Dynamic Updating of Information-flow Policies

Michael Hicks, Stephen Tse, Boniface Hicks

Steve Zdancewic

FCS 2005

Information-flow Security

- Goal is to protect confidential information from leaking to inappropriate principles.
  - Has been studied in computer systems context for > 30 years [Denning, Goguen, Mesegue,…]

- Language-based security is a promising enforcement mechanism:
  - **if** b_H **then** y_L **:= 0 else** y_L **:= 1**

    Idea: use extended type systems to give security labels to data, conservatively track flows, reject programs that don't meet the policy.

  - Implementations: Jif [Myers et al.] , FlowCaml [Simonet & Pottier]

Language-based Enforcement

- *Noninterference:*
  Behavior of the program visible to low-security observers should not depend on high-security information.

- *Sound Execution:*
  The program does not generate errors at run time.

- Both properties are enforced statically.
  - With good reason: purely dynamic enforcement of information flow policies is much too conservative to be useful.

Problems with Practicality

- Noninterference isn't really the property you want:
  - Programs *do* intentionally leak some information
  - So: need mechanisms for controlled downgrading

    [Survey: SS05]
  - But: noninterference is still an essential baseline.

- Static policies are not always sufficient:
  - Some policy-relevant information may not be known until run time (e.g. file permissions)
  - It might be necessary to change the policy for a long running system (e.g. to revoke privileges)

Example Program

```
void access_records(principal{} emp) {
  Query{mgr:div} query;     // query is visible to division
  Data{mgr:emp}  result;    // result is visible to employee
  Data{mgr:}     audit;     // audit info is for managers only

  if (div < emp) {          // employee is member of division
    while (true) {
      query  = get_query();
      result = process_query(query);
      audit  = audit(result);
      display(emp, result);
      if (mgr < emp) {     // employee is a manager
        display(emp, summary); }
    }
  else { abort(); }
}
```

This Paper: Work In Progress

- We consider the problem of dynamically updating information-flow policies.

- Interesting design space (that we're still exploring):
  - In what ways can the policy be changed?
  - When is it safe to update a policy?
  - What does it mean for noninterference to hold when the policy can be changed dynamically?
  - What can we prove about the system as a whole?

- Start simple:
  - noninterference (no downgrading)
  - some dynamically determined policy information (necessary for the policy changes to be useful)

Policy Hierarchies

- Policy hierarchy: $\Pi = (p_1 < q_1, \ldots , p_n < q_n)$
- Ordering on policies determines which labels are more restrictive:

$$(p<q,\ \ q<r);\ \ \vdash\ 1_p\ :\ int_r$$

- In general, the type system is parameterized by the hierarchy:

$$\Pi\ ;\ \Gamma\ \vdash\ e : t$$

- Operational semantics allows for updates:

$$\Pi\ |\ e\ \rightarrow\ \Pi'\ |\ e$$

Dynamic Policy Tests

- Determine policy information at run time:
  [TZ04,ZM04]

$$\frac{\Pi,(p<q);\ \Gamma \vdash \mathbf{e_1} : t \qquad \Pi\ ;\ \Gamma \vdash \mathbf{e_2} : t}{\Pi\ ;\ \Gamma \vdash \texttt{\textbf{if (p<q) then e}}_1\ \texttt{\textbf{else e}}_2 : t}$$

Dynamic Policy Updates

- Could relabel a value: $1_p \rightarrow 1_q$
  - relabeling can violate soundness and noninterference
  - related to declassification

- More interesting: change the *relationship* between labels by altering policy hierarchy.
- Example: $(p<q,\ q<r) \rightarrow (p<q,\ q<s)$
  - New hierarchy disallows old flow $(p < r)$ but it permits the new flow $(p < s)$

What Can Go Wrong?

- Starting with $\Pi = (p<q)$:

  Π | `let x :` $\texttt{int}_q$ `= if (p<q) then` 1_p `else` 2_q
  `in …`

- After one step:
  Π | `let x :` $\texttt{int}_q$ `=` 1_p
  `in …`

- Now, suppose we update to $\Pi' = (q<r)$:
  Π' | `let x :` $\texttt{int}_q$ `=` 1_p
  `in …`

- This program no longer typechecks.

Our Simple Solution: Coercions

- The "tagged" term **`[p<q]e`** coerces e from type t_p to t_q. [B-TCGS'91]

- Operationally: $\mathtt{[p<q]v_p} \rightarrow \mathtt{v_q}$

- Inserting coercions allows the previous example to typecheck even after the policy update:

let x : int$_q$ = if (p<q) then [p<q]1$_p$ else 2$_q$
in …

When Are Updates Allowed?

- Could imagine dynamically "re-typechecking" the continuation of an update under the new policy.
  - Tags allow that process to be optimized
  - Tags are less conservative because they keep information around at run time that would otherwise be erased
- Intuition: The tags record the "active" assumptions about the policy hierarchy.

  `[p<q]e`

  - Computation in **`e`** can safely assume that [p<q] holds.
- Therefore, can't change the policy unless it satisfies all constraints of "exposed" tags.

Examples

- Let $\Pi = (p<q)$ and $\Pi' = (q<r)$

- Can update from Π to Π' when the program is:

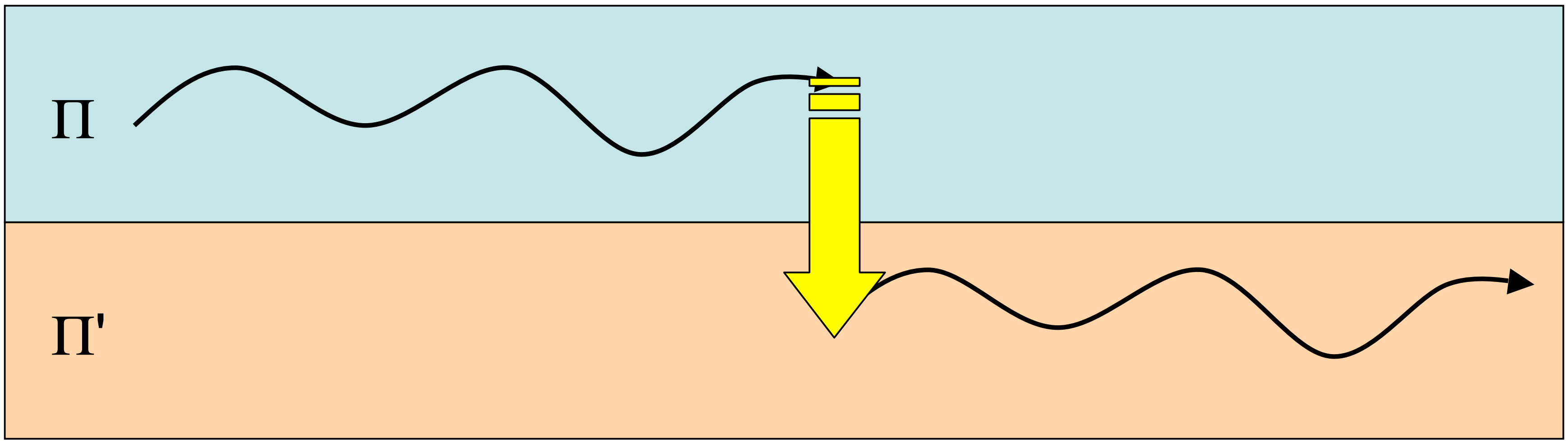
  `if (p<q) then [p<q]`$\mathbf{e_1}$ **`else`** $\mathbf{e_2}$
- Cannot update from Π to Π' when the program is

  `[p<q]`$\mathbf{e_1}$
- Can update from P to P' when the program is:

  $\mathbf{2_q}$

Noninterference Between Updates

- What security property can we get from this type system?



- Easy to show that *between* updates, standard noninterference holds… follows from the soundness of updates.
- But this result doesn't say anything about what happens across updates.

Flows Across Updates

- Purely dynamic tag checks are insufficient:

```
let x = if b_q then (λx.0_q)
        else (λ(p<q)x.[p<q]1_p)
in
let y : int_r = if (p<q) then 1_r
                else 0_r in
… // use x …
```

- If the policy is updated after first lest is evaluated, this program may copy b_q to y , violating noninterference.
- Information flow depends on attacker's knowledge of the hierarchy and policy updates.
- More static constraints can rule out such flows.

Conclusions

- Allowing information-flow enforcement to deal with dynamic policies is important for practical applications.
- This paper presents a first stab at handling dynamic updates to noninterference policies.
- In the paper:
  - Details of the type system and tag checking scheme
  - Proof of soundness for the tagged language
  - Translation from untagged source to tagged language
  - Noninterference between updates

Future Directions

- What can we say about information-flow policies across updates?
  - Related to downgrading and declassification
  - Flows in the program should be explainable in terms of policies in force before and after the updates
- Scaling up these simple ideas of dynamic tags to work with more language features
  - State and other effects
  - Dynamic labels
  - Concurrency
- Implementing dynamic updates to get experience with real software