ProSec

Dimensions of Declassification in Theory and Practice

Andrei Sabelfeld
Chalmers

partly based on joint work with
A. Askarov and D. Sands

ASIAN'07, Dec. 2007

A scenario: free service software

Users freely download and use the software providing a service:

- Grokster, Kazaa, Morpheus,... are file-sharing services helping users exchange files
- Come with “hooks” for automatic updates
- Support advertisement to justify cost

Real story: malware

Users are tricked to download software bundled with:

- Homepage/search hijackers (MySearch)
- Unsolicited pop-up ads
- Rewriting URLs to override original ads with own
- “Hooks” for automatic updates are used to execute the advertiser’s arbitrary code (MediaUpdate, DownLoadware)
- Information gathering—visited URLs and filled forms are forwarded to a third-party (Gator, IPInsight, Transponder)

General problem: malicious and/or buggy code is a threat

- Trends in software
  - mobile code, executable content
  - platform-independence
  - extensibility
- These trends are attackers’ opportunities!
  - easy to distribute worms, viruses, exploits,...
  - write (an attack) once, run everywhere
  - systems are vulnerable to undesirable modifications
- Need to keep the trends without compromising information security

Need for language-based security

- Looking under the street light...
  Common attacker model:
  - eavesdropping on network
  - modifying network traffic
  - trusted communication endpoints

$\Rightarrow$ cryptographic protection of communication

- ...for a key that lies somewhere else!
  Real story [CERT]: Most attacks are
  - remote penetrations (buffer overruns, format strings, RPC vulnerabilities,...)
  - malware (viruses, worms, DDoS slaves,...)

$\Rightarrow$ need protection at application/language level

Information security: confidentiality

- Confidentiality: sensitive information must not be leaked by computation (non-example: spyware attacks)
- End-to-end confidentiality: there is no insecure information flow through the system
- Standard security mechanisms provide no end-to-end guarantees
  - Security policies too low-level (legacy of OS-based security mechanisms)
  - Programs treated as black boxes

Confidentiality: standard security mechanisms

Access control

+prevents "unauthorized" release of information

- but what process should be authorized?

Firewalls

+permit selected communication

- permitted communication might be harmful

Encryption

+secures a communication channel

- even if properly used, endpoints of communication may leak data

Confidentiality: standard security mechanisms

Antivirus scanning

+rejects a “black list” of known attacks

- but doesn’t prevent new attacks

Digital signatures

+help identify code producer

- no security policy or security proof guaranteed

Sandboxing/OS-based monitoring

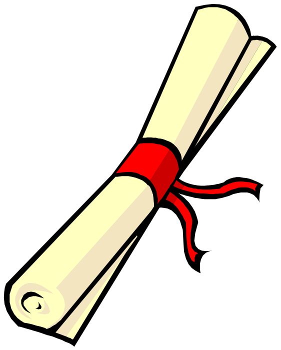
+good for low-level events (such as read a file)

- programs treated as black boxes

$\Rightarrow$ Useful building blocks but no end-to-end security guarantee

Confidentiality: language-based approach

- Counter application-level attacks at the level of a programming language—look inside the black box! Immediate benefits:
- Semantics-based security specification
  - End-to-end security policies
  - Powerful techniques for reasoning about semantics



- Security enforcement
  - Analysis enforcing end-to-end security
  - Track information flow via, e.g., security types
    - Type checking by the compiler removes run-time overhead

Dynamic security enforcement

Java's sandbox, OS-based monitoring, and Mandatory Access Control dynamically enforce security policies; But:

high(secret)

low(public)

```
h:=…;
l:=false;
if h then l:=true
     else skip
```

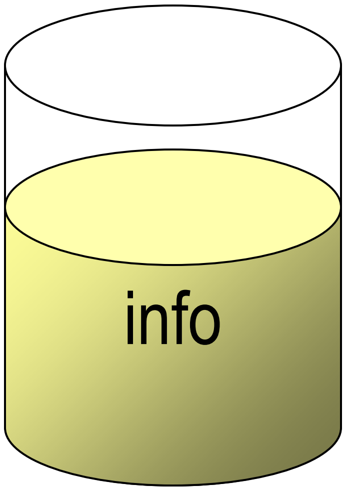
implicit flow from h to l

Problem: monitoring a single execution path is not enough!

Static certification

- Only run programs which can be statically verified as secure before running them
- Static certification for inclusion in a compiler [Denning & Denning'77]
- More precise implicit flow analysis
- Enforcement by static analysis (e.g., security-type systems)

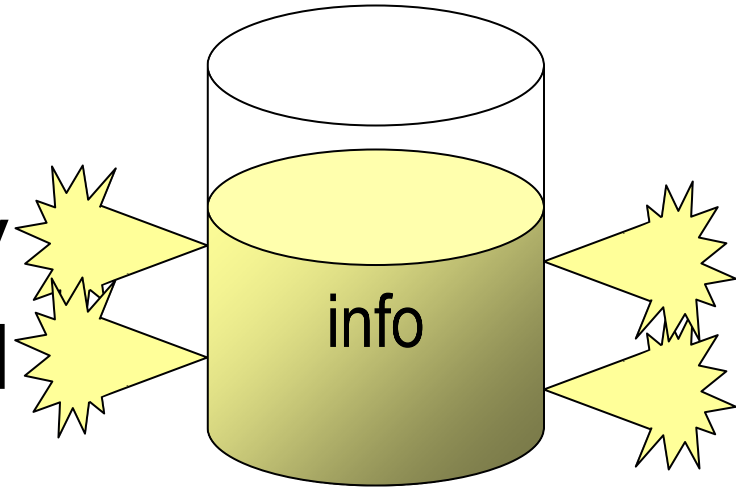
Confidentiality: preventing information leaks

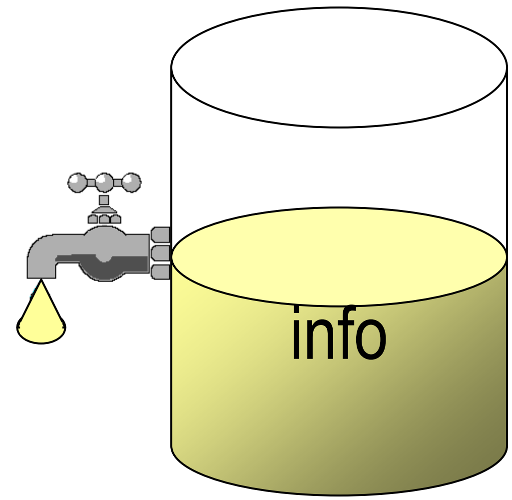


- Untrusted/buggy code should not leak sensitive information
- But some applications depend on intended information leaks
  - password checking
  - information purchase
  - spreadsheet computation
  - ...
- Some leaks must be allowed: need information release (or declassification)

Confidentiality vs. intended leaks

- Allowing leaks might compromise confidentiality
- Noninterference is violated
- How do we know secrets are not laundered via release mechanisms?
- Need for security assurance for programs with release

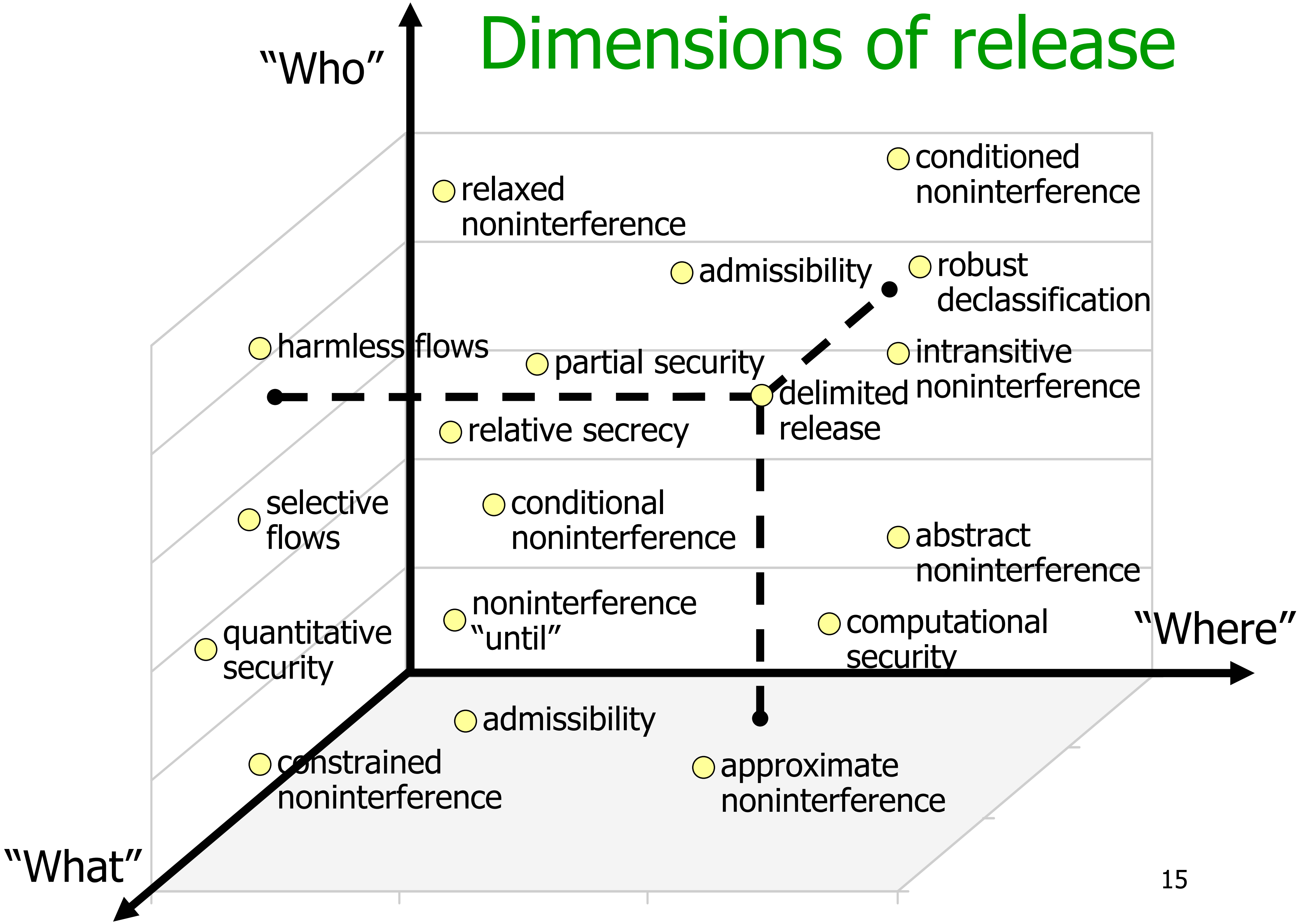




State-of-the-art

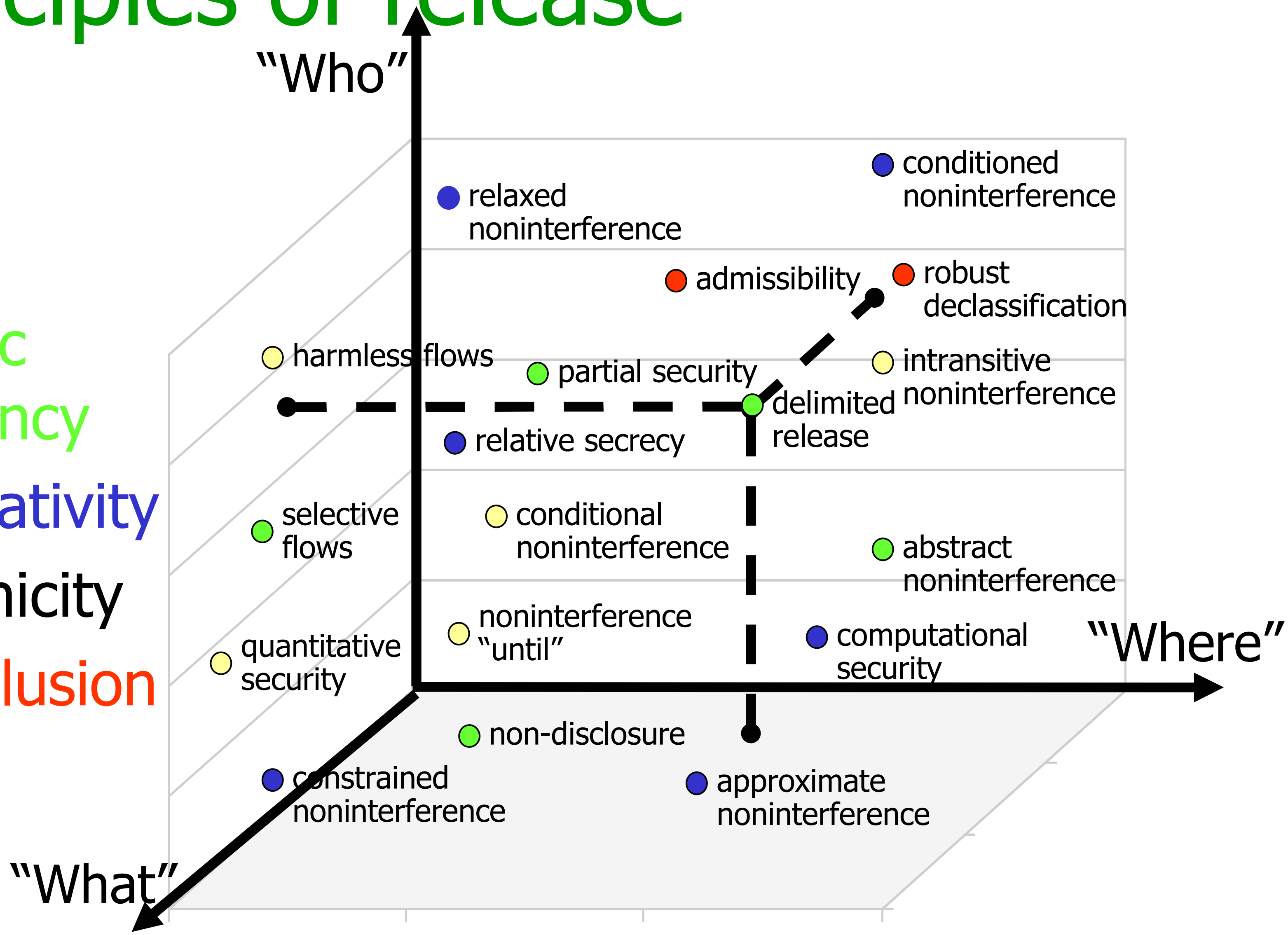
- conditioned noninterference
- relaxed noninterference
- admissibility
- robust declassification
- harmless flows
- partial security
- intransitive noninterference
- delimited release
- relative secrecy
- selective flows
- conditional noninterference
- abstract noninterference
- noninterference "until"
- computational security
- quantitative security
- admissibility
- constrained noninterference
- approximate noninterference

Dimensions of release



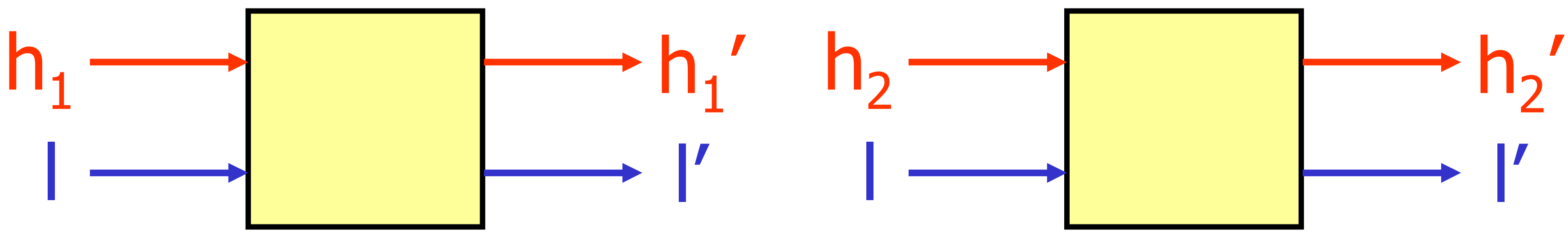
Principles of release

- Semantic consistency
- Conservativity
- Monotonicity
- Non-occlusion



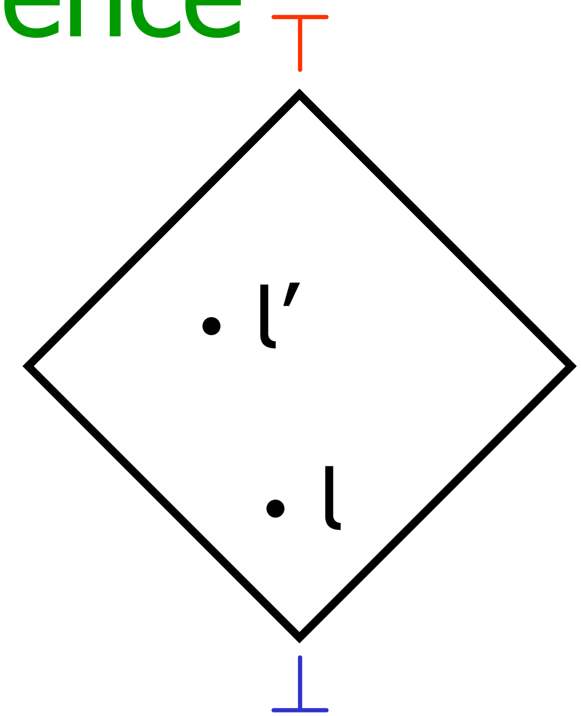
What

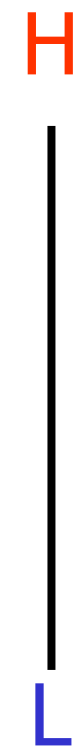
- Noninterference [Goguen & Meseguer]: as high input varied, low-level outputs unchanged



- Selective (partial) flow
  - Noninterference within high sub-domains [Cohen'78, Joshi & Leino'00]
  - Equivalence-relations view [Sabelfeld & Sands'01]
  - Abstract noninterference [Giacobazzi & Mastroeni'04,'05]
  - Delimited release [Sabelfeld & Myers'04]
- Quantitative information flow [Denning'82, Clark et al.'02, Lowe'02]

Security lattice and noninterference

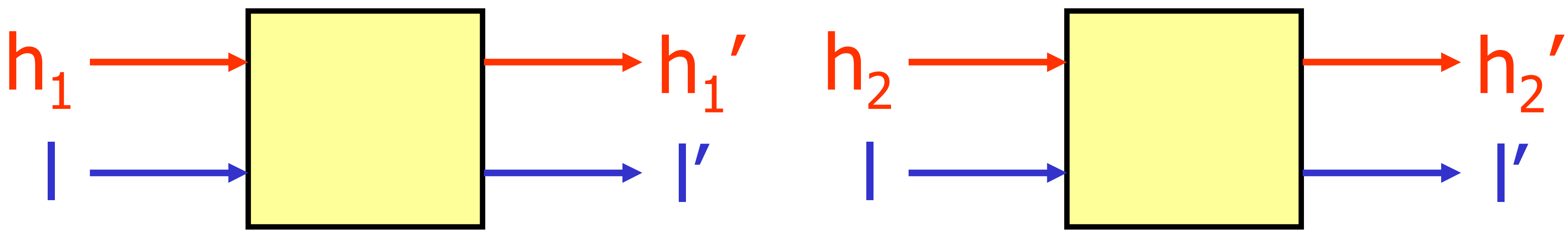
Security lattice: 

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:

$$M_1 =_L M_2 \ \& \ \langle M_1, c\rangle \Downarrow M'_1 \ \& \ \langle M_2, c\rangle \Downarrow M'_2 \Rightarrow M'_1 =_L M'_2$$

Low-memory equality:
$M_1 =_L M_2$ iff $M_1|_L = M_2|_L$

Configuration with M_2 and c

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

[Sabelfeld & Myers, ISSS'03]

- Command c has expressions declassify(e_i,L); c is secure if:

$$M_1 =_L M_2 \;\&\; \langle M_1, c\rangle \Downarrow M'_1 \;\&\; \langle M_2, c\rangle \Downarrow M'_2 \;\&$$
$$\forall i\,.\,eval(M_1, e_i) = eval(M_2, e_i) \Rightarrow$$
$$M'_1 =_L M'_2$$

if M_1 and M_2 are indistinguishable through all e_i…

…then the entire program may not distinguish M_1 and M_2

$\Rightarrow$ security

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

Average salary revisited

- Accepted by delimited release:

```
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
```

Who

- Robust declassification in a language setting [Myers, Sabelfeld & Zdancewic'04/06]
- Command c[•] has robustness if

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

attacks

- If a cannot distinguish bet. M_1 and M_2 through c then no other a' can distinguish bet. M_1 and M_2

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

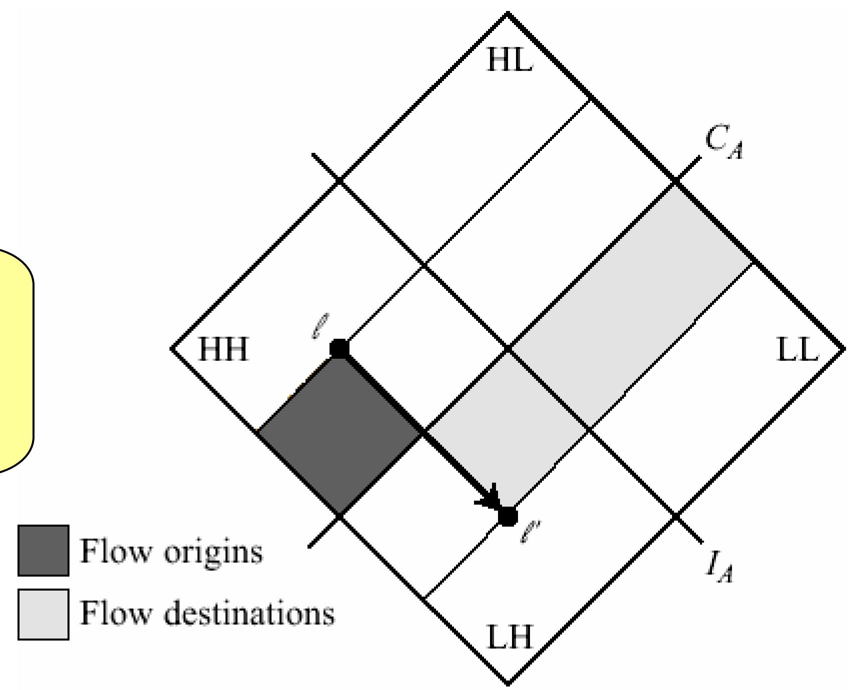
Enforcing robustness

- Security typing for declassification:

context must be high-integrity

data must be high-integrity

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



Where

- Intransitive (non)interference
  - assurance for intransitive flow [Rushby'92, Pinsky'95, Roscoe & Goldsmith'99]
  - nondeterministic systems [Mantel'01]
  - concurrent systems [Mantel & Sands'04]
  - to be declassified data must pass a downgrader [Ryan & Schneider'99, Mullins'00, Dam & Giambiagi'00, Bossi et al.'04, Echahed & Prost'05, Almeida Matos & Boudol'05]

When

- Time-complexity based attacker
  - password matching [Volpano & Smith'00] and one-way functions [Volpano'00]
  - poly-time process calculi [Lincoln et al.'98, Mitchell'01]
  - impact on encryption [Laud'01,'03]
- Probabilistic attacker [DiPierro et al.'02, Backes & Pfitzmann'03]
- Relative: specification-bound attacker [Dam & Giambiagi'00,'03]
- Non-interference "until" [Chong & Myers'04]

Principle I

Semantic consistency

The (in)security of a program is invariant under semantics-preserving transformations of declassification-free subprograms

- Aid in modular design
- "What" definitions generally semantically consistent
- Uncovers semantic anomalies

Principle II

Conservativity

Security for programs with no declassification is equivalent to noninterference

- Straightforward to enforce (by definition); nevertheless:
- Noninterference "until" rejects

```
if h>h then l:=0
```

Principle III

Monotonicity of release

Adding further declassifications to a secure program cannot render it insecure

- Or, equivalently, an insecure program cannot be made secure by *removing* declassification annotations
- "Where": intransitive noninterference (a la M&S) fails it; declassification actions are observable

```
if h then declassify(l=l) else l=l
```

Principle IV

Occlusion

The presence of a declassification operation cannot mask other covert declassifications

Checking the principles

What

Property	Semantic consistency	Conservativity	Monotonicity of release	Non-occlusion
Partial release [Coh78, JL00, SS01, GM04, GM05]	✓	✓	N/A	✓
Delimited release [SM04]	✓	✓	✓	✓
Relaxed noninterference [LZ05a]	×	✓	✓	✓
Naive release	✓	✓	✓	×

Who

Property	Semantic consistency	Conservativity	Monotonicity of release	Non-occlusion
Robust declassification [MSZ04]	✓*	✓	✓	✓
Qualified robust declassification [MSZ04]	✓*	✓	✓	×

Where

Property	Semantic consistency	Conservativity	Monotonicity of release	Non-occlusion
Intransitive noninterference [MS04]	✓*	✓	×	✓

When

Property	Semantic consistency	Conservativity	Monotonicity of release	Non-occlusion
Admissibility [DG00, GD03]	×	✓	×	✓
Noninterference “until” [CM04]	×	×	✓	✓
Typeless noninterference “until”	✓*	✓	×	×

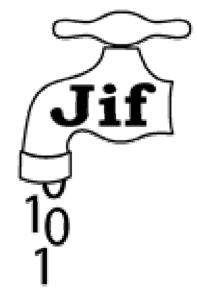
* Semantic anomalies

Declassification in practice: A case study

[Askarov & Sabelfeld, ESORICS'05]

- Use of security-typed languages for implementation of crypto protocols
- Mental Poker protocol by [Roca et.al, 2003]
  - Environment of mutual distrust
  - Efficient

- Jif language [Myers et al., 1999-2005]
  - Java extension with security types
  - Decentralized Label Model
  - Support for declassification



- Largest code written in security-typed language up to publ date [~4500 LOC]

Security assurance/Declassification

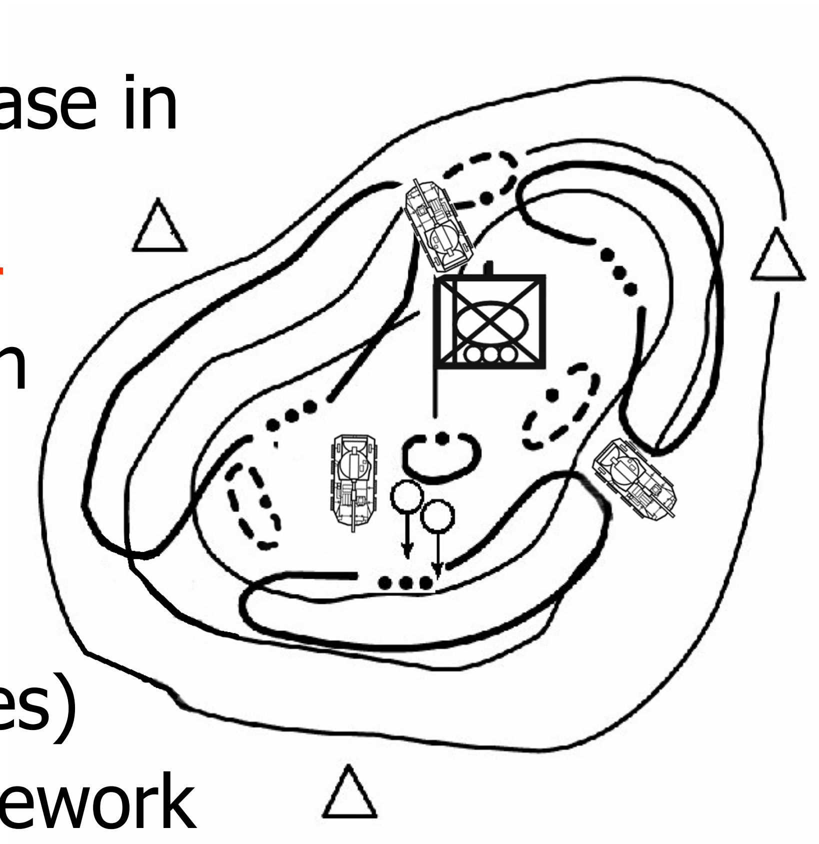
Group	Pt.	What	Who	Where
I	1 2	Public key for signature Public security parameter	Anyone Player	Initialization Initialization
II	3 4-7 8-10	Message signature Protocol initialization data Encrypted permuted card	Player Player Player	Sending msg Initialization Card drawing
III	11	Decryption flag	Player	Card drawing
IV	12-13 14	Player's secret encryption key Player's secret permutation	Player Player	Verification Verification

Group I – naturally public data Group II – required by crypto protocol

Group III – success flag pattern Group IV – revealing keys for verification

Conclusion

- Road map of information release in programs
- Step towards policy perimeter defense: to protect along each dimension
- Prudent principles of declassification (uncovering previously unnoticed anomalies)
- Need for declassification framework for relation and combination along the dimensions



End of talk