



Policies and Mechanisms for Safe Information Release

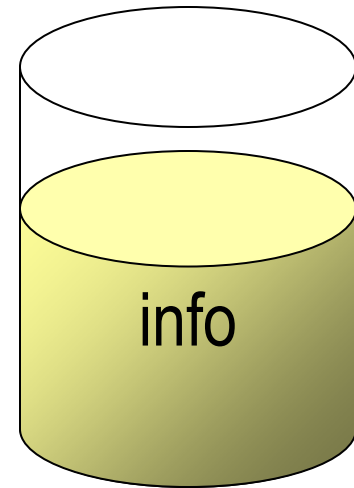
Andrei Sabelfeld
Chalmers

Based on pieces of published joint work with:
Andrew C. Myers (Cornell) and Steve Zdancewic (Penn)
and unpublished joint work with:
David Sands (Chalmers)

PLID, Verona, August 2004

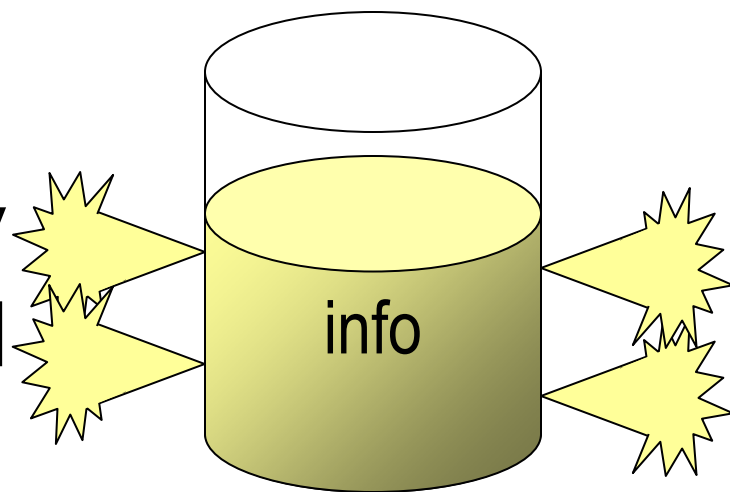
Confidentiality: preventing information leaks

- Untrusted/buggy code should not leak sensitive information
- But some applications depend on **intended** information leaks
 - password checking
 - information purchase
 - spreadsheet computation
 - ...
- Some leaks must be allowed: need **information release** (or **declassification**)



Confidentiality vs. intended leaks

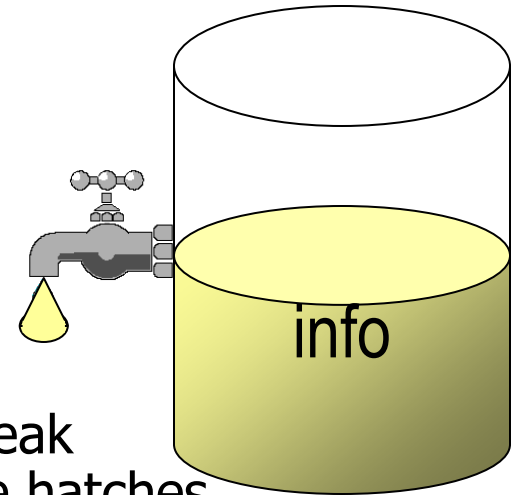
- Allowing leaks might compromise confidentiality
- Noninterference is violated
- How do we know secrets are not **laundered** via release mechanisms?
- Little or no assurance for declassification constructs in many security-typed languages



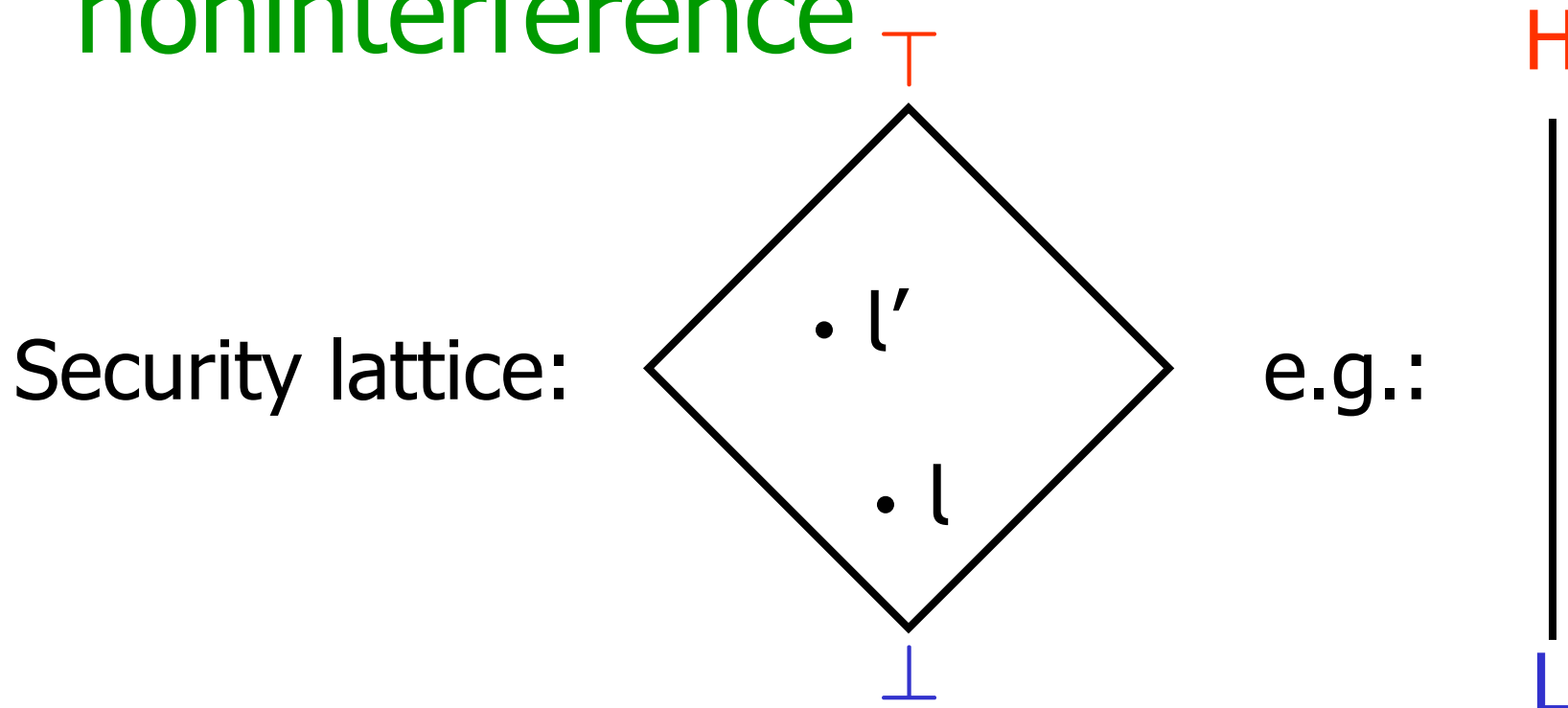
Leveraging assurance in presence of declassification

Two approaches:

- **Delimited release** [ISSS'03]
 - Syntactic “**escape hatches**” for specifying exactly what information is released
 - Guarantee: a program might not release/leak more information than released via escape hatches
- **Robust declassification** [CSFW'01,CSFW'04]
 - An **active** attacker may not learn more sensitive information than a **passive** attacker
- As noninterference, both are **end-to-end** properties
- Provably enforceable by **security-type systems**



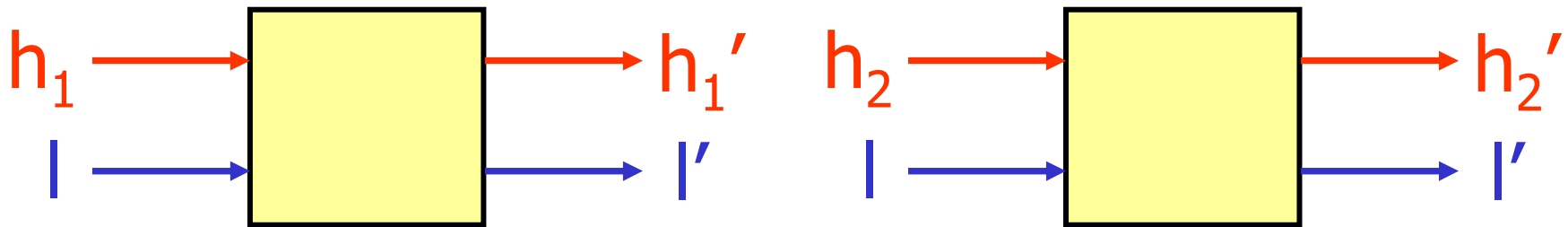
Security lattice and noninterference



Noninterference: flow from l to l' allowed when $l \sqsubseteq l'$

Noninterference

- Noninterference [Goguen & Meseguer]: as **high** input varied, **low**-level outputs unchanged



- Language-based noninterference for C:

$$\forall M_1, M_2. M_1 =_l M_2 \Rightarrow \langle M_1, c \rangle \approx_l \langle M_2, c \rangle$$

Low-memory equality:

$$M_1 =_l M_2 \text{ iff } M_1|_l = M_2|_l$$

Configuration with M_1 and c

Low view \approx_l :

$$M_1 \approx_L M_2 \text{ whenever } M_1 \neq \perp \neq M_2 \Rightarrow M_1 =_L M_2$$

Average salary

- Intention: release average

```
avg := declassify((h1 + ... + hn) / n, low);
```

- Flatly rejected by noninterference
- If accepting, how do we know declassify does not release more than intended?
- Essence of the problem: **what** is released?
- “Only declassified data and no further information”
- Expressions under declassify: **“escape hatches”**

Delimited release

- Command c contains expressions $\text{declassify}(e_i, L_i)$; c is secure if:

$$\forall L, M_1, M_2 (M_1 =_L M_2). \forall i (L_i \sqsubseteq L). \text{eval}(M_1, e_i) = \text{eval}(M_2, e_i) \Rightarrow \llbracket C \rrbracket M_1 \approx_L \llbracket C \rrbracket M_2$$

if m and m' are indistinguishable through all $e_i \dots$

- Noninterference \Rightarrow security
- For programs with no declassification:
Security \Rightarrow noninterference

...then the entire program may not distinguish m and m'

Average salary revisited

- Accepted by delimited release:

```
avg:=declassify((h1+...+hn)/n,low);
```

```
temp:=h1; h1:=h2; h2:=temp;  
avg:=declassify((h1+...+hn)/n,low);
```

- Laundering attack rejected:

```
h2:=h1;...; hn:=h1;  
avg:=declassify((h1+...+hn)/n,low);
```

~ avg:=h₁

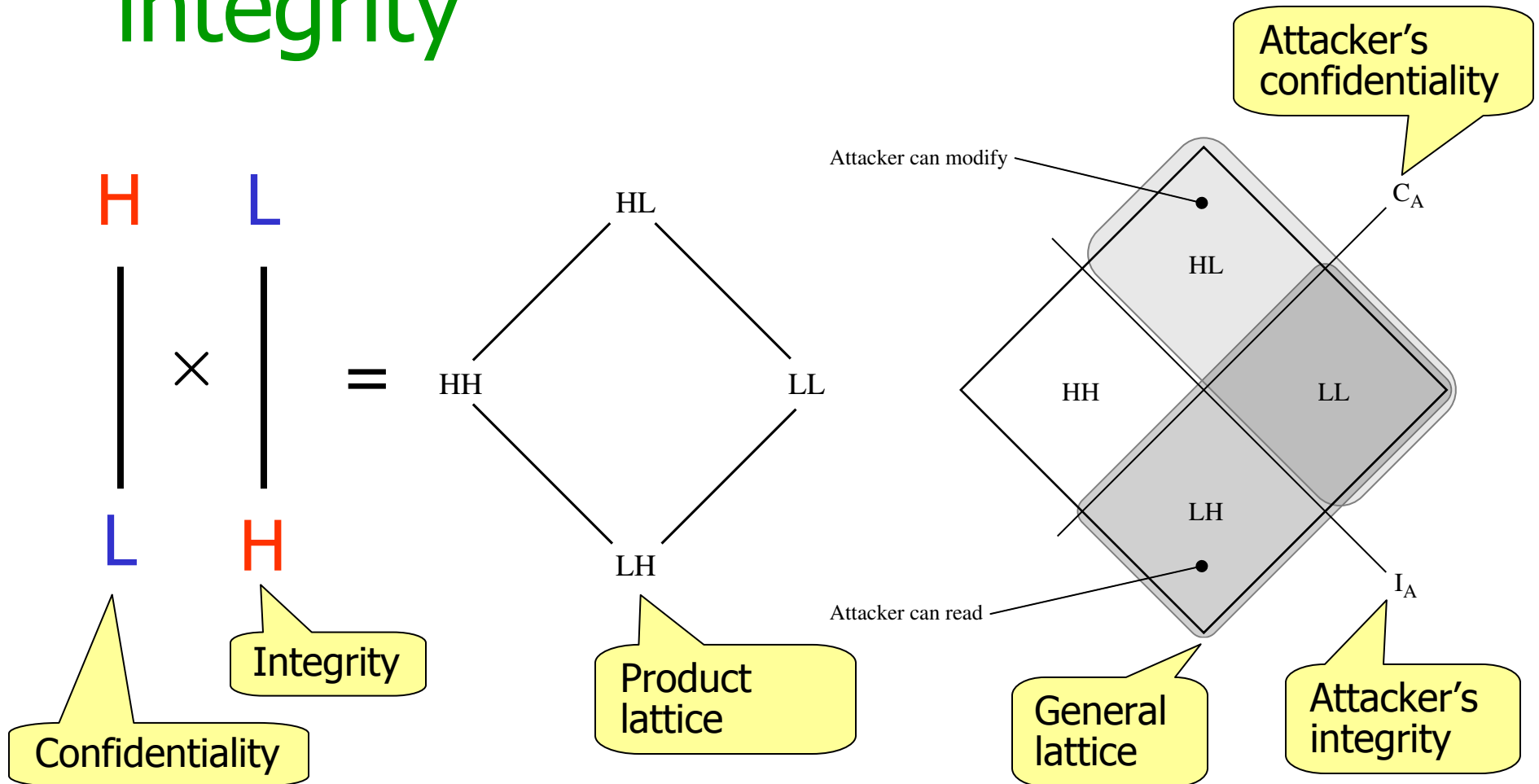
Conditional delimited release and robust declassification

- Only one of h_1 or h_2 is released:

```
if l then l:=declassify( $h_1$ ,low) else l:=declassify( $h_2$ ,low)
```

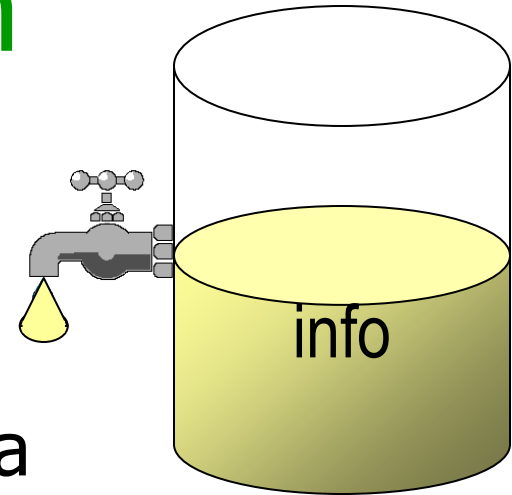
- ...yet both h_1 and h_2 are considered released
- Generally: need to prevent the attacker to control when information is released
- Robust declassification: attacker's actions may not influence observations about secrets [Zdancewic & Myers'01]

Combining confidentiality and integrity



Confidentiality guarantee: Robust declassification

- Attacker may not affect what is released
- Zdancewic & Myers [CSFW01]: An **active** attacker may not learn more sensitive information than a **passive** attacker
- Unresolved questions:
 - What is robust declassification for code?
 - How to represent untrusted code?
 - How to provably enforce robust declassification?
 - How to grant untrusted code a limited ability to control declassification?



Fair attacks

- A command a is a **fair attack** if it may only read and write variables at $l \in LL$
- A program c is high-integrity code interspersed with fair attacks
- High-integrity code $c[\bullet]$ with holes whose contents controlled by attacker
- Can fair attacks lead to laundering?

Robust declassification

- Command $c[\bullet]$ has robustness if

$$\forall M_1, M_2, a, a'. \langle M_1, c[a] \rangle \approx_l \langle M_2, c[a] \rangle \Rightarrow \langle M_1, c[a'] \rangle \approx_l \langle M_2, c[a'] \rangle$$

fair attacks

up to high-confidentiality stuttering

- If a cannot distinguish bet. M_1 and M_2 through c then no other a' can distinguish bet. M_1 and M_2
- Noninterference \Rightarrow robustness
- For programs with no declassification: robustness \Rightarrow noninterference

Robust declassification: examples

- Flatly rejected by noninterference, but secure programs satisfy robustness:

$[\bullet]; x_{LH} := \text{declassify}(y_{HH}, LH)$

$[\bullet]; \text{if } x_{LH} \text{ then } y_{LH} := \text{declassify}(z_{HH}, LH)$

- Insecure program:

$[\bullet]; \text{if } x_{LL} \text{ then } y_{LL} := \text{declassify}(z_{HH}, LH)$

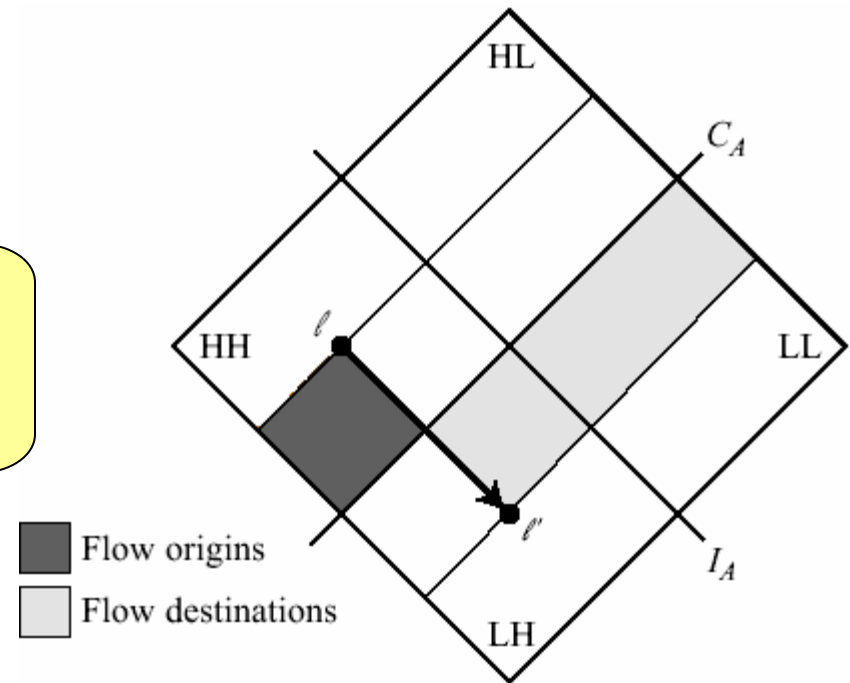
is rejected by robustness

Enforcing robustness

- Security typing for declassification:

context must be high-integrity

data must be high-integrity

$$LH \vdash e : HH$$
$$LH \vdash \text{declassify}(e, l') : LH$$


Endorsement and qualified robustness

- Need to give untrusted code limited ability to affect declassification

```
[•]; if  $x_{LL} = 1$  then  $y_{LH} := \text{declassify}(z_{HH}, LH)$   
      else  $y_{LH} := \text{declassify}(z'_{HH}, LH)$ 
```

- Introduce **endorse** to upgrade trust
- Semantic treatment of endorse:

```
 $\langle M, \text{endorse}(e, l) \rangle \rightarrow \text{val}$  (for some val)
```

- This qualifies robustness: insensitive to how endorsed expressions evaluate

Enforcing qualified robustness

- Qualified robustness:

$$\forall M_1, M_2, a, a'. \langle M_1, c[a] \rangle \approx_l \langle M_2, c[a] \rangle \Rightarrow \langle M_1, c[a'] \rangle \approx_l \langle M_2, c[a'] \rangle$$

possibilistic high-indistinguishability

- Typing rule for endorse:

direct flows

confidentiality unchanged

$$\frac{pc \vdash e:l' \quad l \sqcup pc \sqsubseteq \text{Level}(v) \quad C(l)=C(l')}{pc \vdash v:=\text{endorse}(e,l)}$$

Security typing assures

- c typable and no declassification or endorsement in c
 \Rightarrow noninterference
- c typable and no declassify in c
 \Rightarrow noninterference for confidentiality
- c typable \Rightarrow qualified robustness
- Example of breaking qualified robustness:

```
[•]; if  $x_{LL}$  then  $y_{LH} := \text{endorse}(z_{LL}, LH)$ ;  
      if  $y_{LH}$  then  $v_{LH} := \text{declassify}(w_{HH}, LH)$ 
```

rightfully rejected by type system

Battleship game security

- Players place their ships on their grid boards in secret
- Take turn in firing at locations of the opponent's grid
- Locations disclosed one at a time
- Malicious opponent should not hijack control over declassification

```
while not_done do
  [•1]; m'2 := endorse(m2, LH);
  s1 := apply(s1, m'2);
  m'1 := get_move(s1);
  m1 := declassify(m'1, LH);
  not_done :=
    declassify(not_final(s1), LH);
  [•2]
```

```
Level(s1, m'1) ∈ HH
Level(m1, m'2, not_done) ∈ LH
Level(m2) ∈ LL
```

⇒ Typable and thus secure

Related work on information release

- What? Partial release: noninterference within **high** subdomains [Cohen'78, Joshi & Leino'00, Sabelfeld & Sands'00, Giacobazzi & Mastroeni'04]
- Where? Intransitive (non)interference: to be declassified data must pass a downgrader [Rushby'92, Pinsky'95, Roscoe & Goldsmith'99, Mantel'01, Mantel & Sands'03]
- Who? Decentralized label model: only owner has authority to declassify data [Myers & Liskov'97,'98]
Robust declassification: active attacker may not learn more information than passive attacker [Zdancewic & Myers'01, Zdancewic'03]

Related work on information release

- How much? (where+what) Quantitative information flow [Denning'82, Clark et al.'02, Lowe'02]
- Relative to what?
 - probabilistic attacker [Volpano & Smith'00, Volpano'00, Di Pierro'02]
 - complexity-bound attacker [Laud'01,'03]
 - specification-bound attacker [Dam & Giambiagi'00,'03]

Ongoing/future work

[jointly with David Sands]

“Grand theory” of declassification

- **Scrambling** and **scope-bound release** to connect delimited release, intransitive noninterference, and qualified robustness
- Basic principles of declassification
 - **Security monotonicity of release**: removing declassification from an insecure program should not make the program secure
 - **Undercover release**: Replacing a subprogram with no declassify by a semantically equivalent program should not change the (in)security

Conclusions



Delimited release model

- provides policies for **what** can be leaked
- prevents information laundering
- opportunities for wrapping security-typed code (e.g., Jif) into conventional code (e.g., Java) with no additional leaks

Enforcing robust declassification

- Language-level characterization and enforcement
- Explicit attackers – untrusted code
- Qualified robustness

Need a framework for understanding relation (“**what**”, “**where**”, and “**who**” axes) and enable combination of declassification policies