TU WIEN S&P Security & Privacy CYSEC CYBERSECURITYCENTER

Program Analysis

Introduction to Security (192.019)

Matteo **Maffei**

Security & **Privacy** Research Unit (192-06)
https://**secpriv.wien**

Program Analysis at Scale

- All large IT companies integrate program analysis in their development workflow
- Each line of code is verified: software developers must learn how these tools work!

Program Analysis

All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)

Edward J. Schwartz, Thanassis Avgerinos, David Brumley
Carnegie Mellon University
Pittsburgh, PA
{edmcman, thanassis, dbrumley}@cmu.edu

Program Analysis

Goal: Analyse a program to check certain security properties

- **Security is typically undecidable**: e.g., due to infinite possible inputs
- Hence any program analysis technique has to give up at least one of the following properties
  - **Completeness:** the analysis returns yes if the program is secure (**no false positives**)
  - **Soundness:** the analysis never returns yes if the program is insecure (**no false negatives**)
  - **Termination**: the analysis always terminates

Program Analysis

- We typically distinguish between two analysis techniques

Good to find security vulnerabilities

- **Dynamic Analysis**: monitors certain program runs (i.e., for certain inputs). Typically,
  - **Precise:** can reason about concrete values and concrete program runs
  - **Unsound:** it cannot certify all program runs
  - **Terminating**: it handles a finite number of program runs

Good to obtain security proofs

- **Static analysis**: characterizes all program runs (i.e., for all possible inputs). Typically,
  - **Overapproximating**: values are overapproximated and the analysis might not terminate
  - **Sound**: it can certify all program runs
  - **Terminating or not** (normally, more termination, less precision)

Program Analysis

- We focus on two highly popular analysis techniques

Taint Analysis

Which computations are affected by predefined sources (e.g., inputs) controlled by the attacker, also called tainted sources?

Forward Symbolic Execution

Which inputs lead to a certain program point?

- Both can be designed as static or dynamic analysis techniques, today we will reason about *dynamic* taint analysis and *static* forward symbolic execution

Use Cases

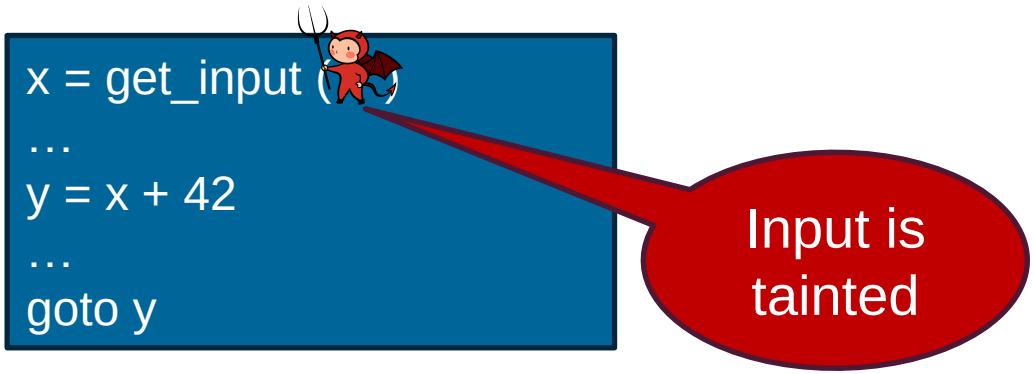
- Unknown vulnerability detection (e.g., code injection): dynamic taint analysis
- Automatic input filter generation (e.g., remove exploits from input stream): forward symbolic execution
- Malware analysis (e.g., information flow in a malware binary, explore trigger-based behaviour, ...): both techniques
- Test case generation (generate inputs for test programs, or generate inputs that can cause the program to behave differently): both techniques
- Web security (e.g., which inputs lead to XSS or SQL injection): symbolic forward execution

A Bird's Eye View on Taint Analysis

Example

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

Example

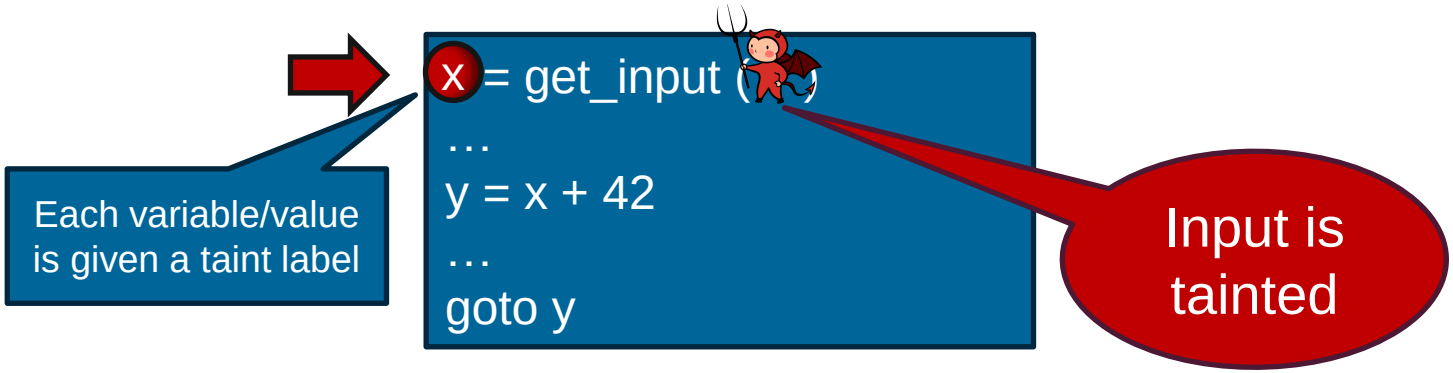


Taint Introduction

For simplicity, we denote and operate on taint labels as logical labels (True and False)

Tainted(T)

Untainted(F)

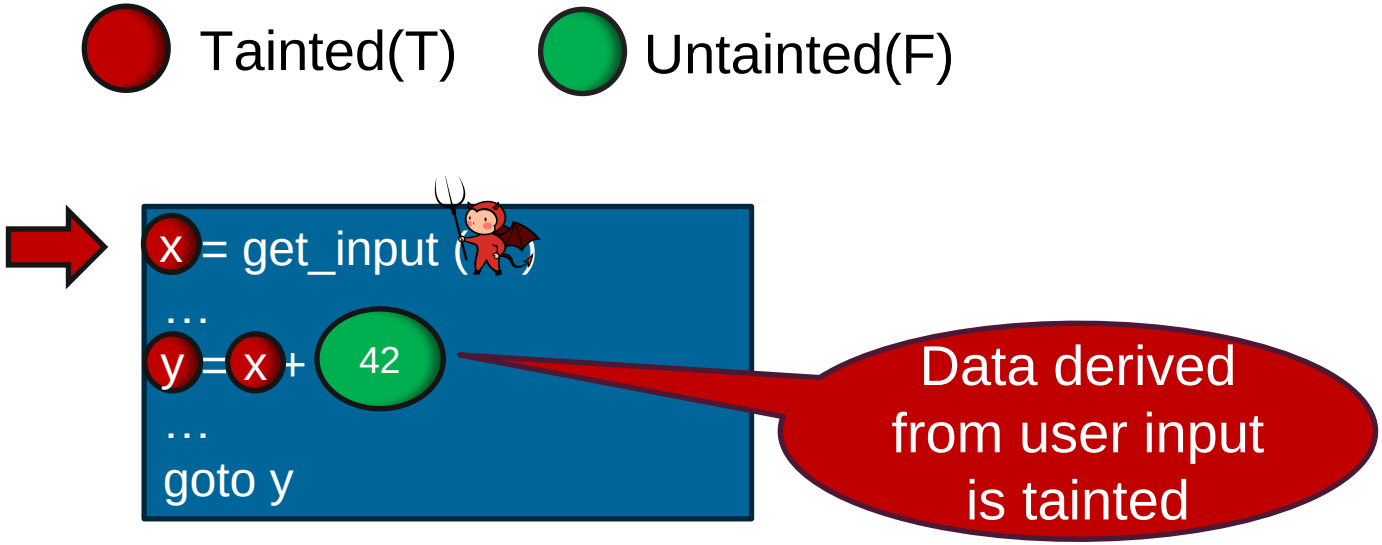


Taint Introduction

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

Var	Val	Taint (T | F)
x	7	T

Taint Propagation

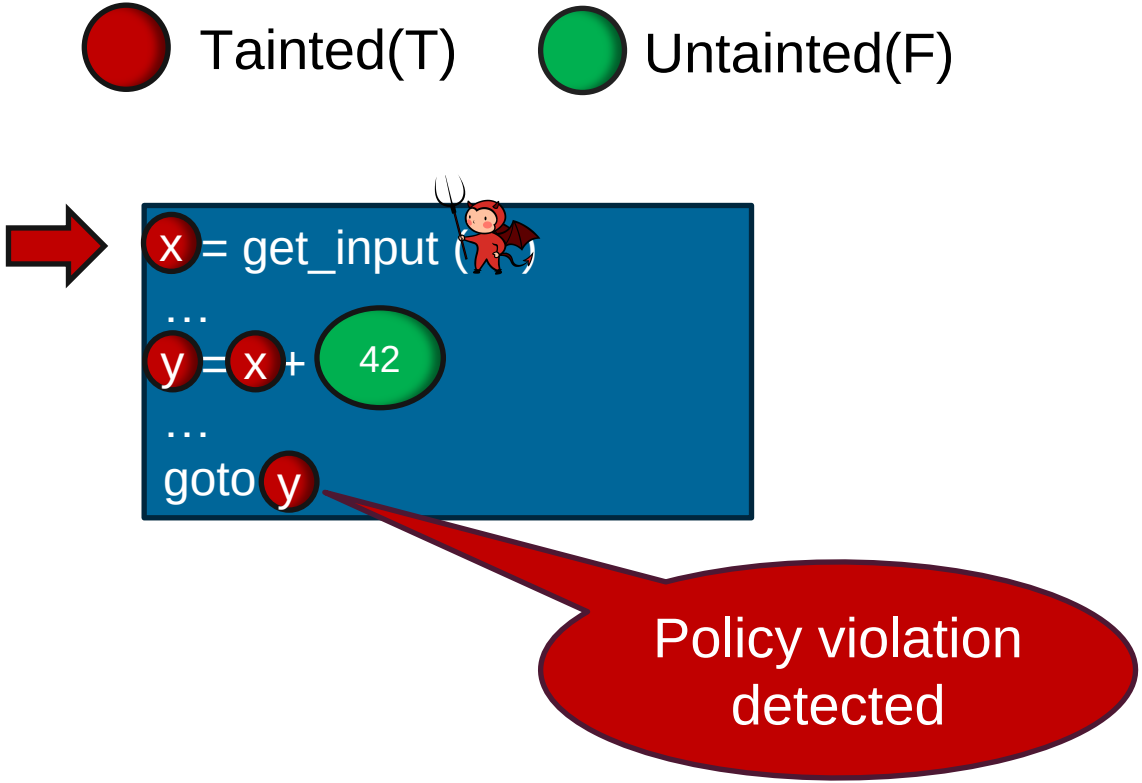


Taint Propagation

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

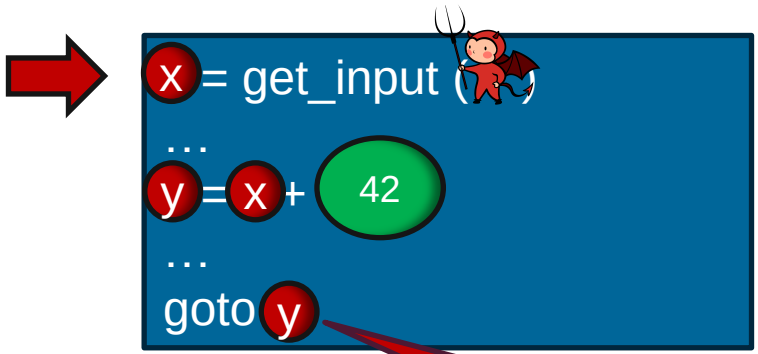
Var	Val	Taint (T | F)
x	7	T
y	49	T

Taint Propagation



So what?

Exploit Detection




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

Tainted return address

Taint Policy

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

(Must be true to execute)

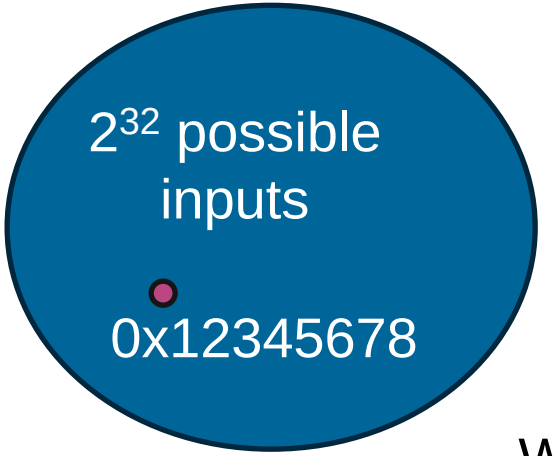
Var	Val	Taint (T | F)
x	7	T
y	49	T

A Bird's Eye View on Forward Symbolic Execution

Example

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

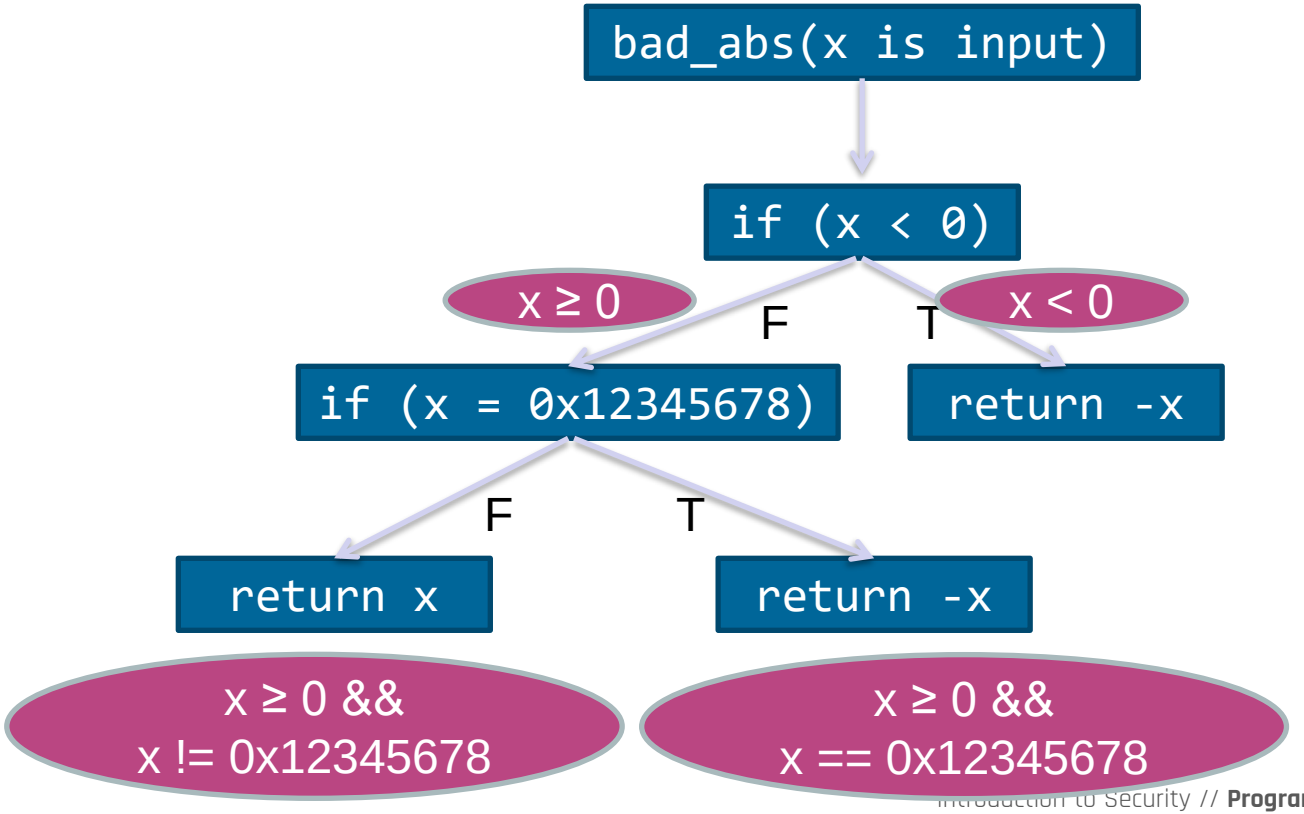
Example




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

What input will execute this line of code?

Working


bad_abs(x is input)
if (x < 0)
x ≥ 0
F
T
x < 0
if (x = 0x12345678)
return -x
F
T
return x
return -x
x ≥ 0 &&
x != 0x12345678
x ≥ 0 &&
x == 0x12345678

First Step: A Simple Intermediate Language (SIMPIL)

Syntax

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

Semantics (Notations)

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

Semantics (Evaluation Rules)

- Evaluation rules take the following form

$$\mu, \Delta \vdash e \Downarrow v$$

- They are read as expression e evaluates to v under memory and variable mappings μ and Δ, respectively

Semantics (Reduction Rules)

- Semantic rules take the following form

$$\frac{\text{computation}}{\langle \text{current state} \rangle, \text{stmt} \rightsquigarrow \langle \text{end state} \rangle, \text{stmt'}}$$

- They are read bottom up, left to right
  - Pattern-match to find the applicable rule
  - Apply computations
    - If they succeed, transition to end state
    - Otherwise, abort abnormally

Semantics (Reduction Rules)

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

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad v' = \Diamond_u v}{\mu, \Delta \vdash \Diamond_u e \Downarrow v'}\ \textsc{Unop} \qquad \frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad v' = v_1 \Diamond_b v_2}{\mu, \Delta \vdash e_1 \Diamond_b e_2 \Downarrow v'}\ \textsc{Binop} \qquad \frac{}{\mu, \Delta \vdash v \Downarrow v}\ \textsc{Const}$$

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

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

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

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

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

Example

- The program

```
1  x := 2 * get_input(·)
```

evaluates as follows

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

Dynamic Taint Analysis

Dynamic Taint Analysis

- **Goal**: track information flow between sources and sinks
- **Method:** assign and propagate a label for each value:
  - those whose computation depend on data derived from a taint source are labeled tainted (denoted $\mathbf{T}$)
  - any other value is considered untainted (denoted $\mathbf{F}$)
- A taint policy P determines how taint flows as a program executes, what sorts of operations introduce new taint, and what checks are performed on tainted values
  - overtainting (false positives, i.e., secure executions are marked as insecure): values are considered tainted although they are not
  - undertainting (false negatives, i.e., attacks are not detected): values are considered untainted although they are not

Notations

taint t	::=	$\mathbf{T} \mid \mathbf{F}$
value	::=	$\langle v, t \rangle$
τ_Δ	::=	Maps variables to taint status
τ_μ	::=	Maps addresses to taint status

- We mark both values, variables, and addresses with a taint label

Tainted Jump Policy

- This policy below is meant to prevent jumps to tainted addresses (control flow hijacking attacks)...

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

- let's see how!

Taint Rules

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

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

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \Diamond_u e \Downarrow \langle \Diamond_u v, P_{\textbf{unop}}(t)\rangle}\ \text{T-UNOP}$$

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

Taint Rules (cont'd)

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

$$\frac{\begin{array}{c}\iota = \Sigma[pc+1] \quad P_{\mathbf{memcheck}}(t_1, t_2) = \mathbf{T} \\ \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1 \rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2 \rangle \quad \mu' = \mu[v_1 \leftarrow v_2] \quad \tau'_\mu = \tau_\mu[v_1 \leftarrow P_{\mathbf{mem}}(t_1, t_2)]\end{array}}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{store}(e_1,\ e_2) \rightsquigarrow \tau'_\mu, \tau_\Delta, \Sigma, \mu', \Delta, pc+1, \iota} \text{ T-STORE}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t \rangle \quad \iota = \Sigma[pc+1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc+1, \iota} \text{ T-ASSERT}$$

Taint Rules (cont'd)

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

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

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

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(·)	$\{x \to 40\}$	$\{x \to \mathbf{T}\}$	T-ASSIGN	2
2	y := 5 + x	$\{x \to 40, y \to 45\}$	$\{x \to \mathbf{T}, y \to \mathbf{T}\}$	T-ASSIGN	3
3	goto y	$\{x \to 40, y \to 45\}$	$\{x \to \mathbf{T}, y \to \mathbf{T}\}$	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(·)	$\{x \to 40\}$	$\{x \to \mathbf{T}\}$	T-ASSIGN	2
2	y := 5 + x	$\{x \to 40, y \to 45\}$	$\{x \to \mathbf{T}, y \to \mathbf{T}\}$	T-ASSIGN	3
3	goto y	$\{x \to 40, y \to 45\}$	$\{x \to \mathbf{T}, y \to \mathbf{T}\}$	T-GOTO	*error*

Example of over- and under-tainting

- The following program is accepted, assuming the memory cell is untainted

```
1  x := get_input(·)
2  y := load(z + x)
3  goto y
```

- But the attacker can basically pick up any value and jump there, violating the intended control flow (undertainting)
- A stricter policy could stop the execution if address or memory cell are tainted

$$\text{Tainted Addresses} \quad \Big| \quad P_{\mathbf{mem}}(t_a, t_v) \equiv t_a \vee t_v$$

- Legitimate programs (e.g., tcpdump) could however be rejected since they work by legit look-up tables with user input (overtainting)

Time of Detection vs Time of Attack

- Dynamic taint analysis raises an alert when tainted values are used in an unsafe way, which could however be too late!
- For instance, consider a typical return address overwrite exploit
  - The exploit overwrites the return address so that this points to attacker's shellcode
- Here the dynamic taint analysis would raise an alarm at the time jumping, but not at the time of overwriting the address
  - The exploit will not be reported until the jump, so any calls done by the vulnerable function will be executed and if these calls have side-effects (file manipulation or networking operations), these will persist after the program is aborted

Forward Symbolic Execution

Forward Symbolic Execution

- **Goal**: determine which inputs lead to a certain program point
- **Method**: build a logical formula that captures program executions
  - Values are symbolic, as opposed to concrete
- **Advantage**: can reason about multiple inputs at one time

Example

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

Initially, we don't know anything about the input, so we use a universally quantified variable

Pass this formula to an SMT prover

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

Semantics for Symbolic Forward Execution

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

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

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

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

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

Challenge 1: Symbolic Addresses

- How does the memory look like after the end of the program?

1. addr1 := **get_input**(·)
2. **store**(addr1 , v)

- Sound strategy: any memory address may contain v, similarly for loads
- Aliasing may also be a problem:

1. **store**(addr1 , v)
2. z = **load**(addr2)

z may contain v or not, depending on whether or not addr1 and addr2 are aliased

- One can let an SMT solver reason about memory, naming each step update

$$mem_1 = mem_0[addr_1 \rightarrow v] \wedge z = mem_1[addr_2]$$

Challenge 2: Symbolic Jumps

- Where does the following program jump to?

1. **jump**(e)

- One can let an SMT solver reason about jumpstoo, e.g., querying $\Pi \wedge e$ where Π is the path predicate. The SMT solver will give us a satisying answer n, which is a possible jump target
- If we want more jump targers, we can query for $\Pi \wedge e \wedge \neg n$ and so on

Challenge 3: Loops

- What happens in case of loops?
- The symbolic execution might not terminate…
- Typically, an upper bound on the number of loop iterations is fixed
  - enforces termination but it is unsound