Dymamic 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 Assemby
- 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 $\rightarrow$ public public $\rightarrow$ secret
secret $\rightarrow$ secret secret $\not\rightarrow$ public

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

reliable $\rightarrow$ reliable reliable $\rightarrow$ dubious
dubious $\rightarrow$ dubious dubious $\not\rightarrow$ 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 Metldown 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.

Meldown 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.

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

Tracking Data Flow to Mitigate Meltdown

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.

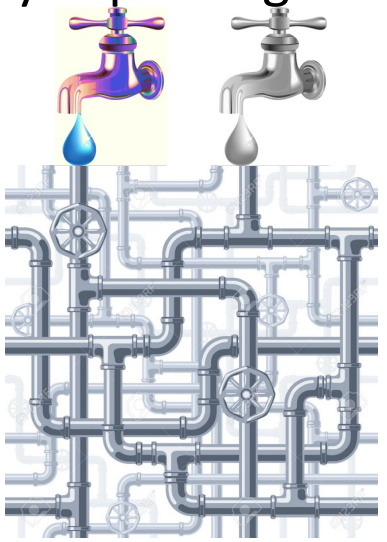
What is Dynamic Taint Analysis?

- **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 inj. is an example in which 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 untainted) 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 prog. 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

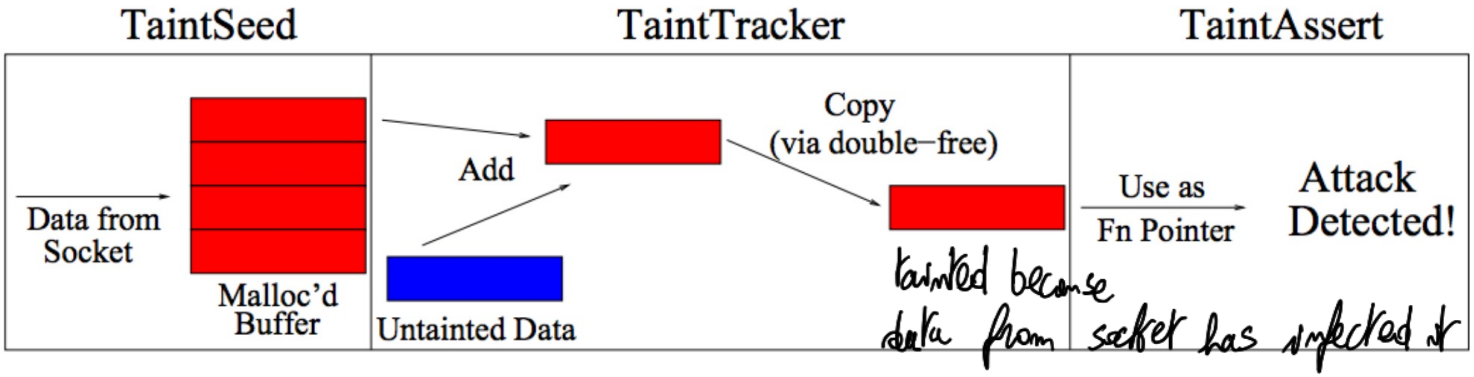
- **Identify sources of taint**
  - Untrusted input is controlled by the attacker
  - Threat model
- **Identify Propagation of taint**
  - Tracks flow that manipulates data in order to determine whether the result is tainted or not
- ***If the analysis finds that tainted data could be used where untainted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

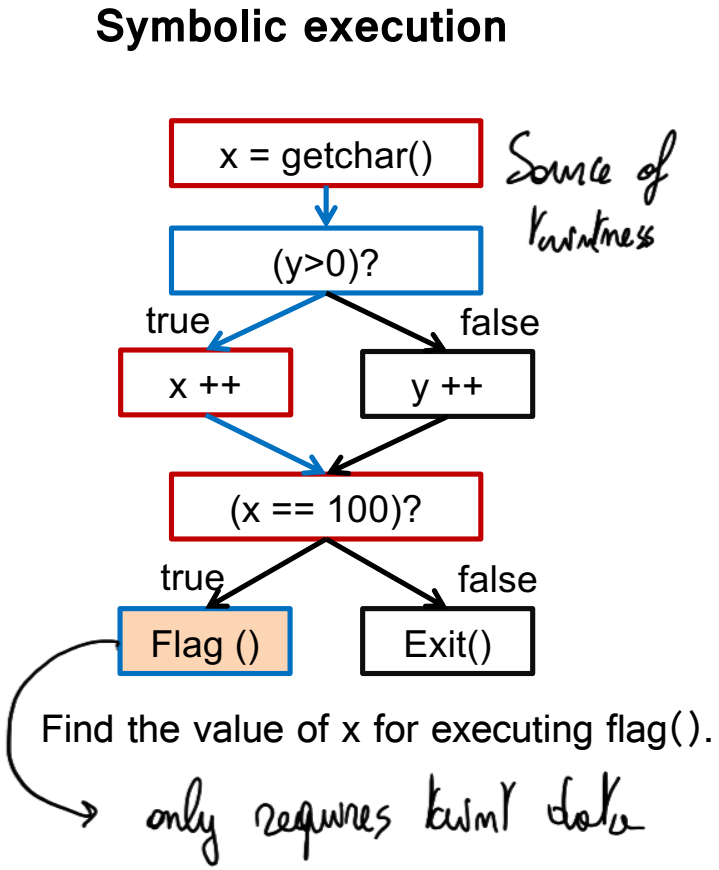
Application: Sanity Check

- Check if untrusted data are used dangerously.



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

Application: Symbolic Execution



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

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

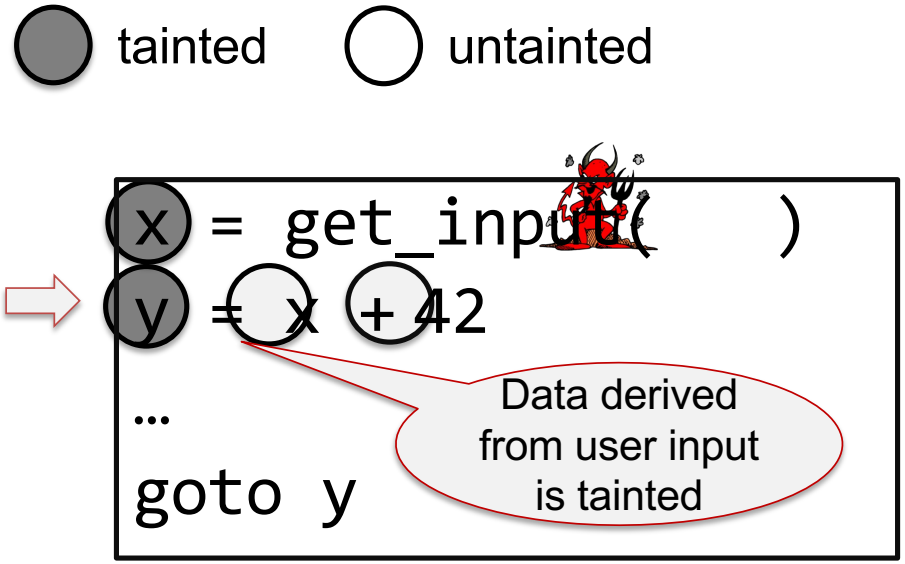
TaintSeed:

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

τ

Var	Tainted?
x	T

if source is untrusted
result is untrusted



μ

Var	Val
x	7
y	49

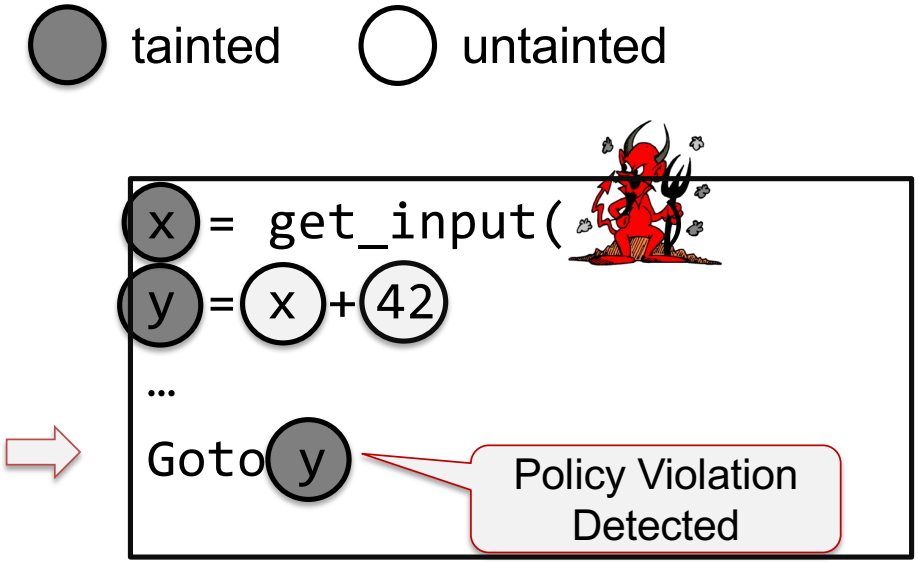
TaintTracker

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

At the level of taintedness we take the or of taintedness

τ

Var	Tainted?
x	T
y	T



μ

Var	Val
x	7
y	49

τ

Var	Tainted?
x	T
y	T

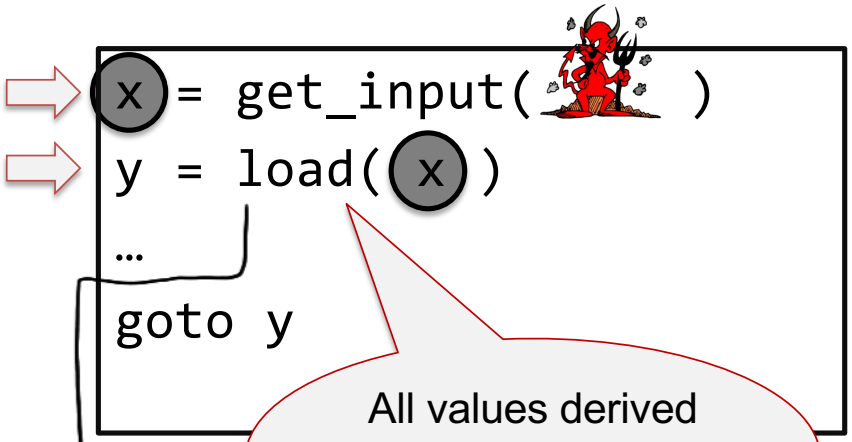
TaintAssert:

$$P_{goto}(t_a) = \neg\, t_{tainted}$$

(Must be true to execute)

Why? Jump/Return address has been overwritten.

Problem: Memory Addresses



ρ

Var	Val
x	7

μ

Addr	Val
7	42

T_μ

Addr	Tainted?
7	F

We load a value, but values will be tainted?

Policy 1:

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

Taint depends only on the memory cell

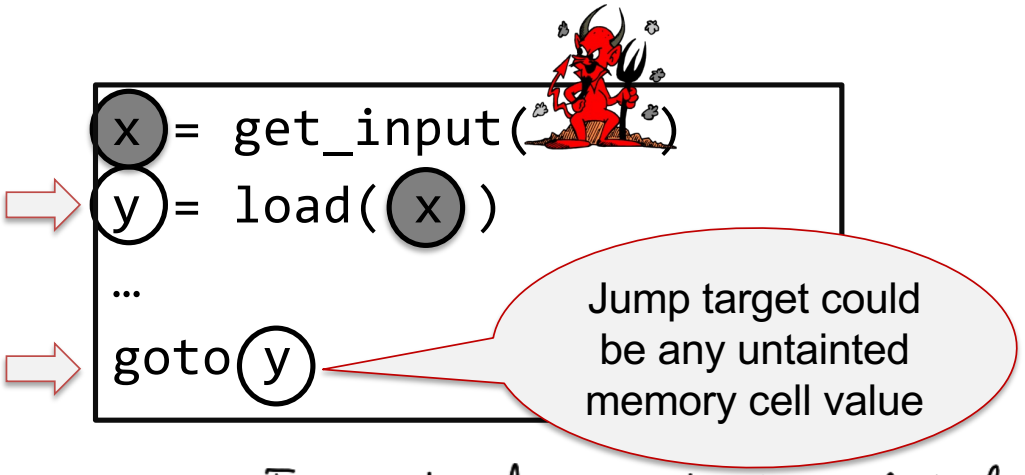
$$\text{Load}\ \frac{l = \rho[x]\ ,\ t = \tau_\mu[l]\ ^*}{\text{load}(x) \downarrow t}$$

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 I look in memory and find value.

*In premise here you find taint of location. So location is tainted.

① l is location in environment associated to x.

② What is the current value associate to location l?

③ γ_M returns association

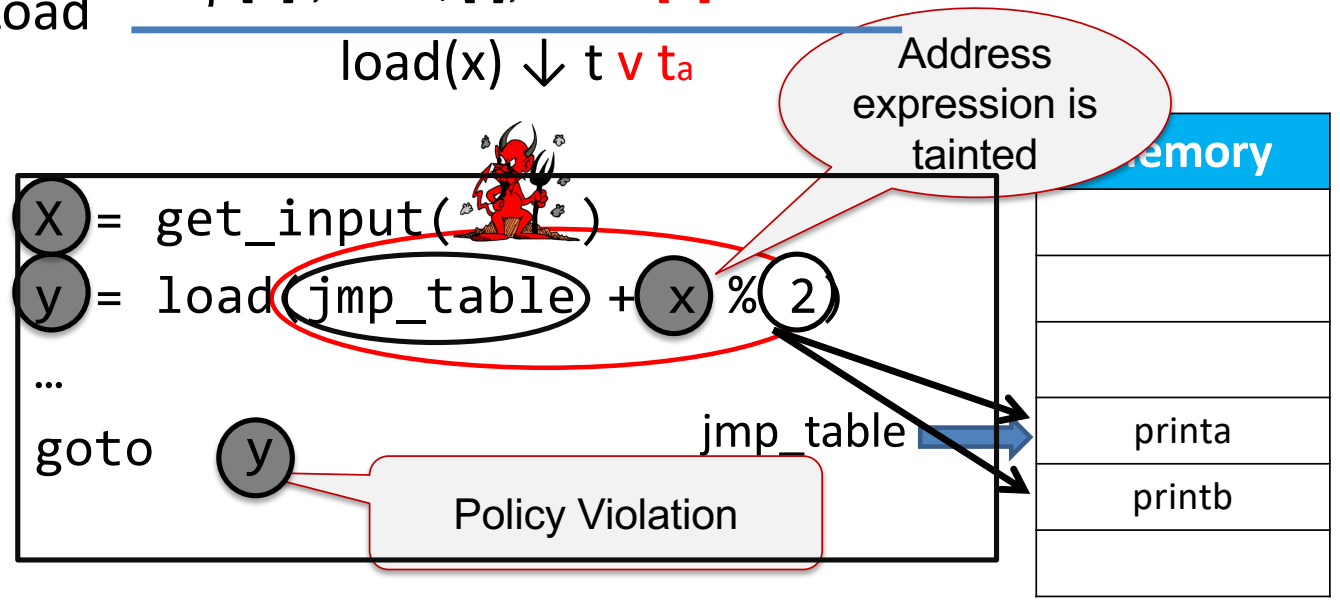
So current value of load depends only loc of variable.

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

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



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* $\Diamond_b$ *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

We consider an intermediate prog. language at similar binary level. Bytecode

Remark

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

↳ only possible source of variable

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 preser op sem., then policy.

Run-time structures

- **Σ**: the ordered sequence of program statements **Σ = Nat -> Stmt**
- **μ**: memory **μ: Loc -> Values**
- **ρ**: environment **ρ: Var -> Loc + Values**
- **pc**: program counter
- **ι**: next instruction

ADDRESSES TO

IDE either to locations or to values.

IMMUTABLE vs MUTABLE?

: language has go-to

node

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

MUTABLE:

Program evolution: expressions

The result is an integer and nothing else

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

Intuition: evaluationg 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 currentr 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')**

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

Remarck: Program evolution-statements

- **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)

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

expression that produces value taken from a list.

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

& LOCATION

⊢ (turnstile): we are evaluating this based on μ and ρ. We resort to memory and environment

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

Rule describes behavior.

A sample of the operational semantics (statement)

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

apply funct. program

old program counter

We execute this in the state*

evaluate expression. We have big step semantics.

*Ev. of exp. is only defined in terms of M, P. While for assignments, is defined in terms of Σ, M, P, PC.

A sample of the operational semantics (statement)

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

The current state of execution

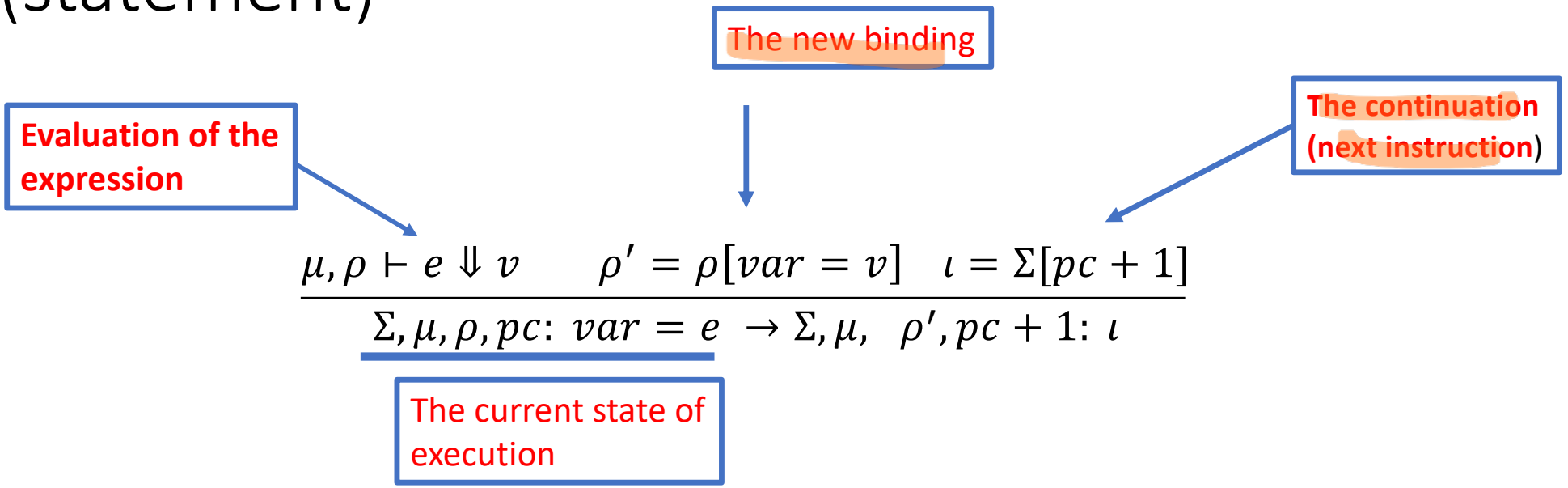
A sample of the operational semantics (statement)

Evaluation of the expression

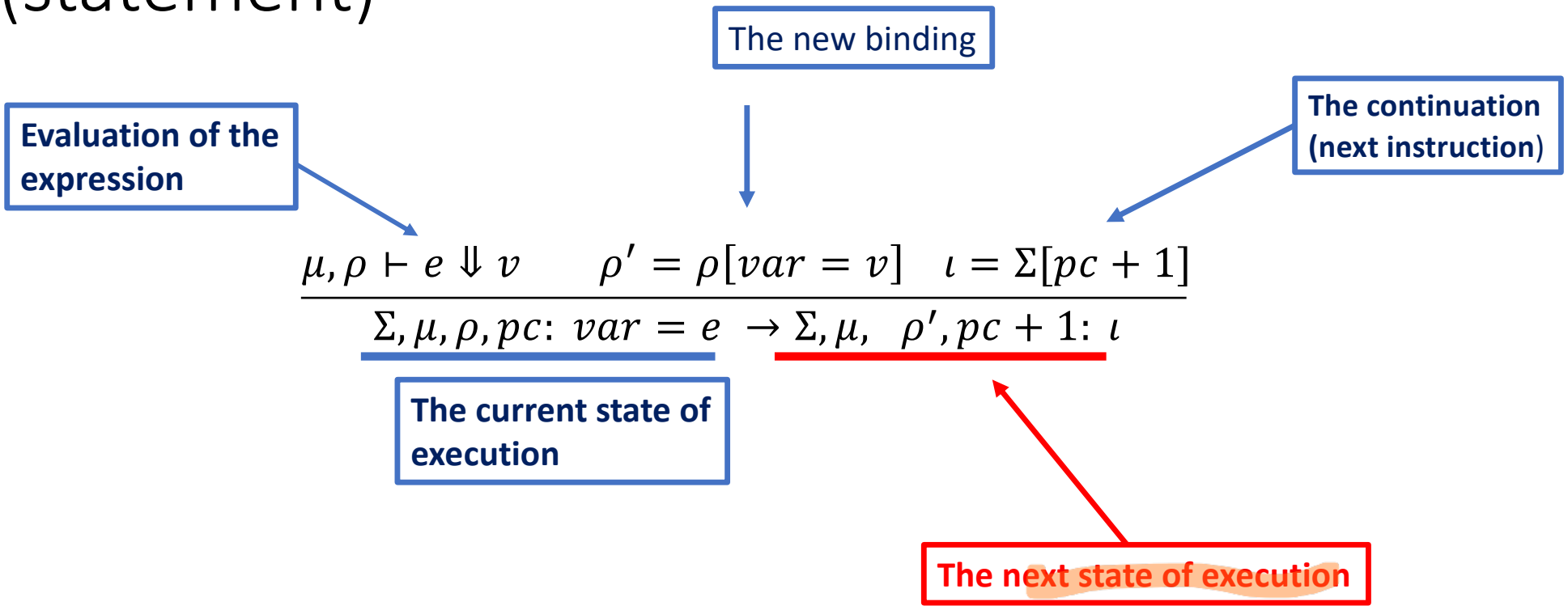
$$\frac{\mu, \rho \vdash e \Downarrow v \qquad \rho' = \rho[var = v] \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \rho, pc{:}\ var = e \ \rightarrow \Sigma, \mu, \ \ \rho', pc + 1{:}\ \iota}$$

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)

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

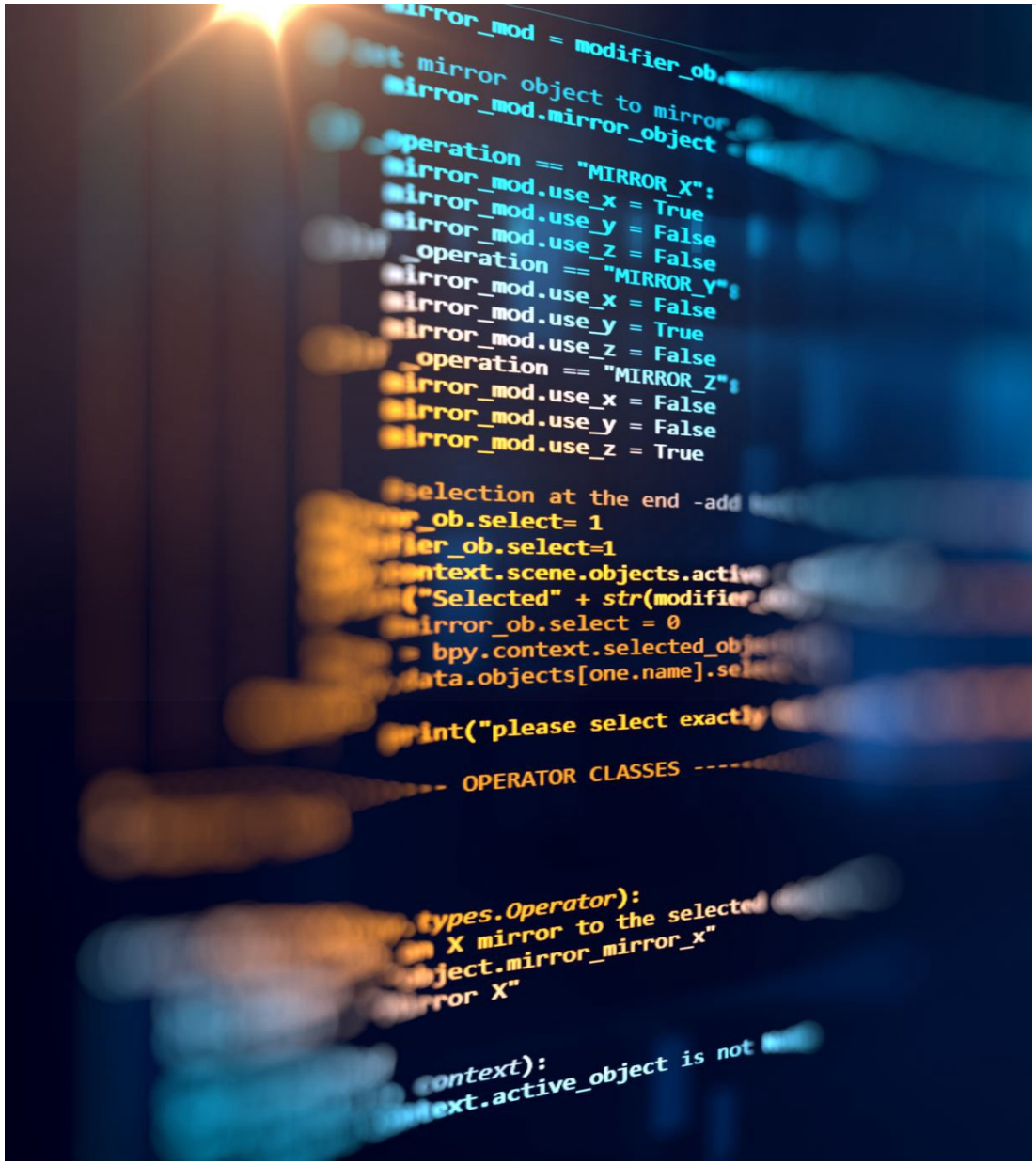
Not structured control flow. You can jump at any possible location

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

True C like approach

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

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



Syntax of our bytecode does not consider any function code, but we can introduce them like we did in WASM. Implement prologue, epilogue and execution. Of course here we are in a register based machine

What about functions?

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

Dynamic taint analysis

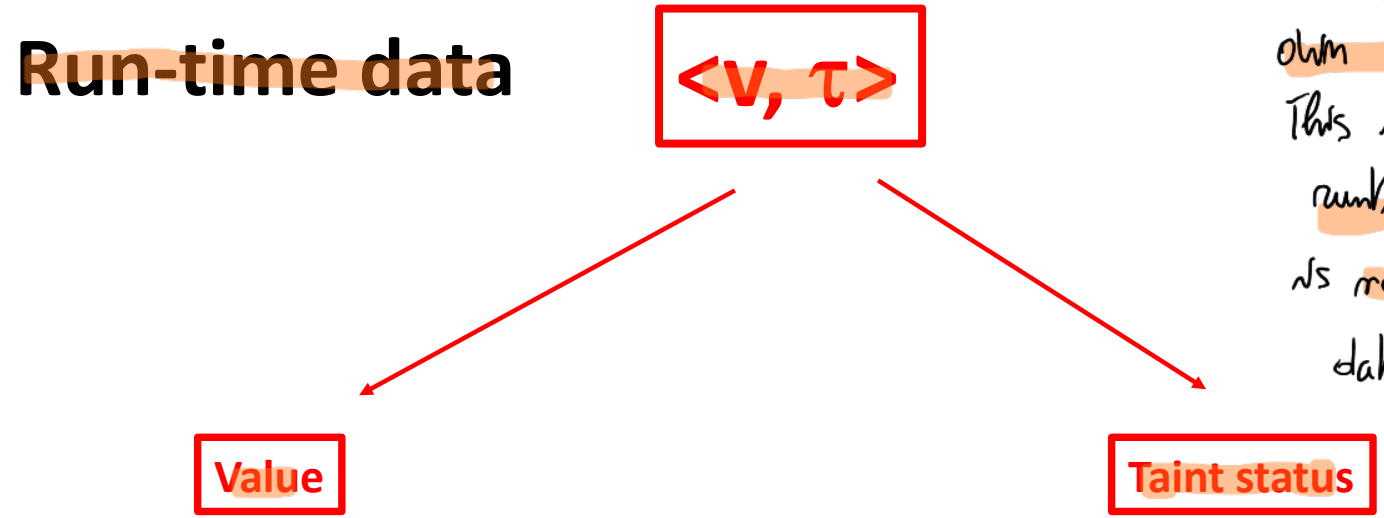
Express the taint propagation in terms of the operational semantics of the intermediate language

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

We keep data with their own taintness status. This must be reflected at runtime. At runtime data is not just values. We have data + tag.



To manage this representation we need a suitable meta-data and choices to make for design

Run-time structure: extension

Taint τ ::= T | F

Trick; taint is just a boolean value

Value ::= <v, τ>

τ_ρ: Maps variable to taint status

τ_μ: Maps addresses to taint status

Environment can map vars to immutable data or mem. locations. But to represent taints, we need two more mappings, from variables/addresses to taint status

Taint Policies

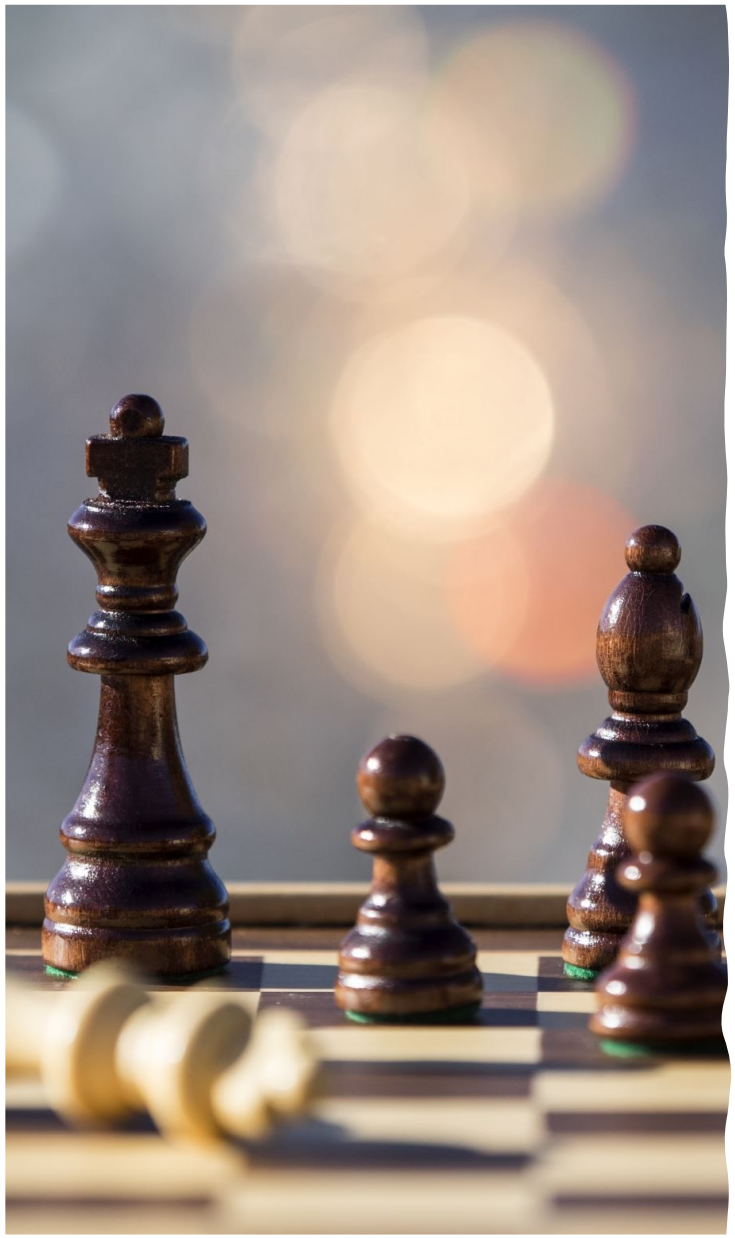
We have to identify the source, the sinks (where program expects untainted value) and how tainted value flows in the program

- 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

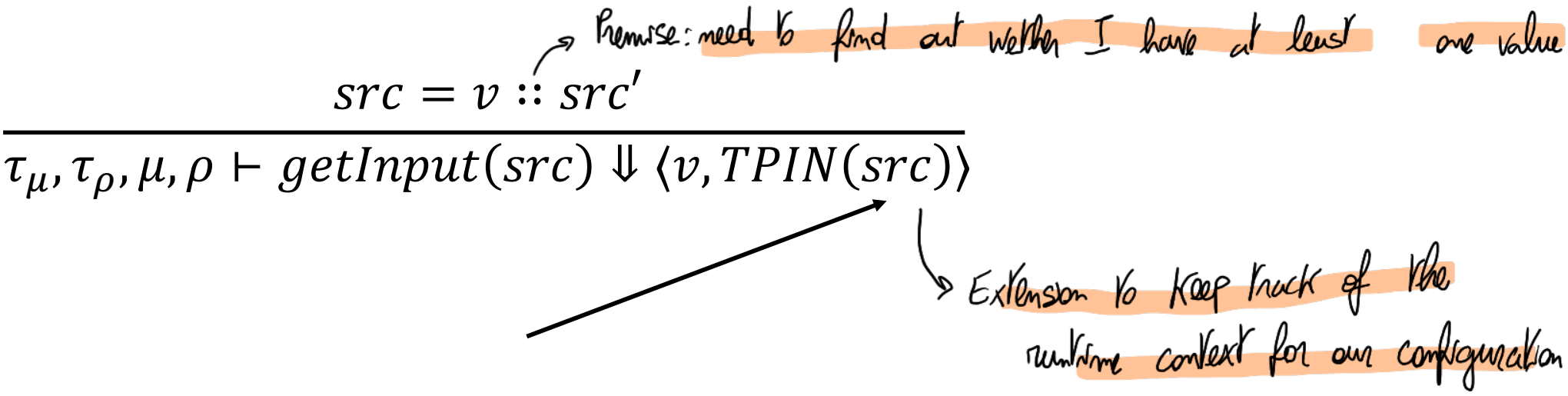
The taint propagation rules specifies how taint is derived from operation or control mechanisms

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)



TPIN is the taint policy associated to the data source src

TPIN: Taint policy IN

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) Assignment: val = exp*	t
TPMEM(t_a t_v) Taint policy takes two values: ①	t_v
TPGOTO(t)	not t
.... ① t_A for address, t_V associated	

To value we want to store in the location. Result is we take the tag associated to value,

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

* exp produces a pair value V and taint r. We take tag associated to value and that will be associated to variable.

GOTO has expression that produces value and target. You cannot jump if value is satisfied. You can jump if target value is false. So policy passes when NOT E is satisfied.

The taint instrumented semantics

$$\frac{src = v :: src'}{\tau_\mu, \tau_\rho, \mu, \rho \vdash getInput(src) \Downarrow \langle v, \underbrace{TPIN(-)}_{\text{We apply taint policies}} \rangle}$$

$$\frac{}{\tau_\mu, \tau_\rho, \mu, \rho \vdash v \Downarrow \langle v, \underbrace{TPCONST(-)}_{\text{Constant is untainted}} \rangle}$$

$$\frac{}{\tau_\mu, \tau_\rho, \mu, \rho \vdash var \Downarrow \langle \rho(var), \underbrace{\tau_\rho(var)}_{\text{Lookup of value and taint value}} \rangle}$$

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

↳ Taint states of env and mem

$$\boldsymbol{TPASN(t) = t}$$

don't you think about your curly hair
clothes that don't exist for you anyway

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

location — value to store — we won't consider virtual statuses for this location

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

when taint policy produces a true value (so t is false)

$$\frac{\tau_\mu, \tau_\rho, \mu, \rho \vdash e \Downarrow \langle v, t \rangle \quad TPgoto(t) = T \quad \iota = \Sigma[v]}{\tau_\mu, \tau_\rho, \Sigma, \mu, \rho, pc; goto\ e \rightarrow \tau_\mu, \tau_\rho, \Sigma, \mu, \rho, v{:}\iota}$$

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

You'll grow up and
grow so rough and charm them

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

τ_ρ	{}
ρ	{}

x = 2*get_input([20]);

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

τ_ρ	[x=T]
ρ	[x=40]

y = 5+ x;

taint status of x is T and we summed it with untainted val.

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

Result is tainted

τ_ρ	[x=T, y=T]
ρ	[x=40, y=45]

goto y

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

error

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

Value of PC

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

We didn't analyse loading operation! We have several design choices to manage loading

Load: take #1

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

location

Taint value here

Taint status is: we have taint status of the address and the taint status of the value in that location.

Here we only take into account the taint status associated to that value in the memory.

Problem: we are ignoring the fact that address can be tainted. Who knows what might happen.

We accept working with tainted data.

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.**

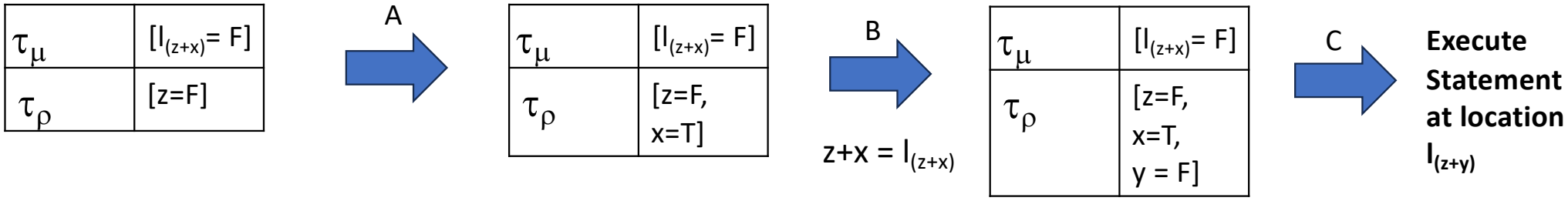
Typical example of RETURN ORIENTED PROGRAMMING

Tainted jump

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

TPmem(t_a, t_v) = t_v

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



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

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

Load: take #2

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

$TPmem(t_a, t_v) = t_a \text{ OR } t_v$ Stronger policy for the memory

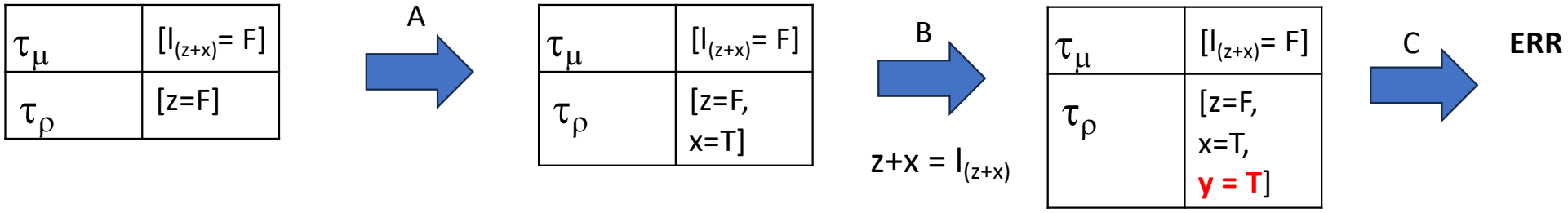
The tainted value of load is a combinatioon of the taint value of the location and the tainted value of the cell

Tainted jump

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

TPmem(t_a, t_v) = t_a OR t_v

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



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

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

We make this decision in terms of a tainted data.

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.

There are in programs some indirect flows that depend on the evaluation of the condition. The choice of the 2 branches is under control of the attacker. We would need a taint status for the program counter. This makes things much more complicated of course.

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

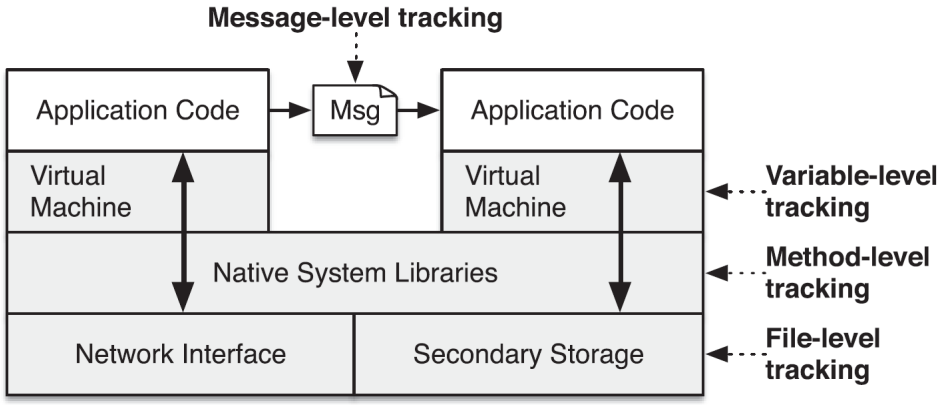
Control Flow Taint

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**
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op 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
binary-op 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
binary-op 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
binary-op 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
aput-op 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
aget-op 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
sput-op 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
sget-op 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
iput-op 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
iget-op 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 programa data in oder 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