

Information flow analysis

Thomas Jensen

SOS, Master Recherche Science Informatique, U. Rennes

Slides by Delphine Demange, David Pichardie and Thomas Jensen

Secure information flow

Overall goal : prevent secret (confidential, private, . . .) data to leak to an attacker.

Technique : follow the flow of secret data during execution

- ▶ **Dynamically** : guarantee the security of an execution, using a security monitor.
- ▶ **Statically** : analyse (prove) the security of the program before execution.

In this lecture :

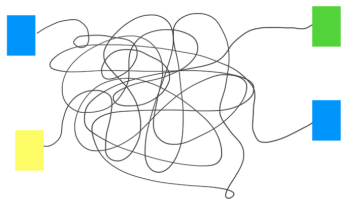
- ▶ what does it mean for a program to **leak** a secret?
- ▶ taint analysis : monitoring information flow,
- ▶ a type system for proving information flow security,
- ▶ different forms of leakage,
- ▶ how to **de-classify** information securely.

Taint analysis : monitoring information flow

Monitoring information flow

Goal : guarantee that secret information is not leaked.

Method : execute program on values tagged (“tainted”) with security labels.



Enforcement : when an illegal flow is detected, stop execution.

A lattice of security levels

Information flow is defined for arbitrary *lattices of security levels*.

We consider here only two security levels :

$$\begin{array}{c} H \\ | \\ L \end{array}$$

- ▶ H stands for “high security : data is secret
- ▶ L stands for “low security” : data is public.

We write \sqsubseteq for the partial order and \sqcup for the least upper bound.

Tainting

In dynamic information flow analysis, values are **tainted** with security levels.

$$42^H \quad true^L$$

When combining values, their taint gets combined as well, eg,

$$42^H + 17^L = 59^H$$

The tainting is $H \sqcup L = H$ to reflect that the result is computed (partially) from secret information.

Dynamic Information Flow analysis

Why dynamic analysis?

- ▶ Static analysis may be imprecise/hard (JavaScript, binary code)
- ▶ Programs may be unsafe — **but certain executions may be safe.**

Example :

```
if test1 then tmp := secret else skip;  
if test2 then x := tmp else skip;
```

Secure, if test1 and test2 cannot be true in the same execution.

Three properties of run-time monitors

Three important properties of Dynamic Information Flow control (and run-time monitors in general) :

Soundness :

executions with security leaks should be stopped.

Transparency :

the result of a monitored execution should be the same as a run without information flow monitoring.

Permissiveness :

the monitor should not halt a secure execution.

Under-tainting

Some taint analyses are **unsound** on purpose. They “under-taint”.

Challenge : assignment to a variable under secret control :

```
x := false;  
if secret then  
  x := true
```

This program copies secret into x via an **indirect flow**.

Two options :

- ▶ Variables assigned under secret control must be tainted “secret”.
- ▶ Ignore, and accept an unsound analysis.

But knowing when you are under secret control is not evident in un-structured languages.

A problem with indirect flows

Just tainting variables assigned under secret control is **not enough** to ensure security.

This program uses indirect flows to **copy the secret value in variable x into z** .

```
y := true;
z := true;
if (x)
  y := false;
if (y)
  z := false;
return z;
```

A problem with indirect flows

Just tainting variables assigned under secret control is **not enough** to ensure security.

This program uses indirect flows to **copy the secret value in variable x into z** .

```

y := true;
z := true;
if (x)
  y := false;
if (y)
  z := false;
return z:

```

Executions with

```

x = false

y := true;
z := true;
-
-
pc := L
z := false;
return false

```

A problem with indirect flows

Just tainting variables assigned under secret control is **not enough** to ensure security.

This program uses indirect flows to **copy the secret value in variable x into z** .

```

y := true;
z := true;
if (x)
  y := false;
if (y)
  z := false;
return z;

```

Executions with

$x = \text{false}$

```

y := true;
z := true;
-
-
pc := L
z := false;
return false

```

$x = \text{true}$

```

y := true;
z := true;
pc = H
y := false
-
-
return true

```

No-sensitive-upgrade

One possible solution : The **Principle of no-sensitive-upgrade** :

No assignment to public variables when control depends on secret values.

On our example, this gives :

	<code>x = false</code>	<code>x = true</code>
<code>y := true;</code>	<code>y := true;</code>	<code>y := true;</code>
<code>z := true;</code>	<code>z := true;</code>	<code>z := true;</code>
<code>if (x)</code>	-	<code>pc = H</code>
<code>y := false;</code>	-	<code>halt</code>
<code>if (y)</code>	<code>pc := L</code>	
<code>z := false;</code>	<code>z := false;</code>	
<code>return z;</code>	<code>return false</code>	

Enough to ensure security (more precisely, **termination-insensitive non-interference**—see later), but **quite restrictive**.

Semantics of dynamic information flow

Dynamic information flow analysis as a big-step operational semantics.

- ▶ Values are tagged with security labels
- ▶ Variables are not tagged — they are implicitly tagged through the values that they contain.

Semantic domains :

$$k, l \in \mathbf{Lab} = \{L, H\}$$

$$v \in \mathbf{Val} = \mathbb{Z} \times \mathbf{Lab}$$

$$\sigma \in \mathbf{State} = \mathbf{Var} \rightarrow \mathbf{Val}$$

Two evaluation relations :

- ▶ for expressions, $e, \sigma \Downarrow_{pc} n^l$
- ▶ for statements, $s, \sigma \Downarrow_{pc} \sigma_1$

Here, the pc indicates the label of the program counter.

Semantics of dynamic information flow I

Expressions

$$\text{Cst} \frac{}{n, \sigma \Downarrow_{pc} n^{pc}} \quad \text{Var} \frac{\sigma(x) = v^k}{x, \sigma \Downarrow_{pc} v^{k \sqcup pc}}$$

$$\text{Binop} \frac{e_i, \sigma \Downarrow_{pc} v_i^{k_i} \quad i = 1, 2}{e_1 + e_2, \sigma \Downarrow_{pc} (v_1 + v_2)^{k_1 \sqcup k_2}}$$

Semantics of dynamic information flow II

Statements

$$\text{Seq} \frac{s_1, \sigma \Downarrow_{pc} \sigma_1 \quad s_2, \sigma_1 \Downarrow_{pc} \sigma_2}{s_1; s_2, \sigma \Downarrow_{pc} \sigma_2}$$

$$\text{If-T} \frac{e, \sigma \Downarrow_{pc} \text{true}^k \quad s_1, \sigma \Downarrow_{pc \sqcup k} \sigma_1}{\text{if } e \text{ then } s_1 \text{ else } s_2, \sigma \Downarrow_{pc} \sigma_1} \quad \text{If-F} \frac{e, \sigma \Downarrow_{pc} \text{false}^k \quad s_2, \sigma \Downarrow_{pc \sqcup k} \sigma_2}{\text{if } e \text{ then } s_1 \text{ else } s_2, \sigma \Downarrow_{pc} \sigma_2}$$

$$\text{While-T} \frac{\begin{array}{c} e, \sigma \Downarrow_{pc} \text{true}^k \\ s, \sigma \Downarrow_{pc \sqcup k} \sigma_1 \\ \text{while } e \text{ do } s, \sigma_1 \Downarrow_{pc \sqcup k} \sigma_n \end{array}}{\text{while } e \text{ do } s, \sigma \Downarrow_{pc} \sigma_n} \quad \text{While-F} \frac{e, \sigma \Downarrow_{pc} \text{false}^k}{\text{while } e \text{ do } s, \sigma \Downarrow_{pc} \sigma}$$

Semantics of dynamic information flow III

The principle of no-sensitive upgrade is formalized in the rule for assignment.

Assignment

$$\text{Asg-NSU} \frac{e, \sigma \Downarrow_{pc} v^k \quad \sigma(x) = w^l \quad pc \sqsubseteq l}{x := e, \sigma \Downarrow_{pc} \sigma[x \mapsto v^k]}$$

This rule relies on the following invariant of the IF semantics for expressions :

$$\text{if } e, \sigma \Downarrow_{pc} v^k \text{ then } pc \sqsubseteq k.$$

Ensures that the label of the value in x stays above the pc label.

Summary on dynamic taint analysis

Taint analysis :

- ▶ track the flow of secret values during execution,
- ▶ in order to avoid making secret data public.

Requires

- ▶ classifying input and output as public or secret,
- ▶ rules for taint propagation for each instruction.

Issues :

- ▶ under-tainting, if ignoring eg indirect flows,
- ▶ over-tainting, because of approximations.

Can also serve to make [integrity analysis](#)

- ▶ track data coming from public (untrusted) sources,
- ▶ can be used to guarantee control flow integrity :
will forbid jmp to addresses dependent on user input.

Static information flow analysis

Secure information flow

Goal : define a static analysis that guarantees that a program only has secure information flow.

There is a **flow of information** from x to y if the value of the variable y depends on the value of the variable x .

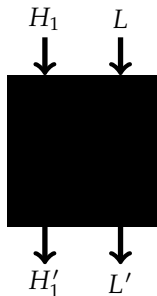
If x is of level k_x and y of level k_y , then the flow from x to y is

- ▶ secure if $k_x \sqsubseteq k_y$
- ▶ illegal if $k_x \not\sqsubseteq k_y$

How can we define “depends on” formally?

Non-interference

"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982

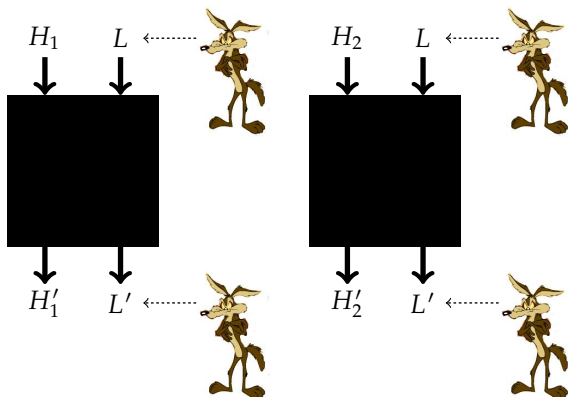


High (H) = confidential

Low (L) = public

Non-interference

"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High (H) = confidential

Low (L) = public

Secure programs

The set of variables is partitioned into two disjoint sets :

- ▶ \mathbb{V}_H : high (or secret) variables
- ▶ \mathbb{V}_L : low (or public) variables

Intuitively¹, a program is *secure* (or *non interfering*) if the **final** values of **low** variables do not depend on the **initial** values of the **high** variables.

Examples : are these programs secure or not?

- ① `h := 1`
- ② `l := h`
- ③ `if (h>0) then {l := 1} else {l := 2}`
- ④ `while (h) do { l := l+1 }; l := 0`

We distinguish between **direct** and **indirect** flows

1. This notion will be defined formally when presenting the semantics of the language.

Program syntax

WHILE language with mixed arithmetic and boolean expressions.

Expr ::=	n	$n \in \mathbb{Z}$
	x	$x \in \mathbb{V}_H \uplus \mathbb{V}_L$
	Expr o Expr	$o \in \{+, -, \times, \dots\}$
	Expr c Expr	$c \in \{=, \neq, <, \leq, \dots\}$
	Expr b Expr	$b \in \{\text{and, or}\}$
		$bop \in o \cup c \cup b$
Stm ::=	$x := \text{Expr}$	
	if Expr then Stm else Stm	
	while Expr do Stm	
	Stm ; Stm	

The set of variables is partitioned into two disjoint sets :

- ▶ \mathbb{V}_H : high (or secret) variables
- ▶ \mathbb{V}_L : low (or public) variables

A simple information flow type system (1/3)

We will present a simple information flow type system² and prove it enforces a semantic non-interference property on well-typed programs.

Typing judgment for expressions : $e \in \mathbf{Expr}$, $\tau \in \{L, H\}$

$$\vdash e : \tau$$

Meaning : the expression e **depends only** on variables of level τ or lower.

Typing rules : (τ_x stands for the ℓ such that $x \in \mathbb{V}_\ell$)

$$\text{CONST} \frac{}{\vdash n : L} \quad \text{VAR} \frac{}{\vdash x : \tau_x} \quad \text{BINOP} \frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash e_1 \text{ *bop* } e_2 : \tau}$$

$$\text{EXP-SUBTYP} \frac{\vdash e : \tau_1 \quad \tau_1 \sqsubseteq \tau_2}{\vdash e : \tau_2}$$

2. equivalent to D. Volpano and G. Smith, *A Type-Based Approach to Program Security*, Theory and Practice of Software Development, 1997.

Example

Assuming $h \in \mathbb{V}_H$, a type derivation for $\vdash h + 1 : H$

$$\text{BINOP} \frac{\text{VAR} \frac{h \in \mathbb{V}_H}{\vdash h : H} \quad \text{EXP-SUBTYP} \frac{\text{CONST} \frac{}{\vdash 1 : L} \quad L \sqsubseteq H}{\vdash 1 : H}}{\vdash h + 1 : H}$$

A simple information flow type system (2/3)

Typing judgment for statements : $S \in \mathbf{Stm}$, $\tau_{pc} \in \{L, H\}$

$$\tau_{pc} \vdash S$$

Intuition :

- ▶ τ_{pc} , the program-counter label, tracks the dependencies of the current program point (to forbid indirect flows).
- ▶ the variables **modified by** statement S are of level τ_{pc} or higher.

Ensures : well-typed programs have no illicit flows.

Typing rules : ($\tau_x = \ell$ means $x \in \mathbb{V}_\ell$)

$$\text{ASSIGN} \frac{\vdash e : \tau \quad \tau \sqcup \tau_{pc} \sqsubseteq \tau_x}{\tau_{pc} \vdash x := e}$$

$$\text{SEQ} \frac{\tau_{pc} \vdash S_1 \quad \tau_{pc} \vdash S_2}{\tau_{pc} \vdash S_1 ; S_2}$$

$$\text{IF} \frac{\vdash e : \tau \quad \tau \sqcup \tau_{pc} \vdash S_i \quad i = 1, 2}{\tau_{pc} \vdash \text{if } e \text{ then } S_1 \text{ else } S_2}$$

$$\text{WHILE} \frac{\vdash e : \tau \quad \tau \sqcup \tau_{pc} \vdash S}{\tau_{pc} \vdash \text{while } e \text{ do } S}$$

A simple information flow type system (3/3)

Sub-typing rule :

$$\text{STM-SUBTYP} \frac{H \vdash S}{L \vdash S}$$



The subtyping relation on statements is *contravariant*!

Intuition : typing S under a high context guarantees that all assignments are to variables of high level, so OK (but not precise) to say that it assigns to variables of high **or** low levels.

More intuition : typing S under a high context is more difficult (because it limits direct and indirect flows), so S is shown to be "more secure".

Exercise

Exercise (Typing derivations)

Assuming $l \in \mathbb{V}_L$ and $h \in \mathbb{V}_H$, try to type the following statements (give a type derivation, if possible) :

- ▶ `if (l) then h := l else l := 0`
- ▶ `if (h) then h := l else l := 0`
- ▶ `if (h) then l := 0 else l := 0`

Type soundness

We want to prove that the type system is indeed ensuring non-interference.

To do so :

- ▶ define the semantics of the language
- ▶ define the semantic property we want to prove (non-interferent program)
- ▶ prove that all well-typed programs satisfy the property

A natural semantics

State = **Var** \rightarrow \mathbb{Z}
 $\llbracket \cdot \rrbracket \in$ **Expr** \rightarrow **State** \rightarrow \mathbb{Z} (semantics of expressions)
 $(\cdot, \cdot) \Downarrow \cdot \subseteq$ (**Stm** \times **State**) \times **State** (semantics of statements)

$$\llbracket n \rrbracket s = \mathcal{N}[\llbracket n \rrbracket]$$

$$\llbracket x \rrbracket s = s(x)$$

$$\llbracket e_1 + e_2 \rrbracket s = \llbracket e_1 \rrbracket s + \llbracket e_2 \rrbracket s$$

...

$$\frac{}{(x := e, s) \Downarrow s[x \mapsto \llbracket e \rrbracket s]} \qquad \frac{(S_1, s) \Downarrow s' \quad (S_2, s') \Downarrow s''}{(S_1; S_2, s) \Downarrow s''}$$

$$\frac{(S_1, s) \Downarrow s' \quad \llbracket e \rrbracket s = 1}{(\text{if } e \text{ then } S_1 \text{ else } S_2, s) \Downarrow s'} \qquad \frac{(S_2, s) \Downarrow s' \quad \llbracket e \rrbracket s = 0}{(\text{if } e \text{ then } S_1 \text{ else } S_2, s) \Downarrow s'}$$

$$\frac{(S, s) \Downarrow s' \quad (\text{while } e \text{ do } S, s') \Downarrow s'' \quad \llbracket e \rrbracket s = 1}{(\text{while } e \text{ do } S, s) \Downarrow s''} \qquad \frac{\llbracket e \rrbracket s = 0}{(\text{while } e \text{ do } S, s) \Downarrow s}$$

The observational power of an attacker

Here, we will consider that the attacker only sees low variables before and after executions.

We model his observational power with an *equivalence* relation between states.

$$\sim \subseteq \mathbf{State} \times \mathbf{State}$$

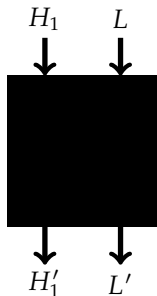
$$s_1 \sim s_2 \text{ iff } \forall x \in \mathbb{V}_L, s_1(x) = s_2(x)$$

Intuition : the attacker cannot distinguish between equivalent states.

NB : This relation can be extended to an arbitrary security lattice.

Non-interference

"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982

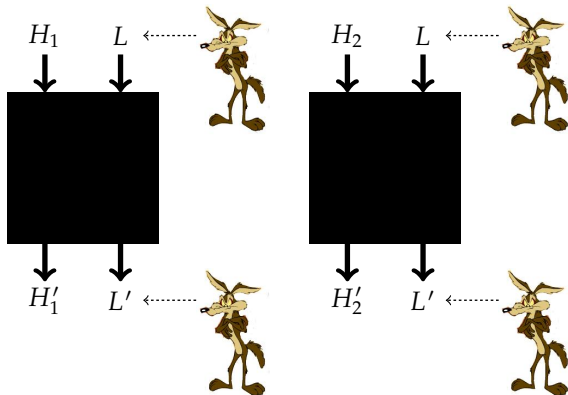


High (H) = confidential

Low (L) = public

Non-interference

"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982

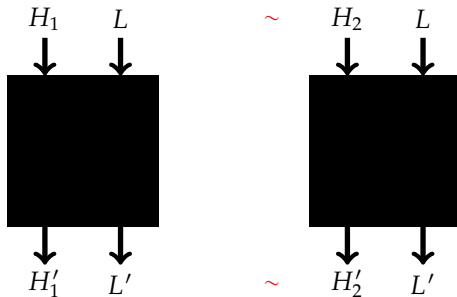


High (H) = confidential

Low (L) = public

Non-interference

"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High (H) = confidential

Low (L) = public

$$\forall s_1, s_2, s'_1, s'_2, \quad s_1 \sim s_2 \wedge (P, s_1) \Downarrow s'_1 \wedge (P, s_2) \Downarrow s'_2 \implies s'_1 \sim s'_2$$

Type soundness

Definition (Non-interference)

A statement S is said *non interferent* iff for all s_1, s_2 such that $s_1 \sim s_2$,

$$\left. \begin{array}{l} (S, s_1) \Downarrow s'_1 \\ (S, s_2) \Downarrow s'_2 \end{array} \right\} \text{implies } s'_1 \sim s'_2$$

Theorem (Type soundness)

Every typable statement (i.e. such that $\exists \tau_{pc}, \tau_{pc} \vdash S$) is non-interferent.

Type soundness proof : step 1

We need a new set of typing rules.

$$\text{CONST}' \frac{}{\vdash_s n : \tau} \quad \text{VAR}' \frac{\tau_x \sqsubseteq \tau'}{\vdash_s x : \tau'} \quad \text{BINOP}' \frac{\vdash_s e_1 : \tau \quad \vdash_s e_2 : \tau}{\vdash_s e_1 \text{ \textit{bop} } e_2 : \tau}$$

$$\text{ASSIGN}' \frac{\vdash_s e : \tau_x \quad \tau' \sqsubseteq \tau_x}{\tau' \vdash_s x := e} \quad \text{SEQ}' \frac{\tau \vdash_s S_1 \quad \tau \vdash_s S_2}{\tau \vdash_s S_1 ; S_2}$$

$$\text{IF}' \frac{\vdash_s e : \tau \quad \tau \vdash_s S_1 \quad \tau \vdash_s S_2 \quad \tau' \sqsubseteq \tau}{\tau' \vdash_s \text{ \textit{if} } e \text{ \textit{then} } S_1 \text{ \textit{else} } S_2}$$

$$\text{WHILE}' \frac{\vdash_s e : \tau \quad \tau \vdash_s S \quad \tau' \sqsubseteq \tau}{\tau' \vdash_s \text{ \textit{while} } e \text{ \textit{do} } S}$$

This type system is *syntax-directed* : at most one rule can be used for each program construct (expression or statement).

Type soundness proof : step 1

Lemma (Sub-typing property)

For all e, τ, τ' , $\vdash_s e : \tau$ and $\tau \sqsubseteq \tau'$ implies $\vdash_s e : \tau'$.

For all S, τ, τ' , $\tau' \vdash_s S$ and $\tau \sqsubseteq \tau'$ implies $\tau \vdash_s S$.

Proof. By induction on the typing judgment.

The new system is equivalent to the previous one.

Lemma

For all e, τ , $\vdash e : \tau$ implies $\vdash_s e : \tau$.

For all S, τ , $\tau \vdash S$ implies $\tau \vdash_s S$.

Proof. By induction on the typing judgment.

Lemma

For all e, τ , $\vdash_s e : \tau$ implies $\vdash e : \tau$.

For all S, τ , $\tau \vdash_s S$ implies $\tau \vdash S$.

Proof. By induction on the typing judgment.

Type soundness proof : step 2

Lemma (Low expressions)

For all $e \in \mathbf{Expr}$, if $\vdash_s e : L$, then for all $s_1, s_2 \in \mathbf{State}$, $s_1 \sim s_2$ implies $\llbracket e \rrbracket_{s_1} = \llbracket e \rrbracket_{s_2}$.

Proof. By induction on type derivation for e .

Lemma (Confinement of high statements)

For all $S \in \mathbf{Stm}$, and $s, s' \in \mathbf{State}$, if $(S, s) \Downarrow s'$ and $H \vdash_s S$, then $s \sim s'$.

Proof. By induction on the judgment $(S, s) \Downarrow s'$.

Theorem (Type soundness)

For all $S \in \mathbf{Stm}$, $s_1, s_2, s'_1, s'_2 \in \mathbf{State}$, $\tau_{pc} \in \{L, H\}$, if $s_1 \sim s_2$, $(S, s_1) \Downarrow s'_1$, $(S, s_2) \Downarrow s'_2$ and $\tau_{pc} \vdash_s S$ then $s'_1 \sim s'_2$.

Proof. By induction on the judgment $(S, s_1) \Downarrow s'_1$.

Be careful with the **while** case.

A few remarks on the type system

- ▶ Type checking is computable but non-interference is not.
Exercise : Give an example of non-interferent program that is not typable.
- ▶ The type system that we have defined is flow-insensitive.
Exercise : Give another example of a non-interferent program that is not typable because of this limitation.
- ▶ The correctness theorem only holds for terminating computations. This means that

```
if secret then skip else loop;
```

is secure.
- ▶ The attacker may have additional observation power : timing, power consumption, sound, ...
(see next section)

Variations on the theme of observation

Observational power of attacker

We have ignored some information channels :

- ▶ timing channels

```
if h>0 then skip else {<huge, non-interfering computation> }
```

measuring the run-time of this program may reveal secret informations.
See lecture on side-channels analysis later in the course.

- ▶ termination channels

```
while h>0 do skip
```

- ▶ power consumption (differential power attacks)

Timing channels

Transforming out timing leaks³

```

s := 1;
i := 0;
while (i < w) {
  if (k[i])
    r := (s*x) mod n
  else
    r := s;
  s := r*r;
  i := i+1
}      (The result is now in r)

```

Figure 11: An implementation of the modular exponentiation algorithm that leaks through timing.

Use padding with “blank” instructions to even out execution time in branches.

3. J. Agat, *Transforming out timing leaks*, POPL 2000

Timing channels

Transformed program :

```

s := 1;
i := 0;
while (i < w) {
  if (k[i])
    r := (s*x) mod n;
    skipAsn r s
  else {
    skipAsn r ((s*x) mod n);
    r := s
  };
  s := r*r;
  i := i+1
}

```

Figure 12: The output of our transformation: a secure implementation of the modular exponentiation algorithm.

Simple solution : cross-copying branches

More challenging : find cross-copying that minimises execution over-head.

Covert channels from power consumption

A bit more challenging : power consumption per processor clock cycle

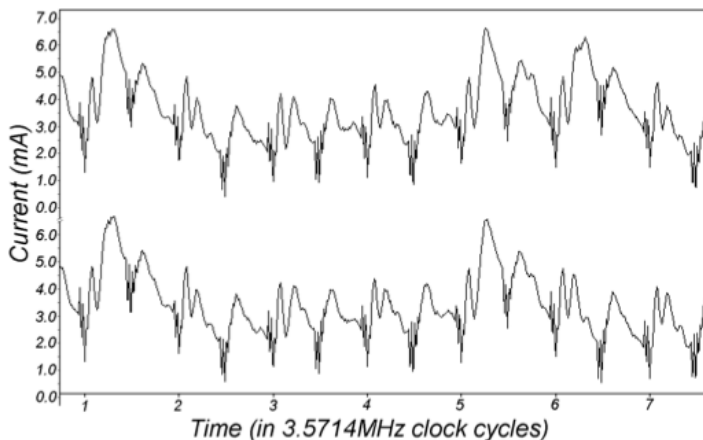


Figure – Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. CRYPTO '99.

JavaScript channels

In JavaScript, records are extensible.
Furthermore, the presence of fields can be tested.

```
myway:InfoFlow demange$ node
```

```
> var o = {}
```

```
> o.secret === undefined
```

```
true
```

```
> o.secret = 1
```

```
> o.secret === undefined
```

```
false
```

The **structure** of data can be used to transmit information!

Scheduler-based channels

Consider two threads

T1: `h := 0; l := h`

and

T2: `h := secret`

Separately, each thread is safe (T1 erases `h`).

Executed concurrently, they may leak the secret.

Implicit flows can also arise :

T1: `(if h > 0 then sleep(100) else skip); l := 1`

and

T2: `sleep(50); l := 0`

Most schedulers will leak `h` into `l`

Making executions **atomic** can remedy this — but is expensive.

Declassification

Giving (some) information away

Code should not leak sensitive information.

Non-interference is sometimes too strong a property.

Some applications **intentionally** leak some confidential information :

- ▶ password checking always reveals some secret
- ▶ statistics
- ▶ ...

Need to give away *some* information.

Need for controlled information release or **declassification**

Declassification

Distinguish several **dimensions**⁴ of declassification :

- ▶ **what** data can be declassified? (*e.g.*, the average of a salary data base)
- ▶ **who** can declassify? (and who can influence the decisions of declassification).
- ▶ **when** can data be declassified (*e.g.*, release highest bid in an auction with secret bids, non-interference “until”)?
- ▶ **where** can data be declassified (*e.g.*, after passing a down-grader)?

4. See Sabelfeld and Sands : *Dimensions of Declassification*, J. Comp Security.

Controlling information release

Declassification might compromise confidentiality.

Ensure that secrets are not leaked via release mechanisms.

Information release violates non-interference!

⇒ we cannot rely on previous type system to ensure security.

What security guarantees for programs with declassification?

An operator for declassification

We introduce a binary operator `declassify(exp, lvl)` that takes as arguments

- ▶ an expression *exp*
- ▶ a security level *lvl* such as high, low, ...

Intention : the information computed by *exp* can be declassified to the level *lvl*. For example, one would like the type system to accept

```
avg := declassify((h_1 + ... + h_n)/n, low)
```

where h_i are secrets, avg, n are public.

Rejected by non-interference.

How to ensure that we are not declassifying more than intended?

Delimited release

Principle : Only release declassified data and no further information

Intuition : Expression exp can be declassified in statement S if making the value of exp visible does not reveal information about secret input.

Formally : All environments that are indistinguishable through exp are indistinguishable through S .

Definition : exp is safe to declassify in S if

$$s_1 \sim s_2 \text{ and } \llbracket exp \rrbracket s_1 = \llbracket exp \rrbracket s_2 \text{ and } (S, s_1) \Downarrow s'_1 \text{ and } (S, s_2) \Downarrow s'_2$$

implies

$$s'_1 \sim s'_2$$

Question : Are non-interferent programs secure wrt. delimited release?

Examples

Exercise

Are the following programs obeying delimited release?

- ▶ `avg := declassify((h_1 + ... + h_n)/n, low)`
- ▶ `tmp := h_1; h_1 := h_2; ... h_n := tmp;`
`avg := declassify((h_1 + ... + h_n)/n, low)`
- ▶ `h_2:=h_1;...; h_n:=h_1;`
`avg:=declassify((h_1+...+h_n)/n,low);`

Examples

Exercise

Are the following programs obeying delimited release?

- ▶ `avg := declassify((h_1 + ... + h_n)/n, low)`
- ▶ `tmp := h_1; h_1 := h_2; ... h_n := tmp;`
`avg := declassify((h_1 + ... + h_n)/n, low)`
- ▶ `h_2:=h_1;...; h_n:=h_1;`
`avg:=declassify((h_1+...+h_n)/n,low);`

Example 1 : accepted. Why?

Examples

Exercise

Are the following programs obeying delimited release?

- ▶ `avg := declassify((h_1 + ... + h_n)/n, low)`
- ▶ `tmp := h_1; h_1 := h_2; ... h_n := tmp;`
`avg := declassify((h_1 + ... + h_n)/n, low)`
- ▶ `h_2:=h_1;...; h_n:=h_1;`
`avg:=declassify((h_1+...+h_n)/n,low);`

Example 1 : accepted. Why?

Example 2 : accepted. Why?

Examples

Exercise

Are the following programs obeying delimited release?

- ▶ `avg := declassify((h_1 + ... + h_n)/n, low)`
- ▶ `tmp := h_1; h_1 := h_2; ... h_n := tmp;`
`avg := declassify((h_1 + ... + h_n)/n, low)`
- ▶ `h_2:=h_1;...; h_n:=h_1;`
`avg:=declassify((h_1+...+h_n)/n,low);`

Example 1 : accepted. Why?

Example 2 : accepted. Why?

Example 3 : rejected. Why?

To see this, set

$$s_1 = [h_1 = 2, h_2 = 4, avg = 0] \text{ and } s_2 = [h_1 = 4, h_2 = 2, avg = 0]$$

Then `declassify((h_1 + ... + h_n)/n, low)` has value 3 in s_1 and s_2 but leads to final states where observable variable `avg` has different values.

Type system for declassification

Idea : prevent new information from flowing into variables used in declassifying expressions

Intuition : `exp` should not contain high variables other than `h` in

`h := exp ; ... ; declassify(h,low);`

Type system

- ▶ $\vdash e : l, D$ where l is a security level and D the variables used in declassified expressions in e .
- ▶ $\tau_{pc} \vdash S : (U, D)$ where U are variables being updated in S and D variables used in declassification operations in S .
- ▶ declassified variables may not be updated prior to declassification

Type system for declassification

Typing rules (a selection)

$$\text{EXP-DECLASS} \frac{\vdash e : l', D}{\vdash \mathbf{declassify}(e, l) : l, \text{Vars}(e)}$$

$$\text{CMD-ASG} \frac{\vdash e : l', D \quad l' \cup \tau_{pc} \sqsubseteq \tau_x}{\tau_{pc} \vdash x := e : \{x\}, D}$$

$$\text{CMD-SEQ} \frac{\tau_{pc} \vdash S_1 : U_1, D_1 \quad \tau_{pc} \vdash S_2 : U_2, D_2 \quad U_1 \cap D_2 = \emptyset}{\tau_{pc} \vdash S_1; S_2 : U_1 \cup U_2, D_1 \cup D_2}$$

References

- ▶ D. Volpano and G. Smith, A Type-Based Approach to Program Security. Theory and Practice of Software Development, 1997.
- ▶ A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. IEEE Journal on selected areas in communications. Vol. 21, NO. 1, 2003.
- ▶ F. Pottier and V. Simonet. Information flow inference for ML. POPL 2002.
- ▶ J. Agat, Transforming out timing leaks. POPL 2000.
- ▶ B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09). 2009
- ▶ A. Sabelfeld, A. C. Myers. A Model for Delimited Information Release. Software Security - Theories and Systems. LNCS 2004.
- ▶ A. C. Myers, A. Sabelfeld, and S. Zdancewic. 2006. Enforcing robust declassification and qualified robustness. J. Comput. Secur. 14, 2 2006, 157-196.