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.

\[
\text{Expr} ::= n \quad n \in \mathbb{Z} \\
| \quad x \quad x \in \mathbb{V}_H \cup \mathbb{V}_L \\
| \quad \text{Expr} \ o \ \text{Expr} \\
| \quad \text{Expr} \ c \ \text{Expr} \\
| \quad \text{Expr} \ b \ \text{Expr} \\
\text{Stm} ::= x := \text{Expr} \\
| \quad \text{if Expr then Stm else Stm} \\
| \quad \text{while Expr do Stm} \\
| \quad \text{Stm} ; \ \text{Stm}
\]

The set of variables is partitioned into two disjoint sets:
- \( \mathbb{V}_H \): high (or secret) variables
- \( \mathbb{V}_L \): low (or public) variables
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. \( \text{if } (h1 > h2) \text{ then } \{ l := 1 \} \text{ else } \{ l := 2 \} \)
4. \( \text{while } (h) \text{ do } \{ l := l+1 \} ; \ l := 0 \)

---

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*.

\[
\begin{array}{c}
H \\
| \\
L
\end{array}
\]

We write \( \subseteq \) 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 \( k_x \subseteq k_y \)
- illicit if \( k_x \nsubseteq k_y \)
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**: \(e \in \text{Expr}, \tau \in \{L, H\}\)

\[ \vdash e : \tau \]

**Meaning**: the expression \(e\) depends only of variable of level \(\tau\) or lower.

**Typing rules**:

\[
\begin{align*}
\text{CONST} & \quad \vdash n : L \\
\text{VAR} & \quad x \in \mathbb{V}_\tau \quad \vdash x : \tau \\
\text{BINOP} & \quad \vdash e_1 : \tau, \vdash e_2 : \tau \quad \vdash e_1 \circ e_2 : \tau \\
\text{EXP-SUBTYP} & \quad \vdash e : \tau_1, \tau_1 \sqsubseteq \tau_2 \quad \vdash e : \tau_2
\end{align*}
\]

---

Example

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

\[
\begin{array}{c}
\text{BINOP}
\hline
\text{VAR} \quad \frac{h \in \mathbb{V}_H}{\vdash h : H} \\
\text{EXP-SUBTYP} \quad \frac{\vdash 1 : L \quad L \sqsubseteq H}{\vdash 1 : H}
\end{array}
\]

$\vdash h + 1 : H$
A simple information flow type system (2/2)

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

$$\vdash S : \tau$$

Meaning: the only variables modified by statement $S$ are of level $\tau$ or higher.

Typing rules:

**ASSIGN**

$\frac{x \in \mathbb{V}_\tau \vdash e : \tau}{\vdash x := e : \tau}$

**SEQ**

$\frac{\vdash S_1 : \tau \vdash S_2 : \tau}{\vdash S_1 ; S_2 : \tau}$

**IF**

$\frac{\vdash e : \tau \vdash S_1 : \tau \vdash S_2 : \tau}{\vdash \text{if } e \text{ then } S_1 \text{ else } S_2 : \tau}$

**WHILE**

$\frac{\vdash e : \tau \vdash S : \tau}{\vdash \text{while } e \text{ do } S : \tau}$

**STM-SUBTYP**

$\frac{\vdash S : \tau_2 \quad \tau_1 \sqsubseteq \tau_2}{\vdash S : \tau_1}$

⚠️ The subtype relation on statements is *contravariant*!
Exercise

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

\[
\text{if } (l) \text{ then } h := l \text{ else } l := 0
\]

\[
\text{if } (h) \text{ then } h := l \text{ else } l := 0
\]
A natural semantics

\[
\text{State} = \text{Var} \rightarrow \mathbb{Z}
\]

\[
\llbracket \cdot \rrbracket \in \text{Expr} \rightarrow \text{State} \rightarrow \mathbb{Z} \quad \text{(semantics of expression)}
\]

\[
(\cdot, \cdot) \Downarrow \cdot \subseteq (\text{Stm} \times \text{State}) \times \text{State} \quad \text{(semantics of statement)}
\]

\[
\llbracket n \rrbracket_s = \mathbb{N}[n]
\]

\[
\llbracket x \rrbracket_s = s(x)
\]

\[
\llbracket e_1 + e_2 \rrbracket_s = \llbracket e_1 \rrbracket_s + \llbracket e_2 \rrbracket_s
\]

\[
(x := e, s) \Downarrow s[x \mapsto \llbracket e \rrbracket_s]
\]

\[
(S_1, s) \Downarrow s' \quad (S_2, s') \Downarrow s'' \quad \frac{(S_1; S_2, s) \Downarrow s''}{(S_1, s) \Downarrow s'}
\]

\[
(S, s) \Downarrow s' \quad (\text{while } e \text{ do } S, s') \Downarrow s'' \quad \frac{\llbracket e \rrbracket_s = 1}{(\text{while } e \text{ do } S, s) \Downarrow s''}
\]

\[
(S, s) \Downarrow s' \quad (\text{if } e \text{ then } S_1 \text{ else } S_2, s) \Downarrow s' \quad \frac{\llbracket e \rrbracket_s = 0}{(S_1, s) \Downarrow s'} \quad \frac{(S_2, s) \Downarrow s'}{(S_2, s') \Downarrow s''}
\]

\[
(\text{while } e \text{ do } S, s) \Downarrow s'' \quad \frac{\llbracket e \rrbracket_s = 0}{(\text{while } e \text{ do } S, s) \Downarrow s}
\]
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 \( \sim \subseteq \text{State} \times \text{State} \) 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

∀s₁, s₂, s₁', s₂', s₁ ~ s₂ ∧ (P, s₁) ↓ s₁' ∧ (P, s₂) ↓ s₂' ⇒ s₁' ~ s₂'

High = confidential     Low = public
A statement $S$ is said non interferent iff $s_1 \sim s_2 \ {\begin{array}{l} (S, s_1) \Downarrow s'_1 \\ (S, s_2) \Downarrow s'_2 \end{array}} \implies s'_1 \sim s'_2$

**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
  Example: give an example of non-interferent program that is not typable.

- The attacker may have additional observation power (timing, power consumption)

- Non-interference 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 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)?

Controlling information release

Declassification might compromise confidentiality.

*Ensure that secrets are not leaked via release mechanisms.*

Information release violates non-interference!

⇒ 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 \texttt{declassify}(exp, lvl) that takes as arguments

- an expression \texttt{exp}
- a security level \texttt{lvl} such as high, low...

and declassifies the result of \texttt{exp} to the level \texttt{lvl}.

For example, the type system can now accept\(^4\)

\[
\texttt{avg} := \texttt{declassify}((h_1 + \ldots + h_n)/n, \texttt{low})
\]

Rejected by non-interference.

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

\(^4\) \texttt{h}_i \text{ 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

$$\llbracket exp \rrbracket s_1 = \llbracket exp \rrbracket s_2 \Rightarrow s'_1 \sim s'_2.$$
Examples

Delimited release accepts

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

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

Rejects

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

To see this, set

```plaintext
s_1 = [h_1 = 2, h_2 = 4, avg = 0] 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**: $\text{exp}$ should not contain high variables other than $\text{h}$ in

\[
\text{h} := \text{exp} ; \ldots ; \text{declassify}(\text{h}, \text{low});
\]

**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**

- **EXP-DECLASS**
  \[
  \frac{}{\vdash \text{declassify}(e, l) : l, \text{Vars}(e)}
  \]

- **CMD-ASG**
  \[
  \frac{}{\vdash x := e : \{x\}, D}
  \]

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

```plaintext
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:

```plaintext
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 overhead.
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:

```javascript
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:  \( h := \text{secret} \)

Separately, each thread is safe.

Executed concurrently, they may leak the secret.

Implicit flows can also arise:

T1:  \( \text{if } h > 0 \text{ then } \text{sleep(100)} \text{ else } \text{skip; } l := 1 \)

and

T2:  \( \text{sleep(50)}; l := 0 \)

Most schedulers will leak \( h \) into \( l \)

Making executions \textit{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 $h = \text{true}$ and $l = \text{false}$.

```plaintext
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

High input → P → High output

undef

Low input → P → Low output
Information flow

Thomas Jensen

INRIA

Ecole Jeunes Chercheurs en Programmation
May 2013