Static information flow analysis

SOS, Master Recherche Science Informatique, U. Rennes

Thomas Jensen

(slides by David Pichardie, Delphine Demange, 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

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

In this lecture :

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

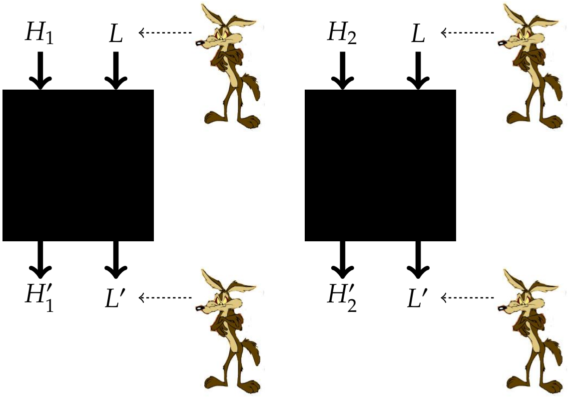
Non-interference

"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

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

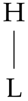
We distinguish between direct and indirect flows

---

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

A lattice of security levels

Information flow can be defined for arbitrary *lattices of security levels*. We consider here only two security levels (low and high).



H

L

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

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 $k_x \not\sqsubseteq k_y$

Program syntax

WHILE language with mixed arithmetic and boolean expressions.

$$\begin{array}{rcl} \mathbf{Expr} ::= & n & \\ \mid & x & \\ \mid & \mathbf{Expr}\ o\ \mathbf{Expr} & \\ \mid & \mathbf{Expr}\ c\ \mathbf{Expr} & \\ \mid & \mathbf{Expr}\ b\ \mathbf{Expr} & \end{array} \qquad \begin{array}{l} n \in \mathbb{Z} \\ x \in \mathbb{V}_H \uplus \mathbb{V}_L \\ o \in \{+, -, \times, \ldots\} \\ c \in \{=, \neq, <, \leq, \ldots\} \\ b \in \{\text{and}, \text{or}\} \\ bop \in o \cup c \cup b \end{array}$$

$$\begin{array}{rl} \mathbf{Stm} ::= & x := \mathbf{Expr} \\ \mid & \texttt{if}\ \mathbf{Expr}\ \texttt{then}\ \mathbf{Stm}\ \texttt{else}\ \mathbf{Stm} \\ \mid & \texttt{while}\ \mathbf{Expr}\ \texttt{do}\ \mathbf{Stm} \\ \mid & \mathbf{Stm}\ ;\ \mathbf{Stm} \end{array}$$

The set of variables is partitioned into two disjoint sets :

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

A simple information flow type system (1/3)

We will present a simple information flow type system [2] 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 $x \in \mathbb{V}_\ell$)

$$\text{CONST}\,\frac{}{\vdash n : L} \qquad \text{VAR}\,\frac{}{\vdash x : \tau_x} \qquad \text{BINOP}\,\frac{\vdash e_1 : \tau \qquad \vdash e_2 : \tau}{\vdash e_1 \; bop \; e_2 : \tau}$$

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

---

[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 $\vdash h + 1 : H$

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

A simple information flow type system (2/3)

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

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

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 $x \in \mathbb{V}_\ell$)

$$\text{ASSIGN}\ \frac{\vdash e : \tau \qquad \tau \sqcup \tau_{pc} \sqsubseteq \tau_x}{\tau_{pc} \vdash x := e} \qquad \text{SEQ}\ \frac{\tau_{pc} \vdash S_1 \qquad \tau_{pc} \vdash S_2}{\tau_{pc} \vdash S_1\ ;\ S_2}$$

$$\text{IF}\ \frac{\vdash e : \tau \qquad \tau \sqcup \tau_{pc} \vdash S_i \qquad i = 1, 2}{\tau_{pc} \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2} \qquad \text{WHILE}\ \frac{\vdash e : \tau \qquad \tau \sqcup \tau_{pc} \vdash S}{\tau_{pc} \vdash \texttt{while}\ e\ \texttt{do}\ S}$$

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

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

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

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

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

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

The observational power of an attacker

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.

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

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

Intuition : the attacker cannot distinguish between equivalent states.

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

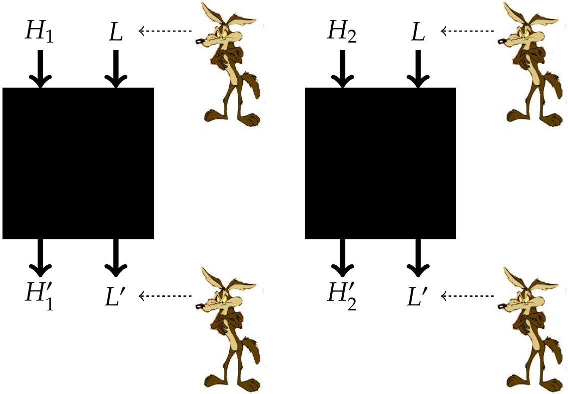
Non-interference

"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

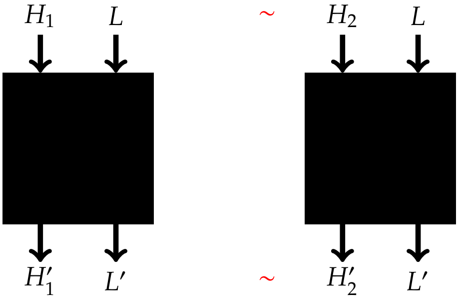
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



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.*

Exercise

Prove this theorem.

Type soundness proof : step 1

We need a new set of typing rules.

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

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

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

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

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* $[\![e]\!]s_1 = [\![e]\!]s_2$.

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

- The attacker may have additional observation power (timing, power consumption)
- Type checking is computable but non-interference is not

Exercise (Type system incompleteness)

Give an example of non-interferent program that is not typable.

Exercise (IFC Challenges)

Solve as many IFC challenges as you can on :
`http://ifc-challenge.appspot.com/`
For each of the challenges :

- give a valid type derivation for your leaky program
- indicate whether (and if so, why) the type system is not restrictive enough
- elaborate on a possible solution to disallow your attack

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)

Covert channels from power consumption

A bit more challenging : power consumption per processor clock cycle

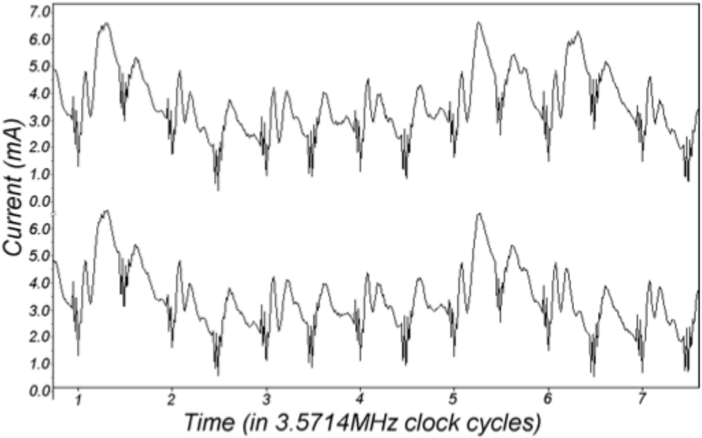


Figure – Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. CRYPTO '99.

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**[3] 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) ?

---

[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!

⇒ 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 [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 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

$$s_1 \sim s_2 \text{ and } [\![exp]\!]s_1 = [\![exp]\!]s_2 \text{ and } (S, s_1) \Downarrow s_1' \text{ and } (S, s_2) \Downarrow s_2'$$

implies

$$s_1' \sim s_2'$$

Exercise (Security property)

Are non-interferent programs secure wrt. delimited release ? If yes, prove it. If not, give a counter example.

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)

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

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

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

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.