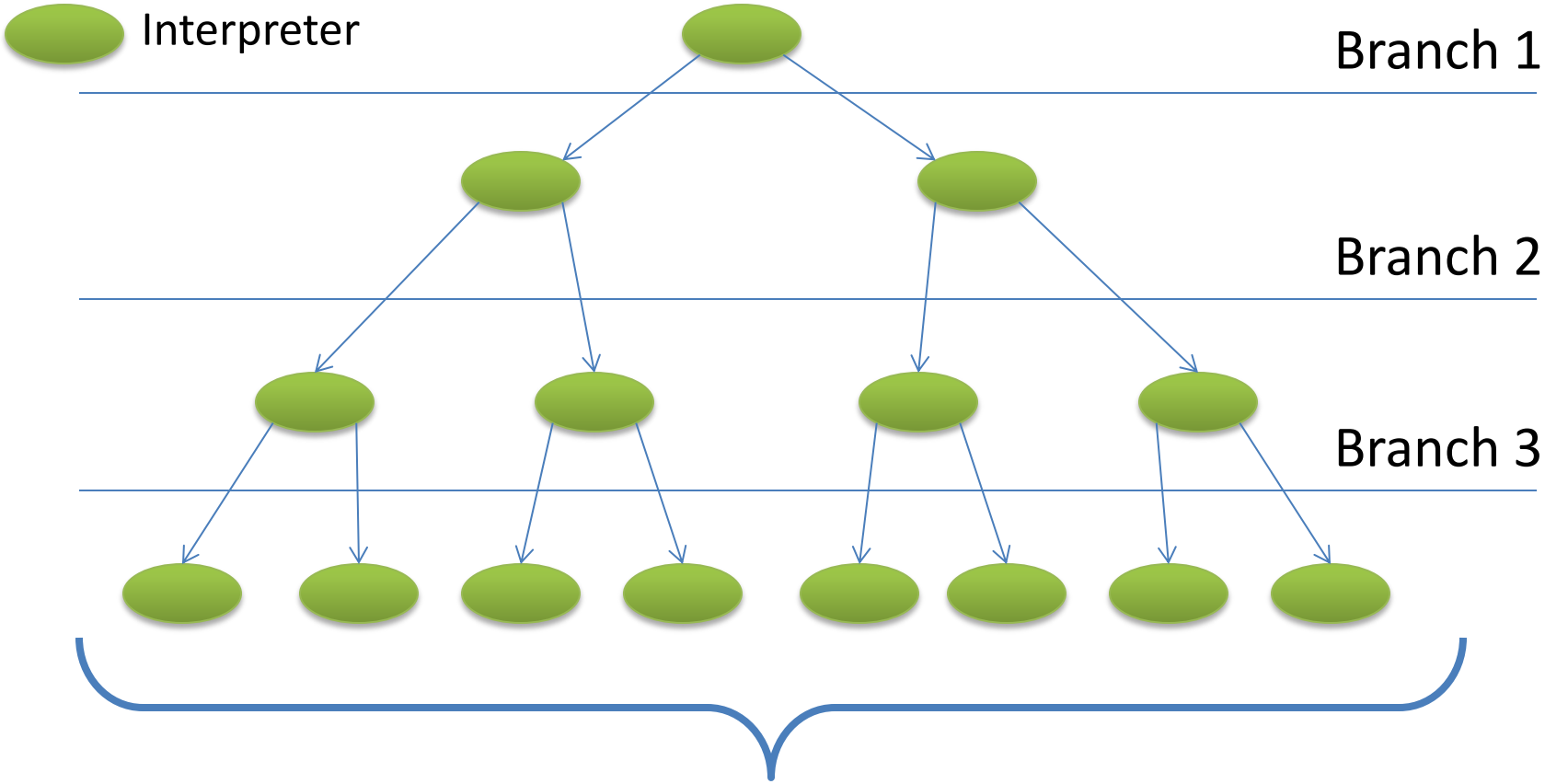

```
bad_abs(x is input)
    if (x < 0) then
            return -x
    if (x = 0x12345678) then
            return  -x
     return x
```

Forward Symbolic Execution:
What input will make execution reach ***this*** line of code?

A Simple Example

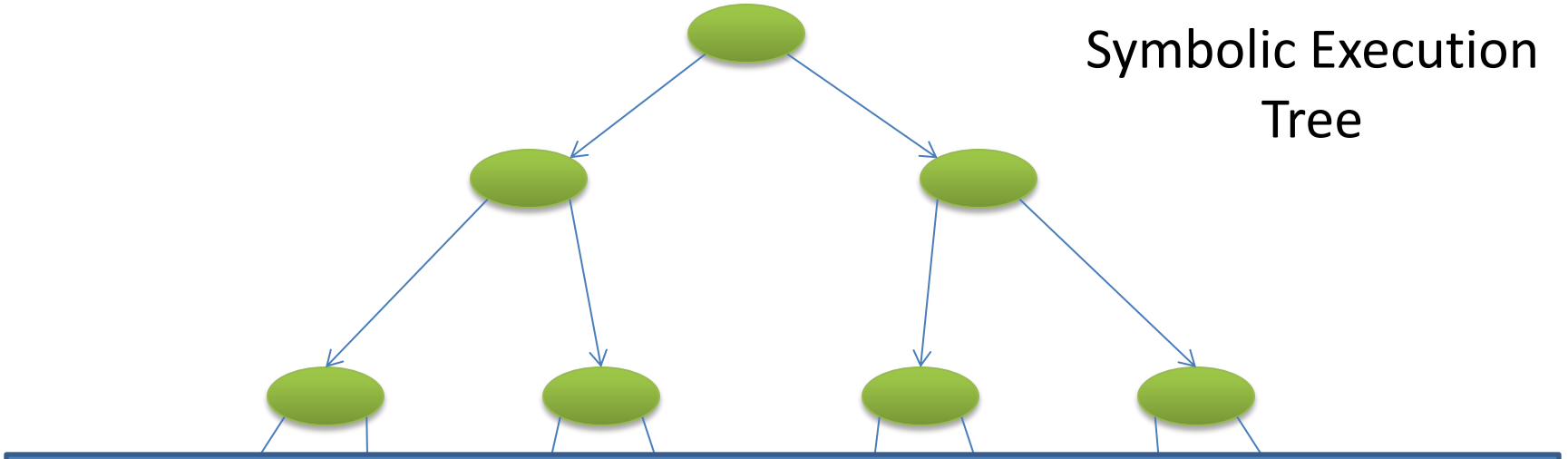


One Problem: Exponential Blowup Due to Branches



Exponential Number of Interpreters/formulas in # of branches

Path Selection Heuristics



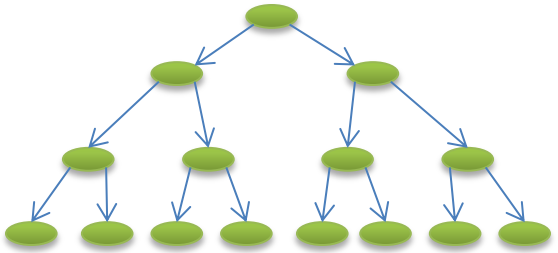
However, these are heuristics. In the worst case all create an exponential number of formulas in the tree height.

…

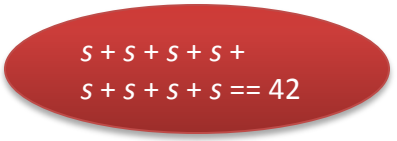
- Depth-First Search (bounded) ,Random Search [Cadar2008]
- Concolic Testing [Sen2005,Godefroid2008]

Symbolic Execution is *not* Easy

- Exponential number of interpreters/formulas

branching 

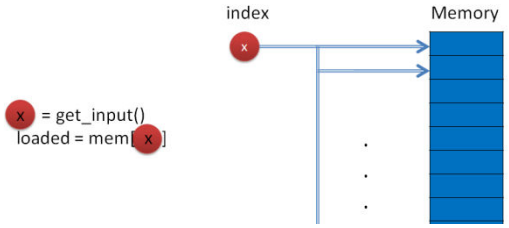
- Exponentially-sized formulas

substitution 

- Solving a formula is NP-Complete!

Other *Important* Issues

Symbolic Memory

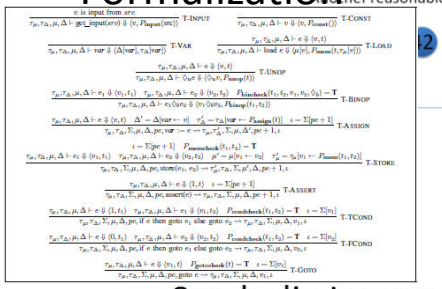


FORWARD SYMBOLIC EXECUTION

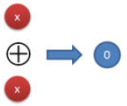
"Like normal execution, where inputs are substituted by symbolic variables"

King *et al.*, 1976

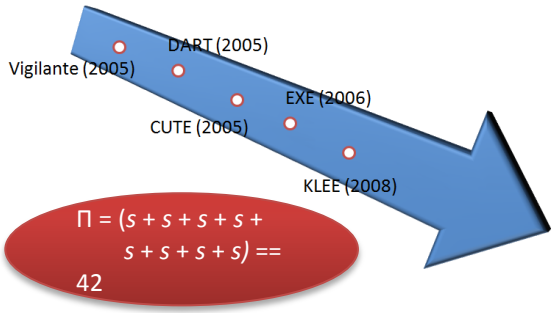
Formalization



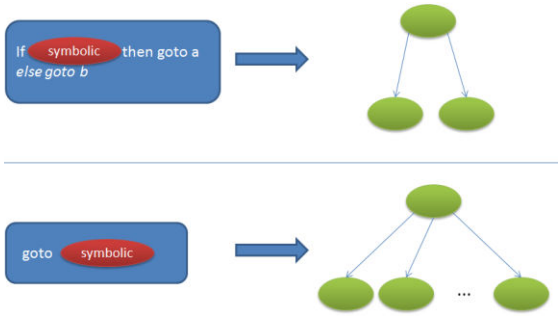
Sanitization



More complex policies



Symbolic Jumps



Summary

- Dynamic taint analysis and forward symbolic execution used extensively in literature
  - Formal algorithm and what is done for each possible step of execution often not emphasized