Language-based software security, second lecture

Information flow

Xavier Leroy

2022-03-17

Collège de France, chair of software sciences
`xavier.leroy@college-de-france.fr`

Multi-level security and information flow

Multi-level security

A computer system that handles data with different levels of confidentiality and integrity.

Example: two confidentiality levels,

- secret (restricted)
- public (unrestricted)

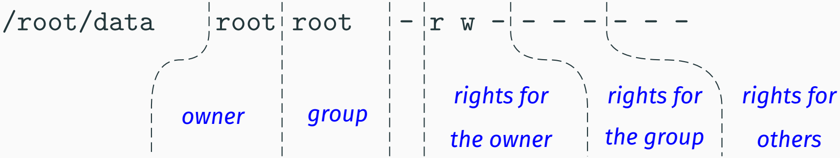
Example: two integrity levels,

- reliable (coming from trusted sources)
- dubious (coming from untrusted sources)

Using access control

Example: a file `/root/data` that must be kept secret and reliable.

Reading and writing restricted to `root`, the superuser.



```
/root/data   root root  - r w - - - - - - -
```

owner

group

rights for the owner

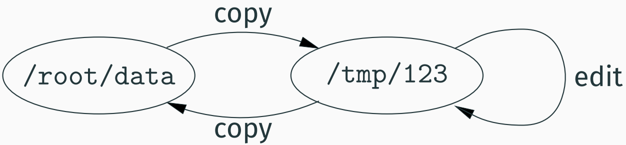
rights for the group

rights for others

Access control is not enough

```
/root/data    root root    -rw-------
```

Editing the file via a temporary file:



If the temporary file was created by the attacker, they can read its contents, and modify it before the copy to `/root/data`.

Information flow

Control not only accesses to resources (data at rest)
but also information flow between resources. (data in transit)

Confidentiality policy: information can flow only from less secret to more secret.

public $\rightarrow$ public   public $\rightarrow$ secret
secret $\rightarrow$ secret   secret $\not\rightarrow$ public

Integrity policy: information can flow only from more reliable to less reliable.

reliable $\rightarrow$ reliable   reliable $\rightarrow$ dubious
dubious $\rightarrow$ dubious   dubious $\not\rightarrow$ reliable

Formalizing confidentiality

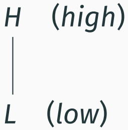
A partial order over confidentiality levels

$$A \sqsubseteq B$$

“B is more secret or as secret as A.”

“Someone with credentials B can access information classified A.”

Example: the public/secret classification.



H (*high*)

L (*low*)

Formalizing confidentiality

Example: the US government classification.

TOP SECRET/A

TOP SECRET/B

TOP SECRET

SECRET/A

SECRET/B

SECRET

CONFIDENTIAL

UNCLASSIFIED

Bell and Lapadula's confidentiality policy

(D. E. Bell et L. J. LaPadula, *Secure Computer Systems: Mathematical Foundations*, 1973, MITRE Corporation.)

No read up: *(Simple Security Property)*

A principal at level ℓ can read only objects of level $\ell' \sqsubseteq \ell$.

No write down: *(Star Security Property)*

A principal at level ℓ can write only objects of level $\ell' \sqsupseteq \ell$.

Object 1

read

$\sqsubseteq$

Principal

write

$\sqsubseteq$

Object 2

Variant: the high-water mark policy

Each principal has two levels:

- R: highest level for reading, fixed;
- W: lowest level for writing, increases over time.

An object at level ℓ can be read if and only if $\ell \sqsubseteq R$.
If so, the writing level is increased: $W \leftarrow W \sqcup \ell$.

An object at level ℓ can be written if and only if $W \sqsubseteq \ell$.

Formalizing integrity

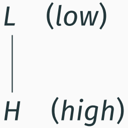
A partial order over integrity levels

$$A \sqsubseteq B$$

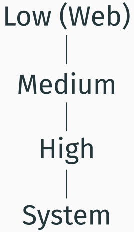
“A is at least as reliable as B.”

“Someone with credentials A can modify level B data.”

Examples: Reliable/dubious



Windows (since Vista)

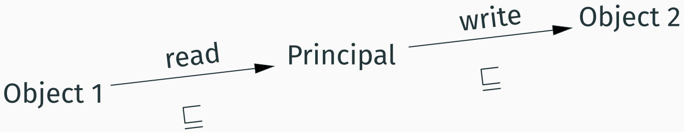


Biba’s integrity policy

K. J. Biba, *Integrity Considerations for Secure Computer Systems*, 1975, MITRE Corporation.

No write down:
a principal at level ℓ can write only objects at level $\ell' \sqsupseteq \ell$.

No read up:
a principal at level ℓ can read only objects at level $\ell' \sqsubseteq \ell$.

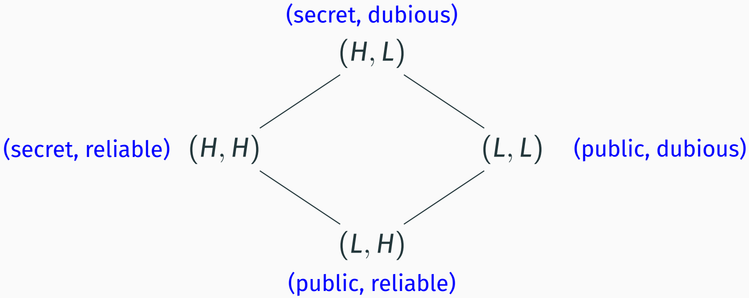


$$\text{Object 1} \xrightarrow[\sqsubseteq]{\text{read}} \text{Principal} \xrightarrow[\sqsubseteq]{\text{write}} \text{Object 2}$$

Combining confidentiality and integrity

Levels = pairs (confidentiality level, integrity level).

Partial order = product of the confidentiality and integrity orders.



(secret, dubious)
(H, L)

(secret, reliable) (H, H)

(L, L) (public, dubious)

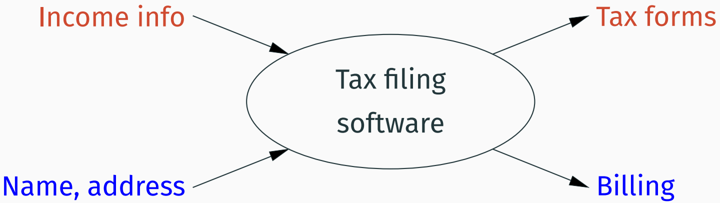
(L, H)
(public, reliable)

Information flow in a program

Multi-level security in a program

A given program can manipulate data at several confidentiality or integrity levels.

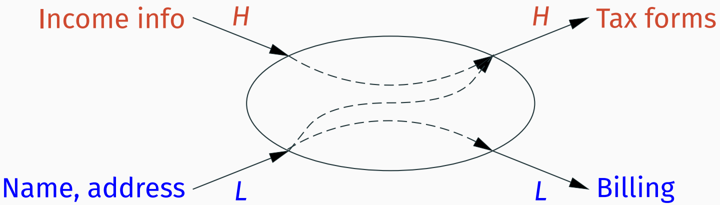
Example: a pay-per-use tax filing program.



To preserve confidentiality, the billing information sent to the software provider must not reveal any of the income info.

Bell-LaPadula for a program

Associate confidentiality levels to inputs, outputs, and intermediate results of the program.



Check that information flows always “go up”:
an output with level ℓ depends only on inputs with levels $\ell' \sqsubseteq \ell$.

Dynamic verification of information flow

Replace each piece of data by a (value, level) pair.

Check the levels at each operation over the data.

```
                       assert (x.level <= z.level);
z := (x + y) / 2  ⇒   assert (y.level <= z.level);
                       z.value := (x.value + y.value) / 2
```

This suffices to control explicit flows.

The problem with implicit flows

Consider two Boolean variables:
`x`, which is secret (level *H*) and `y`, which is public (level *L*).

```
if x then y := true else y := false
```

This code behaves like `y := x`.

Yet, only public values (`true` or `false`) are assigned to `y`.

Dynamic verification of implicit flows

We add a variable *pc* of type “level” to keep track of information revealed by conditional branches.

This variable is updated at conditional statements and loops:

```
if x then ... else ...  ⇒  pc := max(pc, x.level);
                           if x.value then  ... else ...
```

This variable is taken into account during assignments:

```
                        assert (x.level <= z.level);
z := (x + y) / 2  ⇒    assert (y.level <= z.level);
                        assert (pc <= z.level);
                        z.value := (x.value + y.value) / 2
```

Dynamic verification of implicit flows

This succeeds in controlling implicit information flows:

```
if x then y := true else y := false

  ⇒   pc := max(pc, x.level);
      if x.value
      then assert (pc <= y.level); y := true
      else assert (pc <= y.level); y := false
```

Label creep

The level of the *pc* never decreases!

A single test on a *H*-level variable forces the remainder of the program to operate at level *H*.

```
    if x^H then y^H := true else y^H := false;
✘   z^L := false                           // rejected, since pc = H
```

We are tempted to reset *pc* to its level before the conditional:

```
if x then ... else ...  ⇒  pc1 := pc; pc := max(pc, x.level);
                           if x.value then  ... else ...;
                           pc := pc1
```

But this would be unsound…

Label creep

The lack of an assignment can also create an implicit flow!

$\texttt{y}^L$ `:= false;`
`if` $\texttt{x}^H$ `then` $\texttt{y}^L$ `:= true else skip;`
C

If the program reaches point C, without having failed the assertion `pc <= y.level`, it knows the secret "$\texttt{x}^H$ is `false`".

Therefore, the program must execute C with $pc = H$.

Static typing of information flow

Static verification of information flow

By dataflow analysis (Denning & Denning, 1973).

As a type system (Volpano, Irvine, Smith, 1996):

$\vdash a : \ell$	the value of expression *a* has level ℓ
$pc \vdash c : *$	command *c* is safe in a context of level *pc*

IMP: a small imperative language with structured control

Arithmetic expressions:

$$\begin{array}{rll}
a ::= & \mathtt{x}^{\ell} & \text{variables with their levels } \ell \\
| & 0 \mid 1 \mid \ldots & \text{constants} \\
| & a_1 + a_2 \mid a_1 \times a_2 \mid \ldots & \text{operations}
\end{array}$$

Boolean expressions:

$$\begin{array}{rll}
b ::= & a_1 \leq a_2 \mid \ldots & \text{comparisons} \\
| & b_1 \ \mathtt{and} \ b_2 \mid \mathtt{not} \ b \mid \ldots & \text{connectives}
\end{array}$$

Commands:

$$\begin{array}{rll}
c ::= & \mathtt{skip} & \text{empty command} \\
| & \mathtt{x}^{\ell} := a & \text{assignment} \\
| & c_1 ; c_2 & \text{sequence} \\
| & \mathtt{if} \ b \ \mathtt{then} \ c_1 \ \mathtt{else} \ c_2 & \text{conditional} \\
| & \mathtt{while} \ b \ \mathtt{do} \ c & \text{loop}
\end{array}$$

Typing rules for information flow in IMP

Arithmetic and Boolean expressions:

$$\frac{\ell' \sqsubseteq \ell \ \text{ for every variable } x^{\ell'} \text{ free in } a}{\vdash a : \ell}$$

$$\frac{\ell' \sqsubseteq \ell \ \text{ for every variable } x^{\ell'} \text{ free in } b}{\vdash b : \ell}$$

Typing rules for information flow in IMP

$$pc \vdash \texttt{skip} : * \qquad \frac{\vdash a : \ell' \quad \ell' \sqsubseteq \ell \quad pc \sqsubseteq \ell}{pc \vdash x^{\ell} := a : *}$$

$$\frac{pc \vdash c_1 : * \quad pc \vdash c_2 : *}{pc \vdash c_1 ; c_2 : *}$$

$$\frac{\vdash b : \ell \quad pc \sqcup \ell \vdash c_1 : * \quad pc \sqcup \ell \vdash c_2 : *}{pc \vdash \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : *}$$

$$\frac{\vdash b : \ell \quad pc \sqcup \ell \vdash c : *}{pc \vdash \texttt{while}\ b\ \texttt{do}\ c}$$

(Explicit flow control) (Implicit flow control)

Example of typing: controlling an implicit flow

$$(\texttt{if x}^{H}\texttt{ = 0 then }y^{\ell}\texttt{ := 0 else skip); z}^{L}\texttt{ := 1}$$

Typing derivation:

$$\dfrac{\dfrac{\vdash \mathtt{x}^{H} = 0 : H \qquad \dfrac{\vdash 0 : L \quad L \sqsubseteq \ell \quad H \sqsubseteq \ell}{H \vdash \mathtt{y}^{\ell} := 0 : *} \qquad H \vdash \mathtt{skip} : *}{L \vdash \mathtt{if}\ \ \mathtt{x}^{H} = 0\ \ \mathtt{then}\ \ \mathtt{y}^{\ell} := 0\ \ \mathtt{else}\ \ \mathtt{skip} : *} \qquad \dfrac{\begin{array}{c} \vdash 1 : L \\ L \sqsubseteq L \\ L \sqsubseteq L \end{array}}{L \vdash \mathtt{z}^{L} := 1 : *}}{L \vdash (\mathtt{if}\ \ \mathtt{x}^{H} = 0\ \ \mathtt{then}\ \ \mathtt{y}^{\ell} := 0\ \ \mathtt{else}\ \ \mathtt{skip}); \mathtt{z}^{L} := 1 : *}$$

The program is accepted if $\ell = H$, rejected if $\ell = L$.

No label creep because we check both branches of the `if`
(unlike dynamic verification, which checks only one branch).

The non-interference property

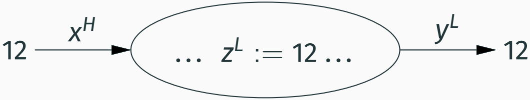
A semantic characterization of correct information flow control.

> *The values of outputs of level* ℓ *must not depend on the values of inputs of level* $\ell' \sqsupset \ell$.

The non-interference property

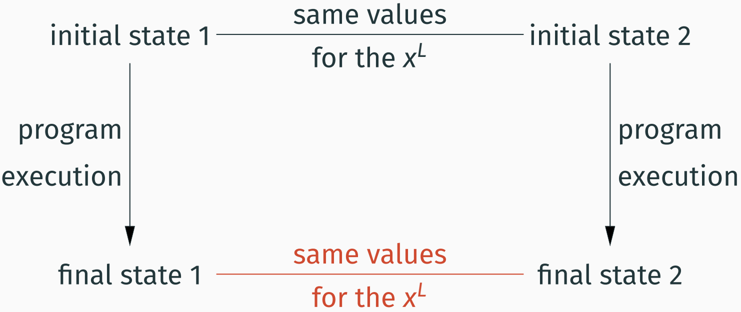
Non-interference is not a property of a single run of the program.

Example: in the execution below, is there interference between the output y^L and the input x^H ?



The non-interference property

Non-interference is a hyperproperty that relates two runs of the program.



initial state 1

same values

for the x^L

initial state 2

program

execution

program

execution

final state 1

same values

for the x^L

final state 2

Natural semantics for IMP

A predicate $c/s \Downarrow s'$ meaning

command c, started in state s, terminates in state s'.

$$\texttt{skip}/s \Downarrow s \qquad\qquad x := a/s \Downarrow s[x \leftarrow [\![a]\!]\, s]$$

$$\frac{c_1/s \Downarrow s' \qquad c_2/s' \Downarrow s''}{c_1; c_2/s \Downarrow s''} \qquad \frac{\begin{array}{c} c_1/s \Downarrow s' \text{ if } [\![b]\!]\, s = \texttt{true} \\ c_2/s \Downarrow s' \text{ if } [\![b]\!]\, s = \texttt{false} \end{array}}{\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2/s \Downarrow s'}$$

$$\frac{[\![b]\!]\, s = \texttt{false}}{\texttt{while } b \texttt{ do } c/s \Downarrow s}$$

$$\frac{[\![b]\!]\, s = \texttt{true} \qquad c/s \Downarrow s' \qquad \texttt{while } b \texttt{ do } c/s' \Downarrow s''}{\texttt{while } b \texttt{ do } c/s \Downarrow s''}$$

Formalizing non-interference

(We restrict ourselves to two levels, L and H.)

Define equality at level L between two states:

$$s_1 \overset{L}{\approx} s_2 \quad \overset{def}{=} \quad \forall x^L,\ s_1(x^L) = s_2(x^L)$$

A L expression has the same value in two states that are $\overset{L}{\approx}$:

$$\begin{array}{lllll} [\![a]\!]\ s_1 = [\![a]\!]\ s_2 & \text{if} \ \vdash a : L & \text{and} & s_1 \overset{L}{\approx} s_2 \\ [\![b]\!]\ s_1 = [\![b]\!]\ s_2 & \text{if} \ \vdash b : L & \text{and} & s_1 \overset{L}{\approx} s_2 \end{array}$$

A H command does not modify L variables:

$$\text{if} \ \ H \vdash c : * \ \ \text{and} \ \ c/s \Downarrow s' \ \ \text{then} \ \ s \overset{L}{\approx} s'$$

Proof of non-interference

Theorem

If $pc \vdash c : *$ *and* $s_1 \overset{L}{\approx} s_2$ *and* $c/s_1 \Downarrow s_1'$ *et* $c/s_2 \Downarrow s_2'$*, then* $s_1' \overset{L}{\approx} s_2'$.

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\overset{L}{\approx}"] \arrow[d, "\textit{execution of } c"'] & s_2 \arrow[d, "\textit{execution of } c"] \\
s_1' \arrow[r, dash, red, "\overset{L}{\approx}"] & s_2'
\end{tikzcd}$$

Proof.

By induction on the derivation of $c/s_1 \Downarrow s_1'$ and case over c. $\square$

Proof of non-interference

Assignment case $x^\ell := a$:

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\overset{L}{\approx}"] \arrow[d, "\text{execution of } x^\ell := a"'] & s_2 \arrow[d, "\text{execution of } x^\ell := a"] \\
s_1[x^\ell \leftarrow [\![a]\!]\, s_1] \arrow[r, dash, "\overset{L}{\approx}"] & s_2[x^\ell \leftarrow [\![a]\!]\, s_2]
\end{tikzcd}$$

If $\ell = L$, by typing hypothesis $\vdash a : L$, hence $[\![a]\!]\, s_1 = [\![a]\!]\, s_2$ and $\overset{L}{\approx}$ holds for the modified states.

If $\ell = H$, no L variable is modified, therefore $\overset{L}{\approx}$ holds for the modified states.

Proving non-interference

Conditional case `if` b `then` c' `else` c'':

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\overset{L}{\approx}"] \arrow[d, "\substack{\text{execution of } c' \\ \text{or of } c''}"'] & s_2 \arrow[d, "\substack{\text{execution of } c' \\ \text{or of } c''}"] \\
s_1' \arrow[r, dash, "\overset{L}{\approx}"] & s_2'
\end{tikzcd}$$

If $\vdash b : L$, we have $[\![b]\!]\ s_1 = [\![b]\!]\ s_2$, hence both executions take the same branch c' or c''. By induction hypothesis we have $s_1' \overset{L}{\approx} s_2'$.

If $\vdash b : H$, both executions can take different branches. But by typing hypothesis we have $H \vdash c' : *$ and $H \vdash c'' : *$.
Hence $s_1 \overset{L}{\approx} s_1'$ and $s_2 \overset{L}{\approx} s_2'$, and finally $s_1' \overset{L}{\approx} s_2'$.

Termination as an information flow

The preceding type system and non-interference criterion are termination insensitive: we consider only the case where both program runs terminate.

However, a program can terminate or diverge depending on the value of a secret:

$$\texttt{if } \texttt{s}^H \texttt{ < 0 then skip else diverge}$$

where `diverge` is the infinite loop `while true do skip done`.

This program “leaks” the sign bit of $\texttt{s}^H$.

Generally, we consider that observing termination or divergence transmits at most one bit of information to an attacker.

Termination as an information flow

(Askarov, Hunt, Sabelfeld et Sands, *Termination-Insensitive Noninterference Leaks More Than Just a Bit*, ESORICS 2008.)

This is no longer true if the program can communicate over a public channel:

```
i^L := 0;
while true do
    output i^L;
    if i^L = s^H then diverge else skip;
    i^L := i^L + 1
done
```

The secret value of $\texttt{s}^H$ is the last integer sent by the program before diverging and going silent.

This kind of attack leaks k bits of information in time $O(2^k)$.

Concurrency and termination

If the language supports concurrent executions, we can easily leak k bits in time $O(k)$:

$$\begin{array}{l||c||l}
\texttt{if s}^H\ \texttt{land}\ 1 = 0 & & \texttt{if s}^H\ \texttt{land}\ 2^{k-1} = 0 \\
\texttt{then diverge} & \dots & \texttt{then diverge} \\
\texttt{else skip;} & & \texttt{else skip;} \\
\texttt{output}\ 0 & & \texttt{output}\ (k-1)
\end{array}$$

Enforcing equitermination by typing

We can strengthen the typing rule for `while` loops:

$$\frac{\vdash b : L \qquad L \vdash c : *}{L \vdash \texttt{while } b \texttt{ do } c}$$

This guarantees that

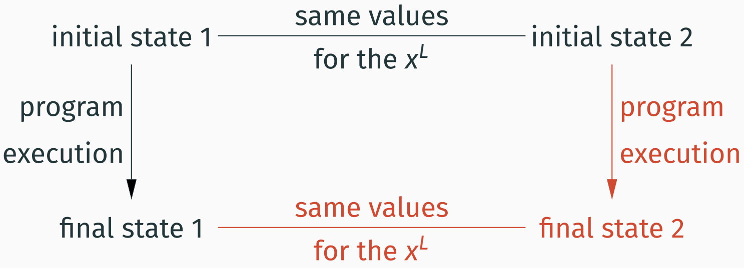
- The loop condition does not depend on H variables.
  (No `while` $x^H = 0$ `do skip done`.)
- A conditional at level H contains no loops.
  (No `if` $x^H = 0$ `then diverge else skip`.)

Termination-sensitive non-interference

Two runs of a program c from two states related by $\overset{L}{\approx}$ either both terminate or both diverge.

Theorem

If $pc \vdash c : *$ *and* $s_1 \overset{L}{\approx} s_2$ *and* $c/s_1 \Downarrow s'_1$,
there exists s'_2 *such that* $c/s_2 \Downarrow s'_2$ *and* $s'_1 \overset{L}{\approx} s'_2$.



initial state 1 — same values for the x^L — initial state 2

program execution ↓ (initial state 1 → final state 1)

program execution ↓ (initial state 2 → final state 2)

final state 1 — same values for the x^L — final state 2

Higher order languages: functions as values

Starting with a standard type system, we add levels on types:

$$\begin{array}{lll} \tau ::= \texttt{int}^{\ell} \mid \texttt{bool}^{\ell} & & \text{base types} \\ \quad\;\; \mid (\sigma \to \tau)^{\ell} & & \text{functions} \\ \quad\;\; \mid \texttt{list}(\tau)^{\ell} & & \text{lists} \end{array}$$

The $\sqsubseteq$ order over levels induces a subtyping relation:

$$\frac{\ell \sqsubseteq \ell'}{\texttt{int}^{\ell} <: \texttt{int}^{\ell'}} \qquad \frac{\sigma' <: \sigma \quad \tau <: \tau' \quad \ell \sqsubseteq \ell'}{(\sigma \to \tau)^{\ell} <: (\sigma' \to \tau')^{\ell'}}$$

Typing rules for a purely functional language

$$\Gamma \vdash n : \mathtt{int}^{\ell} \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma <: \tau}{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : (\sigma \to \tau)^{\ell}} \qquad \frac{\Gamma \vdash e_1 : (\sigma \to \tau)^{\ell} \qquad \Gamma \vdash e_2 : \sigma \qquad \ell \sqsubseteq \mathit{Label}(\tau)}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{bool}^{\ell} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau \qquad \ell \sqsubseteq \mathit{Label}(\tau)}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau}$$

Where $\mathit{Label}(\mathtt{int}^{\ell}) = \ell$, $\mathit{Label}((\sigma \to \tau)^{\ell}) = \ell$, etc.

Adding mutable state

(F. Pottier, V. Simonet, *Information flow inference for ML*, 2002, 2003.)

Types:	$\tau ::= \texttt{int}^\ell \mid \texttt{bool}^\ell$	base types
	$\mid (\sigma \xrightarrow{pc} \tau)^\ell$	functions
	$\mid \texttt{list}(\tau)^\ell$	lists
	$\mid \texttt{ref}(\tau)^\ell$	mutable references

We now need to track implicit flows by adding a level *pc* both to the typing judgment

$$pc, \Gamma \vdash e : \tau$$

and to function types as a latent effect

$$(\sigma \xrightarrow{pc} \tau)^\ell$$

Typing rules for functions + mutable state

$$\frac{pc, \Gamma \vdash e_1 : \texttt{ref}(\tau)^\ell \qquad pc, \Gamma \vdash e_2 : \tau \qquad \ell \sqcup pc \sqsubseteq \mathit{Label}(\tau)}{pc, \Gamma \vdash e_1 := e_2 : \texttt{unit}}$$

$$\frac{pc, \Gamma \vdash e_1 : \texttt{bool}^\ell \qquad pc \sqcup \ell, \Gamma \vdash e_2 : \tau \qquad pc \sqcup \ell, \Gamma \vdash e_3 : \tau \qquad \ell \sqsubseteq \mathit{Label}(\tau)}{pc, \Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\frac{pc', \Gamma, x : \sigma \vdash e : \tau}{pc, \Gamma \vdash \lambda x.e : (\sigma \xrightarrow{pc'} \tau)^\ell}$$

$$\frac{\Gamma \vdash e_1 : (\sigma \xrightarrow{pc'} \tau)^\ell \qquad \Gamma \vdash e_2 : \sigma \qquad \ell \sqsubseteq \mathit{Label}(\tau) \qquad pc \sqsubseteq pc'}{pc, \Gamma \vdash e_1\ e_2 : \tau}$$

Program logics and self-composition

Limitations of type systems for information flow

Type systems for information flow are sometimes too strict and reject programs that contain no dangerous flows and satisfy the non-interference property.

Examples: (`s` is at level *H* and `x` at level *L*)

```
x := s; x := 0
```

```
x := x + s;  ...;  x := x - s
```

```
assert (s >= 0);
x := s;
while x > 0 do ... ; x := x - 1 done
```

In all three programs, the final value of `x` doesn't depend on the initial value of `s`.

Verifying non-interference with a program logic

For a given program c, we would like to verify directly the non-interference property

$$s_1 \overset{L}{\approx} s_2 \ \wedge\ c/s_1 \Downarrow s'_1 \ \wedge\ c/s_2 \Downarrow s'_2 \implies s'_1 \overset{L}{\approx} s'_2$$

using an appropriate program logic.

(See also: my 2020–2021 lectures on program logics.)

Hoare logic (recap)

A set of deduction rules for the predicate

$$\{\,P\,\}\;c\;\{\,Q\,\}$$

which means

$$\forall s, s',\;\; P(s) \land c/s \Downarrow s' \Rightarrow Q(s')$$

Preconditions P and postconditions Q are predicates over memory states s.

Relational Hoare logic

A set of deduction rules for the predicate

$$\{\, P \,\}\ c_1 \mid c_2\ \{\, Q \,\}$$

which means

$$\forall s_1, s_2, s_1', s_2',\ \ P(s_1, s_2) \wedge c_1/s_1 \Downarrow s_1' \wedge c_2/s_2 \Downarrow s_2' \Rightarrow Q(s_1', s_2')$$

Preconditions P and postconditions Q are relations between two memory states s_1, s_2.

Application to non-interference: a program c satisfies the non-interference condition if and only if

$$\{\, \overset{L}{\approx} \,\}\ c \mid c\ \{\, \overset{L}{\approx} \,\}$$

Selected "diagonal" rules

(D. A. Naumann, *37 years of relational Hoare logic*, 2020)

$$\{\, Q[x_1 \leftarrow a_1, x_2 \leftarrow a_2] \,\}\; x_1 := a_1 \mid x_2 := a_2 \;\{\, Q \,\}$$

$$\frac{\{\, P \,\}\; c_1 \mid c_2 \;\{\, Q \,\} \qquad \{\, Q \,\}\; c'_1 \mid c'_2 \;\{\, R \,\}}{\{\, P \,\}\; c_1; c'_1 \mid c_2; c'_2 \;\{\, R \,\}}$$

$$\frac{\begin{array}{cc} \{\, P \wedge b_1 \wedge b_2 \,\}\; c_1 \mid c_2 \;\{\, Q \,\} & \{\, P \wedge \neg b_1 \wedge \neg b_2 \,\}\; c'_1 \mid c'_2 \;\{\, Q \,\} \\ \{\, P \wedge b_1 \wedge \neg b_2 \,\}\; c_1 \mid c'_2 \;\{\, Q \,\} & \{\, P \wedge \neg b_1 \wedge b_2 \,\}\; c'_1 \mid c_2 \;\{\, Q \,\} \end{array}}{\{\, P \,\}\; \texttt{if}\ b_1\ \texttt{then}\ c_1\ \texttt{else}\ c'_1 \mid \texttt{if}\ b_2\ \texttt{then}\ c_2\ \texttt{else}\ c'_2 \;\{\, Q \,\}}$$

$$\frac{\begin{array}{c} Q \Rightarrow b_1 = b_2 \vee (b_1 \wedge L) \vee (b_2 \wedge R) \\ \{\, Q \wedge b_1 \wedge b_2 \wedge \neg L \wedge \neg R \,\}\; c_1 \mid c_2 \;\{\, P \,\} \\ \{\, Q \wedge b_1 \wedge L \,\}\; c_1 \mid \texttt{skip} \;\{\, Q \,\} \qquad \{\, Q \wedge b_2 \wedge R \,\}\; \texttt{skip} \mid c_2 \;\{\, Q \,\} \end{array}}{\{\, Q \,\}\; \texttt{while}\ b_1\ \texttt{do}\ c_1\ \texttt{done} \mid \texttt{while}\ b_2\ \texttt{do}\ c_2\ \texttt{done} \;\{\, Q \wedge \neg b_1 \wedge \neg b_2 \,\}}$$

Selected “left” rules (+ “right” rules by symmetry)

$$\{\,Q[x_1 \leftarrow a_1]\,\}\; x_1 := a_1 \mid \mathtt{skip}\; \{\,Q\,\}$$

$$\frac{\{\,P\,\}\; c_1 \mid \mathtt{skip}\; \{\,Q\,\} \qquad \{\,Q\,\}\; c_1' \mid c_2'\; \{\,R\,\}}{\{\,P\,\}\; c_1; c_1' \mid c_2'\; \{\,R\,\}}$$

$$\frac{\{\,P \wedge b_1\,\}\; c_1 \mid c_2\; \{\,Q\,\} \qquad \{\,P \wedge \neg b_1\,\}\; c_1' \mid c_2\; \{\,Q\,\}}{\{\,P\,\}\; \mathtt{if}\ b_1\ \mathtt{then}\ c_1\ \mathtt{else}\ c_1' \mid c_2\; \{\,Q\,\}}$$

$$\frac{\begin{array}{c}\{\,P\,\}\; \mathtt{while}\ b_1\ \mathtt{do}\ c_1\ \mathtt{done} \mid c_2\; \{\,Q\,\} \\ \{\,Q\,\}\; \mathtt{while}\ b_1\ \mathtt{do}\ c_1\ \mathtt{done} \mid c_2'\; \{\,R\,\} \qquad Q \wedge \neg b_1 \Rightarrow R\end{array}}{\{\,P\,\}\; \mathtt{while}\ b_1\ \mathtt{do}\ c_1\ \mathtt{done} \mid c_2; c_2'\; \{\,R\,\}}$$

Reduction to usual Hoare logic

(N. Francez, *Product properties and their verification*, 1983)

If the variables V_1 used by c_1 are distinct from the variables V_2 used by c_2, then

$$\{\,\bar{P}\,\}\; c_1; c_2\; \{\,\bar{Q}\,\} \quad \text{implies} \quad \{\,P\,\}\; c_1 \mid c_2\; \{\,Q\,\}$$

where $\bar{P}$ is the predicate obtained from the relation P by

$$\bar{P}(s) \overset{def}{=} P(s_{|V_1}, s_{|V_2})$$

Proof sketch: if $c_1/s_1 \Downarrow s_1'$ and $c_2/s_2 \Downarrow s_2'$, we can assume $Dom(s_i) \subseteq V_i$ and $Dom(s_i') \subseteq V_i$. Then, we can derive

$$\frac{c_1/(s_1 \uplus s_2) \Downarrow (s_1' \uplus s_2) \qquad c_2/(s_1' \uplus s_2) \Downarrow (s_1' \uplus s_2')}{c_1; c_2/(s_1 \uplus s_2) \Downarrow (s_1' \uplus s_2')}$$

Self-composition

(G. Barthe, P. D'Argenio, T. Rezk, *Secure information flow by self-composition*, 2004)

To show non-interference for a program c, it therefore suffices to take two copies of c where variables are renamed:

$$c_1 = c\{x \leftarrow x_1 \mid x \in \mathit{Vars}(c)\} \qquad c_2 = c\{x \leftarrow x_2 \mid x \in \mathit{Vars}(c)\}$$

then show, in usual Hoare logic, that

$$\{\, L \,\}\; c_1; c_2 \;\{\, L \,\}$$

where L is the assertion "renamed L variables are equal":

$$L = \bigwedge \{x_1 = x_2 \mid x \in \mathit{Vars}(c), x \text{ at level } L\}$$

Example of verification by self-composition

Consider the program $x := x + s; x := x - s$

$$\{ x_1 = x_2 \} \Rightarrow$$

$$\{ (x_1 + s_1) - s_1 = (x_2 + s_2) - s_2 \}$$

$x_1 := x_1 + s_1;$

$$\{ x_1 - s_1 = (x_2 + s_2) - s_2 \}$$

$x_1 := x_1 - s_1;$

$$\{ x_1 = (x_2 + s_2) - s_2 \}$$

$x_2 := x_2 + s_2;$

$$\{ x_1 = x_2 - s_2 \}$$

$x_2 := x_2 - s_2;$

$$\{ x_1 = x_2 \}$$

Example of verification by self-composition

Consider $\texttt{assert}(s \geq 0); x := s; \texttt{while } x > 0 \texttt{ do } x := x - 1 \texttt{ done}$

$$
\begin{array}{ll}
 & \{ x_1 = x_2 \} \Rightarrow \\
 & \{ T \} \\
\texttt{assert}(s_1 \geq 0); & \{ s_1 \geq 0 \} \\
x_1 := s_1; & \{ x_1 \geq 0 \} \\
\texttt{while } x_1 > 0 \texttt{ do} & \{ x_1 > 0 \} \\
\quad x_1 := x_1 - 1 & \{ x_1 \geq 0 \} \\
\texttt{done}; & \{ x_1 = 0 \} \\
\texttt{assert}(s_2 \geq 0); & \{ x_1 = 0 \wedge s_2 \geq 0 \} \\
x_2 := s_2; & \{ x_1 = 0 \wedge x_2 \geq 0 \} \\
\texttt{while } x_2 > 0 \texttt{ do} & \{ x_1 = 0 \wedge x_2 > 0 \} \\
\quad x_2 := x_2 - 1 & \{ x_1 = 0 \wedge x_2 \geq 0 \} \\
\texttt{done}; & \{ x_1 = 0 \wedge x_2 = 0 \} \\
 & \Rightarrow \{ x_1 = x_2 \}
\end{array}
$$

Declassification and endorsement

A declassified document

Attachment 1

[redacted]

Work Scope

1. Introduction

Raytheon seeks to sponsor fundamental research in order to [redacted]

[redacted]

[redacted]

Declassification

Voluntary downgrading of the confidentiality level for some results.

Example: checking a password.

```
let checkpwd (input: string^H) (hashed_password: string^H)
          : bool^L =
  let res : bool^H = (hash(input) = hashed_password) in
  declassify(res)
```

Some declassification technique:

- Manual redaction + stamp of approval / crypto signature
- Reveal a very small part of the secret (as in `checkpwd`)
- Encrypt or hash the secret

Endorsement

Voluntary upgrading of the integrity level for some inputs.

Example: validating a ZIP code entered on a Web page.

```
let checkzip (input: string^L) : string^H =
  if DB.search zip_database input = Found
  then endorse(input)
  else raise "bad ZIP code"
```

Some endorsemet techniques:

- Manual checks + stamp of approval / crypto signature
- Cross-checking against reliable databases (as in `checkzip`).

Declassification as a function?

It is dangerous to offer declassification as a function $H \to L$ that can be used arbitrarily many times.

Example: assuming $\texttt{checkpwd} : \texttt{string}^H \to \texttt{string}^H \to \texttt{bool}^L$, we can leak all the bits of a secret s^H.

```
for b^H in bits(s^H) do
    let c^H = if b^H then "1" else "0" in
    let z^L = checkpwd c^H (hash("1")) in
    output(z^L)
done
```

Declassification as a function?

It is dangerous to offer declassification as a function $H \to L$ that can be used arbitrarily many times.

Example: with an encryption function $\texttt{encrypt} : \texttt{key}^H \to \texttt{string}^H \to \texttt{string}^L$.

```
for b^H in bits(s^H) do
    let c^H = if b^H then "X" else "" in
    let z^L = enc k^H c^H in
    output(z^L)
done
```

All the bits leak if encryption preserves the length of the cleartext, or if encryption is deterministic.

Global declassification policies

(Li & Zdancewicz, *Downgrading Policies and Relaxed Noninterference*, 2015.)

A set of functions F_i that can be applied to the secret inputs of the program (but not to other arguments) to produce declassified data.

Consider the following program:

```
let checkpwd (input: string^H) (hashed: string^H) =
  hash(input) = hashed
```

Take the declassification function $F(i, h) = (\mathtt{hash}(i) = h)$.

The value $F(\mathtt{input}, \mathtt{hashed}) = (\mathtt{hash}(\mathtt{input}) = \mathtt{hashed})$ is declassified and usable at level L.

Any other comparison of hashes is not declassified.

Declassification and non-interference

Relaxed non-interference criterion: the outputs of level L depend only on inputs of level L and on declassified values, that is, the values of the F_i applied to H inputs.

In relational Hoare logic:

$$\{ \overset{L}{\approx} \ \wedge \ \mathcal{D} \} \; c \mid c \; \{ \overset{L}{\approx} \}$$

where $\mathcal{D}$ expresses equality of declassified values in both states:

$$\mathcal{D}(s_1, s_2) \ \overset{def}{=} \ \bigwedge_i F_i(s_1^H) = F_i(s_2^H)$$

Summary

Summary on multi-level security and information flow

“Military-style” multi-level security systems, as studied by Bell and LaPadula, are not very common …

… but many systems (Android, iOS, Windows) include integrity policies in the style of Biba …

… and similar confidentiality and integrity problems appear in many other contexts, notably Web pages.

The notion of information flow is crucial to ensure confidentiality and integrity of data.

Summary on information flow analysis

The analysis of information flow (by typing or by program logics) is very strict…

… but very effective to identify the points in the code where declassification or endorsement takes place

… and has other uses,
for instance to ensure “constant time” execution ($\rightarrow$ 4th lecture).

Going further

Typing information flow for “real” languages:
functions, objects, exceptions, concurrency, nondeterminism, …

(See for instance JIF, “Java + Information Flow” by Myers *et al*, `https://www.cs.cornell.edu/jif/`.)

Accounting for other information channels:
execution time (→ 4th lecture), power usage, electromagnetic emissions, …

Reasoning over the quantity of information that leaks:
information theory, Bayesian models.

(See Alvim *et al*, *The Science of Quantitative Information Flow*, Springer, 2020.)