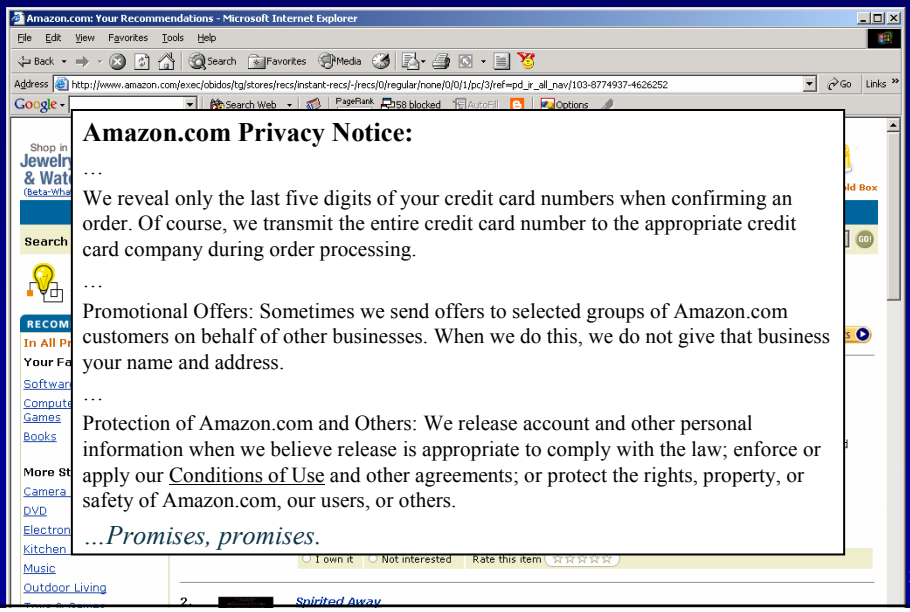
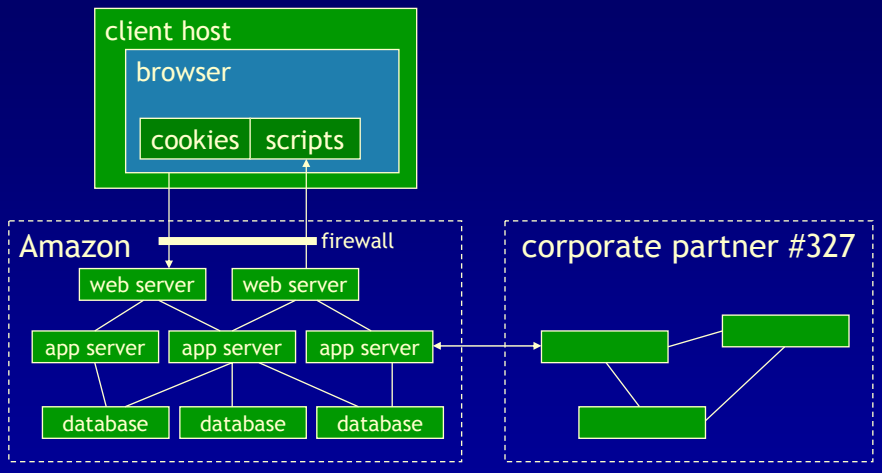
Programming with Explicit Security Policies

Andrew Myers
Cornell University

Information security



Implementor's view



Complex system -- how does Amazon know they are meeting their legal obligations?

3

Where we are now

- Complex systems, sensitive information
  - Online shopping, financial systems, medical information systems, B2B transactions, email, user profile management, …
  - Not particularly secure!
    - "High-tech financial firms are…hemorrhaging personal data like a leaky dam." MSNBC, 4/4/2005
- Weak validation techniques
  - Limited set of checkable properties
  - Individual mechanisms, but not whole systems
- Need: end-to-end security

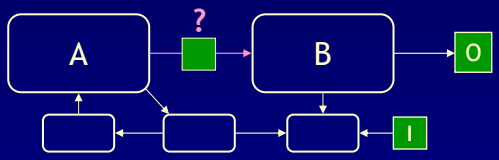
4

Explicit policies

- Too hard to give an end-to-end argument that large systems are secure
- Solution: use policies during development
  - Add policies to programming language or development environment
  - Use a compositional security analysis to build up from small secure components to big ones

5

End-to-end security

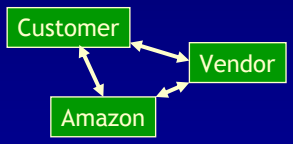


- Access control policies, mechanisms: useful, but…
  - Controls access but not propagation
    - E.g., Java stack inspection
    - What code to trust?
  - How to enforce end-to-end policy?
    - e.g., information I cannot be transmitted to output O
    - Access control mechanisms are necessary, access control policies are insufficient

6

The problems

- How to specify security requirements?
  - Needs to be simple
  - Diverse, dynamic, end-to-end security requirements
  - Non-uniform trust, mutual distrust

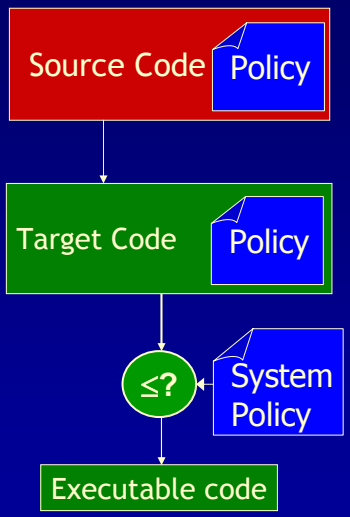


- How to enforce them?
  - Needs to be automatic
  - Want to catch problems early ⇒ static analysis
  - … and automatic program transformation

7

Enforcing language-based security

- Programs are annotated with security policies
- Compiler checks, possibly transforms program to ensure that all executions obey rules
- Loader, run-time system validates program policy against system policies



8

Outline

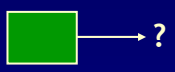
1. Expressive, tractable policy language [ZM05]
   - Generalization of decentralized labels [ML98]
   - Richer information flow controls: release, endorsement, robustness, and erasure (3.)
2. Security by construction for distributed programs

- Concurrency and timing channels [ZM03]
- Dynamic policies, run-time checking [ZM04]
- Enforcing availability policies [ZM05]
- Quantitative information flow [CMS05]
- & much more language-based security work elsewhere…

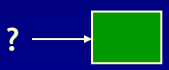
9

1. Information security properties

- Confidentiality (secrecy, privacy)
  - Ensuring that information isn't released improperly



- Integrity
  - Ensuring information only comes from the right places

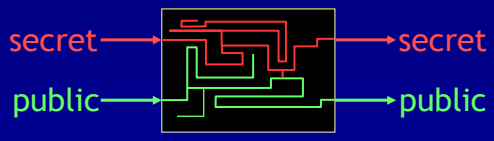


- Availability
  - Ensuring things happen

10

Secure information flow

- Goal: enforcement of end-to-end information security properties.
  Must track information flows = dependencies



11

Tracking information flow

- Goal: ensure information flows only upward in lattice of sensitivity labels

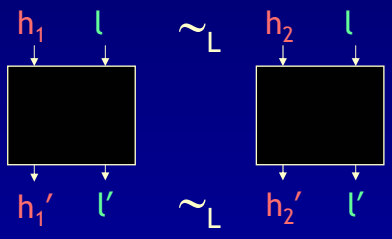
- Dynamically: mandatory access control
  - Evaluation augmented to track labels
  - Flexible, but expensive, imprecise
- Statically: program analysis [DD77]
  - Can be phrased as type system where type checking provably enforces noninterference [VSI96, HR98, P00, ZM01, PS02, BN02, ZM04, …]

12

Noninterference

High-security inputs to the system do not "interfere" with low-security outputs. [GM82]

-- an end-to-end security condition



Confidentiality: high = confidential, low = public

Integrity, Availability [ZM05]: low = trusted, high = untrusted

13

Noninterference on programs

- If observer cannot distinguish states s_1, s_2, cannot distinguish their executions either:

$$s_1 \sim_L s_2 \Rightarrow [\![s_1]\!] \approx_L [\![s_2]\!]$$

  - $[\![s]\!]$ denotes executions of s (e.g., set of traces, denotation)

- Different definitions of $\sim_L$, $\approx_L$ induce different notions of security
  - Possibilistic: $T_1 \approx_L T_2$ iff $\forall t_1 \in T_1.\ \exists t_2 \in T_2.\ t_1 \approx_L t_2 \wedge \ldots$
  - Observational determinism: $\forall t_1 \in S_1.\ \forall t_2 \in S_2.\ t_1 \approx_L t_2$
  - Termination insensitive: $t_1 \prec t_2 \Rightarrow t_1 \approx_L t_2$
  - Race insensitive [ZM03]: $t_1 \approx_L t_2$ defined per location

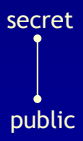
- Caveat: not all program equivalences are security

14

Jif: a security-typed language

- Jif = "Java information flow" (a.k.a. JFlow [POPL99])
- Available at **http://www.cs.cornell.edu/jif**
- Idea: attach security labels to types to obtain security types

```
int{secret} x;
int{public} y;
x = y; // ok if public ⊑ secret
y = x; // not ok if secret ⋢ public
```



- Both static and dynamic information-flow checking

15

Jif: a security-typed language

- Jif is a full-fledged programming language
  - Objects, classes, exceptions, label inference, label polymorphism, dependent labels, …
  - NI implications of features understood separately
- Some applications
  - Secure smartcard programming
  - Distributed auctions
  - Enforcing P3P privacy policies [HA04]
  - Distributed games (Battleship, poker [AS05])

16

Labels as a policy language

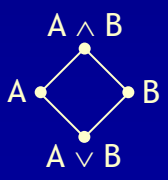
Need a richer label structure for expressing real security requirements:

- Principal
  - Whose security is being enforced?
  - Who can compromise security?
- Policy
  - A restriction on the use of some information
  - Pertains to a specific aspect of security
- Label
  - Combined policies governing an information resource

17

Principals

- Entities with security requirements or with ability to affect enforcement of those requirements
  - Users: Alice, Bob, root
  - Groups: students, faculty
  - Roles: Bob as chair
  - Environmental factors: power, Linux, Word
- Some principals act for others ($\Rightarrow$)
  - A $\Rightarrow$ B means A is as powerful as, as trusted as B (or more)
  - Bob $\Rightarrow$ Bob as chair Alice $\Rightarrow$ students
- Composite principals formed using $\wedge$, $\vee$
  - Alice $\wedge$ Bob students $\vee$ faculty
- Principal hierarchy



18

Owned policies

- Policy has form $u : p$ where
  - u is the owner of the policy (a principal)
  - p is the trustee of the policy (a principal)
- Confidentiality:
  - u trusts p not to leak
  - p is the reader of the information
  - Bob: (Alice $\vee$ Bob) means:
    Bob thinks Alice and Bob can read the data
- Integrity:
  - u trusts p not to damage information
  - p is a writer of the information
- Availability:
  - u trusts data from p to remain available

19

Owned policies, cont'd

- Simple, expressive common policy form
  - Know who must trust security analysis (owner)
  - Owner can control requirements (e.g., declassify)
    - Bob: (Alice $\vee$ Bob) $\rightarrow$ Bob: (Charlie $\vee$ Bob)
- Security guarantee when principals misbehave:
  Security requirement $u{:}p$ compromised only when:
  - Relevant trustee p violates security, or
  - An assumption of a principal acting for u is false
  - Formalized in a semantics [ZM05]
- Policy ordering
  $u_1{:}p_1 \leq u_2{:}p_2$ means $u_2{:}p_2$ at least as strong as $u_1{:}p_1$

$$\frac{u_2 \Rightarrow u_1 \qquad p_2 \Rightarrow p_1}{u_1{:}p_1 \leq u_2{:}p_2}$$

20

Labels

- Label is a set of zero or more policies
  - Policies may have different owners
  - Policies may govern different security aspects
  - All policies equally enforced
  - Lattice ordering $\sqsubseteq$ is lifted from policy ordering $\leq$
- Confidentiality label: `int{Bob: Bob} a1;`
  "Bob's private int"
- Integrity label: `int{Bob:! Alice ∧ Bob} a2;`
  "Bob allows effects if Alice & Bob agree."
- Combined labels:
  `int{Bob: Bob, !Alice ∧ Bob} a3;` "both"

OK	BAD

a1 = a2;
a1 = a3;
a3 = a2;
	a2 = a1;
	a3 = a1;
	a2 = a3;

21

2. Distributed systems

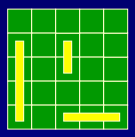
- The interesting security issues are in distributed systems: Online shopping, financial and medical information systems, B2B transactions, email, user profile management, …
- Many mechanisms:
  - encapsulation, access control lists, distributed protocols, encryption, signing,…
- But how to validate? Hard!

- Our goal: systems **secure by construction**
  - Programs annotated with explicit security policies
  - Policy-driven program transformation by compiler

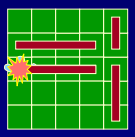
22

Distributed Battleship

- Two-player game in which each player tries to sink other's ships



"A3"

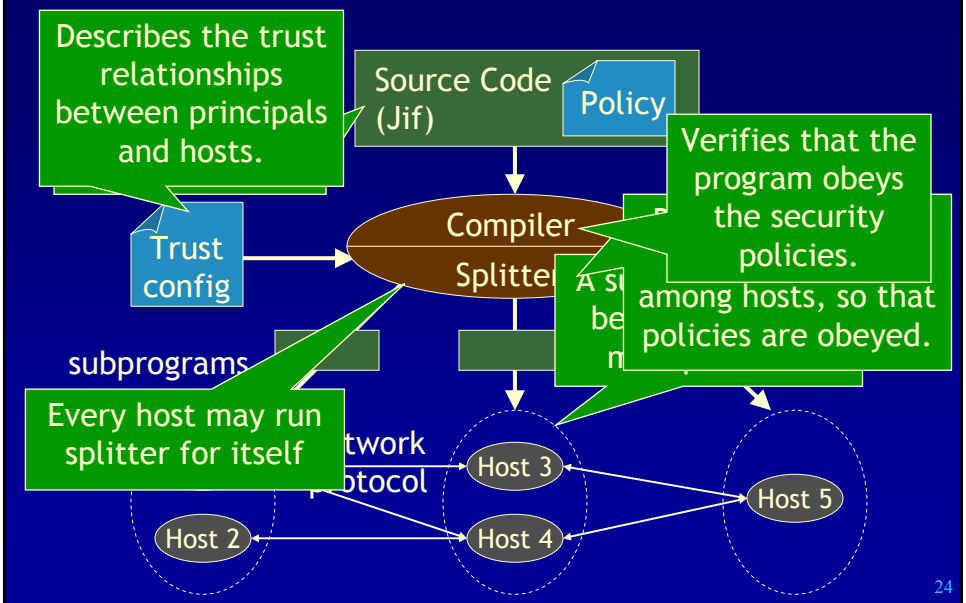


"hit" "missed"

- General problem for distributed systems: hard to prevent cheating
- Idea: write program once, compiler figures out how it can run securely on untrusted hosts

23

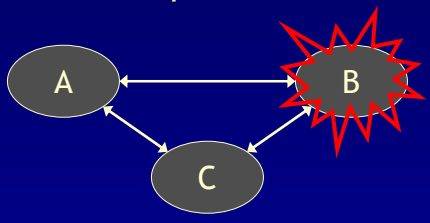
Secure partitioning [ZZNM01, ZCMZ03]



24

Security for distrusting principals

- Principals vs. hosts



"Alice trusts hosts A & C"
"Bob trusts hosts B & C"

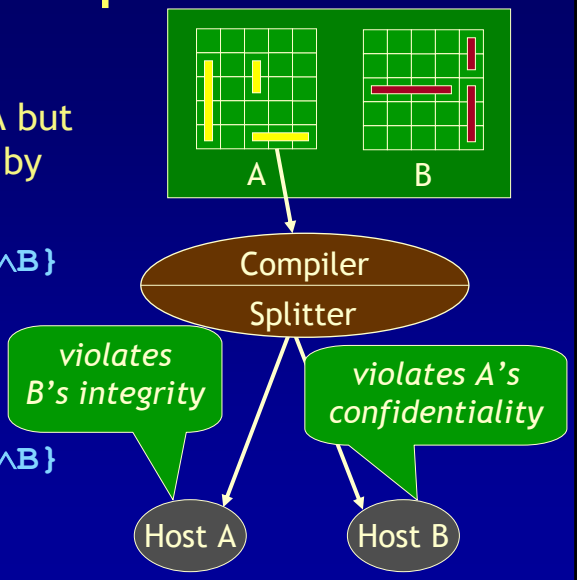
- Security guarantee:
  Principal P's security policy can be violated only if a host h that P trusts fails.
  (an assumption of P is false)

If B is subverted, Alice's policy is obeyed; Bob's policy might be violated.

25

Battleship example

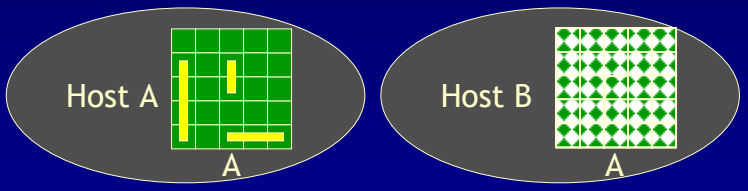
- A's board is confidential to A but must be trusted by both A and B:
  {A:A; A∧B:!A∧B}
- B's board is symmetrical:
  {B:B; A∧B:!A∧B}



26

Replication

- Idea 1: replicate both boards onto *both* hosts so both principals trust the data.

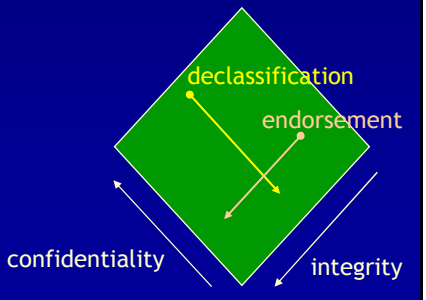


- Problem: host B now has A's confidential data.
- Idea 2: host B stores a one-way hash of cells
  - Cleartext cells checked against hashed cells to provide assurance data is trusted by both A & B.
  - Compiler automatically generates this solution!

27

3. Information release

- Noninterference policies are too strong!
- Confidentiality:
  - Real programs release some sensitive information
  - Example: password checker
- One solution: a downgrading escape hatch
  - Mediated by static authorization check [ML97]
  - No authorization ⇒ noninterference



28

Downgrading in Battleship

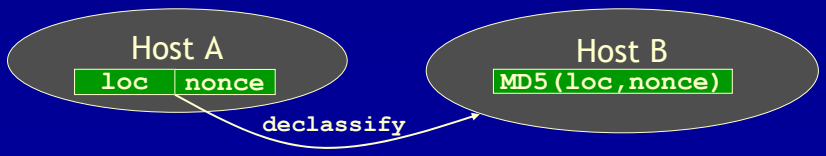
- **Declassification:** board location (i,j) not confidential once bomb dropped on it:

```
loc = declassify(board[move],
                 {A:A; A∧B:!A∧B} to {A∧B:!A∧B})
```

- **Endorsement:** opponent can make any legal move, and can initially position ships wherever desired.

```
move = endorse(move_ , {A:!B} to {A:!A∧B})
```

- `declassify, endorse` often correspond to network data transfers, hash value checks



29

Robustness

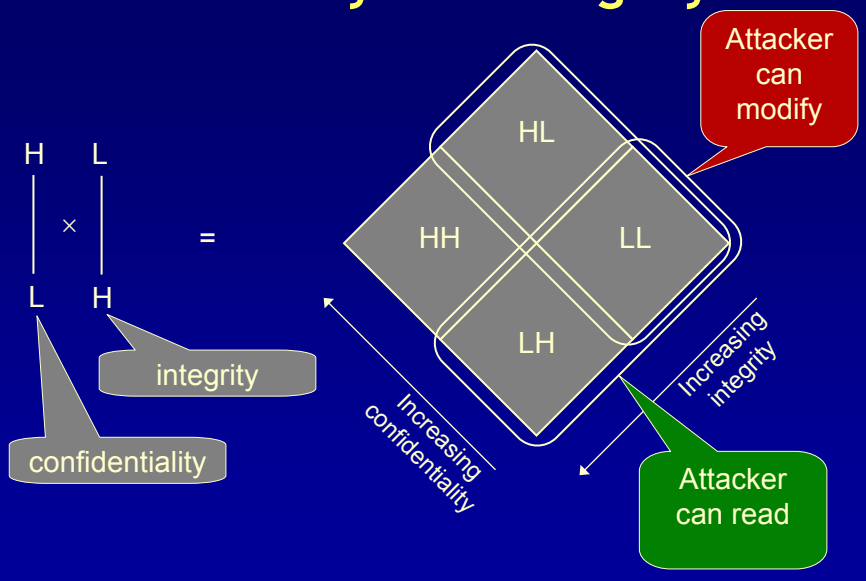
- What if attackers can exploit declassification?
  - Affect what information is released
  - Affect whether anything is released

  ⇒Static authorization is insufficient

- Intuition: a system is robust against attackers if declassification only releases information intentionally [ZM01b]
  - New building block for characterizing end-to-end system security in presence of declassification

30

Confidentiality and integrity



31

Robustness

- Noninterference at L: $s_1 \sim_L s_2 \Rightarrow [\![s_1]\!] \approx_L [\![s_2]\!]$
  - Observing execution gives no new information
- Define $s_1 \approx_L s_2$ if $[\![s_1]\!] \approx_L [\![s_2]\!]$ (related traces)
  - Noninterference at L: $\sim_L \subseteq \approx_L$
- Let $s[a]$ be attack a applied to program s
  - a replaces low-integrity code, data
- Robustness: for all valid a, a',

$$s_1[a] \cong_L s_2[a] \Rightarrow s_1[a'] \approx_L s_2[a']$$

$\Rightarrow$ Applying attack doesn't change observations

32

Checking declassify

- `declassify(`e`, `L_1` to `L_2`)`
  - gives value of e with label L_2
    (Note: $L_e \sqsubseteq L_1$, $L_e \not\sqsubseteq L_2$)
- Static checking rule: [ZZNM01,MSZ04]
  - integrity of e must be high
    attacker cannot affect what is declassified
  - integrity of pc (control flow) must be high
    attacker cannot affect when declassification happens
- Provably enforces robustness and qualified robustness [MSZ04]

33

Information erasure

- Some systems need to guarantee information becomes unavailable
  - Voting system dissociates voter id from ballot
  - Cryptographic hardware forgets keys
  - Online shopping: forget credit card number
- Can satisfy by
  - Erasing information, or more generally,
  - upgrading label to be more sensitive
  - Dependent information must also be upgraded (need information flow!)
- Noninterference is also too weak!

34

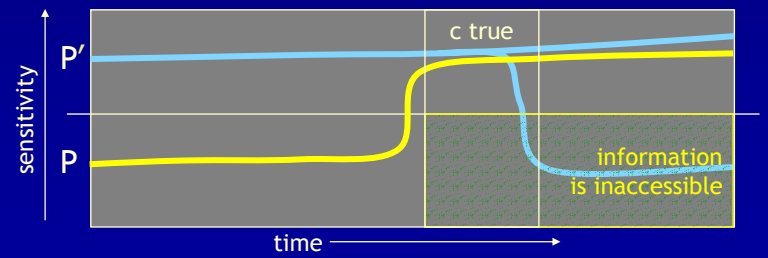
Downgrading & erasure policies

- Given labels l from lattice $\mathcal{L}$ (e.g., DLM), information release policy P has forms

$$l \mid P \rightsquigarrow^{c} P' \mid P \nearrow^{c} P'$$

Enforce P on information; when condition c is true, may downgrade to P′ [CM04]

Enforce P on information; when condition c is true, must enforce P′ [CM05]



Example

- Making purchase over Internet
  - Customer provides CC#, but merchant should not store it
  - Merchant must send CC# to bank, which will keep record
- Policy for CC#: $(M \rightsquigarrow^{pur} B) \nearrow^{end} B$
  - M = security label for merchant
  - B = security label for bank
  - *pur* is true when purchase approved
  - *end* is true when transaction ended
- Downgrading and erasure policies expressively connect future information use to abstract description of program behavior
- Erasure introduces "must" flows

Related work

Information flow, noninterference
- Language-based security and static information flow (see [SM02])
  - mostly ignores distribution, distrust
- Multilevel security and mandatory access control
- Noninterference properties for process algebras
- Program slicing and dependency analysis
- Tainting

End-to-end security by construction
- Uniform replication
  - replicated state machines, BFT, file systems
- Commitment protocols

Information release and declassification
- Selective declassification [Pottier00]
- Intransitive noninterference
  - [Rushby92, Pinsky95, RG99, Mantel01, Mantel&Sands04]
- Quantitative information flow
- Attacker strength [VS00, DiPierro02, Laud01, DG00]

37

Future work

- Information release policies and properties
- Integrating access control, authentication, auditing
- Availability
- Revocation
- Concurrent and distributed systems
  - Incorporating more construction techniques
- Designing support libraries
  - data structures, UI, …

38

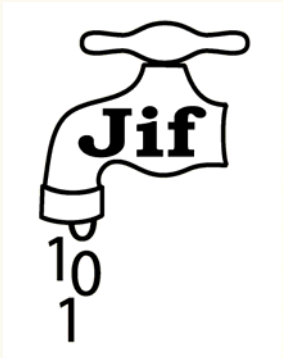
Conclusions

- Need end-to-end techniques for validating security and privacy assurance
- Make policies explicit in programs
  - expose security requirements and assumptions
  - enable analysis and transformation
- Information flow is a necessary component and already useful
- A lot left to do!

39

Acknowledgments

- Lantian Zheng
- Stephen Chong
- Nate Nystrom
- Andrei Sabelfeld (Chalmers)
- Steve Zdancewic (U.Penn)



http://www.cs.cornell.edu/Jif

40