Information flow

Thomas Jensen

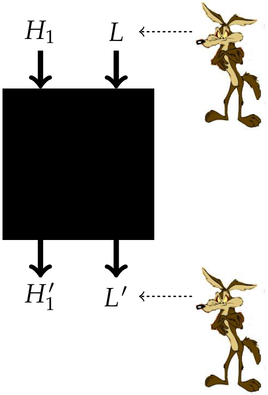
INRIA

Ecole Jeunes Chercheurs en Programmation
May 2013

Information flow type system

Non-interference

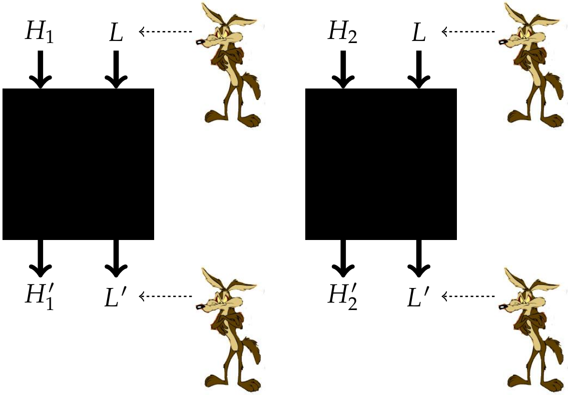
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidential   Low = public

Non-interference

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



High = confidential Low = public

Program syntax

We consider a standard WHILE language but we mix arithmetic and boolean expressions.

$$
\begin{array}{rcl}
\textbf{Expr} ::= & n & \\
| & x & \\
| & \textbf{Expr}\ o\ \textbf{Expr} & \\
| & \textbf{Expr}\ c\ \textbf{Expr} & \\
| & \textbf{Expr}\ b\ \textbf{Expr} & \\
\textbf{Stm} ::= & x := \textbf{Expr} & \\
| & \texttt{if}\ \textbf{Expr}\ \texttt{then}\ \textbf{Stm}\ \texttt{else}\ \textbf{Stm} & \\
| & \texttt{while}\ \textbf{Expr}\ \texttt{do}\ \textbf{Stm} & \\
| & \textbf{Stm}\ ;\ \textbf{Stm} &
\end{array}
$$

$$
\begin{array}{l}
n \in \mathbb{Z} \\
x \in \mathbb{V}_H \uplus \mathbb{V}_L \\
o \in \{+, -, \times, \ldots\} \\
c \in \{=, \neq, <, \leqslant, \ldots\} \\
b \in \{\text{and}, \text{or}\}
\end{array}
$$

The set of variables is partitioned into two disjoint sets :

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

Secure programs

Intuitively [1], a program is *secure* (or *non interferent*) if the final values of low variables do not depend on the initial values of the high variables.

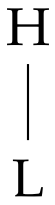
Examples : are these programs secure or not ?

1. `h := l`
2. `l := h`
3. `if (h1>h2) then {l := 1} else {l := 2}`
4. `while (h) do { l := l+1 }; l := 0`

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

A lattice of security levels

We consider here only two kind of informations (low and high), but the information flow policy can be defined as a lattice of *security levels*.



We write $\sqsubseteq$ for the corresponding partial order.

We say 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 (resp. y) is of level k_x (resp. k_y), the flow is

- licit if $k_x \sqsubseteq k_y$
- illicit if $k_x \not\sqsubseteq k_y$

A simple information flow type system (1/2)

We will present a simple information flow type system [2] and prove it enforces the 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 of variable of level τ or lower.

Typing rules :

$$\text{CONST}\ \frac{}{\vdash n : L} \qquad \text{VAR}\ \frac{x \in \mathbb{V}_\tau}{\vdash x : \tau} \qquad \text{BINOP}\ \frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash e_1\ o\ e_2 : \tau}$$

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

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

Example

A type derivation for $\vdash h + 1 : H$

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

A simple information flow type system (2/2)

Typing judgment for statements : $S \in \textbf{Stm}, \tau \in \{L, H\}$

$$\vdash S : \tau$$

Meaning : the only variables modified by statement S are of level τ or higher.

Typing rules :

$$\text{ASSIGN}\ \frac{x \in \mathbb{V}_\tau \qquad \vdash e : \tau}{\vdash x := e : \tau} \qquad \text{SEQ}\ \frac{\vdash S_1 : \tau \qquad \vdash S_2 : \tau}{\vdash S_1\ ;\ S_2 : \tau}$$

$$\text{IF}\ \frac{\vdash e : \tau \qquad \vdash S_1 : \tau \qquad \vdash S_2 : \tau}{\vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \tau} \qquad \text{WHILE}\ \frac{\vdash e : \tau \qquad \vdash S : \tau}{\vdash \texttt{while}\ e\ \texttt{do}\ S : \tau}$$

$$\text{STM-SUBTYP}\ \frac{\vdash S : \tau_2 \qquad \tau_1 \sqsubseteq \tau_2}{\vdash S : \tau_1}$$

The subtype relation on statements is *contravariant* !

Exercise

Try to type the following statements (give a type derivation, if possible) :

$$\texttt{if}\ (l)\ \texttt{then}\ h := l\ \texttt{else}\ l := 0$$

$$\texttt{if}\ (h)\ \texttt{then}\ h := l\ \texttt{else}\ l := 0$$

A natural semantics

$$\begin{array}{rcl} \textbf{State} & = & \textbf{Var} \to \mathbb{Z} \\ [\![\cdot]\!] & \in & \textbf{Expr} \to \textbf{State} \to \mathbb{Z} \ \text{(semantics of expression)} \\ (\cdot,\cdot) \Downarrow \cdot & \subseteq & (\textbf{Stm} \times \textbf{State}) \times \textbf{State} \ \text{(semantics of statement)} \end{array}$$

$$\begin{array}{rcl} [\![n]\!]s & = & \mathcal{N}[\![n]\!] \\ [\![x]\!]s & = & s(x) \\ [\![e_1 + e_2]\!]s & = & [\![e_1]\!]s + [\![e_2]\!]s \\ & \cdots & \end{array}$$

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

$$\frac{(S_1, s) \Downarrow s' \qquad [\![e]\!]s = 1}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s) \Downarrow s'} \qquad \frac{(S_2, s) \Downarrow s' \qquad [\![e]\!]s = 0}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s) \Downarrow s'}$$

$$\frac{(S, s) \Downarrow s' \qquad (\texttt{while}\ e\ \texttt{do}\ S, s') \Downarrow s'' \qquad [\![e]\!]s = 1}{(\texttt{while}\ e\ \texttt{do}\ S, s) \Downarrow s''} \qquad \frac{[\![e]\!]s = 0}{(\texttt{while}\ e\ \texttt{do}\ S, s) \Downarrow s}$$

The observational power of an attacker

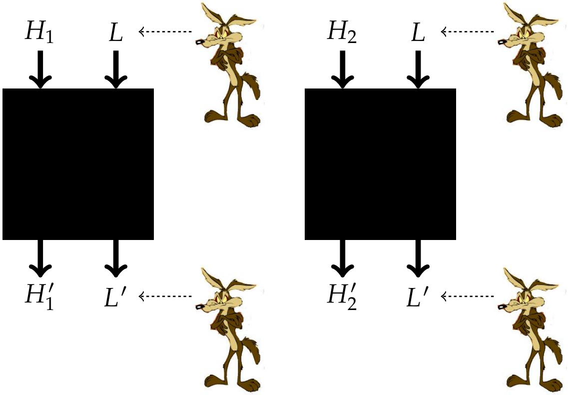
The attacker only sees low variable before and after executions.

We model his observational power with an *indistinguishability* relation $\sim \subseteq \textbf{State} \times \textbf{State}$ between states.

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

Non-interference

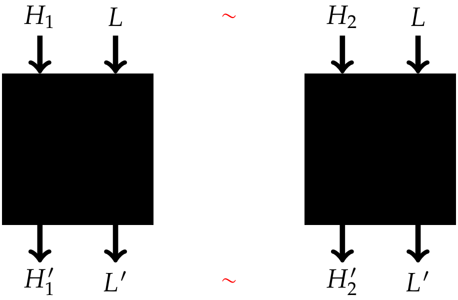
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidential Low = public

Non-interference

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



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

High = confidential   Low = public

Type soundness

A statement S is said *non interferent* iff $\left.\begin{array}{r} s_1 \sim s_2 \\ (S, s_1) \Downarrow s_1' \\ (S, s_2) \Downarrow s_2' \end{array}\right\}$ implies $s_1' \sim s_2'$

Theorem

Every typable statement (i.e. such that $\exists \tau, \vdash S : \tau$) is non interferent.

A few remarks on the type system

- Type checking is computable but non-interference is not
  Exercice : give an example of non-interferent program that is not typable.
- The attacker may have additional observation power (timing, power consumption)
- Non-inteference is sometimes too strong a property.
  Example : a password checker always reveals some secret.
  Need to give away *some* information.

Information flow challenge

http://ifc.hvergi.net/

Declassification

Giving (some) information away

Code should not leak sensitive information.

However, some applications **intentionally** leak some confidential information :

- password checking
- statistics
- ...

Need for controlled information release or **declassification**

Declassification

Distinguish several **dimensions**[3] of declassification :

- **what** data can be declassified ?
- **who** can declassify ?
- **when** can data be declassified (non-interference “until”) ?
- **where** can data be declassified (*e.g.*, after passing a down-grader) ?

3. 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 !

$\Rightarrow$ 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...

and declassifies the result of *exp* to the level *lvl*.

For example, the type system can now accept [4]

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

Rejected by non-interference.

But how to ensure that we are not declassifying more than intended ?

4. $\mathtt{h}_i$ are secrets, `avg`, `n` are low variables

Delimited release

Principle :

Only release declassified data and no further information

Intuition : expression *exp* can be declassified in statement S if

all environments that are indistinguishable through *exp*
are indistinguishable through S.

Definition :
Assume

$$s_1 \sim s_2 \text{ and } (S, s_1) \Downarrow s_1' \text{ and } (S, s_2) \Downarrow s_2'.$$

Then we must have

$$[\![exp]\!]s_1 = [\![exp]\!]s_2 \Rightarrow s_1' \sim s_2'.$$

Examples

Delimited release accepts

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

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

Rejects

```
h2:=h1;...; hn:=h1;
avg:=declassify((h1+...+hn)/n,low);
```

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 $\vdash S : (U, D)$ where U are variables being updated in S and D variables used in declassification operations in S.

Typing rules

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

$$\text{CMD-ASG}\ \frac{\vdash e : l', D}{\vdash x := e : \{x\}, D}$$

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

Variations on the theme of observation

Observational power of attacker

We have ignored some information channels :

- timing channels

  ```
  if h>0 then l:=0 else {<huge, non-interfereing computation>; l:=0 }
  ```

  measuring the run-time of this program may reveal secret informations.
- termination channels

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

- power consumption (differential power attacks)

Timing channels

Transforming out timing leaks [5]

```
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.

5. 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
}
```

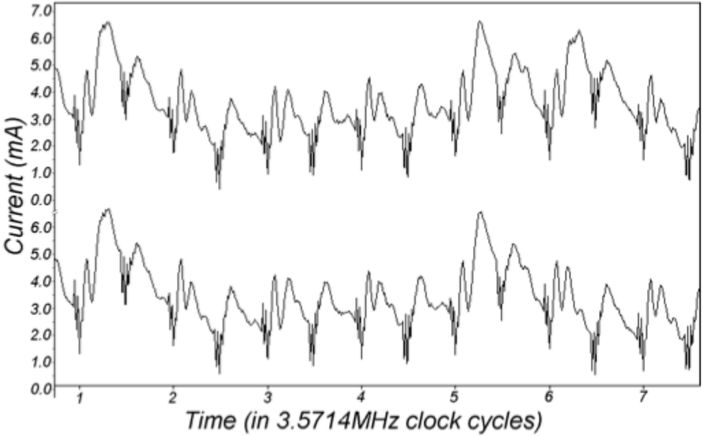
Figure 12: The output of our transformation: a secure implementation of the modular exponentiation algorithm.

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



Executions are identical except for the jump instruction at cycle 6.

JavaScript channels

In JavaScript, records are extensible :

```
o = { };
o[h] = 1;
o.has(true)?
```

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.
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 executioons atomic can remedy this — but is expensive.

Dynamic information flow analysis

Dynamic Information Flow analysis

Programs may be unsafe — but certain executions may be safe.

Dynamic information flow analysis

- execute program on labelled data
- track information flows (both explicit and implicit)
- can be more precise than static techniques
- alleviates the need for supporting static analyses (think JavaScript !)

Comes in different flavors

- taint analysis
- information flow monitors
- secure multi-execution

All have to address : non-interference is a property of **sets of traces**.

Monitoring information flow

Where monitoring pays off :

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

Non-interference, if `test1` and `test2` cannot be true at the same time.

Various ways of stopping the execution (halt, output of dummy values) that may create other channels

Monitoring information flow

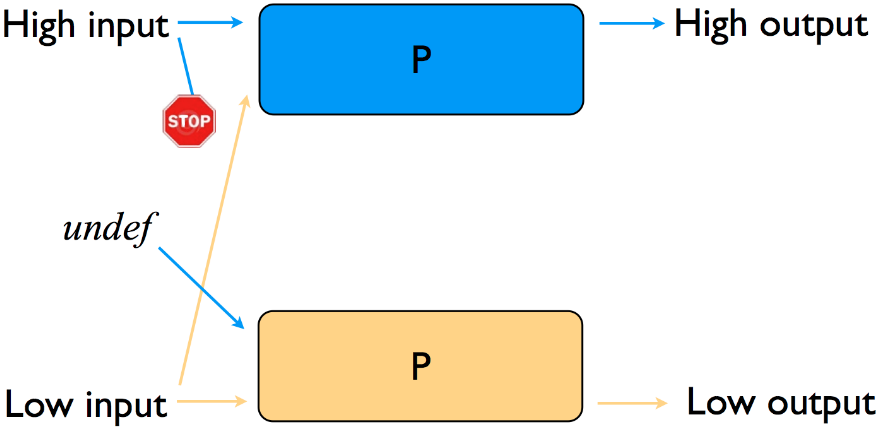
Let `h = true` and `l = false`.

```
x:= 0; y := 1;
if h then
   skip
else
   if l then x := y else skip end;
end;
output x;
```

Non-interference - but need to ensure that `x := y` is not executed.

Combining with a static analysis of the non-executed branches may accept more executions

Secure multi-execution



Information flow

Thomas Jensen

INRIA

Ecole Jeunes Chercheurs en Programmation
May 2013