

Dynamic Taint Analysis

Where we are
now?

Security in Programming Languages (Recap)

Run-time Support

- Stack Canaries
- Layout randomization
- Shadow stack
- NX memory
- Enclaves
- FAT Pointer & Data Descriptor
- Web Assembly
- IRM and Code Instrumentation

Static Analysis

- Later

Application Level Reqs: Fine grained control

- Modern apps make security decisions with respect to the abstraction provided by the application level
 - E-Commerce: no goods until payments
 - IoT Apps: avoid data leaks
 - ...



Taking decisions in terms of data:]

Information Security

Valuable (secret) information should
not be leaked by computation

Only authorized entities can read
from a file (access control approach)

But what happen to information
when the entity is authorized?

Idea: what happens to data when auth. is granted?

INFORMATION SECURITY

Typical approach is:

Software systems handles data with different levels of confidentiality and integrity.

Two confidentiality levels,
secret (restricted)
public (unrestricted)

Two integrity levels,
reliable (coming from trusted sources)
dubious (coming from untrusted sources)

INFORMATION FLOW: *how the data flow among these levels*

Control not only accesses to resources (data at rest)
but also information flow between resources. (data in transit)

Confidentiality policy: information can flow only from less secret to more secret.

"flow"
public → public public → secret
secret → secret secret ↗ public

Integrity policy: information can flow only from more reliable to less reliable.

reliable → reliable reliable → dubious
dubious → dubious dubious ↗ reliable

↑ data is assumed to be reliable

A (STRONGER) MOTIVATION

- Meltdown, Spectre Attacks
- Isolation is a cornerstone of our Trusted Execution Environment (TEE)
 - Modern processors: isolation between the kernel and user processes is realized by a supervisor bit of the processor that defines whether a memory page of the kernel can be accessed or not.
- Meltdown allows overcoming memory isolation by covert channels caused by *out-of-order execution*.
 - Speculative executions

What is speculative execution?

- Speculative execution is an optimization technique used in modern processors. It allows the CPU to guess and execute future instructions before knowing whether they are needed.
 - If the guess is correct, the execution is sped up.
 - If incorrect, the results are discarded, and the correct path is followed.
- This technique is widely implemented in out-of-order execution architectures, where the CPU executes instructions as resources become available rather than strictly following program order.

How speculative execution works

- **Branch Prediction:**
The processor predicts the outcome of a conditional branch (e.g., if-else) before the condition is fully evaluated.
- **Execution of Instructions Speculatively:**
If the processor predicts that a certain branch is likely to execute, it starts computing instructions in that path.
- **Commit or Rollback:**
If the prediction is correct, the results are committed to memory. If incorrect, the processor discards the speculative results and rolls back to the correct path.

Meltdown Attacks

- Meltdown is a hardware vulnerability that exploits speculative execution to leak sensitive data from protected memory regions, such as kernel memory.
- The attack was discovered in 2018 and affects many processors, especially Intel CPUs.

How Meltdown Works

- Meltdown allows an attacker to **read privileged memory** (e.g., kernel space) that should be inaccessible to user-level applications.
- It exploits the fact that speculative execution can access **memory that is normally restricted**.

A software view

```
try { i = val;  
      V = PM[i];  
      :  
}  
catch (IllegalReadException)  
{for j in 0 to Max  
  { read PM[j]  
  }  
}
```

A software view

```
try { i = val;      // Attempt to read  
    v = PM[i];    // private data (*)  
    :  
}  
catch (IllegalReadException) {  
    for j in 0 to Max  
        { read PM[j]  
    }  
}
```

Trigger Speculative Execution on Unauthorized Memory

The attacker forces the CPU to execute an instruction that accesses privileged memory (*). Normally, this would cause raising a protection exception, preventing attacker from reading the private data.

A software view

```
try { i = val;      // Attempt to read
      V = PM[i];    // private data (*)
      :              // (speculative execution)
}
catch (IllegalReadException) {
  for j in 0 to Max
    { read PM[j]
  }
}
```

Speculative Execution Reads Restricted Data

Due to **out-of-order execution**, the CPU **speculatively** fetches the restricted data before checking access permissions. This means the data is temporarily loaded into the **CPU cache**.

A software view

```
try { i = val;      // Attempt to read
      V = PM[i];    // private data (*)
      :
      // (speculative execution)
}
catch (IllegalReadException) {
  for j in 0 to Max
    { read PM[j]
  }
}
```

Side-Channel Attack via Cache

While the CPU eventually realizes the access was illegal and raises the security exception, the cached data remains accessible.

The attacker then uses cache (side-channel attack) to read the leaked data.

Summary

- Meltdown highlights a fundamental weakness in speculative execution: security checks occur too late. While speculative execution boosts performance, it inadvertently exposes memory data that attackers can exploit via side-channel attacks.
- The introduction of suitable hardware fixes helps mitigate the risk, but the trade-off between performance and security remains a challenge for modern processor design.

Meltdown attack

- Meltdown by Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, 2018

Software Mitigation?

- The Meltdown attack exposes vulnerabilities in speculative execution, allowing unauthorized access to privileged memory.
 - Since Meltdown does not rely on traditional software vulnerabilities (e.g., buffer overflows), it is not easily detected by standard security tools.
 - The attack operates at the microarchitectural level, making it invisible to software-level security mechanisms.

Tracking Data Flow to Mitigate Meltdown

↑ approach is track how data flows in execution, and resulted if data under control of the attacker (tainted) is used to make decision, execution is stopped.

Dynamic Taint Analysis (DTA)

Concept: DTA tracks the flow of **tainted data** (e.g., user input) and ensures it does not interact with **sensitive memory**.

Mitigation: If speculative execution tries to **private memory into cache**, **taint analysis** can detect the violation and trigger a response.

What is Dynamic Taint Analysis?

- **Dynamic Taint Analysis (DTA)** is a security technique used in programming languages and runtime environments to track the flow of potentially untrusted data (tainted data) through a program.
- It helps identify vulnerabilities such as **injection attacks, buffer overflows, and unauthorized data leakage**.
- This technique is widely used for detecting and preventing security exploits in languages like **C, C++, Python, Java, and Rust**.

SQL injection is an example where source of query is under control of attacker.

Taint Analysis

- A data-flow analysis mechanism.
 - Purpose: to track the propagations of data.
 - Rule: the variables whose values are computed by exploiting tainted data are also tainted.
- Components:
 - Sources: Taint Seed
 - Untrusted input
 - Sensitive data
 - Propagation: Taint Policy
 - Sinks: Taint Checking/Assert



We have several sources (tainted and unknown) and a lot of mixing. You then have a sink in which data flows to. In the sink you expect only untainted data, otherwise, we stop. So you identify SINK and SOURCES. If you identify a TAINTED PATH, that's bad.

Taint Propagation

- DTA marks specific data (e.g., user input, network data, file input) as tainted.
- When tainted data flows through computations (e.g., string concatenations, arithmetic operations), the taint is propagated to derived values.
- If untainted data combines with tainted data, the result typically remains tainted.



Taint Tracking at Runtime

- Dynamic taint analysis operates at runtime: It tracks taint information across memory, registers, and variables as the program executes. Works at very low level
- This approach ensures precise detection of vulnerabilities but can incur performance overhead.

Taint analysis is an example of an integrity policy.

Policy Enforcement

- The analysis enforces **security policies** to prevent unintended behavior
 - Prevent execution of tainted data in sensitive functions (e.g., SQL queries, shell commands).
 - Block tainted data from reaching unauthorized memory locations.
 - Restrict transmission of tainted data over the network without sanitization.

Fine Grain Taint Tracking

- What do we do at level of phys. language? ①. But taint analysis can go even deeper
- Some DTA approaches track taint information at different granularities:
 - **Variable-Level:** Taint is assigned to whole variables. ①
 - **Byte-Level:** Tracks taint at a finer granularity within buffers.
 - **Bit-Level:** Provides the highest precision, though at a higher performance cost.

Taint Analysis for Security

- **Identify sources of taint**
 - Untrusted input is controlled by the attacker
 - Threat model
- **Identify Propagation of taint**
 - Tracks flow that manipulates data in order to determine whether the result is tainted or not
- *If the analysis finds that tainted data could be used where untainted data is expected, there could be a potential security vulnerability*



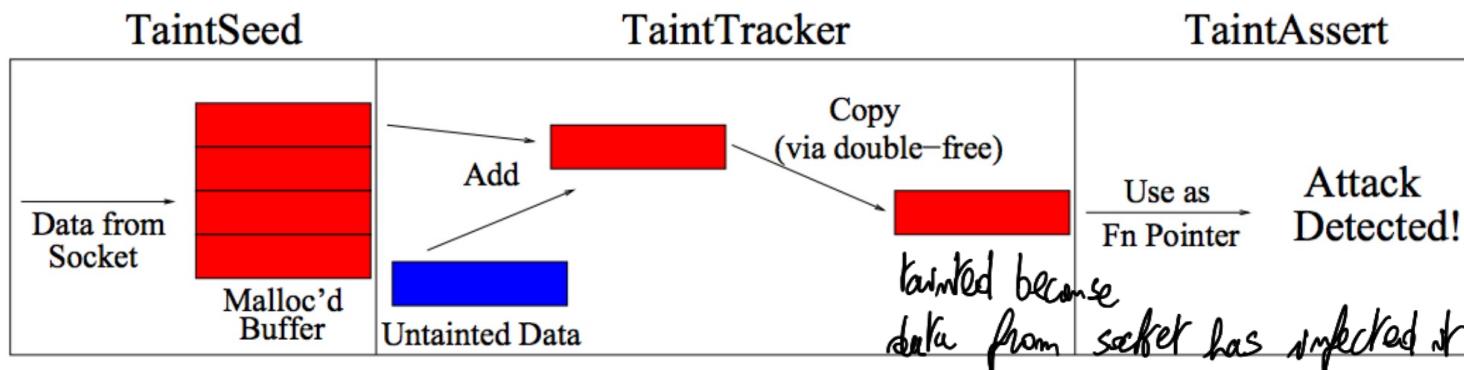
How can DTA be used?

- Tainted data occur in
 - format string attacks.
 - construction of SQL queries, which could be used in SQL injection attacks.
 - Network applications where the attacker controls the communication channels
 - :



Application: Sanity Check

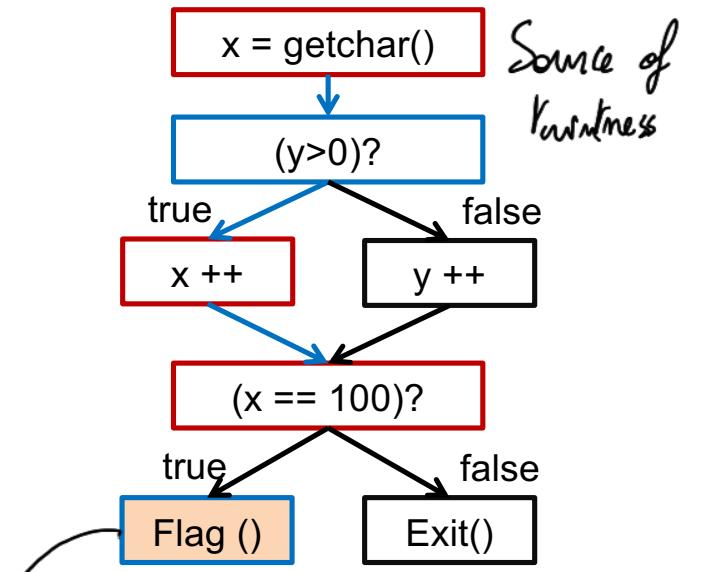
- Check if untrusted data are used dangerously.



James Newsome and Dawn Song, Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software, 2004

Application: Symbolic Execution

Symbolic execution



- Track the propagation of Variable
- Extract the constraint for executing a particular path.

Find the value of x for executing flag().

only requires liveness info

Sources of data

- Sources of information
- Untrusted – tainted – public
 - Possibly controlled by the attacker
- Trusted – untainted -- secret



MORE INTUITION



tainted



untainted

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

Input is tainted

μ	
Var	Val
x	7

τ	
Var	Tainted?
x	T

TaintSeed:

Input $t = \text{IsUntrusted}(\text{src})$
 $\text{get_input}(\text{src}) \downarrow t$

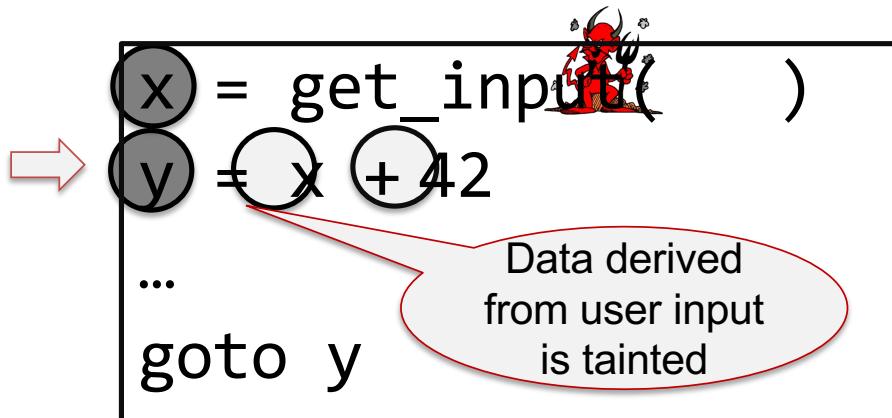
*If source is untrusted
result is untrusted*



tainted



untainted

 μ

Var	Val
x	7
y	49

 τ

Var	Tainted?
x	T
y	T

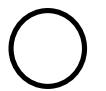
TaintTracker

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

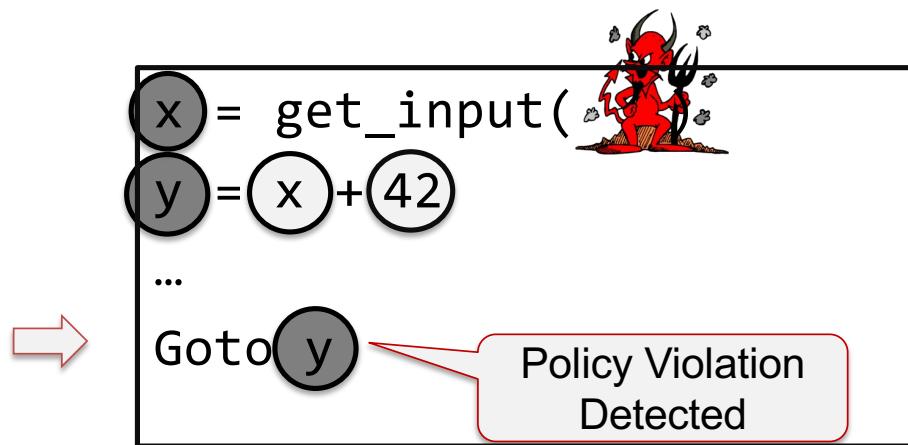
↓ At the level of taintness we take the or of taintness



tainted



untainted



μ	
Var	Val
x	7
y	49

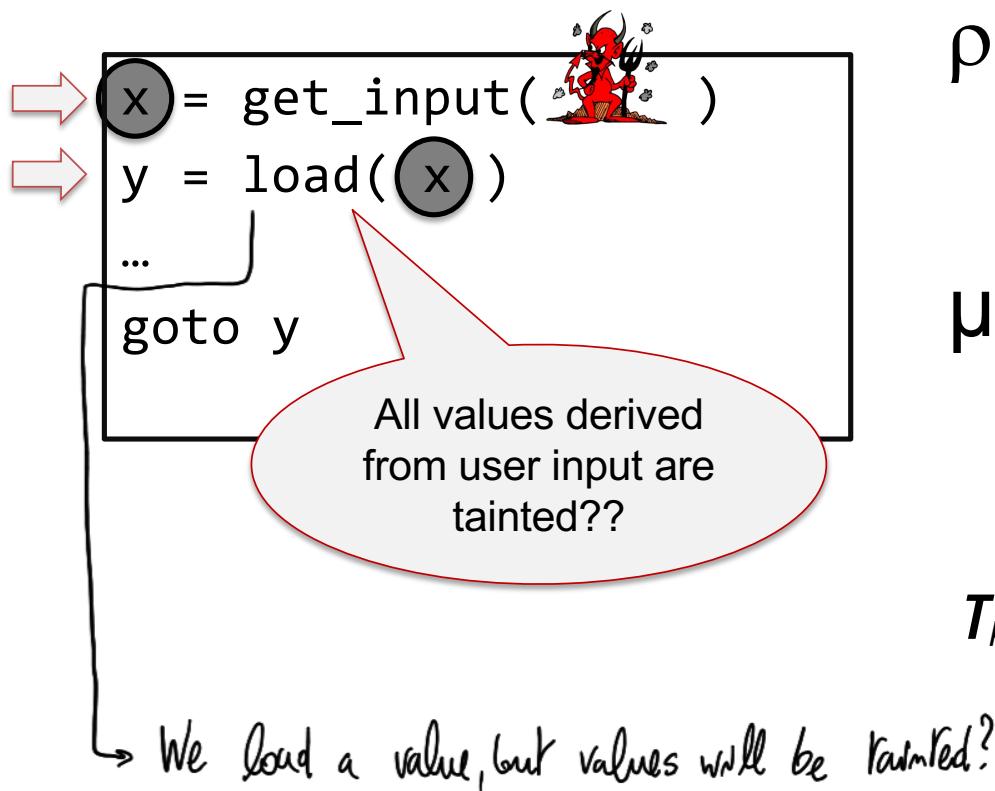
τ	
Var	Tainted?
x	T
y	T

TaintAssert:

$P_{\text{goto}(ta)} = \neg \text{tainted}$
(Must be true to execute)

Why? Jump/Return address has been overwritten.

Problem: Memory Addresses



	Var	Val
ρ	x	7
	Addr	Val
μ	7	42

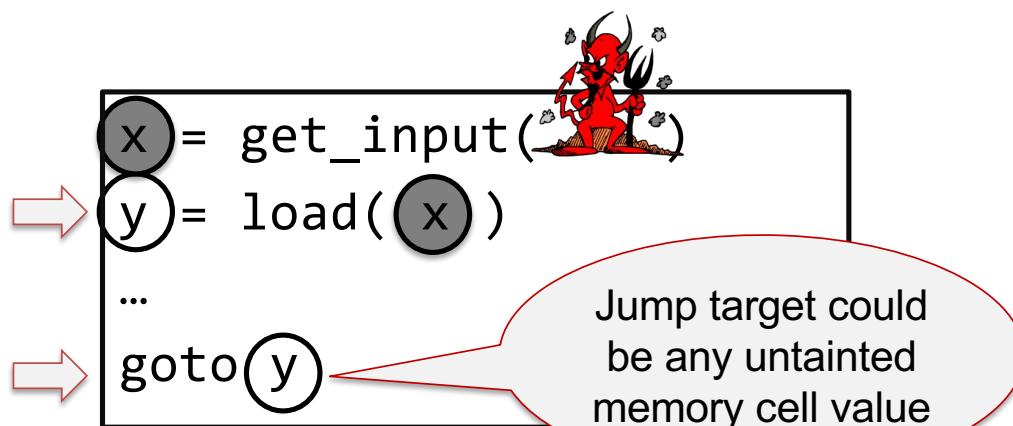
	Addr	Tainted?
T_μ	7	F

Policy 1:

- Propagation of tainted value depends on policy -
It might depend on memory, var., loc.

Taint depends only on the memory cell

$$\text{Load} \quad l = \rho[x], t = \tau_{\mu}[l] \quad * \\ \text{load}(x) \downarrow t$$



ρ	Var	Val
	x	7

μ	Addr	Val
	7	42

T_μ	Addr	Tainted?
	7	F

I associate value associated to x. And find the location. Now it looks in memory and find value.

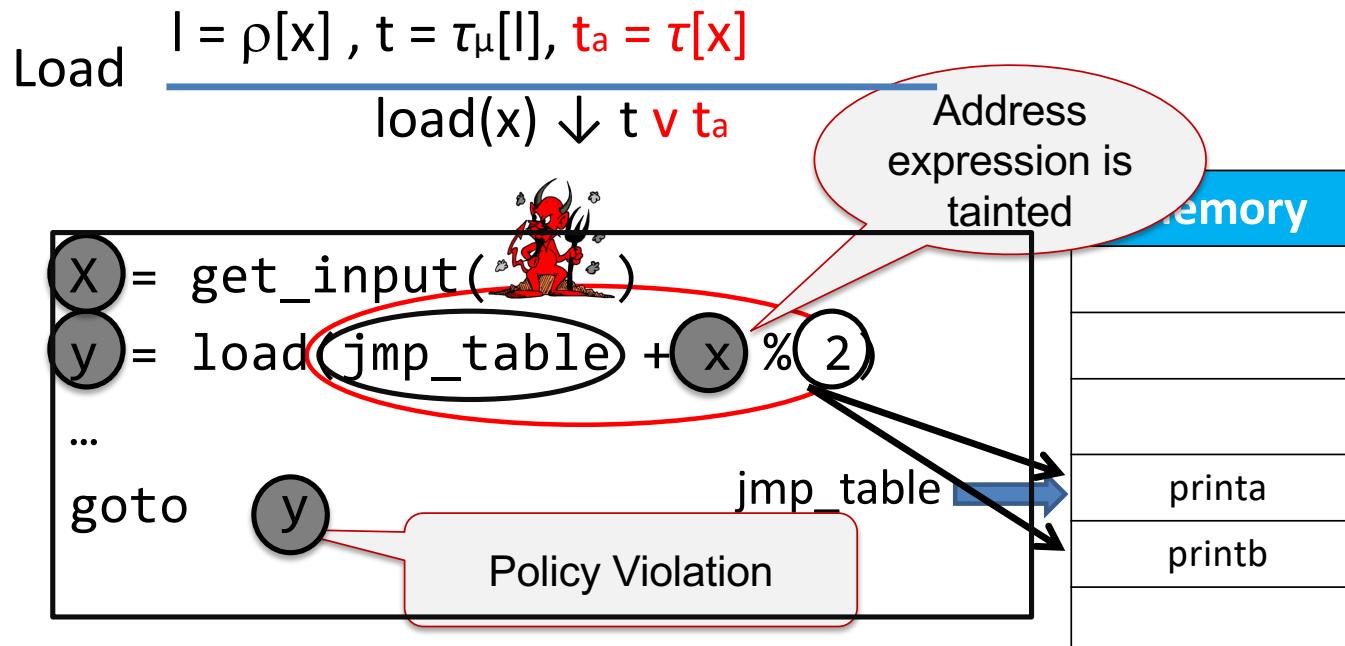
* In premise here you find kind of location. So location is kind.

- ① l is location in environment associated to x .
- ② What is the k_{ml} value associate to location l ?
- ③ γ_M returns association

So k_{ml} value of load depends only loc of variable.
We could also take into account environment etc..

Policy 2:

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



The approach

- We explain the main features of dynamic taint analysis by exploiting a simple yet expressive intermediate programming language.
- The language includes features
 - from Java bytecode
 - from assembly languages

An Intermediate programming language (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 ◊b exp | ◊u exp
          | var | get_input(src) | v
◊b ::= typical binary operators
◊u ::= typical unary operators
value v ::= 32-bit unsigned integer
```

We consider an intermediate prog. language at similar binary level. By code

Remark

The expression
`get_input(src)` returns
input from the source
stream `src`.

↳ only possible
source of input

We model input stream
as a suitable list, ie. `scr =`
`v:: src'`

↑ first el. of input stream and rest of list

We omit the type-
checking mechanism of our
language and
assume things are
well-typed in the
obvious way,

Operational semantics

Operational semantics of a programming language describes the interpreter: how to execute a program in terms of the run-time values and the run-time data structures

Idea: since dynamic taint analysis is defined in terms of program execution the operational semantics is the natural mechanism on which to base the dynamic taint analysis

First we present op sem., then policy.

Run-time structures

- Σ : the ordered sequence of program statements $\Sigma = \text{Nat} \rightarrow \text{Stmt}$
- μ : memory $\mu: \text{Loc} \rightarrow \text{Values}$ ADDRESSES TO →
IDE entries to locations or to values.
- ρ : environment $\rho: \text{Var} \rightarrow \text{Loc + Values}$
- pc: program counter
- i: next instruction: language has go-to
[nota]

IMMUTABLE: if var goes to loc then value: mutable, otherwise immutable.

MUTABLE:

Program evolution: expressions

The result is our target and witness

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

Intuition: evaluating the expression e in the run-time context provided by the memory μ and the environment ρ produces v as result

Program evolution: statements

for statements we are interested to
see how machine progresses step
by step

$$\Sigma, \mu, \rho, pc : smt \rightarrow \Sigma, \mu', \rho', pc' : smt'$$

↳ Program is left unchanged (no JIT compilation)

- Intuition: the execution of the statement smt in the run-time context given by
 - the program list (Σ),
 - the current memory state (μ),
 - the current binding for variable (ρ)
 - the current program counter (pc)
- yields a new state of program execution (Σ, μ', ρ', pc')

Remark:
Program evolution-
statements

$$\Sigma, \mu, \rho, pc : smt \rightarrow \Sigma, \mu', \rho', pc' : smt'$$

- Intuition: the execution of the statement smt in the run-time context yields a new state of program execution (Σ, μ', ρ', pc')
- The program Σ does is not modified by transitions.
 - We do not allow programs with dynamically generated code.

A sample of the operational semantics (expressions)



$$\frac{src = v :: src'}{\mu, \rho \vdash \text{getInput}(src) \Downarrow v}$$

expression that produces value v from a list.

$$\frac{\mu, \rho \vdash e \Downarrow v_1 \quad v = \mu(v_1)}{\mu, \rho \vdash \text{load } e \Downarrow v}$$

↳ V dash: we are evaluating this based on M and P.
We resort to memory and environment

$$\frac{}{\mu, \rho \vdash \text{var} \Downarrow \rho(\text{var})}$$

↳ Rule describes behavior.

A sample of the operational semantics (statement)

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

↑
evaluate expression. We have long step
semantics.

We execute this
in the state*

apply funct. program
↑
old program contn

* Ev. of exp. is only defined in terms of M, P. While for assignments, it's defined in terms of Σ, M, P, pc .

A sample of the operational semantics (statement)

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

The current state of
execution

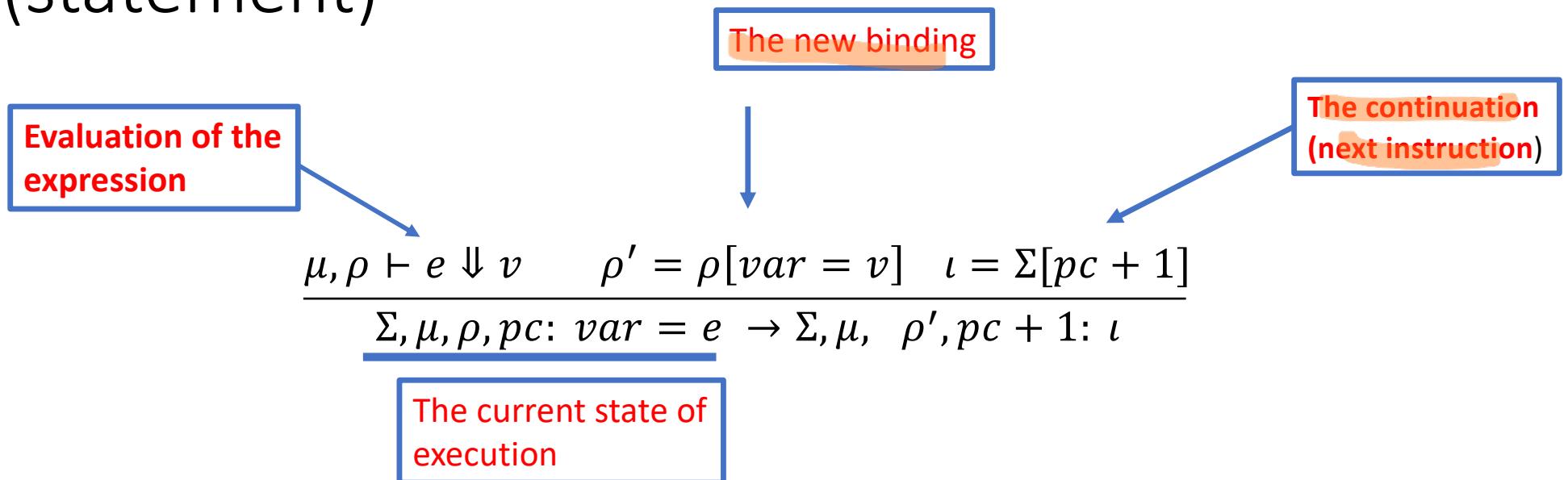
A sample of the operational semantics (statement)

Evaluation of the expression

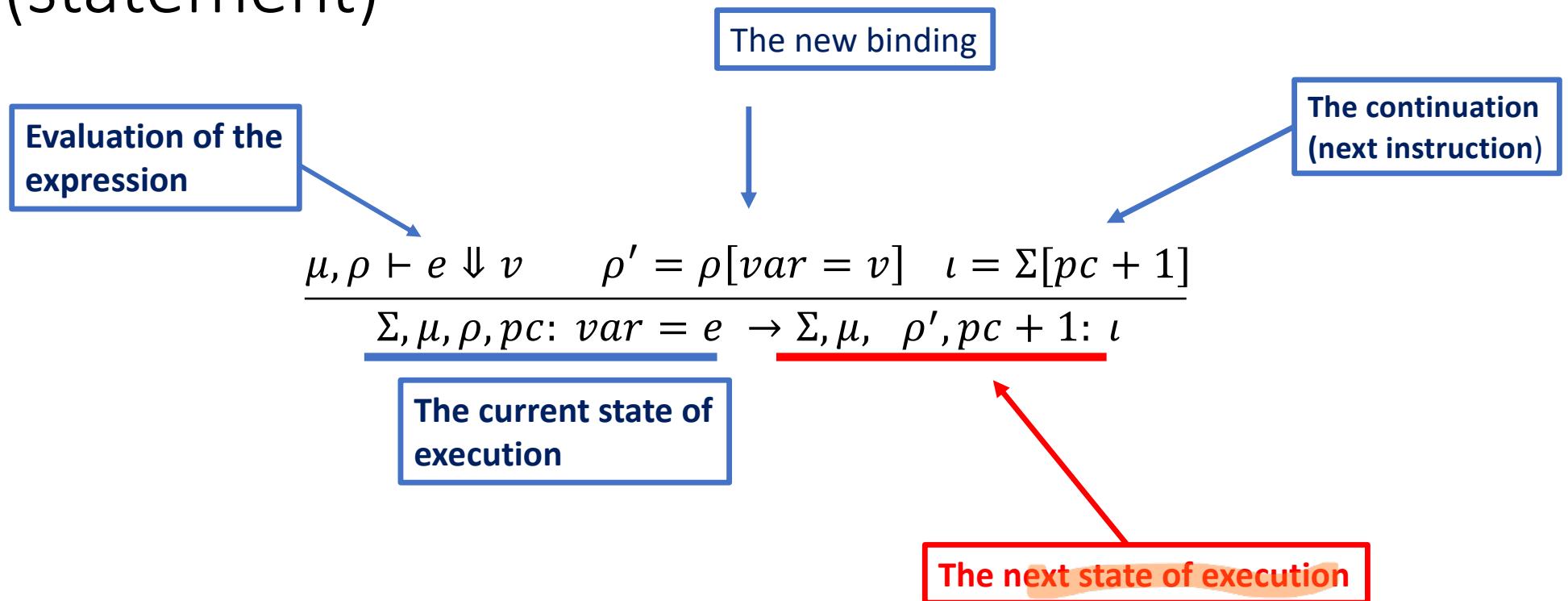
$$\frac{\mu, \rho \vdash e \Downarrow v \quad \rho' = \rho[\text{var} = v] \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \rho, pc: \text{var} = e \rightarrow \Sigma, \mu, \rho', pc + 1: \iota}$$

The current state of execution

A sample of the operational semantics (statement)



A sample of the operational semantics (statement)



A sample of the operational semantics (statements)

$$\frac{\mu, \rho \vdash e \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \rho, pc: goto e \rightarrow \Sigma, \mu, \rho, v_1: \iota}$$

Note structured control flow. You can jump at any possible location

$$\frac{\mu, \rho \vdash e_1 \Downarrow v_1 \quad \mu, \rho \vdash e_2 \Downarrow v_2 \quad \iota = \Sigma[pc + 1] \quad \mu' = \mu[v_1 = v_2]}{\Sigma, \mu, \rho, pc: Store(e_1, e_2) \rightarrow \Sigma, \mu', \rho, pc + 1: \iota}$$

$$\frac{\mu, \rho \vdash e \Downarrow 1 \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \rho, pc: assert(e) \rightarrow \Sigma, \mu, \rho, pc + 1: \iota}$$

↑ we check where at a certain point of a program we have something we expect. We find whether certain values have been produced.



What about
functions?

***Function calls in
high-level
programming
language are
compiled by storing
the return address
and transferring
control flow.***



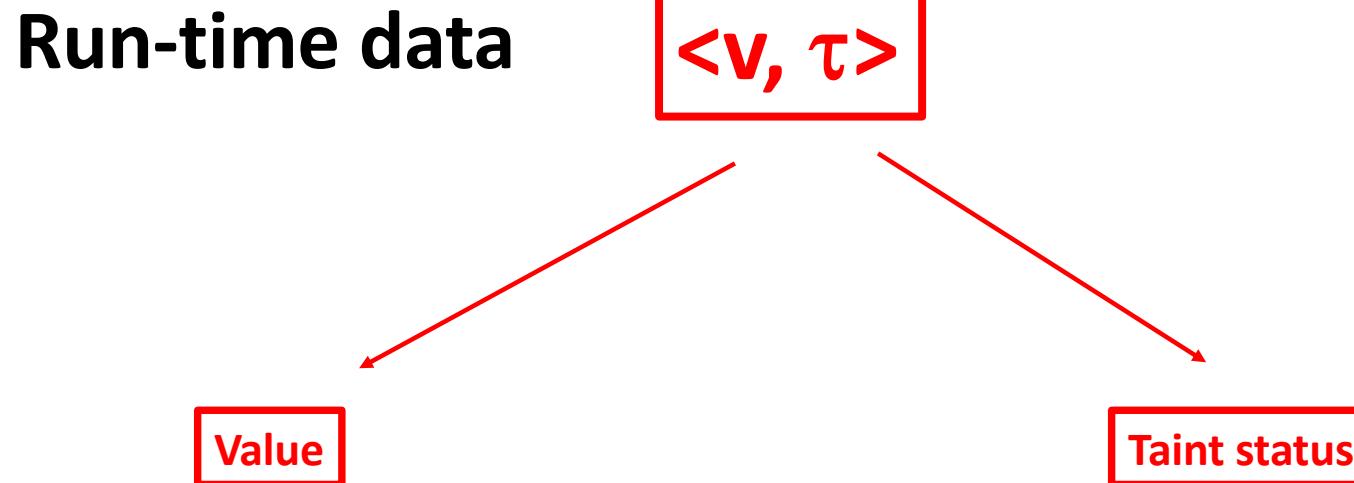
Dynamic taint analysis

Express the taint propagation in terms of the operational semantics of the intermediate language

Dynamic taint analysis is obtained by monitoring the program execution via suitable taint checkers

Design issues

keep track of the taint status of run-time data



Run-time structure: extension

Taint $\tau ::= T \mid F$

Value ::= $\langle v, \tau \rangle$

τ_p : Maps variable to taint status

τ_μ : Maps addresses to taint status

Taint Policies

How new taint is introduced

How taint propagates when execution progresses

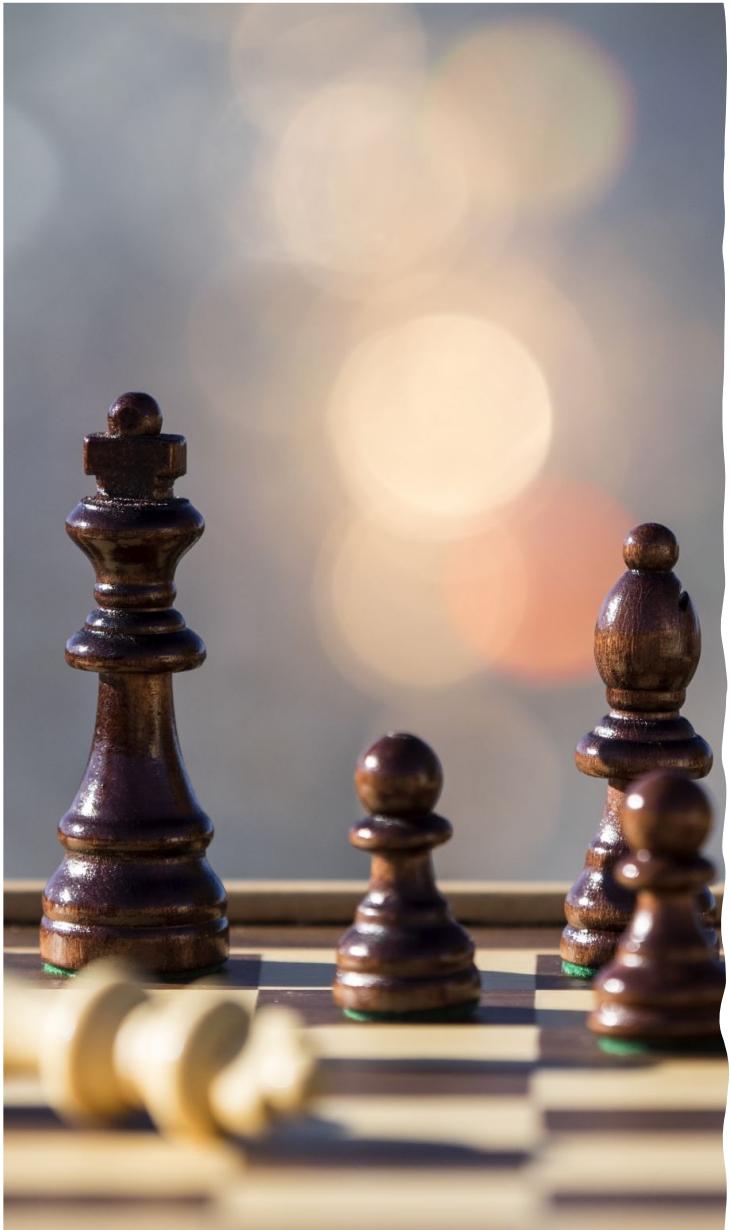
How taint is checked during execution



Taint introducion

Operational rules describe how taint values are introduced in the system

In our language we have a single source: the **getInput** operation.



Taint propagation

The taint propagation rules specifies how taint is derived from operation or control mechanisms

Taint Checking

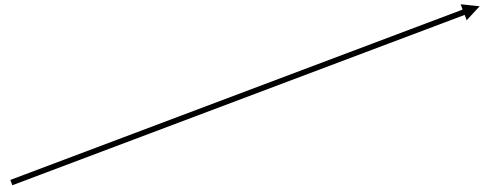
The taint value of run-time data impacts over the behaviour of programs: the detector may stop the execution if the address of a jump is tainted



Program instrumentation: we perform checking of taint policies before applying execution rules

Taint checking (example)

$$\frac{src = v :: src'}{\tau_\mu, \tau_\rho, \mu, \rho \vdash getInput(src) \Downarrow \langle v, TPIN(src) \rangle}$$



TPIN is the taint policy associated to the data source src

Taint checking (example)

TPIN is the taint policy associated to the data source src

Assume that src is under the control of the attacker

$$\frac{src = 5 :: src'}{\tau_\mu, \tau_\rho, \mu, \rho \vdash getInput(src) \Downarrow \langle 5, T \rangle}$$

A summary of Tainted Policies

POLICY COMPONENT	POLICY CHECK
TPIN(-)	T
TPCONST(-)	F
TPASN(t)	t
TPMEM($t_a t_v$)	t_v
TPGOTO(t)	not t
....	

A taint status is converted to a boolean value in the natural way, e.g., T maps to true, and F maps to false.

The taint instrumented semantics

$$\frac{src = v :: src'}{\tau_\mu, \tau_\rho, \mu, \rho \vdash getInput(src) \Downarrow \langle v, TPIN(-) \rangle}$$

$$\frac{}{\tau_\mu, \tau_\rho, \mu, \rho \vdash v \Downarrow \langle v, TPCONST(-) \rangle}$$

$$\frac{}{\tau_\mu, \tau_\rho, \mu, \rho \vdash var \Downarrow \langle \rho(var), \tau_\rho(var) \rangle}$$

$$\frac{\tau_\mu, \tau_\rho, \mu, \rho \vdash e \Downarrow \langle v, t \rangle \ \rho' = \rho[\textit{var} = v] \ \ \tau'_\rho = \tau_\rho[\textit{var} = \textit{TPASN}(t)] \ \iota = \Sigma[\textit{pc} + 1]}{\tau_\mu, \tau_\rho, \Sigma, \mu, \rho, \textit{pc}; \ \textit{var} = e \rightarrow \tau_\mu, \tau'_\rho, \Sigma, \mu, \rho', \textit{pc} + 1 : \iota}$$

$$\textcolor{red}{\textit{TPASN}(t) = t}$$

$$\tau_\mu, \tau_\rho, \mu, \rho \vdash e_1 \Downarrow \langle v_1, t_1 \rangle. \quad \tau_\mu, \tau_\rho, \mu, \rho \vdash e_2 \Downarrow \langle v_2, t_2 \rangle$$

$$\frac{\mu' = \mu[v_1 = v_2] \ \tau'_\mu = \tau_\mu[v_1 = TPMEM(t_1, t_2)] \iota = \Sigma[pc + 1]}{\tau_\mu, \tau_\rho, \Sigma, \mu, \rho, pc; store(e_1, e_2) \rightarrow \tau'_\mu, \tau_\rho, \Sigma, \mu', \rho, pc + 1 : \iota}$$

$$TPMEM(t_1,t_2)=t_2$$

$$\frac{\tau_\mu, \tau_\rho, \mu, \rho \vdash e \Downarrow \langle v, t \rangle \quad TPgoto(t) = T \quad \iota = \Sigma[v]}{\tau_\mu, \tau_\rho, \Sigma, \mu, \rho, pc; goto\ e \rightarrow \tau_\mu, \tau_\rho, \Sigma, \mu, \rho, v : \iota} \quad \text{TPGOTO}(t) = \text{NOT } t$$

The rule is applied when it is safe to perform a jump operation
 This holds when $TPgoto(t)$ returns T , i.e. the value t is F (untainted)

When the target address is tainted, $TPgoto(t)$ returns F and the premises of the rule is not satisfied and an exception is raised

1. $x = 2 * \text{get_input}([20]);$
2. $y = 5 + x;$
3. $\text{goto } y$

$$\frac{src = 20 :: []}{\tau_\mu, \tau_\rho, \mu, \rho \vdash \text{getInput}([20]) \Downarrow \langle 20, T \rangle}$$

τ_ρ	{}
ρ	{}

$x = 2 * \text{get_input}([20]);$

$$\frac{\tau_\mu, \tau_\rho, \mu, \rho \vdash 2 * \text{getInput}([20]) \Downarrow \langle 40, T \rangle \quad \rho' = [x = 40] \quad \tau'_\rho = [x = T] \quad \iota = 2}{\tau_\mu, \tau_\rho, \Sigma, \mu, \rho, pc; x = 2 * \text{getInput}([20]) \rightarrow \tau_\mu, \tau'_\rho, \Sigma, \mu, \rho', 2: \iota}$$

τ_ρ	$[x=T]$
ρ	$[x=40]$

$y = 5 + x;$

$$\frac{\tau_\mu, \tau_\rho, \mu, \rho \vdash 5 + x \Downarrow \langle 45, T \rangle \quad \rho' = [x = 40, = 45] \quad \tau'_\rho = [x = T, y = -T] \quad \iota = 3}{\tau_\mu, \tau_\rho, \Sigma, \mu, \rho, pc; y = 5 + x \rightarrow \tau_\mu, \tau'_\rho, \Sigma, \mu, \rho', 3: \iota}$$

τ_ρ	$[x=T,$ $y=T]$
ρ	$[x=40,$ $y=45]$

$\text{goto } y$

$$\frac{\tau_\mu, \tau_\rho, \mu, \rho \vdash y \Downarrow \langle 45, T \rangle \quad F = T \quad \iota = \Sigma[45]}{\tau_\mu, \tau_\rho, \Sigma, \mu, \rho, pc; \text{goto } y \rightarrow err}$$

1. `x = 2*get_input([20]);`
2. `y = 5+ x;`
3. `goto y`

Line #	Stm	ρ	τ_ρ	Rule	pc
	start	{}	{}		1
1	<code>x = 2*getInput(20::[]))</code>	{x=40}	{x = T}	ASSIGN	2
2	<code>y = 5 + x</code>	{x=40, y = 45}	{x=T, y = T}	ASSIGN	3
3	<code>goto y</code>	{x=40, y = 45}	{x=T, y = T}	GOTO	err

Load: take #1

$$\frac{\tau_\mu, \tau_\rho, \mu, \rho \vdash e \Downarrow \langle v, t \rangle}{\tau_\mu, \tau_\rho, \mu, \rho \vdash \text{load } e \Downarrow \langle \mu(v), \text{TPmem}(t, \tau_\mu(v)) \rangle}$$

$$\text{TPmem}(t_a, t_v) = t_v$$

Only the tainted value of the cell is considered

A main design issue

- A. `x = get_input()`
- B. `y = load(z+x)`
- C. `goto y`

Assumptions:

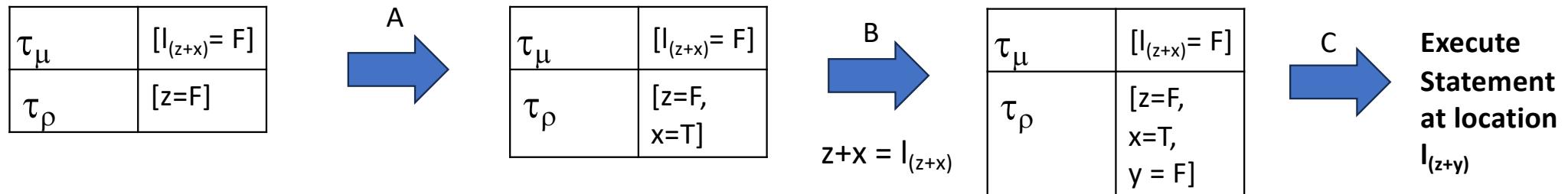
- The value associated to variable z has been already defined
- The attacker provides input x to the program that is used as a table index offset
- The attacker can provide an appropriate value of x to address any value in memory that is untainted.
- The result of the table lookup is used as the target address for a jump.

Tainted jump

$$\frac{\tau_\mu, \tau_\rho, \mu, \rho \vdash e \Downarrow \langle v, t \rangle}{\tau_\mu, \tau_\rho, \mu, \rho \vdash \text{load } e \Downarrow \langle \mu(v), \text{TPmem}(t, \tau_\mu(v)) \rangle}$$

A. `x = get_input()`
 B. `y = load(z+x)`
 C. `goto y`

$$\text{TPmem}(t_a, t_v) = t_v$$



$$\text{load}(x + y) \Downarrow \langle l_{\{z+x\}}, T \rangle$$

$$\text{TPmem}(T, \tau_\mu(I_{(z+x)})) = \tau_\mu(I_{(z+x)}) = F$$

Load: take #2

$$\frac{\tau_\mu, \tau_\rho, \mu, \rho \vdash e \Downarrow \langle v, t \rangle}{\tau_\mu, \tau_\rho, \mu, \rho \vdash \text{load } e \Downarrow \langle \mu(v), \text{TPmem}(t, \tau_\mu(v)) \rangle}$$

$$\text{TPmem}(t_a, t_v) = t_a \text{ OR } t_v$$

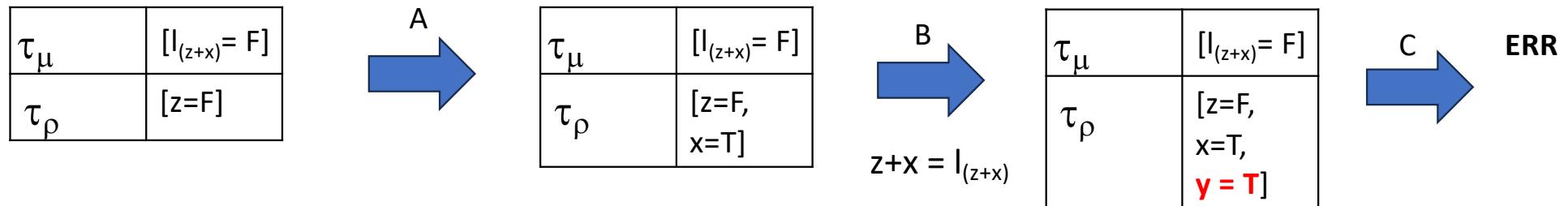
The tainted value of load is a combination of the taint value of the location and the tainted value of the cell

Tainted jump

$$\frac{\tau_\mu, \tau_\rho, \mu, \rho \vdash e \Downarrow \langle v, t \rangle}{\tau_\mu, \tau_\rho, \mu, \rho \vdash \text{load } e \Downarrow \langle \mu(v), \text{TPmem}(t, \tau_\mu(v)) \rangle}$$

A. `x = get_input()`
 B. `y = load(z+x)`
 C. `goto y`

$$\text{TPmem}(t_a, t_v) = t_a \text{ OR } t_v$$



$$\text{load}(x + y) \Downarrow \langle l_{\{z+x\}}, T \rangle$$

$$\text{TPmem}(T, \tau_\mu(l_{(z+x)})) \text{ TPmem}(T, F) = T \text{ OR } F = T$$

Control flow taint

```
1  x := get_input(·)
2  if x = 1 then goto 3 else goto 4
3  y := 1
4  z := 42
```

*The assignment to y is control-dependent on line 2,
since the branching outcome determines whether or not line 3 is executed.*

*The assignment to z is not control-dependent on line 2, since
z will be assigned the value 42 regardless of which branch is taken.*

```
1  x := get_input(·)
2  if x = 1 then goto 3 else goto 4
3  y := 1
4  z := 42
```

Control Flow Taint

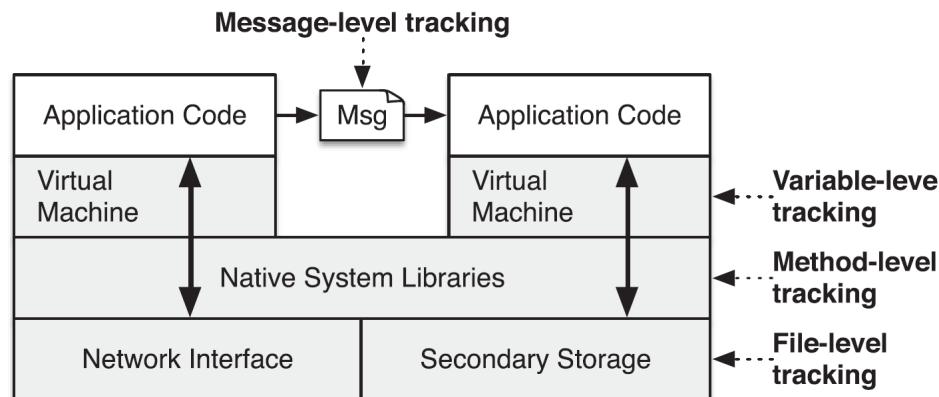
Dynamic taint analysis cannot compute control dependencies,
thus cannot determine control-flow-based taint.

Reasoning about control dependencies requires reasoning about
multiple paths, and dynamic analysis executes on a single path at
a time.



Dynamic Taint Analysis in practice: TaintDroid

- William Enck, *et al.* “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones,” OSDI, 2010.
- Privacy leakage (misuse) detection for Android applications.
- Employ dynamic taint analysis and report leakage during runtime.
- Challenge: Track the propagations of private data in Android platform, *e.g.*, Inter-process communication.



Taint Propagation of Interpreted Code

- Variable-level taint tracking

Op Format	Op Semantics	Taint Propagation	Description
<i>const-op</i> $v_A \ C$	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
<i>move-op</i> $v_A \ v_B$	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
<i>move-op-R</i> v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
<i>return-op</i> v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint (\emptyset if void)
<i>move-op-E</i> v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
<i>throw-op</i> v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
<i>unary-op</i> $v_A \ v_B$	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
<i>binary-op</i> $v_A \ v_B \ v_C$	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint \cup v_C taint
<i>binary-op</i> $v_A \ v_B$	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
<i>binary-op</i> $v_A \ v_B \ C$	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
<i>aput-op</i> $v_A \ v_B \ v_C$	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
<i>aget-op</i> $v_A \ v_B \ v_C$	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
<i>sput-op</i> $v_A \ f_B$	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
<i>sget-op</i> $v_A \ f_B$	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
<i>iput-op</i> $v_A \ v_B \ f_C$	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
<i>iget-op</i> $v_A \ v_B \ f_C$	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to f_C and obj. ref. taint

TAINT ANALYSIS: SUMMARY

- Model: *Model of programming language behaviour*
- Threat: The attacker controls some data and attempts to taint program data in order to create security vulnerabilities (buffer overflow, string attacks, injections)
- Countermeasures: Analysis to track flow of data

References

- E. J. Schwartz, T. Avgerinos, D.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: 317-331,
- The paper defines the algorithms and summarizes the critical issues
that arise when taint analyses are used in typical security context