0x1A Great Papers in Computer Security

Vitaly Shmatikov

http://www.cs.utexas.edu/~shmat/courses/cs380s/

B. Lampson

A Note on the Confinement Problem

(CACM 1973)

Information Channels

End-to-end security requires controlling information channels

- **Legitimate channels:** declared outputs
- **Storage channels:** transmit explicitly
  - Assign to variables, write to files, sockets
- **Covert channels:** transmit by mechanisms not intended for transmitting information
  - System load, locks, power consumption, etc. etc.
  - Timing channels: transmit information by when something happens (rather than what)

Confinement Properties

- Confinement is established through isolation
  - Restrict a process' access
  - Enforce the principle of least privilege (means what?)
- Total isolation: a process that cannot communicate with any other process and cannot be observed cannot leak information
  - In practice, any process uses observable resources such as CPU, secondary storage, networks, etc.
- Confinement must be transitive
  - If a confined process invokes a second process, the second process must be as confined as the caller

Simulating a Shared Variable

```
Procedure settrue (file)
  1: try opening file - if already open, then goto 1;
Procedure setfalse (file)
    close file;
Procedure value (file)
    value = true;
    try opening file - if already open, then goto 2;
    value = false;
    close file;
  2: return value;
```

Covert Channel via File Open/Close

Three files: data, sendlock, receivelock

sender:	settrue(data) or setfalse(data) -- sends 1 bit
	settrue(sendlock)
receiver:	wait for value(sendlock)=true
	value(data) $\rightarrow$ received bit
	settrue(receivelock)
sender:	wait for value(receivelock)=true
	setfalse(sendlock)
receiver:	wait for value(sendlock)=false
	setfalse(receivelock)
sender:	wait for value(receivelock)=false

Lipner's Notes on Time

- All processes can obtain rough idea of time
  - Read system clock or wall clock time
  - Determine number of instructions executed
- All processes can manipulate time
  - Wait some interval of wall clock time
  - Execute a set number of instructions, then block

We'll see some timing attacks later in the course

Example of a Timing Channel

- System has two VMs: sender S and receiver R
- To send 0, S immediately yields CPU
  - For example, run a process that instantly blocks
- To send 1, S uses full quantum
  - For example, run a CPU-intensive process
- To receive, R measures how quickly it gets CPU
  - Uses real-time clock to measure intervals between accesses to a shared resource (CPU in this case)

Covert Channels Without Time

- Two VMs share disk cylinders 100 to 200, SCAN algorithm schedules disk accesses
- Receiver: read data on cylinder 150
  - Yields CPU when done, disk arm now at 150
- Sender: to send "1", read data on cylinder 140; to send "0", read data on cylinder 160
  - Yields CPU when done, disk arm now at 140 or 160
- Receiver: read data on cylinders 139 and 161
  - SCAN: if arm is at 140, then reads 139 first; if arm is at 160, reads 161 first - this leaks 1 bit (why?)

Analysis of Secure Xenix

- 140 variables both visible and alterable
  - 90 out of those shared
  - 25 can be used as covert channels
- Resource exhaustion channels
  - Example: signal by exhausting free inodes
- Event-count channels
  - Example: number of files created
- Unexploitable channels
  - Example: cause system crash

Covert vs. Side Channels

- Covert channel: an unanticipated path of communication exploited by an attacker to convey confidential information
  - Insider exfiltration, steganography …
- Side channel: an unanticipated information leak that an attacker uses to obtain confidential information
  - Pizza orders at the Pentagon, Tempest, power analysis of smart cards, acoustic emanations, compromising reflections …

Modern Confinement Mechanisms

- Memory protection
- Sandboxes
  - Java virtual machine
  - Inline reference monitors
  - System-call interposition
- Virtual machine monitors

Access Control Model

- Principal makes a request to access a resource (object)
  - Example: process tries to write into a file
- Reference monitor permits or denies request
  - Example: file permissions in Unix

Rights and Actions

- Access control matrix
  - For each subject and object, lists subject's rights
- Subjects, objects, rights can be created…
  - Example: new users, new files
  - Creation of rights is sometimes called "delegation"
    - Example: grant right R to subject S with respect to object O
- …or deleted
- Access control is undecidable (in general)
  - In general, can't determine if a given subject can get a particular right with respect to a given object
    - Harrison, Ruzzo, Ullman (1976)

ACL: Access Control Lists

- For each object, store a list of (Subject x Rights) pairs
  - Resolving queries is linear in the size of the list
- Easy to answer "who can access this object?"
- Easy to revoke rights to a single object
- Lists can get long
- Authentication at every access can be expensive

Capability Lists

- For each subject, store a list of (Object x Rights) pairs – called capabilities
  - Capabilities should be unforgeable (why?)
- Authentication takes place when capability is granted - don't need to check at every access
- Revocation is harder (why?)

Implementing Capabilities

- Unique identifiers that map to objects
  - Extra level of indirection to access an object
  - Integrity of the map must be protected
- Capabilities must be unforgeable
  - Special hardware: tagged words in memory
    - Can't be copied or modified
  - Store capabilities in protected address space
  - Use static scoping in programming languages
    - "Private" fields in Java
  - Cryptography
    - Shared keys; OS could digitally sign capabilities

OS: Coarse-Grained Access Control

- Enforce security properties at the system call layer (what are the issues?)
- Enforcement decisions are made at the level of "large" objects
  - Files, sockets, processes …
- Coarse notion of subject / "principal"
  - UID

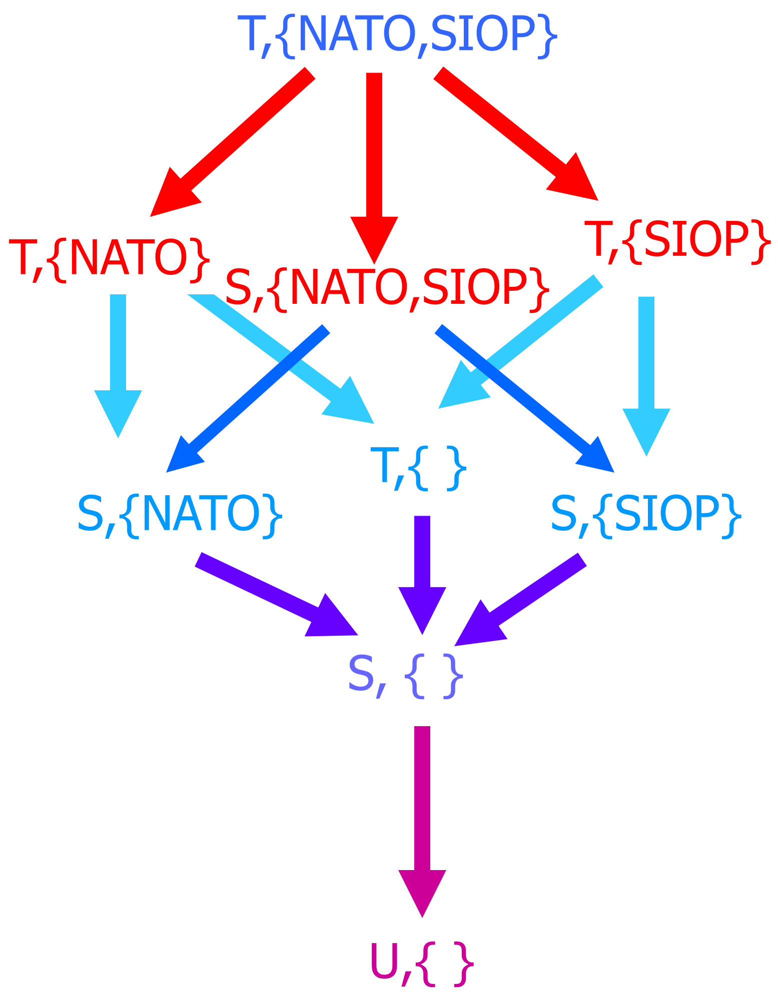
DAC vs. MAC

- Discretionary access control (DAC)
  - Individual user may, at his own discretion, determine who is authorized to access the objects he creates
    - Example: Unix files
- Mandatory access control (MAC)
  - Creator of an object does not necessarily have the ability to determine who has authorized access to it
  - Policy typically governed by a central authority
    - Recent research on decentralized information flow control
  - Policy on an object depends on what object or information was used to create it

Multi-Level Security (Military)

- Classification of personnel and data
  - Class D = ⟨rank, compartment⟩
- Dominance relation
  - D1 ≤ D2 iff rank1 ≤ rank2 and compart1 ⊆ compart2
    - Example: ⟨Restricted, Iraq⟩ ≤ ⟨Secret, CENTCOM⟩
- Subjects: users or processes
  - Class(S) = clearance of S
- Objects: documents or resources
  - Class(O) = classification of O

Example of a Label Lattice



Bell-LaPadula Model

"No read up, no write down"

- Principals are assigned clearance levels drawn from a lattice of security labels
- A principal may <u>read</u> objects with lower or equal security label: $C(O) \leq C(S)$
- A principal may <u>write</u> objects with higher or equal security label: $C(S) \leq C(O)$
  - Example: a user with Secret clearance can read objects with Public and Secret labels, but can only write objects with Secret label (why?)
  - "Tainted" may not flow into "untainted"

SELinux

- Security-enhanced Linux system from NSA
- MAC built into the OS kernel
  - Each process has an associated domain
  - Each object has an associated type (label)
  - Configuration files specify how domains may access types, interact, transition between domains
- Role-based access control
  - Each process has an associated role
    - Separate system and user processes
  - Configuration files specify the set of domains that may be entered by each role

Other MAC Policies

- "Chinese Wall" [Brewer & Nash 1989]
  - Object labels are classified into "conflict classes"
  - If subject accesses an object with a particular label from a conflict class, all accesses to objects labeled with other labels from the conflict class are denied
  - Policy changes dynamically
- "Separation of Duties"
  - Division of responsibilities among subjects
    - Example: Bank auditor cannot issue checks

D. Denning and P. Denning

Certification of Programs for Secure Information Flow

(CACM 1976)

Beyond Access Control

- Finer-grained data confidentiality policies
  - At the level of principals rather than hosts or processes
- Security enforcement decisions at the level of application abstractions
  - User interface: access control at window level
  - Mobile code: no network send after file read
  - E-commerce: no goods until payment
  - Make security policies part of the programming language itself
- End-to-end security: control propagation of sensitive data <u>after</u> it has been accessed

Information Flow Within Programs

- Access control for program variables
  - Finer-grained than processes
- Use program analysis to prove that the program has no undesirable flows

Confidentiality

- Confidentiality via basic access control …
  - "Only authorized processes can read a file"
    - When should a process be "authorized"?
  - Encryption provides end-to-end confidentiality, but it's difficult to compute on encrypted data
- … vs. end-to-end confidentiality
  - Information should not be improperly released by a computation no matter how it is used

Integrity

- Integrity via basic access control …
  - "Only authorized processes can write a file"
    - When should a process be "authorized"?
  - Digital signatures provide end-to-end integrity, but cannot change signed data
- … vs. end-to-end integrity
  - Information should not be updated on the basis of less trustworthy information

Explicit and Implicit Flows

- Goal: prevent information flow from “high” variables to “low” variables (why?)
- Flow can be explicit …

  h := <secret>

  x := h

  l := x
- … or implicit

  boolean h := <secret>

  if (h) { l := true} else { l := false }

Compile-Time Certification

- Declare classification of information allowed to be stored in each variable
  - x: integer class { A,B }
- Classification of function parameter = classification of argument
- Classification of function result = union of parameter classes
  - … unless function has been verified as stricter
- Certification becomes type checking!

Assignments and Compound Stmts

- Assignment: left-hand side must be able to receive all classes in right-hand side

  x = w+y+z

  requires lub{w,y,z} ≤ x

- Compound statement

```
begin
   x = y+z;
   a = b+c –x
end
```

requires lub{y,z} ≤ x and lub{b,c,x} ≤ a

Conditionals and Functions

- Conditional:

  classification of "then/else" must contain classification of "if" part (why?)

- Functions:

```
int sum (int x class{A}) {
    int out class{A,B} ;
    out = out + x;
}
```

requires $A \leq B$ and $B \leq B$

Iterative Statements

- In iterative statements, information can flow from the absence of execution

  while $f(x_1, x_2, \ldots, x_n)$ do S

  - Information flows from variables in the conditional statement to variables assigned in S (why?)

- For an iterative statement to be secure …
  - Statement terminates
  - Body S is secure
  - $\text{lub}\{x_1, x_2, \ldots, x_n\} \leq$ glb{target of an assignment in S}

Non-Interference

[Goguen and Meseguer]

- Observable behavior of the program should not depend on confidential data
  - Example: private local data should not "interfere" with network communications

Declassification

- Non-interference can be too strong
  - Programs release confidential information as part of normal operation
  - "Alice will release her data after you pay her $10"
- Idea: allow the program to release confidential data, but only through a certain computation
- Example: logging in using a secure password

```
if (password == input) login(); else fail();
```

  - Information about password must be released …
    … but only through the result of comparison

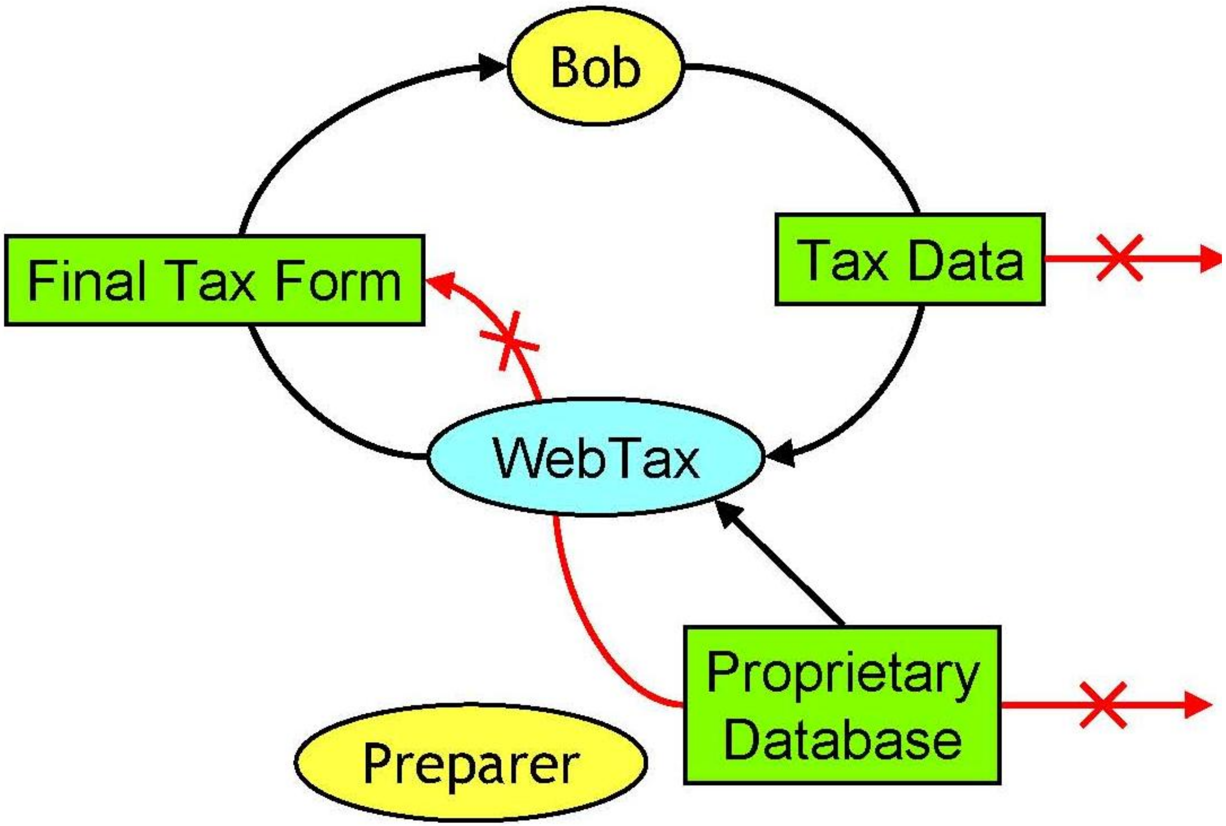
A. Myers and B. Liskov

A Decentralized Model for Information Flow Control

(SOSP 1997)

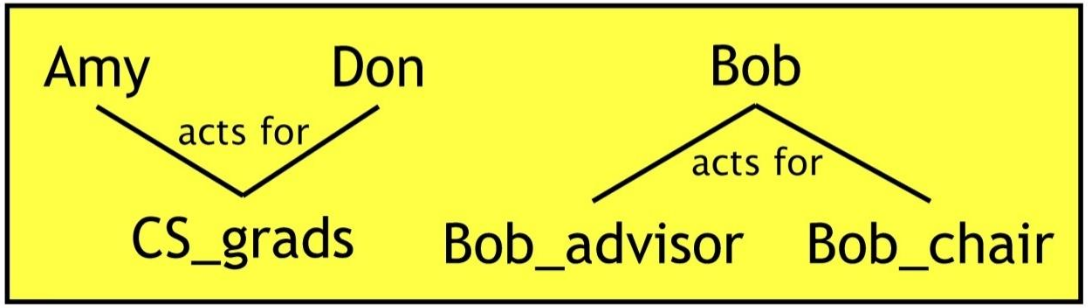
Web Tax Example

[Myers]



Principals

- Principals are users, groups of users, etc.
- Used to express fine-grained policies controlling use of data
  - Individual users and groups
  - Close to the semantics of data usage policies
- Principal hierarchy generated by the acts-for relation



Data Labels

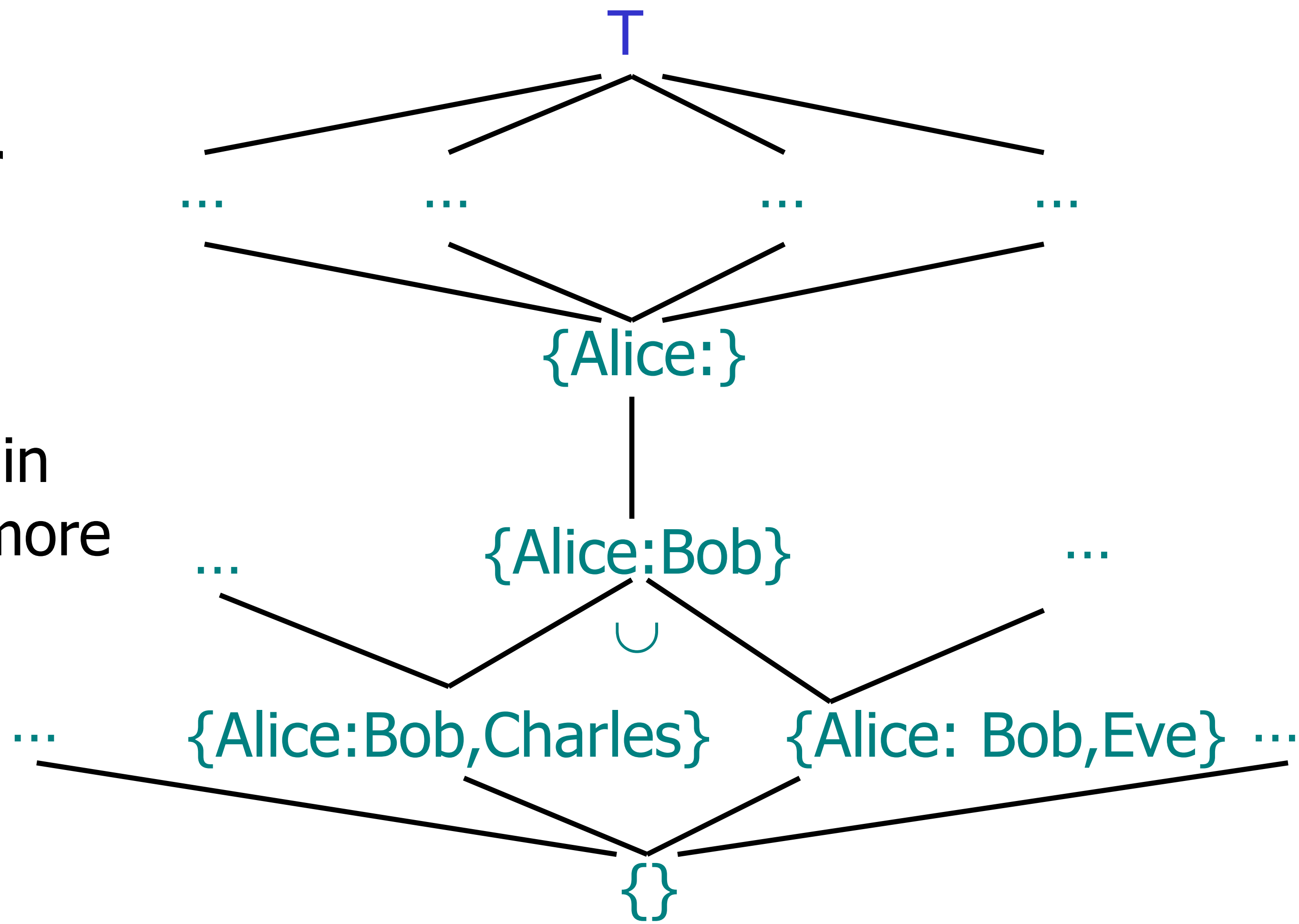
[Myers and Liskov]

- Label each piece of data to indicate permitted information flows (to and from)
  - Label specifies a set of policies
- Confidentiality constraints: who may read it?
  - {Alice: Bob, Eve} label means that Alice owns this data, and Bob and Eve are permitted to read it
  - {Alice: Charles; Bob: Charles} label means that Alice and Bob own this data, but only Charles can read it
- Integrity constraints: who may write it?
  - {Alice ? Bob} label means that Alice owns this data, and Bob is permitted to change it

Label Lattice

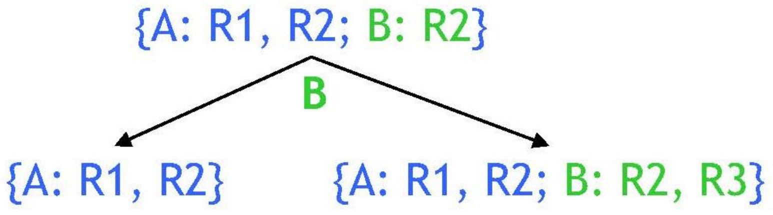
$\sqsubseteq$ order
$\sqcup$ join

Labels higher in the lattice are more restrictive



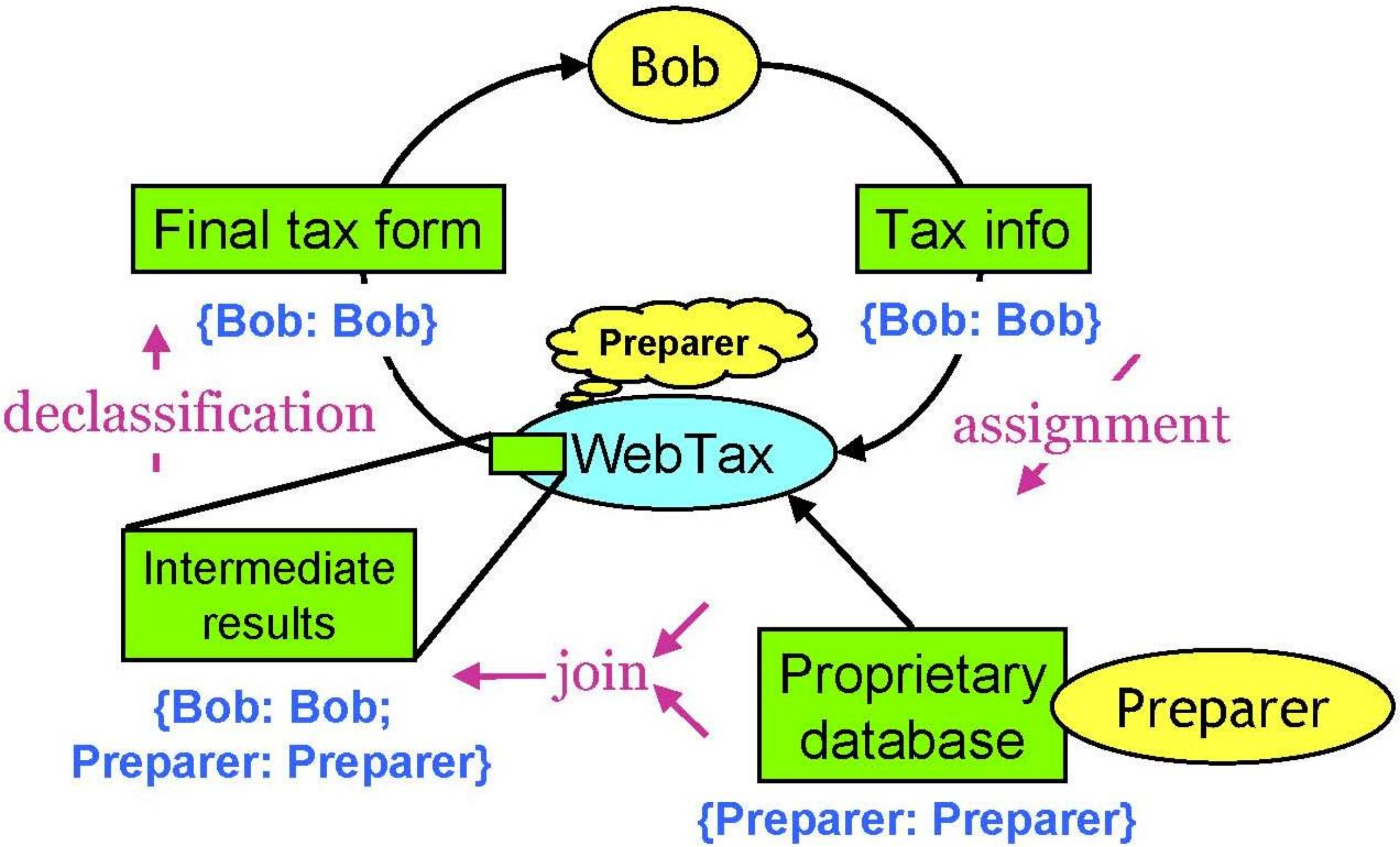
Computation Changes Labels

- **Assignment (X=Y) relabels a variable**
  - For every policy in the label of Y, there must be a policy in the label of X that is at least as restrictive
- **Combining values (when does this happen?)**
  - Join labels – move up in the lattice
  - Label on data reflects all of its sources
- **Declassification**
  - A principal can rewrite <u>its own</u> part of the label



Web Tax Example

[Myers]



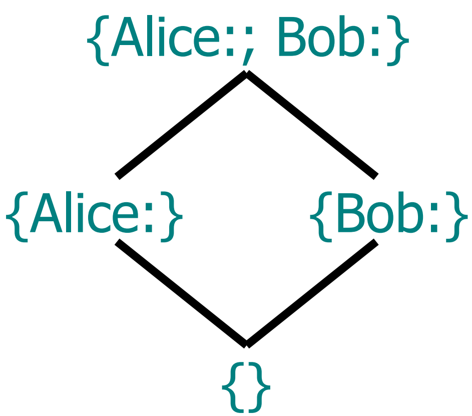
Jif

[Myers]

- Jif: Java with information flow control
- Represent principals as Java classes
- Jif augments Java types with labels
  - int {Alice:Bob} x;
  - Object {L} o;
- Subtyping follows the $\sqsubseteq$ lattice order
- Type inference
  - Programmer may omit types - Jif will infer them from how values are used in expressions

Implicit Flows (1)

[Zdancewic]



PC label

```
int{Alice:} a;
int{Bob:} b;
...
if (a > 0) then {
   b = 4;
}
```

{} → (before `if`)

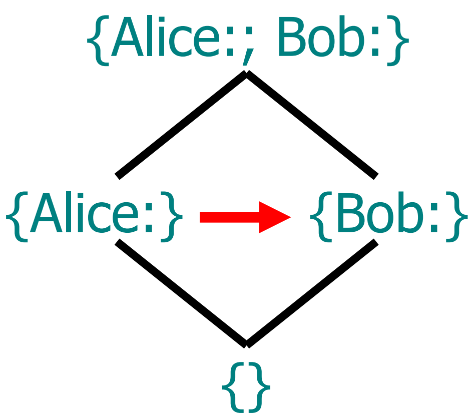
{}∪{Alice:}={Alice:} → `b = 4;`

{} → (after `}`)

This assignment leaks information contained in program counter (PC)

Implicit Flows (2)

[Zdancewic]



PC label

```
int{Alice:} a;
int{Bob:} b;
...
if (a > 0) then {
  b = 4;
}
```

{}

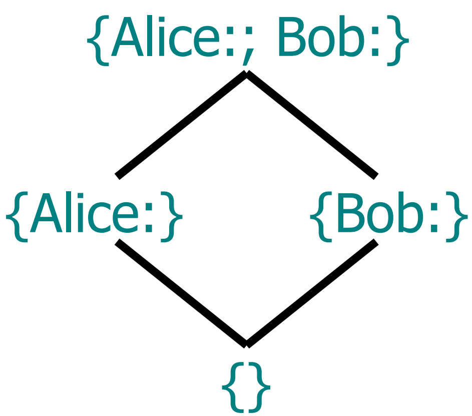
{}∪{Alice:}={Alice:}

{}

To assign to variable with label X, must have PC ⊆ X

Function Calls

[Zdancewic]



PC label

```
int{Alice:} a;
int{Bob:} b;
...
if (a > 0) then {
  f(4);
}
```

{} → (before if)

{}∪{Alice:}={Alice:} → f(4);

{} → (after })

Effects inside function can leak information about program counter

Method Types

```
int{L1} method{B} (int{L2} arg) : {E}
      where authority(Alice)
{
   …
}
```

- Constrain labels before and after method call
  - To call the method, need PC $\sqsubseteq$ B
  - On return, should have PC $\sqsubseteq$ E
  - "where" clauses may be used to specify authority (set of principals)

Declassification

```
int{Alice:} a;
int Paid;
...   // compute Paid
if (Paid==10) {
      int{Alice:Bob} b = declassify(a, {Alice:Bob});
      ...
}
```

"downcast" int{Alice:} to int{Alice:Bob}

Robust Declassification

[Zdancewic and Myers]

```
int{Alice:} a;
int Paid;
...   // compute Paid
if (Paid==10) {
      int{Alice:Bob} b = declassify(a, {Alice:Bob});
      ...
}
```

Alice needs to trust the contents of Paid

Introduces constraint $PC \sqsubseteq \{Alice?\}$

Jif Caveats

- No threads
  - Information flow hard to control
    - Depends on scheduling, etc.
  - Active area of current research
- Timing channels not controlled
  - Explicit choice for practicality
- Differences from Java
  - Some exceptions are fatal
  - Restricted access to some system calls