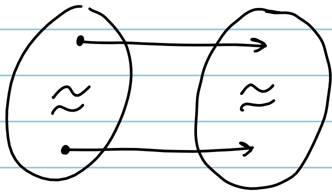
OPLSS '12

Lecture 4 - Downgrading & other future directions

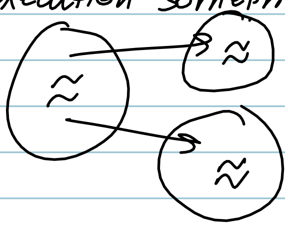
Noninterference: execution preserves equivalence

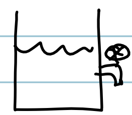


But: real systems need to release some information as part of their intended function.

- password checker: passwords
- reviewing system: reviews (eventually)
- distributed games: opponent actions (integrity)

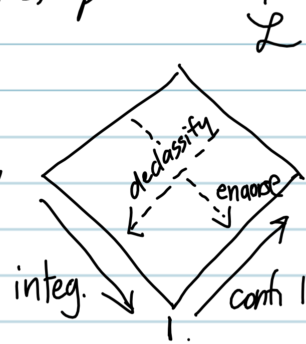
Result: execution sometimes does this.





Noninterference $\Leftrightarrow$ 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: ~~lower~~ raise integrity

Problem: Justification for downgrading is application-specific.

<u>Ex. 1</u> Password checking.

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

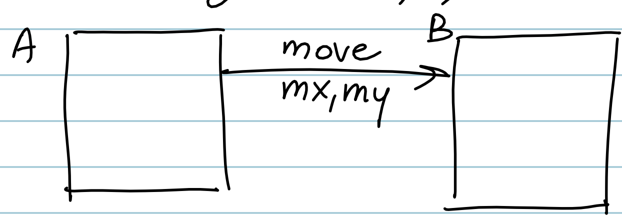
Justification: adversary learns very little.

<u>Ex2</u>. Auction.

if (auction_done)
 released_bid = declassify(bid, H to L)

Justification: information no longer confidential.

<u>Ex3</u> Distributed game (e.g., Battleship)



mx, my : A (integrity)

~~if (legal_move(mx, my, board))~~

endorse (mx, my, A to B)

if (legal_move(mx, my, board)) {

 ⋮

 // use mx, my at B integrity

}

Justification: adversary gets to make legal moves.

Many justifications ⇒ many different approaches to relaxing noninterference.

Sabelfeld & Sands, "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 [SOSP'97] - 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 low)

Relaxed noninterference generalizes this [Li & Zdancewic] to specify types that say what computations may be declassified (password : $\lambda p.\ p == guess$)

A "when" approach: Steve Chong's
downgrading policies $\ell_1 \xrightarrow{c} \ell_2$

"flow allowed if condition c holds"
e.g., bid : $H \xrightarrow{\text{auction end}} L$

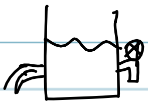
Logic c unspecified; Dimitrova et al [VMCAI'11] explore instantiating c with temporal logic, using model checking for enforcement. Practical?

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. (undecidable, 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; $\ell_1 := h$; $\ell_2 :=$ declassify(h) is semantically "secure"

Laundering

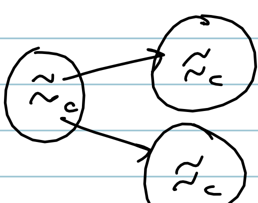
- aspect of non-occlusion
- Adversary can exploit declassify to create unintended information release.
- Particularly an issue in distributed systems where adversary may supply some of the code.

adversary code

```
[ pwd := 0
[ guess := secret & 2^i      <- "bitwise and"
if (declassify(guess == pwd)) {
    login := true;
}
[restore pwd]
```

Integrity can affect confidentiality — not fully dual.

To prevent: adversary should not be able to affect
- what is declassified
- whether declassification happens } rob ! dec ss
- whether endorsement happens



with or without adversary.

Extensionally:

Let s[a] be state s with low-integrity parts replaced by adversary "fair attack" a.

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 $\ell = (p_C, p_I)$, define:
$\text{enforces}(\ell) = (p_I, \bot)$

$$\boxed{\frac{\Gamma, pc \vdash e : \ell_e \quad \ell_e \sqsubseteq \ell_1 \quad \begin{array}{l} \ell_1 \sqsubseteq \ell_2 \sqcup \text{enforces}(pc) \\ \ell_1 \sqsubseteq \ell_2 \sqcup \text{enforces}(\ell_1) \end{array}}{\Gamma, pc \vdash \text{declassify}(e, \ell_1 \text{ to } \ell_2) : \ell_2}}$$

See Askarov et al (LMCS) for ~~most~~ 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: <u>mandatory upgrading</u>

<u>erasure policies</u> require information label to increase, disappear from orig. level.

Chong: policy $l_1 \nearrow^{c} l_2$ means

- info. at level l_1 <u>must</u> upgrade to level l_2 when condition c holds. — including <u>all derived info.</u>
- vs. <u>may</u> downgrade policies
- can combine both to achieve interesting temporal policies
- implemented in $\text{Jif}_{\text{until}}$ using static + dynamic enforcement.
- used to implement Civitas voting system.

<u>Open problems / interesting directions</u>

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

$$[\![S_1]\!] \approx [\![S_2]\!] \iff \forall t_1 \in S_1\, \forall t_2 \in S_2 \,.\, t_1 \approx t_2$$

Problem: how to enforce compositionally?

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

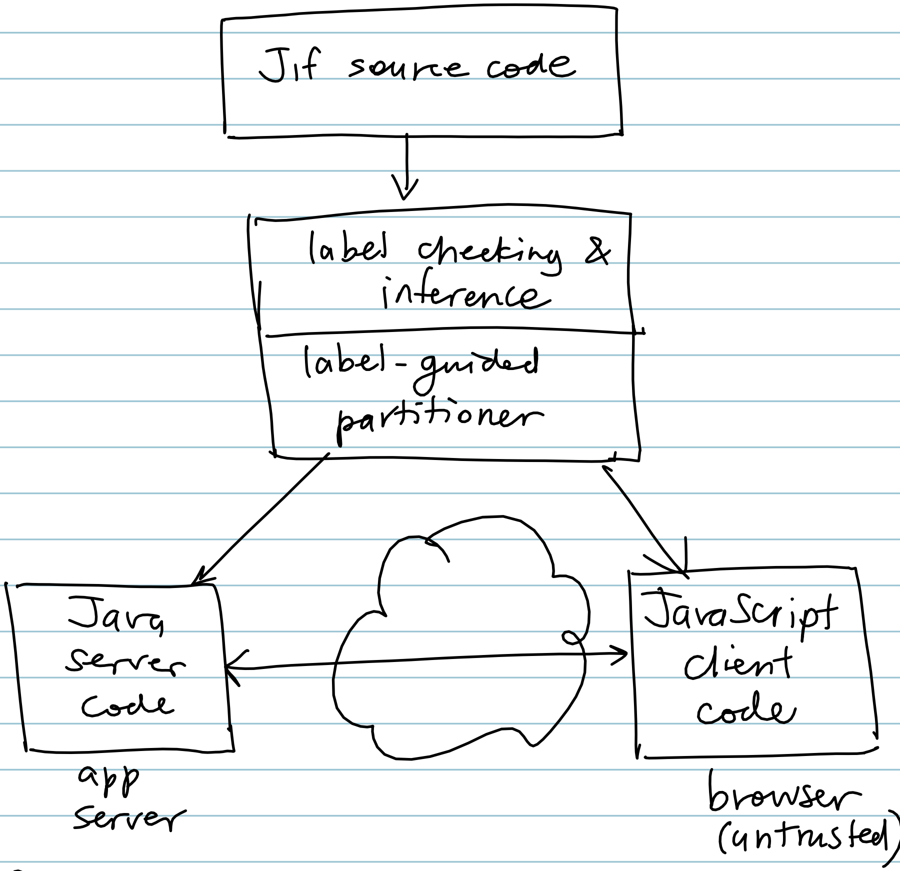
costs	benefits
- added annotation burden (Jif, FlowCaml do do inference)	+ increased security assurance
- challenge of mapping system requirements to labels.	
- non-local errors that are hard to diagnose.	

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

type system enforcement: compositional

⇒ Can transform code without breaking security.

Example: Swift: automatically securely 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

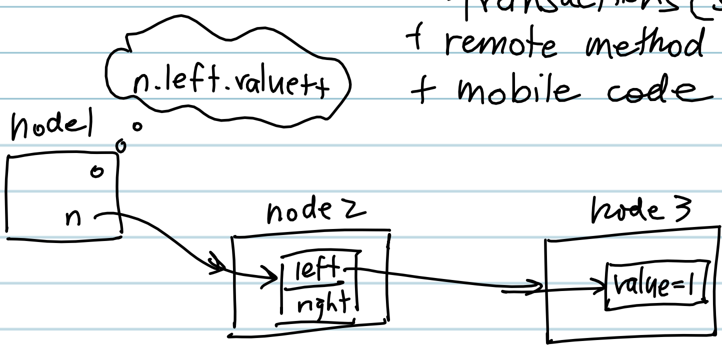
- partitioning done to minimize control transfers (= network delays)
- Fine-grained partitioning: splits individual objects, statements, according to labels. See also: work by Fournet et al. on secure partitioning

Another higher-level programming model: Fabric [SOSP'09, S&P'12].

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 transparent persistence (no DB!)
+ secure federated transactions (strong consistency)
+ remote method calls
+ mobile code



- Nodes are principals

```
atomic {
    n.left_value++;
    n.append @ node3 (n')
}
```

transaction: atomic, isolated

check: $pc \sqsubseteq$ 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!

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