Systems Security
Ruhr-University Bochum

Program Analysis

Lecture 12: *Dynamic Taint Analysis and Symbolic Execution*
Winter term 2011/2012

Prof. Thorsten Holz

Horst Görtz Institut
für IT-Sicherheit

Recapitulation: SimpIL

program	::=	*stmt**
stmt s	::=	*var* := *exp* | store(*exp*, *exp*)
		| goto *exp* | assert *exp*
		| if *exp* then goto *exp*
		else goto *exp*
exp e	::=	load(*exp*) | *exp* $\Diamond_b$ *exp* | $\Diamond_u$ *exp*
		| *var* | get_input(src) | v
$\Diamond_b$	::=	typical binary operators
$\Diamond_u$	::=	typical unary operators
value v	::=	32-bit unsigned integer

Language definition

Recap: Operational Semantics

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

Context	Meaning
Σ	Maps a statement number to a statement
μ	Maps a memory address to the current value at that address
Δ	Maps a variable name to its value
pc	The program counter
ι	The next instruction

$\mu, \Delta \vdash e \Downarrow v$ denotes evaluating an expression e to a value v in the current state given by memory state μ and variables Δ

Recapitulation: Rule Evaluation

Systems Security
Ruhr-University Bochum

Rules are quite complex, evaluating rules is also complex ☹

```
1: x := 2 * get_input(·)
```

is evaluated for the input *20* to:

$$\dfrac{\dfrac{\dfrac{}{\mu,\Delta \vdash 2 \Downarrow 2}\ \textsc{Const} \quad \dfrac{20 \text{ is input}}{\mu,\Delta \vdash \text{get\_input}(\cdot) \Downarrow 20}\ \textsc{Input} \quad v' = 2 * 20}{\mu,\Delta \vdash 2\text{*get\_input}(\cdot) \Downarrow 40}\ \textsc{Binop} \quad \Delta' = \Delta[x \leftarrow 40] \quad \iota = \Sigma[pc+1]}{\Sigma,\mu,\Delta,pc,x := 2\text{*get\_input}(\cdot) \rightsquigarrow \Sigma,\mu,\Delta',pc+1,\iota}\ \textsc{Assign}$$

Since the ASSIGN rule requires the expression e in var := e to be evaluated, we have to recurse to other rules (BINOP, INPUT, CONST) to evaluate the complete expression

Dynamic Taint Analysis

Motivation

- Quite often we want to study how specific input influences a program
  - What input do we need to feed to take a specific jump in a program?
  - Can we trigger a vulnerability by feeding specific input to a program?
- *Taint analysis* enables us to track information flow within a given program

Intuition

- We want to study for a web application if the data received over the network can influence the program
- "Mark" memory location that stores input with a specific "color"
- Track how this memory location is used within the program and propagate "taint information"
- If **x** is tainted and `y := x + 5` is executed, then **y** is also tainted (`x := 23` removes taint info from **x**)

Example: Perl


```
$query = new CGI;
$username = $query->param(“username”);
$password = $query->param(“password”);
...
$sql_command = “select * from users where
                    username=‘$username’ and
                    password=‘$password’”;
$sth = $dbh->prepare($sql_command)
```

Example: Perl


```
$query = new CGI;
$username = $query->param(“username”);
$password = $query->param(“password”);
...
$sql_command = “select * from users where
                    username=‘$username’ and
                    password=‘$password’”;
$sth = $dbh->prepare($sql_command)
```

SQL Injection via
$passwort: ’; UPDATE users SET password = ‘foo

Example: Perl

- Such an attack can be prevented via different ways, for example via whitelisting only specific characters
- Perl can enforce that a check is performed before external input (= tainted variable) can be used in a function: taint analysis for variables
- Enable via `#!/usr/bin/perl -T`
- Learn more at http://perldoc.perl.org/perlsec.html#Taint-mode

Dynamic Taint Analysis

- Track information flow between *sources* and *sinks*
- Any program value whose computation depends on data derived from a taint source is considered *tainted* (denoted **T**)
- Any other value is considered untainted (denoted **F**)
- *Taint policy P* determines exactly how taint flows as a program executes, what sorts of operations introduce new taint, and what checks are performed on tainted values

DTA Semantics I

- Dynamic taint analysis is performed on code at runtime, thus we need to express DTA in terms of the operational semantics of the language
  - Yes, we need all these definitions again...
- Besides the operational semantics, we also add *taint policy actions*
  - How is taint info propagated, introduced or checked?

DTA Semantics II

- We want to keep track of taint status of each program value and thus need to extend semantics
  - We simply store taint status together with variable
  - Tuple $\langle \mathbf{v}, \boldsymbol{\tau} \rangle$ means **v** is a value in the initial language, and **τ** is the taint status of **v**
- We keep maps of this info
  - τ_Δ maps variables to taint status
  - τ_μ maps address to taint status

Initialized so that all values are marked untainted

Taint Policy

- Again, we need to precisely define how taint information is handled, similar to previous definitions
  - Introduce a *taint policy P*
  - Defined how new taint is introduced, how taint propagates as instructions execute, and how taint is checked during execution
  - Yes, this seems to be boring, but it is necessary
- 8/12 rules are covered in the lecture, the remaining rules can be found in Figure 5 of the paper

Taint Introduction

$$\frac{v \text{ is input from } src}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{get\_input}(src) \Downarrow \langle v, P_{\textbf{input}}(\text{src}) \rangle} \quad \text{T-INPUT}$$

Context	Meaning
Σ	Maps a statement number to a statement
μ	Maps a memory address to the current value at that address
Δ	Maps a variable name to its value
pc	The program counter
ι	The next instruction

$\mu,\Delta \vdash e \Downarrow \langle v,\tau \rangle$ denotes evaluating an expression e to a value $\langle v,\tau \rangle$ in the current state given by memory state μ and variables Δ

τ_Δ	::=	Maps variables to taint status
τ_μ	::=	Maps addresses to taint status

Taint Introduction

$$\frac{v \text{ is input from } src}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{get\_input}(src) \Downarrow \langle v, P_{\textbf{input}}(\text{src})\rangle} \text{ T-INPUT}$$

Note that we do not deal with *taint removal*, a hard problem in itself!

Context	Meaning
Σ	Maps a statement number to a statement
μ	Maps a memory address to the current value at that address
Δ	Maps a variable name to its value
pc	The program counter
ι	The next instruction

μ,Δ ⊢ e ⇓ ⟨v,τ⟩ denotes evaluating an expression e to a value ⟨v,τ⟩ in the current state given by memory state μ and variables Δ

τ_Δ	::=	Maps variables to taint status
τ_μ	::=	Maps addresses to taint status

Taint Propagation

$$\frac{}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash v \Downarrow \langle v, P_{\mathbf{const}}() \rangle} \text{ T-CONST}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t \rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{load } e \Downarrow \langle \mu[v], P_{\mathbf{mem}}(t, \tau_\mu[v]) \rangle} \text{ T-LOAD}$$

Taint Propagation

$$\frac{}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash v \Downarrow \langle v, P_{\mathbf{const}}() \rangle} \text{ T-CONST}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t \rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{load } e \Downarrow \langle \mu[v], P_{\mathbf{mem}}(t, \tau_\mu[v]) \rangle} \text{ T-LOAD}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1 \rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2 \rangle \quad P_{\mathbf{bincheck}}(t_1, t_2, v_1, v_2, \lozenge_b) = \mathbf{T}}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \lozenge_b e_2 \Downarrow \langle v_1 \lozenge_b v_2, P_{\mathbf{binop}}(t_1, t_2) \rangle} \text{ T-BINOP}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t \rangle \quad \Delta' = \Delta[var \leftarrow v] \quad \tau'_\Delta = \tau_\Delta[var \leftarrow P_{\mathbf{assign}}(t)] \quad \iota = \Sigma[pc + 1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, var := e \rightsquigarrow \tau_\mu, \tau'_\Delta, \Sigma, \mu, \Delta', pc + 1, \iota} \text{ T-ASSIGN}$$

Taint Checking

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v_1, t \rangle \quad P_{\mathbf{gotocheck}}(t) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{goto } e \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota} \text{ T-GOTO}$$

$P_{\text{gotocheck}}(\mathbf{t})$ returns **T** if it is safe to perform a jump operation when the target address has taint value **t**, and returns **F** otherwise. If **F** is returned, the premise for the rule is not met and the machine terminates abnormally.

Taint Checking

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v_1, t\rangle \quad P_{\mathbf{gotocheck}}(t) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{goto } e \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota} \text{ T-GOTO}$$

$P_{gotocheck}$(**t**) returns **T** if it is safe to perform a jump operation when the target address has taint value **t**, and returns **F** otherwise. If **F** is returned, the premise for the rule is not met and the machine terminates abnormally.

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_2\rangle \quad P_{\mathbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota} \text{ T-TCOND}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 0, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad P_{\mathbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_2]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_2, \iota} \text{ T-FCOND}$$

Tainted Jump Policy

We want to protect a program from control flow hijacking attacks: input-derived value should never overwrite a control-flow value (i.e., tainted jump targets are never used)

Tainted Jump Policy

We want to protect a program from control flow hijacking attacks: input-derived value should never overwrite a control-flow value (i.e., tainted jump targets are never used)

Component	Policy Check
$P_{\textbf{input}}(\cdot)$, $P_{\textbf{bincheck}}(\cdot)$, $P_{\textbf{memcheck}}(\cdot)$	$\mathbf{T}$
$P_{\textbf{const}}()$	$\mathbf{F}$
$P_{\textbf{unop}}(t)$, $P_{\textbf{assign}}(t)$	t
$P_{\textbf{binop}}(t_1, t_2)$	$t_1 \vee t_2$
$P_{\textbf{mem}}(t_a, t_v)$	t_v
$P_{\textbf{condcheck}}(t_e, t_a)$	$\neg t_a$
$P_{\textbf{gotocheck}}(t_a)$	$\neg t_a$

Example


```
x    :=   2*get_input(·)
y    :=   5  +  x
goto  y
```

Example


```
1  x := 2*get_input(·)
2  y := 5 + x
3  goto y
```

Line #	Statement	Δ	τ_Δ	Rule	pc
	start	$\{\}$	$\{\}$		1
1	x := 2*get_input(·)	$\{x \rightarrow 40\}$	$\{x \rightarrow \mathbf{T}\}$	T-ASSIGN	2
2	y := 5 + x	$\{x \rightarrow 40, y \rightarrow 45\}$	$\{x \rightarrow \mathbf{T}, y \rightarrow \mathbf{T}\}$	T-ASSIGN	3
3	goto y	$\{x \rightarrow 40, y \rightarrow 45\}$	$\{x \rightarrow \mathbf{T}, y \rightarrow \mathbf{T}\}$	T-GOTO	*error*

Example


```
1  x := 2*get_input(·)
2  y := 5 + x
3  goto y
```

Line #	Statement	Δ	τ_Δ	Rule	*pc*
	start	$\{\}$	$\{\}$		1
1	x := 2*get_input(·)	$\{x \rightarrow 40\}$	$\{x \rightarrow \mathbf{T}\}$	T-ASSIGN	2
2	y := 5 + x	$\{x \rightarrow 40, y \rightarrow 45\}$	$\{x \rightarrow \mathbf{T}, y \rightarrow \mathbf{T}\}$	T-ASSIGN	3
3	goto y	$\{x \rightarrow 40, y \rightarrow 45\}$	$\{x \rightarrow \mathbf{T}, y \rightarrow \mathbf{T}\}$	T-GOTO	*error*

Note: Tainted jump policy does not consider whether memory addresses are tainted. Thus, it may miss some attacks!

Problems I

- Two types of errors can occur
  - *Overtainting* means that a value is marked as tainted when it is not derived from a taint source
  - *Undertainting* means that the information flow from a source to a sink is missed
- A dynamic taint analysis system is *precise* if no undertainting or overtainting occurs

Problems II

- Taint problems can often occur in practice
  - If an array index is tainted, what else should be tainted?
  - How to handle pointers?
  - All kinds of instructions need to be included in practice, for example also FPU or SSE instructions
- Implementing a good taint analysis system is hard

Summary

- Today's lecture was rather theoretic...
  - but we need this formalism to be able to be precise
- SimpIL as an example of an intermediate language
  - Enables us to study programs
- Taint analysis enables us to study how information flows within a given program

Forward Symbolic Execution

Motivation

- If we execute a program we can only analyze one specific run of the program
  - We see what the program does given specific input
  - But what other paths can be taken?
- Can we somehow analyze more paths / examine what a program does given another input?
- Can we maybe even analyze what a program does given any input?

Motivation


```
1  x := 2*get_input(·)
2  if x−5 == 14 then goto 3 else goto 4
3  // catastrophic failure
4  // normal behavior
```

Motivation


```
1  x := 2*get_input(·)
2  if x−5 == 14 then goto 3 else goto 4
3  // catastrophic failure
4  // normal behavior
```

Only 1 out of 2^{32} possible inputs will trigger the failure

We generalize get_input() and treat it is symbol instead concrete value

Motivation

concrete / arithmetic

arithmetic execution

Symbolic (algebraic) input covers whole classes of arithmetic cases

Motivation

concrete / arithmetic ⟷ algebra / symbolic

↕

arithmetic execution

Symbolic (algebraic) input covers whole classes of arithmetic cases

Motivation

concrete / arithmetic

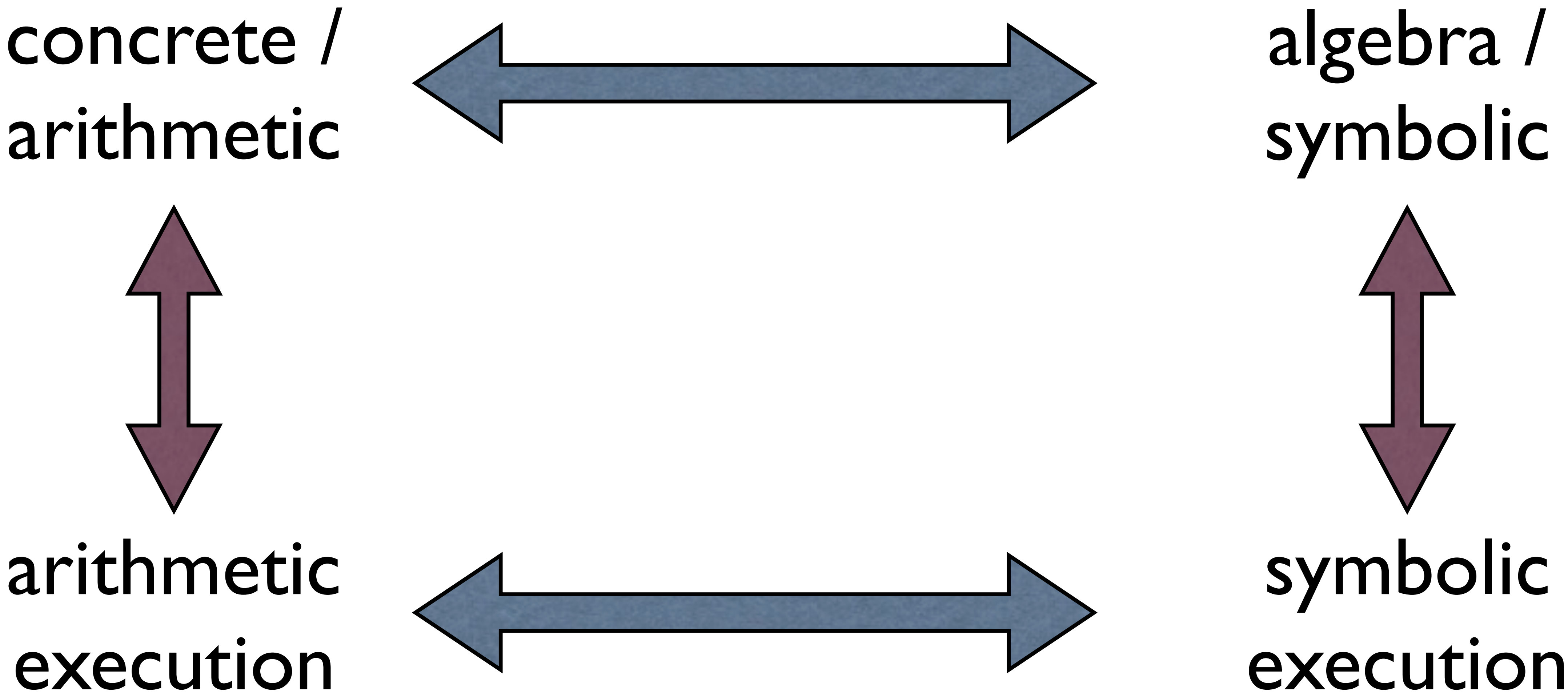
algebra / symbolic

arithmetic execution

symbolic execution

Symbolic (algebraic) input covers whole classes of arithmetic cases

Motivation



Symbolic (algebraic) input covers whole classes of arithmetic cases

Motivation

- When get_input() is evaluated symbolically, it returns a *symbol* instead of a concrete value
  - When a new symbol is first returned, there are no constraints on its value: it represents *any possible* value
  - Expressions involving symbols cannot be fully evaluated to a concrete value (e.g., `s + 5` cannot be reduced further)
- SimpIL must be extended again to deal with this

Changes to SimpIL

value v	$::=$	32-bit unsigned integer $\mid$ *exp*
Π	$::=$	Contains the current constraints on symbolic variables due to path choices

- Value can be integer or expression
- There can be constraints (i.e., *path conditions*) on symbolic variables
  - Branches constrain the values of symbolic variables to the set of values that would execute the path

Changes to SimpIL

$$\frac{v \text{ is a fresh symbol}}{\mu, \Delta \vdash \text{get\_input}(\cdot) \Downarrow v} \text{ S-INPUT}$$

$$\frac{\mu, \Delta \vdash e \Downarrow e' \quad \Pi' = \Pi \wedge e' \quad \iota = \Sigma[pc + 1]}{\Pi, \Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \Pi', \Sigma, \mu, \Delta, pc + 1, \iota} \text{ S-ASSERT}$$

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

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

Example


```
1  x := 2*get_input(·)
2  if x-5 == 14 then goto 3 else goto 4
3  // catastrophic failure
4  // normal behavior
```

- Constraints are added to *true* and *false* branch
- After an assert statement, the values of symbols must be constrained such that they satisfy the asserted expression

Example


```
1 x := 2*get_input(·)
2 if x-5 == 14 then goto 3 else goto 4
3 // catastrophic failure
4 // normal behavior
```

Statement	Δ	Π	Rule	*pc*
start	$\{\}$	$true$		1
x := 2*get_input(·)	$\{x \to 2 * s\}$	$true$	S-ASSIGN	2
if x-5 == 14 goto 3 else goto 4	$\{x \to 2 * s\}$	$[(2 * s) - 5 == 14]$	S-TCOND	3
if x-5 == 14 goto 3 else goto 4	$\{x \to 2 * s\}$	$\neg[(2 * s) - 5 == 14]$	S-FCOND	4

When forward symbolic execution reaches a branch, as in Line 2, it must choose which path to take

Forward Symbolic Execution

- General procedure: take the operational semantics of the language and change the definition of a value to include symbolic expressions. However:
  - *Symbolic memory*: What should we do when the analysis uses the μ context with a symbolic index?
  - *System calls*: How should our analysis deal with external interfaces such as system calls?
  - *Path selection*: How should we decide which branches to take?

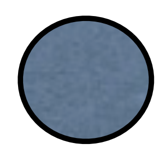
Symbolic Memory Addresses

- Load and Store rules need to evaluate expressions
  - When we load from a symbolic expression, a sound strategy is to consider it a load from any possible satisfying assignment for the expression
  - Similar for Store
  - Example are table lookups
- *Alias problem:*

```
store(addr1, v)
z = load(addr2)
```

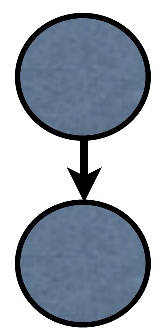
Path Selection

- When forward symbolic execution encounters a branch, it must decide which branch to follow first



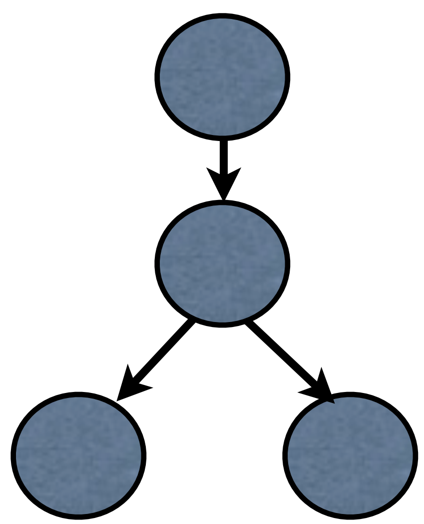
Path Selection

- When forward symbolic execution encounters a branch, it must decide which branch to follow first



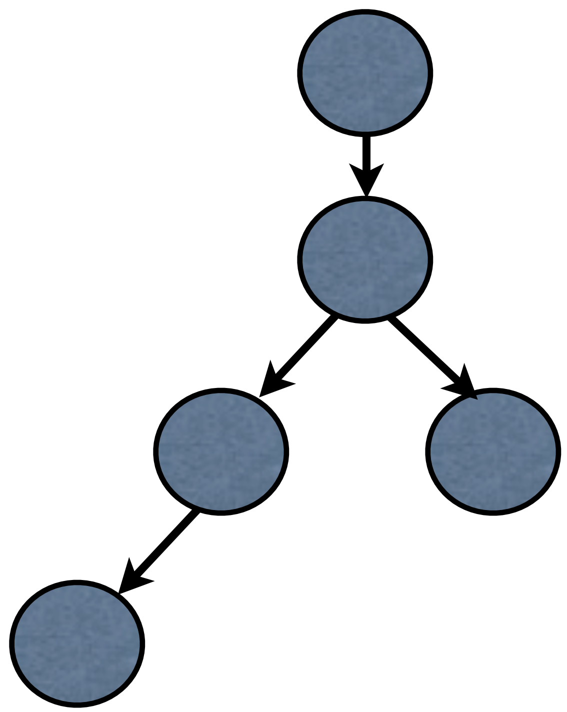
Path Selection

- When forward symbolic execution encounters a branch, it must decide which branch to follow first



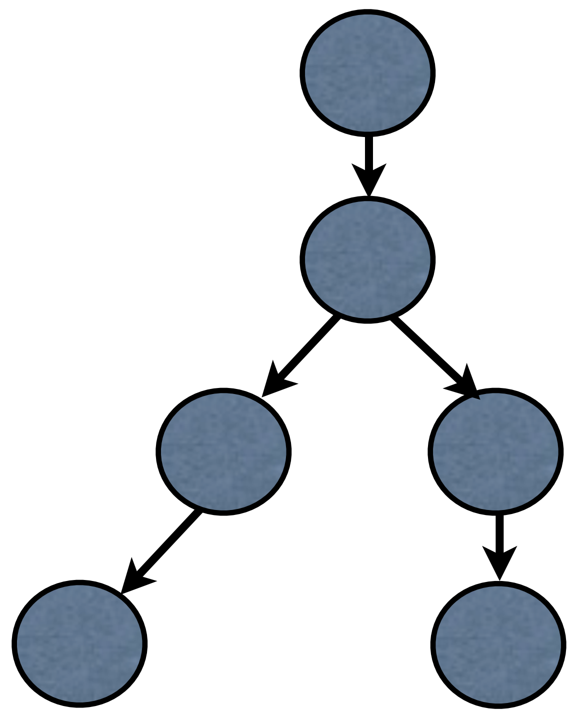
Path Selection

- When forward symbolic execution encounters a branch, it must decide which branch to follow first



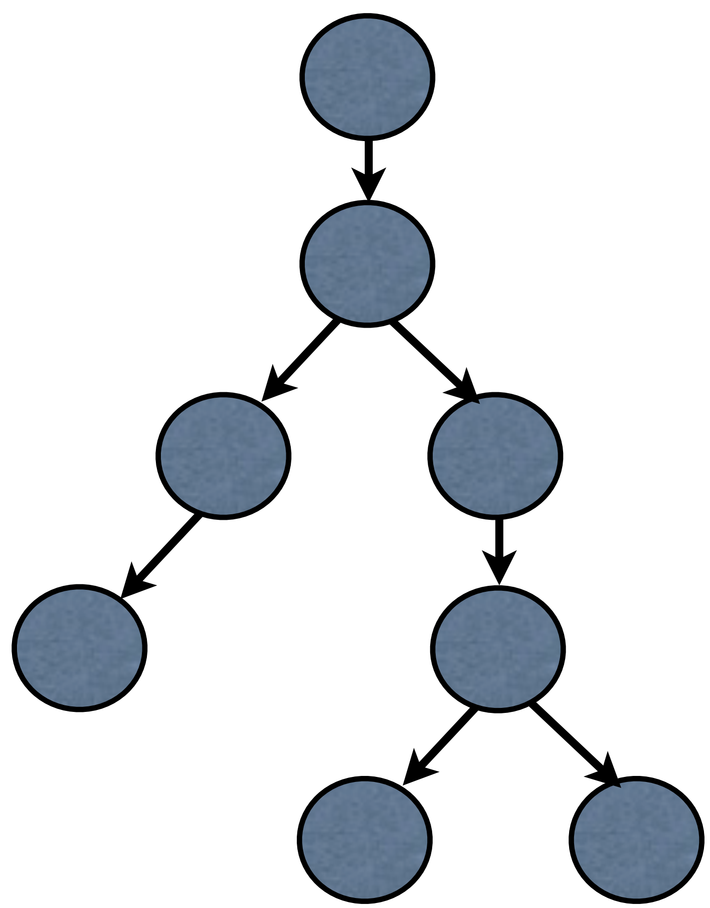
Path Selection

- When forward symbolic execution encounters a branch, it must decide which branch to follow first



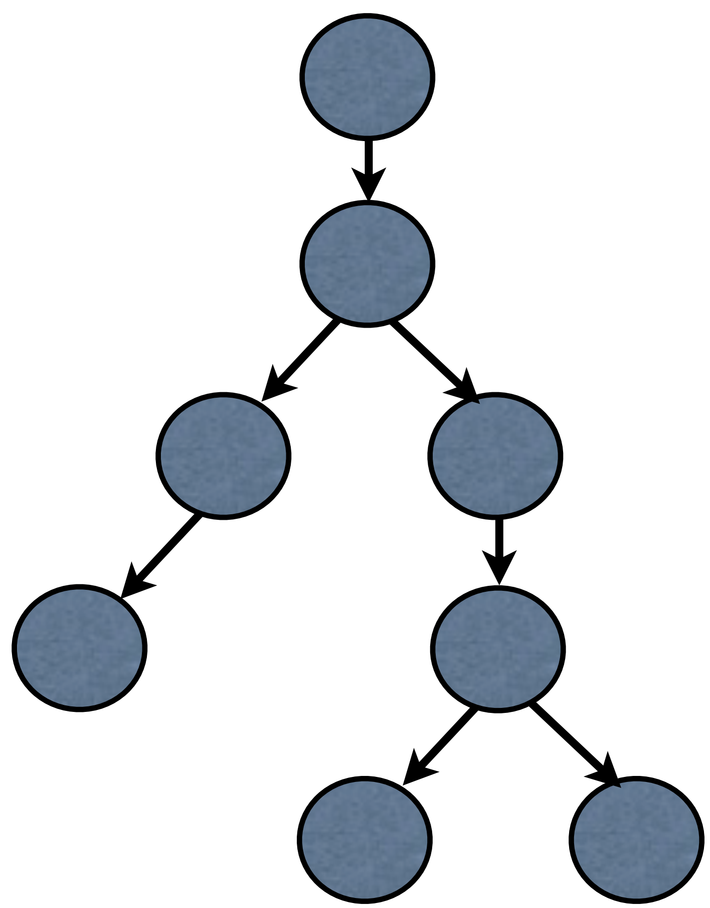
Path Selection

- When forward symbolic execution encounters a branch, it must decide which branch to follow first



Path Selection

- When forward symbolic execution encounters a branch, it must decide which branch to follow first



Note: loops with symbolic conditions may never terminate

Path Selection

- Possible strategies
  - *Depth-first search*: maximum depth should be specified to prevent endless runs
  - *Concolic testing*: use concrete execution to produce trace; follow this path (can be used to generate inputs for specific paths)
  - *Random paths*
  - *Heuristics*

Symbolic Jumps

- Jump target of Goto statement may be an expression instead of a concrete location
  - For example a jump table / switch statement
- Possible strategies
  - Concolic execution
  - SMT solver
  - Static analysis

Discussion

- Straightforward implementation of forward symbolic execution will lead to
  - a running time exponential in the number of program branches
  - an exponential number of formulas/conditions
  - an exponentially-sized formula per branch
- There is a lot of space for improvements!

Summary

- Symbolic execution enables us to analyze paths in an abstract manner
- Systems such as EXE or KLEE show the practical use of such techniques
  - Automatically generate input that leads to crashes
  - Can also be used to computer input that leads to specific paths in a program
  - Powerful technique, but requires some time

Questions?

Systems Security
Ruhr-University Bochum

Contact:

Prof. Thorsten Holz

thorsten.holz@rub.de
@thorstenholz on Twitter

More information:

http://syssec.rub.de

http://moodle.rub.de

Sources

- Paper by Schwartz et al.: "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)" - IEEE S&P'10
  - http://www.ece.cmu.edu/~ejschwar/papers/oakland10.pdf