Dynamic Taint Analysis

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)

$$
\begin{array}{lcl}
\textit{program} & ::= & \textit{stmt*} \\
\textit{stmt}\ s & ::= & \textit{var} := \textit{exp} \mid \text{store}(\textit{exp}, \textit{exp}) \\
 & & \mid \text{goto } \textit{exp} \mid \text{assert } \textit{exp} \\
 & & \mid \text{if } \textit{exp} \text{ then goto } \textit{exp} \\
 & & \quad \text{else goto } \textit{exp} \\
\textit{exp}\ e & ::= & \text{load}(\textit{exp}) \mid \textit{exp} \lozenge_b \textit{exp} \mid \lozenge_u \textit{exp} \\
 & & \mid \textit{var} \mid \text{get\_input}(src) \mid v \\
\lozenge_b & ::= & \text{typical binary operators} \\
\lozenge_u & ::= & \text{typical unary operators} \\
\textit{value}\ v & ::= & \text{32-bit unsigned integer}
\end{array}
$$

Remark

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

We model input stream as a suitable list, ie. Scr = v:: src’

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

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

Program evolution: expressions

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

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

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

- **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’)**

Remarck: Program evolution-statements

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

- **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)

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

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

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

A sample of the operational semantics (statement)

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

A sample of the operational semantics (statement)

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

The current state of execution

A sample of the operational semantics (statement)

Evaluation of the expression

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

The current state of execution

A sample of the operational semantics (statement)

The new binding

Evaluation of the expression

The continuation (next instruction)

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

The current state of execution

A sample of the operational semantics (statement)

The new binding

Evaluation of the expression

The continuation (next instruction)

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

The current state of execution

The next state of execution

A sample of the operational semantics (statements)

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

$$\frac{\mu, \rho \vdash e_1 \Downarrow v_1 \quad \mu, \rho \vdash e_2 \Downarrow v_{2.}\ \iota = \Sigma[pc + 1]\ \ \mu' = \mu[v_1 = v_2]}{\Sigma, \mu, \rho, pc: Store(e_1, e_2) \rightarrow \Sigma, \mu', \rho, pc + 1: \iota}$$

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

What about functions?

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

```
1  /* Caller function */
2  esp := esp + 4
3  store(esp, 6) /* retaddr is 6 */
4  goto 9
5  /* The call will return here */
6  halt
7
8  /* Callee function */
9  ...
10 goto load(esp)
```

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 data $\boxed{\langle v, \tau \rangle}$

Value

Taint status

Run-time structure: extension

Taint τ ::= T | F

Value ::= <v, τ>

τ_ρ: Maps variable to taint status

τ_μ: Maps addresses to taint status

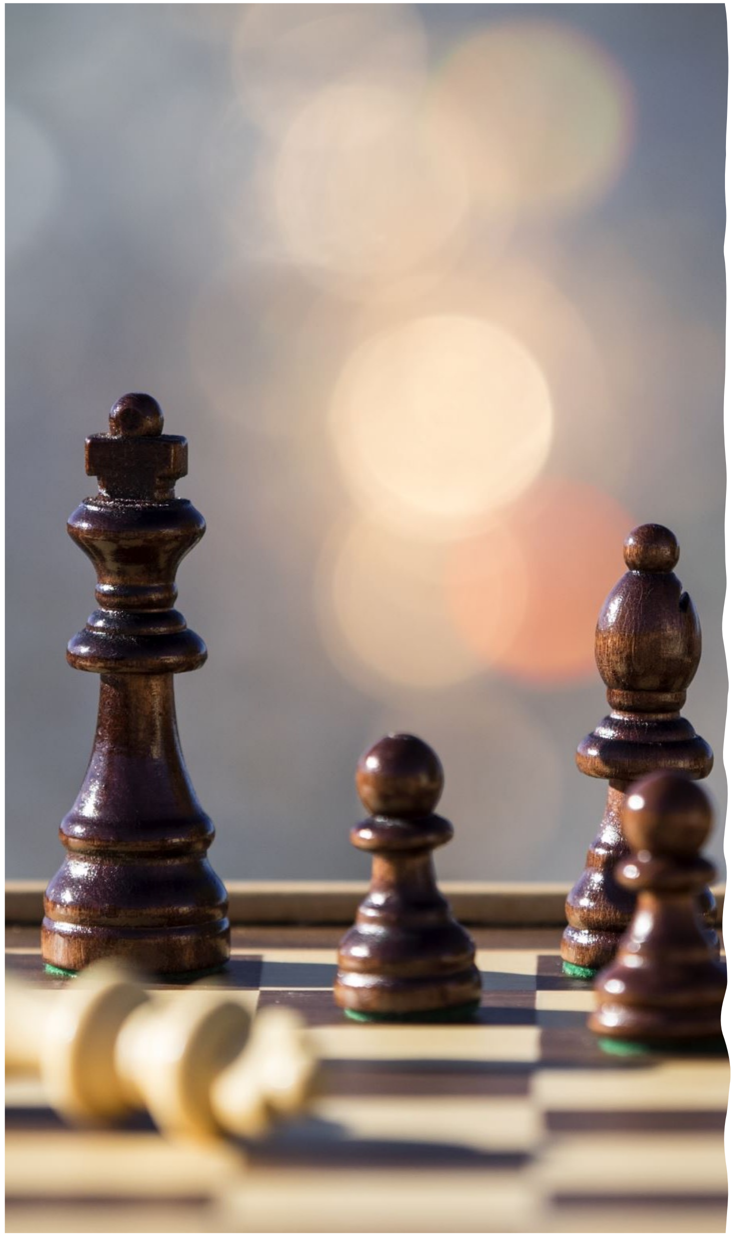
Taint Policies

- 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 ran-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
TPCON(t_a, t_v)	not 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

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

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

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

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

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

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

$TPMEM(t_1, t_2) = t_2$

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

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

$$\frac{\tau_\mu, \tau_\rho, \mu, \rho \vdash e \Downarrow \langle v, t \rangle. \quad \tau_\mu, \tau_\rho, \mu, \rho \vdash e_1 \Downarrow \langle v_1, t_1 \rangle \quad TPCOND(t, t_1) = T \; \iota = \Sigma[v]}{\tau_\mu, \tau_\rho, \Sigma, \mu, \rho, pc; if \; e \; then \; goto \; e_1 else \; goto \; e_2 \; \rightarrow \tau, \tau_\rho, \Sigma, \mu, \rho, v: \iota}$$

TPCOND(t, t_1) = NOT t_1

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

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

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;

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

τ_ρ	[x=T, y=T]
ρ	[x=40, y=45]

goto y

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

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

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

Load: take #1

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

$TPmem(t_a, t_v) = t_v$

Only the tainted value of the cell is considered

A main design issue

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

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

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

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

τ_μ	[$l_{(z+x)}$= F]
τ_ρ	[z=F]

A

τ_μ	[$l_{(z+x)}$= F]
τ_ρ	[z=F, x=T]

B

z+x = $l_{(z+x)}$

τ_μ	[$l_{(z+x)}$= F]
τ_ρ	[z=F, x=T, y = F]

C

Execute Statement at location $l_{(z+y)}$

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

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

Load: take #2

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

$TPmem(t_a, t_v) = t_a$ OR t_v

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

τ_μ	[$l_{(z+x)}$= F]
τ_ρ	[z=F]

A

τ_μ	[$l_{(z+x)}$= F]
τ_ρ	[z=F, x=T]

B

z+x = $l_{(z+x)}$

τ_μ	[$l_{(z+x)}$= F]
τ_ρ	[z=F, x=T, **y = T**]

C

ERR

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

TPmem(T, τ_μ($l_{(z+x)}$)) 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
```

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
```

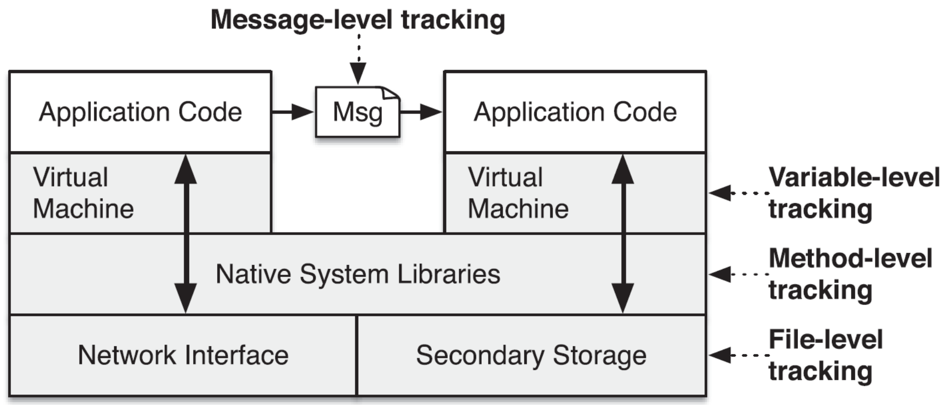
Control Flow Taint

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**
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

OCAML SIMULATION

As usual … the AST

```
type expr =
  | EInt of int
  | EBool of bool
  | Var of string
  | Let of string * expr * expr          (* let x = e1 in e2 *)
  | Prim of string * expr * expr           (* binop e1 e2 *)
  | If of expr * expr * expr             (* if e1 then e2 else e3 *)
  | Fun of string * expr                (* param identifier * funct body *)
  | Call of expr * expr                (* fun identifier * param *)
  | GetInput of expr                   (* function that takes input, taint source*)
```

Run Time Values … standard

```
type value =
  | Int of int
  | Bool of bool
  | Closure of string * expr * value env
```

Environment: handling bindings and taint

```
(* environment *)
type 'v env = (string * 'v * bool) list

(* binding *)
let rec lookup env x =
  match env with
  | [] -> failwith (x ^ "not found")
  | (y, v, _) :: r -> if x = y then v else lookup r x

(* taintness of a variable *)
let rec t_lookup env x =
  match env with
  | [] -> failwith (x ^ "not found")
  | (y, _, t) :: r -> if x = y then t else t_lookup r x
```

The environment maps variables to pairs consisting of a value and taint status

Interpreter

```
let rec eval (e : expr) (env:value env) (t : bool) : value * bool =
 match e with
 | EInt n -> (Int n, t)
 | EBool b -> (Bool b, t)
 | Var x -> (lookup env x, t_lookup env x)
 | Prim (op, e1, e2) ->
  begin
   let v1, t1 = eval e1 env t in
   let v2, t2 = eval e2 env t in
    match (op, v1, v2) with
    (* taintness of binary ops is given by the OR of the taintness of the args *)
    | "*", Int i1, Int i2 -> (Int (i1 * i2), t1 || t2)
    | "+", Int i1, Int i2 -> (Int (i1 + i2), t1 || t2)
    | "-", Int i1, Int i2 -> (Int (i1 - i2), t1 || t2)
    | "=", Int i1, Int i2 -> (Bool (if i1 = i2 then true else false), t1 || t2)
    | "<", Int i1, Int i2 -> (Bool (if i1 < i2 then true else false), t1 || t2)
    | ">", Int i1, Int i2 -> (Bool (if i1 > i2 then true else false), t1 || t2)
    | _, _, _ -> failwith "Unexpected primitive."
  end
```

Interpreter (cont.)

```
| If (e1, e2, e3) ->
    begin
     let v1, t1 = eval e1 env t in
     match v1 with
       | Bool true -> let v2, t2 = eval e2 env t in (v2, t1 || t2)
       | Bool false -> let v3, t3 = eval e3 env t in (v3, t1 || t3)
       | _ -> failwith "Unexpected condition."
    end
```

Interpreter (cont)

```
| Fun (f_param, f_body) -> (Closure (f_param, f_body, env), t)
| Call (f, param) ->
   let f_closure, f_t = eval f env t in
   begin
     match f_closure with
     | Closure (f_param, f_body, f_dec_env) ->
         let f_param_val, f_param_t = eval param env t in
           let env' = (f_param, f_param_val, f_param_t)::f_dec_env in
             let f_res, t_res = eval f_body env' t
               in (f_res, f_t || f_param_t || t_res)

     | _ -> failwith "Function expected error"
   end
```

Interpreter (cont)

```
| Fun (f_param, f_body) -> (Closure (f_param, f_body, env), t)
| Call (f, param) ->
   let f_closure, f_t = eval f env t in
   begin
     match f_closure with
     | Closure (f_param, f_body, f_dec_env) ->
         let f_param_val, f_param_t = eval param env t in
           let env' = (f_param, f_param_val, f_param_t)::f_dec_env in
             let f_res, t_res = eval f_body env' t
               in (f_res, f_t || f_param_t || t_res)

     | _ -> failwith "Function expected error"
   end
```

SPOT
THE ERROR

Interpreter (cont.)

```
| GetInput(e) -> eval e env true
```