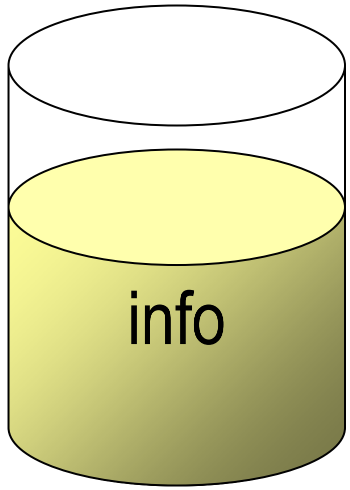
Course outline: the four hours

1. Language-Based Security: motivation
2. Language-Based Security: the big picture
3. Confidentiality for sequential and multithreaded programs (on the board)
4. Mechanisms for safe information release

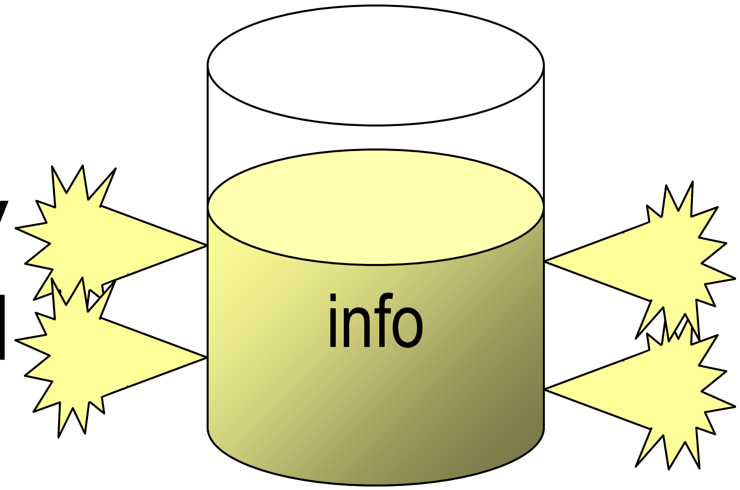
today (items 3–4)

Confidentiality: preventing information leaks



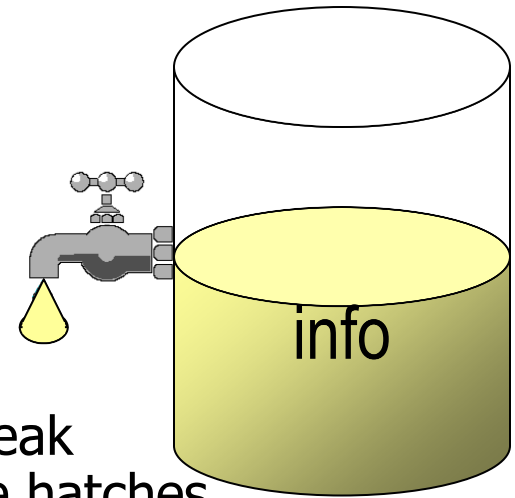
- Untrusted/buggy code should not leak sensitive information
- But some applications depend on intended information leaks
  - password checking
  - information purchase
  - spreadsheet computation
  - ...
- Some leaks must be allowed: need information release (or declassification)

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

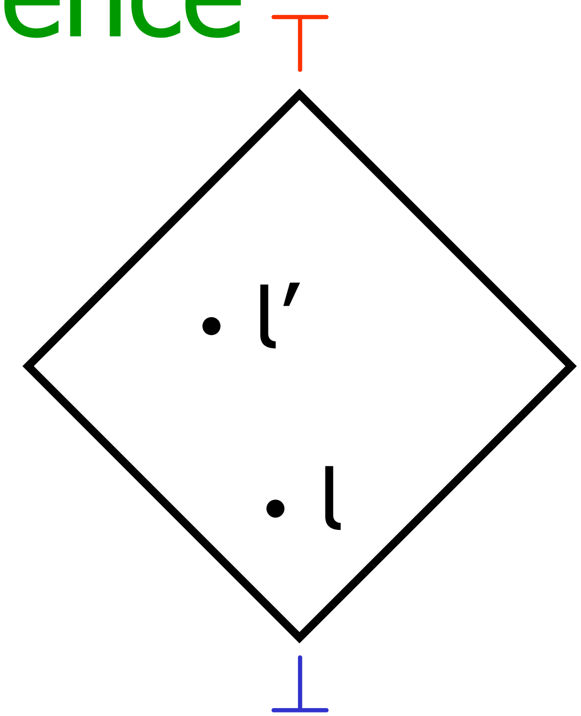
Leveraging assurance in presence of declassification

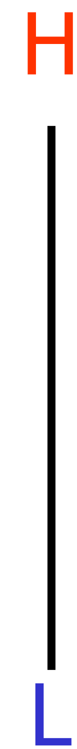


Two approaches:

- 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

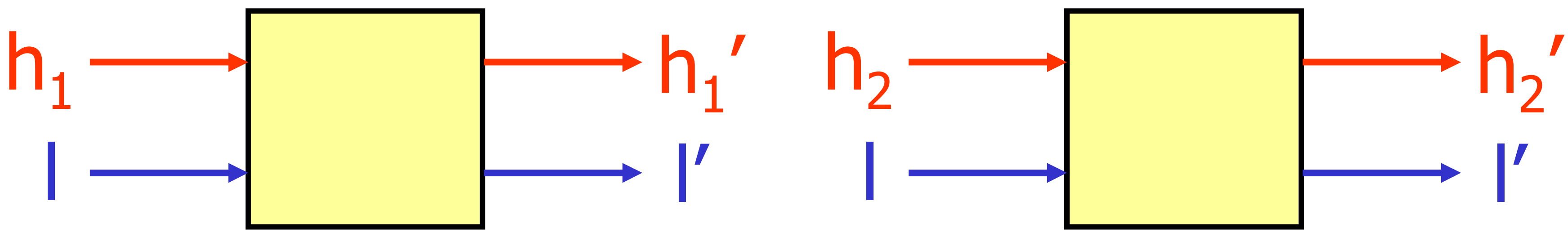
Security lattice: 

e.g.: 

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

Configuration with M_1 and c

Low view $\approx_l$:
$M_1 \approx_L M_2$ whenever $M_1 \neq \bot \neq M_2 \Rightarrow M_1 =_L M_2$

Average salary

- Intention: release average

$$avg:=declassify((h_1+...+h_n)/n,low);$$

- 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

if M_1 and M_2 are indistinguishable through all e_i…

- Command c contains expressions declassify(e_i,L_i); c is secure if:

$$\forall L, M_1, M_2\ (M_1 =_L M_2).\forall i\ (L_i \sqsubseteq L).\mathrm{eval}(M_1, e_i) = \mathrm{eval}(M_2, e_i) \Rightarrow [\![C]\!]M_1 \approx_L [\![C]\!]M_2$$

…then entire program may not distinguish M_1 and M_2

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

Average salary revisited

- Accepted by delimited release:

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

$avg := declassify((h_1 + ... + h_n)/n, low);$

$temp := h_1;\ h_1 := h_2;\ h_2 := temp;$
$avg := declassify((h_1 + ... + h_n)/n, low);$

- Laundering attack rejected:

$h_2 := h_1; ...;\ h_n := h_1;$
$avg := declassify((h_1 + ... + h_n)/n, low);$ $\sim$ $avg := h_1$

Electronic wallet

- If enough money then purchase

```
if declassify(h≥k,low) then (h:=h-k; l:=l+k);
```

amount in wallet (h) · cost (k) · spent (l)

- Accepted by delimited release

Electronic wallet attack

- Laundering bit-by-bit attack (h is an n-bit integer)

```
l:=0;
while(n≥0) do
   k:=2^(n-1);
   if declassify(h≥k,low)
      then (h:=h-k; l:=l+k);
   n:=n-1;
```

$\sim$

```
l:=h
```

- Rejected by delimited release

Security type system

- Basic idea: prevent new information from flowing into variables used in escape hatch expressions

```
h:=…;
…
declassify(h,low)
```

may not use other (than h) high variables

```
while … do
  declassify(h,low)
  …
  h:=…;
```

may not use other (than h) high variables

- Theorem:
  c is typable $\Rightarrow$ c is secure

Delimited release in password checking

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

$$\text{hash(pwd, salt)}: L_{pwd} \times L_{salt} \rightarrow \text{low}$$
$$= \text{declassify(buildHash(pwd||salt), low)}$$

- User query+salt is matched with the image

$$\text{match(pwdImg, salt, query)}: L_{pwdImg} \times L_{salt} \times L_{query} \rightarrow L_{pwdImg} \sqcup \text{low}$$
$$= \text{pwdImg==hash(query, salt)}$$

Delimited release in password checking

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

```
update(pwdImg, salt, oldPwd, newPwd)  (low ⊑ L_pwdImg)
   = if match(pwdImg, salt, oldPwd)
        then pwdImg:=hash(newPwd, salt)
```

- hash, match, and update are typable and thus secure

Delimited release in password checking: instantiation

- Honest user hashing password:
  hash(pwd, salt): high $\times$ low $\to$ low
- Attacker hashing password (with user's salt):
  hash(pwd, salt): low $\times$ low $\to$ low
- Honest user matching his password:
  match(pwdImg, salt, query): low $\times$ low $\times$ high $\to$ low
- Attacker guessing a password (with user's salt):
  match(pwdImg, salt, query): low $\times$ low $\times$ low $\to$ low

Delimited release in password checking: instantiation

- Honest user updating password:
  update(pwdImg, salt, oldPwd, newPwd):
  low × low × high× high
- Attacker attempting to update the honest users's password:
  update(pwdImg, salt, oldPwd, newPwd):
  low × low × low × low

Rationale for security:
to succeed the attacker needs to guess secrets

Password checking laundering

```
l:=0;
while(n≥0) do
  k:=2^(n-1);
  if hash(sign(h-k+1),salt)=hash(1,salt)
    then (h:=h-k; l:=l+k);
  n:=n-1;
```

$\sim$ h≥k

$\sim$ l:=h

- Attack rejected by type system
- Average salary and electronic wallet are accepted; respective laundering attacks are rejected

Programming with delimited release

- Program intended to release the parity of h':

```
h:=parity(h');
if declassify(h=1,low) then l:=1 else l:=0;
```

- …insecure and rejected by type system
- Programmer forced to rewrite:

```
h:=parity(h');
if declassify(parity(h'),low) then l:=1 else l:=0;
```

- …secure and typable
- Potential for automated transformation

Security ordering

l_1:=declassify(h_1);
l_2:=declassify((h_1+h_2)/2);

…leaks as much as:

l_1:=declassify(h_1);
l_2:=declassify(h_2);

- In other words: $C_1 \approx_S C_2$ under
- $C_1 \sqsubseteq_S C_2$ if for all L, M_1, M_2 ($M_1 =_L M_2$) whenever $[\![C_1]\!]M_1 \approx_L [\![C_1]\!]M_2$ then $[\![C_2]\!]M_1 \approx_L [\![C_2]\!]M_2$
- $\approx_S$ can be decidably approximated
- $\approx_S$ different from semantic equivalence

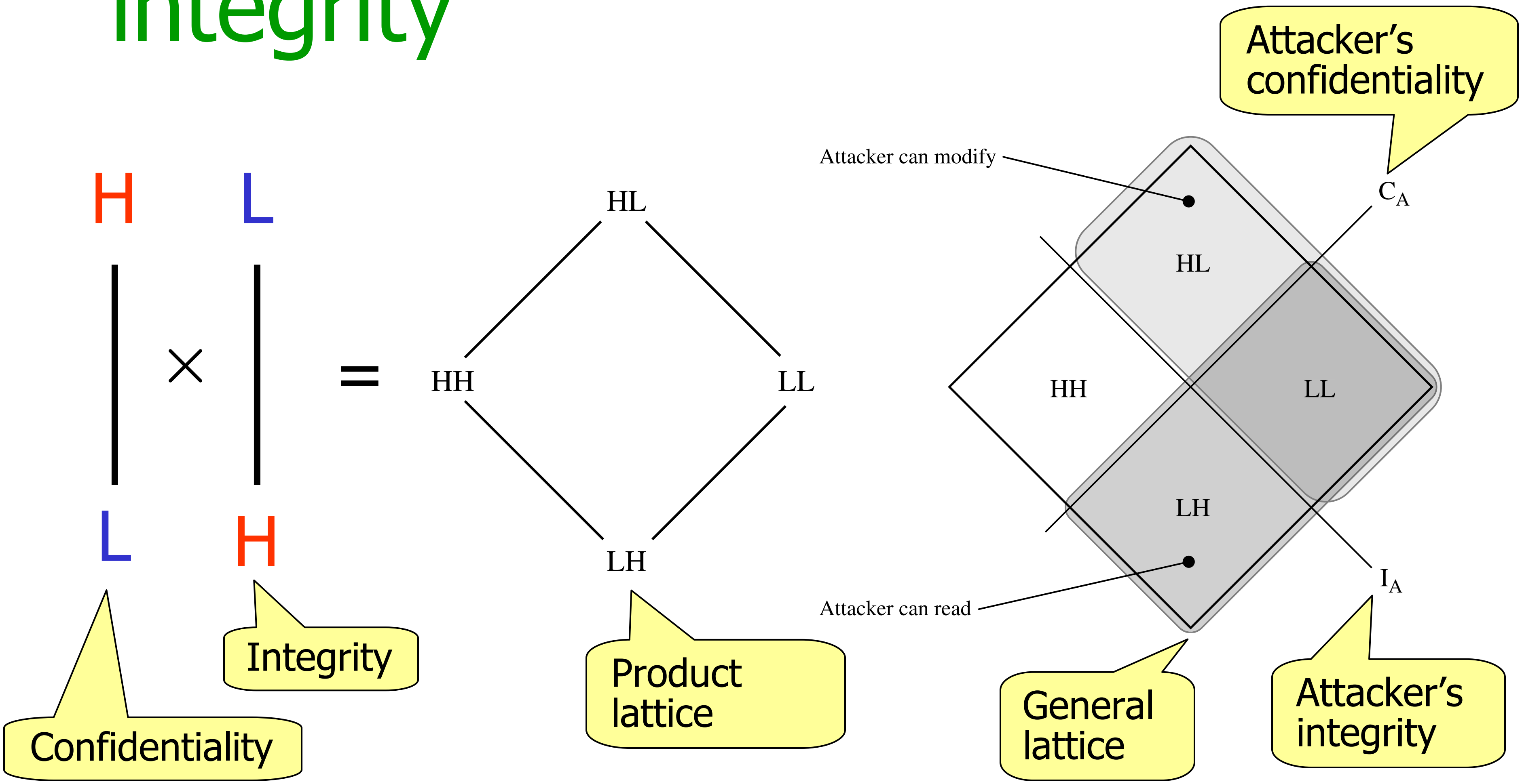
Conditional delimited release and robust declassification

- Only one of h_1 or h_2 is released:

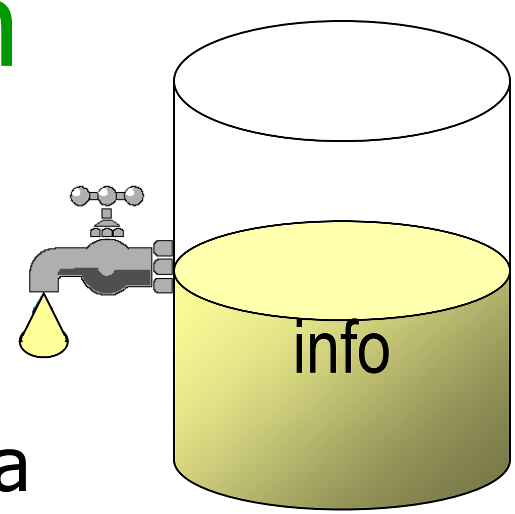
```
if l then l:=declassify(h1,low) else l:=declassify(h2,low)
```

- …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 $\ell \in LL$
- 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

Robust declassification: examples

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

[•]; x_{LH}:=declassify(y_{HH},LH)

[•]; if x_{LH} then
  y_{LH}:=declassify(z_{HH},LH)

- Insecure program:

[•]; if x_{LL} then y_{LL}:=declassify(z_{HH},LH)

is rejected by robustness

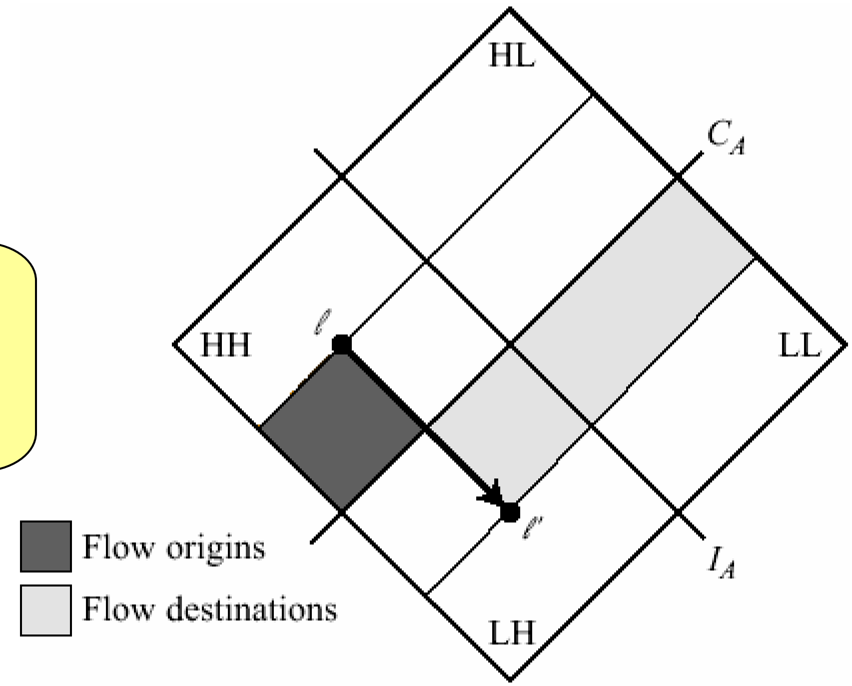
Enforcing robustness

- Security typing for declassification:

context must be high-integrity

data must be high-integrity

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



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

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

- User query+salt is matched with the image

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

$$
\begin{array}{l}
LH \vdash \text{update(pwdImg, salt, oldPwd, newPwd)} : \\
\qquad LH \times LH \times HH \times HH \\
\quad = \text{if match(pwdImg, salt, oldPwd)} \\
\qquad\quad \text{then pwdImg:=hash(newPwd, salt)}
\end{array}
$$

$\Rightarrow$ 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:

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

- 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:

[•]; if x_{LL} then y_{LH}:=endorse(z_{LL},LH);
    if y_{LH} then v_{LH}:=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
  $[\bullet_1]$; m'_2:=endorse(m_2,LH);
  s_1:=apply(s_1,m'_2);
  m'_1:=get_move(s_1);
  m_1:=declassify(m'_1,LH);
  not_done:=
    declassify(not_final(s_1),LH);
  $[\bullet_2]$

Level(s_1,m'_1) $\in$ HH
Level(m_1,m'_2,not_done) $\in$ LH
Level(m_2) $\in$ LL

$\Rightarrow$ 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 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]

Ongoing/future work
[jointly with David Sands]

Grand theory of declassification

- Scrambling 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

Delimited release model

- provides policies for what can be leaked
- prevents information laundering
- line of defense in addition to compartmentalization
- opportunities for wrapping security-typed code (e.g., Jif) into conventional code (e.g., Java) with no additional leaks

Conclusions: robust declassification

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

Course outline: the four hours

1. Language-Based Security: motivation
2. Language-Based Security: the big picture
3. Confidentiality for sequential and multithreaded programs (on the board)
4. Mechanisms for safe information release

today (items 3–4)