Noninterference: execution preserves equivalence

But: real systems need to release some information as part of their intended function.
- password checker: passwords
- reviewing system: reviews (eventually)
- distributed games: opponent actions (integrity)

Result: execution sometimes does this.

Noninterference $\iff$ 0 information flow
Need to be able to enforce it, but need information flow control, not just prevention.

Can add downgrading to system
- declassify: lower confidentiality
- integrity: raise integrity
Problem: Justification for downgrading is application-specific.

Ex. 1  Password checking.

\[
\text{if (declassify(guess = \text{security, L to H}))} \\
\text{login} := \text{true};
\]

Justification: adversary learns very little.

Ex. 2  Auction.

\[
\text{if (auction\_done)} \\
\text{released\_bid} = \text{declassify(bid, H to L)}
\]

Justification: information no longer confidential.

Ex. 3  Distributed game (e.g., Battleship).

\[
\text{move (mx, my)} \\
\text{mx, my : A (integrity)}
\]

\[
\text{if (legal\_move(mx, my, board))} \\
\text{endorse (mx, my, A to B)}
\]

\[
\text{if (legal\_move(mx, my, board))} \\
\text{use mx, my at B integrity}
\]

Justification: adversary gets to make legal moves.
Many justifications ⇒ many different approaches to relaxing noninterference.

Sobel and Sanders, "Dimensions and Principles" of declassification: categorize mechanism
who - who decides to declassify
what - which information, or how much,
is declassified
when - under what conditions, temporally related policies
where - using notions of locality

All have been explored.

An early idea: restrict declassification to trusted/authorized code - selective declassification [SOsp97] - a "who" approach.
- need to have labels that talk about principals - "decentralized labels"
- problem: untrusted code/agents can still influence trusted code.

A "what" approach: delimited release
- specifies which expressions may be downgraded; semantic condition
  says \( S_1 \sim_L S_2 \) if \( S_1, S_2 \) equal at those expressions (at \( \Lambda \))

Relaxed noninterference generalizes this [Li&Zaner92] to specify types that say what computations may be declassified (password: \( \lambda p . p = \text{guess} \))
A "when" approach: Steve Chong's downgrading policies \( l_1 \leq l_2 \).

"Flow allowed if condition c holds" e.g., bid: \( H \uparrow L \).

Logic c unspecified; Dimitrova et al. [VMCAI'11] explore instantiating c with temporal logic, using model checking for enforcement. Practical?

S&S also identify some useful principles declassification mechanisms should aim for, such as:

Semantic consistency:

* semantics-preserving program changes shouldn't affect security judgment. (undeniable, but a good goal)

  - Want extensional security that depends on behavior, not intensional security based on details of code.

Non-occlusion

* Use of declassify should not hide other leaks. E.g., in delimited release: \( l_1 := h \); \( l_2 := \text{declassify}(h) \) is semantically "secure."
Laundering

- aspect of non-disclosure
- Adversary can exploit declassification to create unintended information release.
- Particularly an issue in distributed systems where adversary may supply some of the code.

\[
\begin{align*}
\text{adversary code} & \\
\text{pwd} & := 0 \\
guess & := \text{secret} \& \ 2^i \\
\text{if} \ & (\text{declassify}(\text{guess} == \text{pwd})) \ \\ & \text{login} := \text{true}; \\
\text{restore \ pwd} & \\
\end{align*}
\]

Integrity can affect confidentiality — not fully dual.

To prevent: adversary should not be able to affect:
- what is declassified
- whether declassification happens
- whether endorsement happens with or without adversary.

\[\text{Extensionally:}\]

Let \(s[a]\) be states with low-integrity parts replaced by adversary "fair attack" on.
Robust declassification:

\[ \forall s, s', a, a'. s[a] \approx_{L} s'[a] \Rightarrow s[a'] \approx s'[a'] \]

termination-sensitive insensitive

- "No attack is worse than the dummy attack"
- Extensional, enforceable by a type system

To connect confidentiality & integrity need to map integrity to the confidentiality it enforces

Eg if \( l = (p_c, p_i) \), define:

\[
\text{enforces}(l) = (p_i, 1)
\]

\[ \Gamma, p_c \vdash e : l_e \quad l_e \in l_1 \quad l_1 \in l_2 \quad \text{enforces}(l_1) \]

\[ \Gamma, p_c \vdash \text{declassify}(e, l_1 \text{ to } l_2) : l_2 \]

See Askarov et al (LMCS) for more precise (progress-sensitive) security conditions.

- Most interesting directions seem to be in the "what" and "when" dimensions.

One more direction: NI too weak!

- crypto device should forget keys
- voting machine should forget voter-ballot linkage
- legal requirements to forget information (medicine, finance, govt, ...
Idea: mandatory upgrading

Erasure policies require information label to increase, disappear from orig. level.

Chong: policy \( l_1 \rightarrow l_2 \) means

- info. at level \( l_1 \) must upgrade to level \( l_2 \) when condition \( c \) holds — including all derived info.
- vs. may downgrade policies
- can combine both to achieve interesting temporal policies
- implemented in Jif using static + dynamic enforcement.
- used to implement Civitas voting system.

Open problems / interesting directions:

- Downgrading policies and their connection to extensional system security requirements.
- Integrating information flow with cryptography.
- Concurrency — how to avoid internal timing channels. Starting point: low determinism (Roscoe 95)

\[ [s_1] \equiv [s_2] \iff \forall t_1 \in s_1, \forall t_2 \in s_2, t_1 \equiv t_2 \]

\[ \equiv \]
Problem: how to enforce compositionally?

- Soundness of Jif, etc.
  - complex language: objects, exceptions, dependent labels, parametric polymorphism.
- How to get developer buy-in?

<table>
<thead>
<tr>
<th>Costs</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>added annotation burden (Jif, Flow, Caml) do do inference)</td>
<td>+ increased security assurance</td>
</tr>
<tr>
<td>challenge of mapping system requirements to labels.</td>
<td></td>
</tr>
<tr>
<td>non-local errors that are hard to diagnose.</td>
<td></td>
</tr>
</tbody>
</table>

Idea: information flow enables + higher-level programming model.

Type system enforce: Compositional

⇒ Can transform code without breaking security.

Example: Swift: automatically secure partitioning web applications [SOSP'07]
Partitioner keeps secrets off client, does not accept high-integrity results from client.
- High-integrity public values may be replicated automatically.
- e.g., input validation typically replicated to both sides
  - Low latency (client side)
  - Security (server side)
  - Consistent
- Partitioning done to minimize control transfers (= network delays)
- Fine-grained partitioning: splits individual objects, statements, according to labels. See also: work by Fournet et al. on secure partitioning.

Another higher-level programming model: Fabric [Sosp09, Srp12].

Enforcing security one machine at a time makes little sense! How do we program networks as if they are computers? (+ consistency, persistence, ...)

One challenge: heterogeneous trust / generalizing beyond Swift.

Fabric: everything is a Jif (roughly) object, language = Jif (Java + labels) + orthogonal transparency, persistence (no DB!), + secure federated transactions (strong consistency) + remote method calls + mobile code

Nodes are principals
atomic { 
  n.left.value++;
  n.append @ node3 (n');
} 

\( \text{transaction:} \quad \text{atomic, isolated} \)

\( \text{check:} \quad p \in E \text{ node3 ?} \)

- can write complex systems including
  “mashups” much more concisely.
- lots of challenges left
  - side channels from distributed
    protocols such as transactions (e.g., timing)
  - scalability
  - verification!

\text{Summary: language-based security is a great application area for languages & verification.}