Enforcing Robust Declassification

Andrei Sabelfeld
Chalmers

joint work with

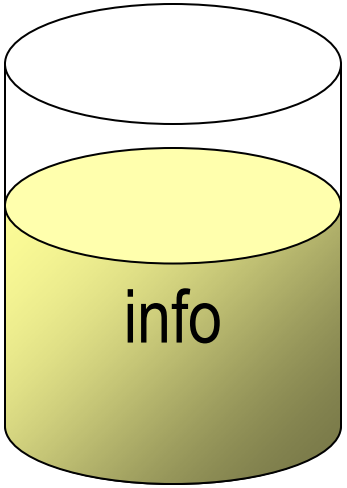
Andrew C. Myers
Cornell

&

Steve Zdancewic
U. Penn

CSFW04, June 2004

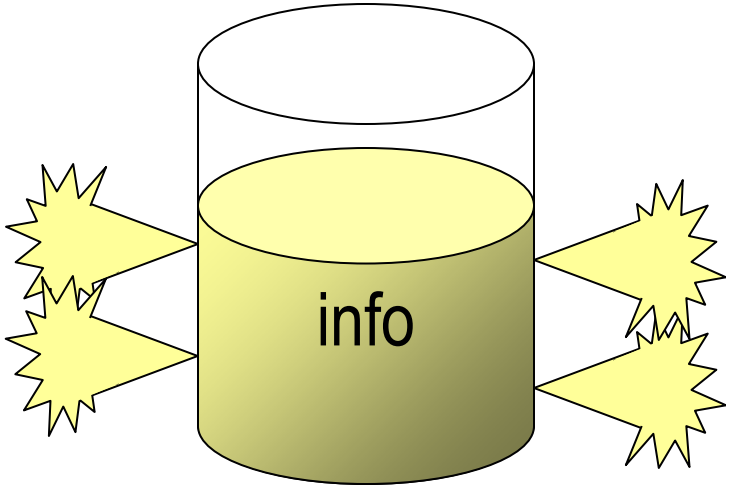
Confidentiality: preventing information leaks



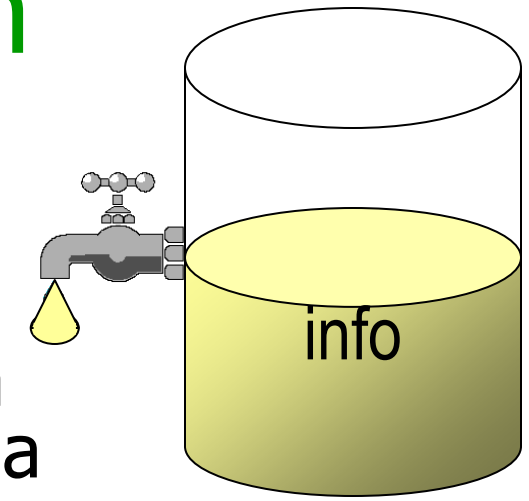
- Untrusted/buggy code should not leak sensitive information
- But some applications depend on intended information leaks
  - password checking
  - information purchase
  - spreadsheet computation
  - …
- Some leaks must be allowed: need information release (or declassification)

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 guarantee for declassification constructs in many security-typed languages



Confidentiality guarantee: Robust declassification



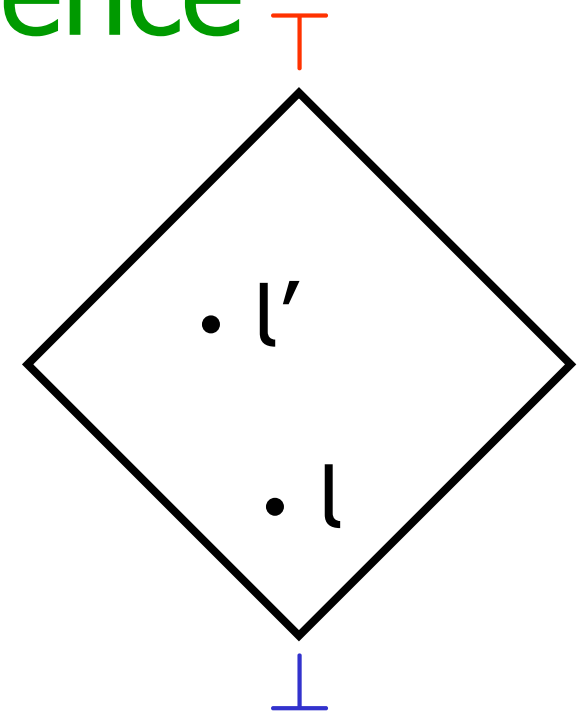
- Attacker may not affect what is released
- Zdancewic & Myers [CSFW01]: An active attacker may not learn more sensitive information than a passive attacker
- Unresolved questions:
  - What is robust declassification for code?
  - How to represent untrusted code?
  - How to provably enforce robust declassification?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

This talk

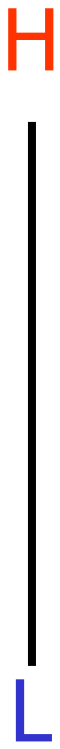
- Language-level end-to-end robust declassification
- Explicit attackers – untrusted code
- Robust declassification enforcement by security typing
- Qualified robustness – limited ability for untrusted code to affect declassification
- Non-dual view – integrity represents whether code has enough authority to declassify
- Related/ongoing work & conclusions

Security lattice and noninterference

Security lattice:

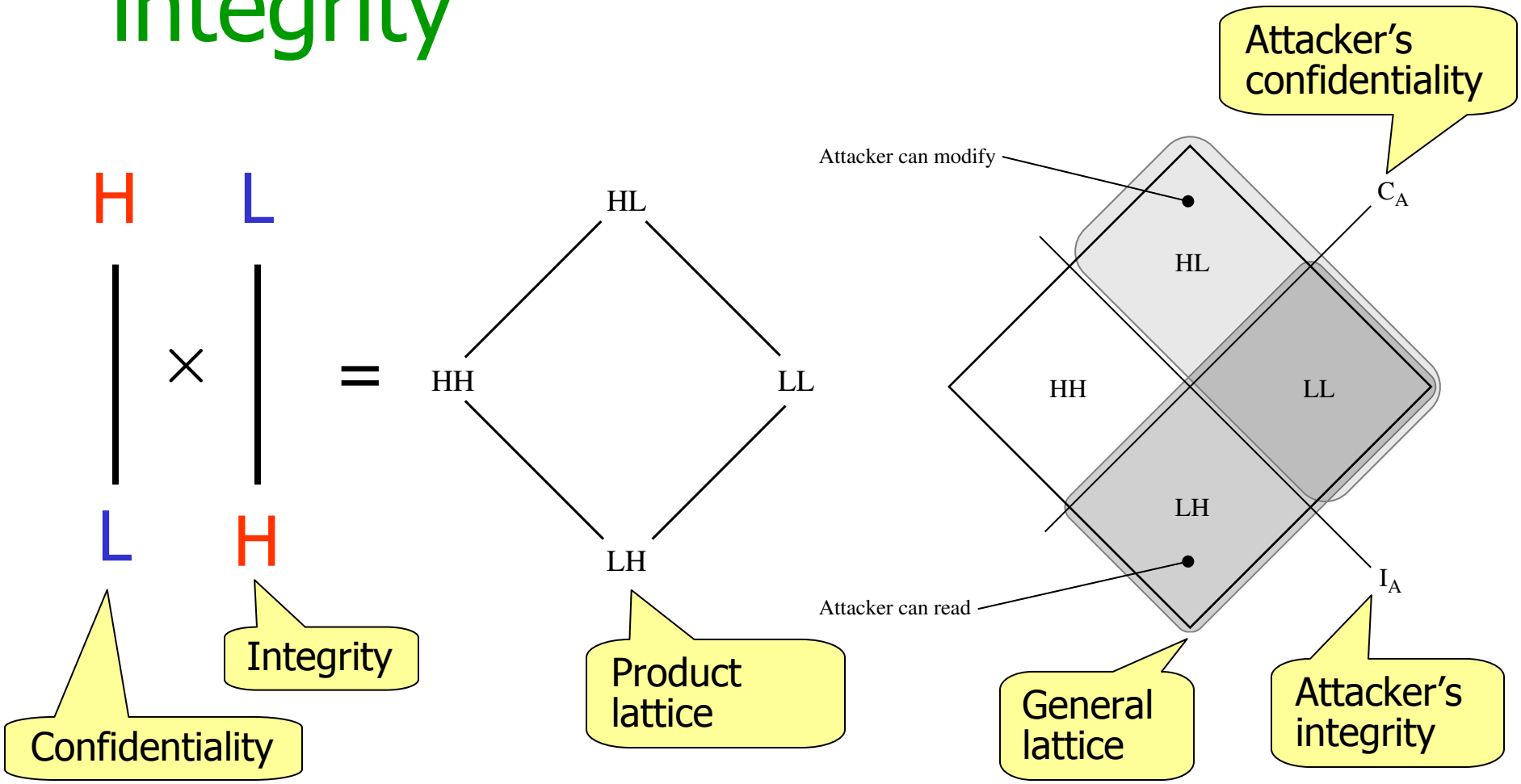


e.g.:



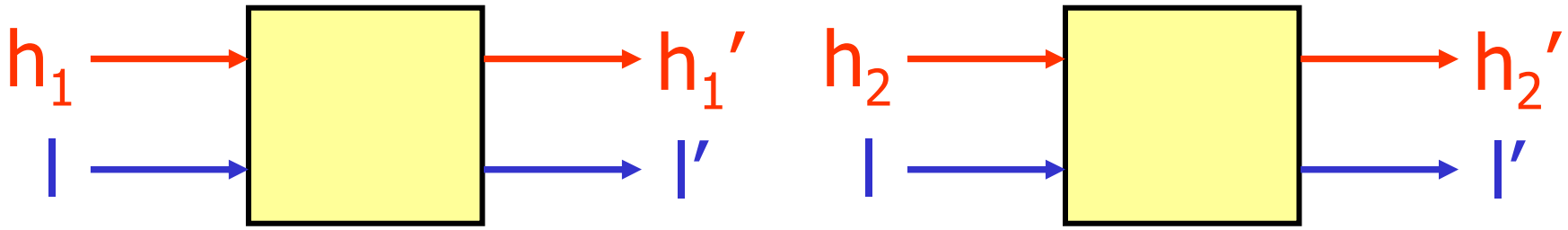
Noninterference: flow from l to l′ allowed when $l \sqsubseteq l'$

Combining confidentiality and integrity



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:
$M_1 =_l M_2$ iff $M_1|_l = M_2|_l$

Configuration with M_1 and c

Low view $\approx_l$: trace indistinguishability up to high stuttering

From noninterference to robustness

- Noninterference too restrictive
- Need to allow declassification but so that the attacker may not affect it
- Zdancewic & Myers [CSFW01]: An active attacker may not learn more sensitive information than a passive attacker

⇒ Model both kinds of attackers relative to a point in security lattice

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[•] with holes whose contents controlled by attacker
- Can fair attacks lead to laundering?

Robust declassification

- Command c[•] has robustness if

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

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

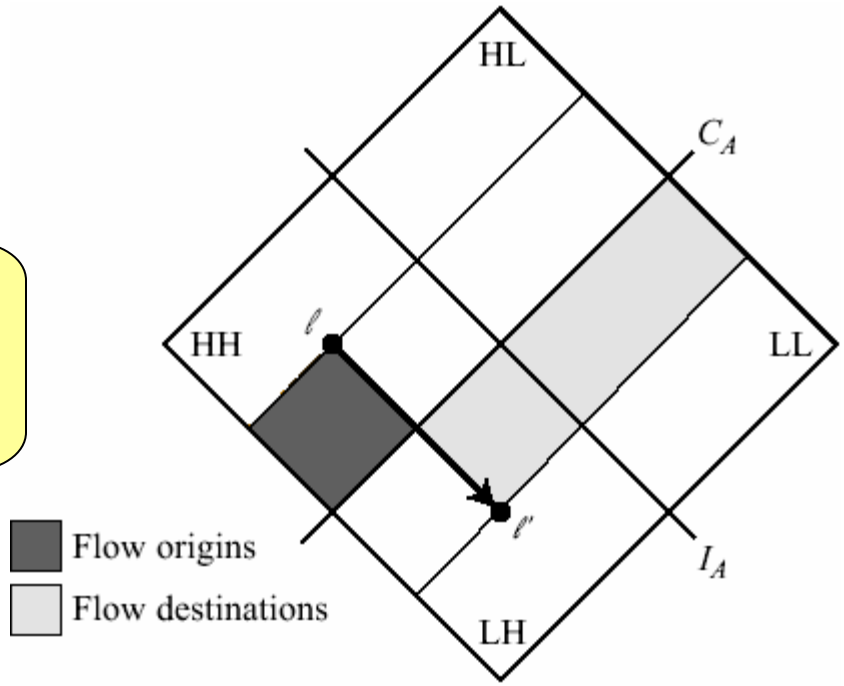
Enforcing robustness

- Security typing for declassification:

context must be high-integrity

data must be high-integrity

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



Security typing assures

- c typable and no declassification in c $\Rightarrow$ noninterference
- c typable $\Rightarrow$ noninterference for integrity (no downward flows along the integrity axis)
- c typable $\Rightarrow$ robustness

Password checking security

- Password+salt are hashed in a (public) image database

$$LH \vdash \text{hash(pwd, salt)}: HH \times LH \rightarrow LH$$
$$= \text{declassify(buildHash(pwd||salt), } LH)$$

- User query+salt is matched with the image

$$LH \vdash \text{match(pwdImg, salt, query)}: LH \times LH \times HH \rightarrow LH$$
$$= \text{pwdImg==hash(query, salt)}$$

$\Rightarrow$ Typable and thus secure

Password laundering attack

- Program leaking the parity of x_{HH}

  $[\bullet]$; match(hash(parity(x_{HH}),salt), salt, y_{LL})

  is rejected by type system

- Password updated with newPwd if hashing oldPwd+salt matches the image:

$LH \vdash$ update(pwdImg, salt, oldPwd, newPwd) :
$LH \times LH \times HH \times HH$
= if match(pwdImg, salt, oldPwd)
then pwdImg:=hash(newPwd, salt)

$\Rightarrow$ Typable and thus secure

Endorsement and qualified robustness

- Need to give untrusted code limited ability to affect declassification

$[\bullet]$; if $x_{LL}=1$ then y_{LH}:=declassify(z_{HH},LH)
else y_{LH}:=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:

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

possibilistic high-indistinguishability

- Typing rule for endorse:

direct flows

confidentiality unchanged

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

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:

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

rightfully rejected by type system

Battleship game security

- Players places 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

$\Rightarrow$ Typable and thus secure

```
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]
```

$Level(s_1, m'_1) \in HH$
$Level(m_1, m'_2, not\_done) \in LH$
$Level(m_2) \in LL$

Related work on information release

- What? Partial release: noninterference within high subdomains [Cohen'78, Joshi & Leino'00, Sabelfeld & Sands'00, Giacobazzi & Mastroeni'04, Sabelfeld & Myers'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 that passive attacker [Zdancewic & Myers'01, Zdancewic'03]

Related work on information release

- How much? 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]

Conclusions

Enforcing robust declassification

- Language-level characterization and enforcement
- Explicit attackers – untrusted code
- Qualified robustness – limited ability for untrusted code to affect declassification
- Non-dual view – integrity represents whether code has enough authority to declassify

Future work: generalizations to concurrent attackers and combination with intransitive noninterference