

Access Control Information-Flow Security

Vitaly Shmatikov

Reading Assignment

- ◆ Lampson. "A Note on the Confinement Problem" (CACM 1973).
- ◆ Myers and Liskov. "A Decentralized Model for Information Flow Control" (SOSP 1997).

Access Control Model

- ◆ Classic method for preventing “bad things” from happening
- ◆ **Principal** makes a **request** to access a resource (**object**)
 - Objects have owners
 - 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
- ◆ When is authentication performed?
 - If at every access, can be very 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
- 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 = $\langle \text{rank}, \text{compartment} \rangle$

◆ Dominance relation

- $D1 \leq D2$ iff $\text{rank1} \leq \text{rank2}$ & $\text{compart1} \subseteq \text{compart2}$
 - Example: $\langle \text{Restricted}, \text{Iraq} \rangle \leq \langle \text{Secret}, \text{CENTCOM} \rangle$

◆ Subjects: users or processes

- Class(S) = clearance of S

◆ Objects : documents or resources

- Class(O) = classification of O

Bell-LaPadula Model

- ◆ “No read up, no write down”
- ◆ Principals are assigned clearance levels drawn from a lattice of security labels
- ◆ A principal may read objects with lower (or equal) security label: $C(O) \leq C(S)$
- ◆ A principal may write 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?)
 - Integrity: “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

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 after it has been accessed

Confidentiality

◆ Confidentiality via 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

◆ End-to-end confidentiality

- Information should not be improperly released by a computation no matter how it is used

Integrity

◆ Integrity via access control

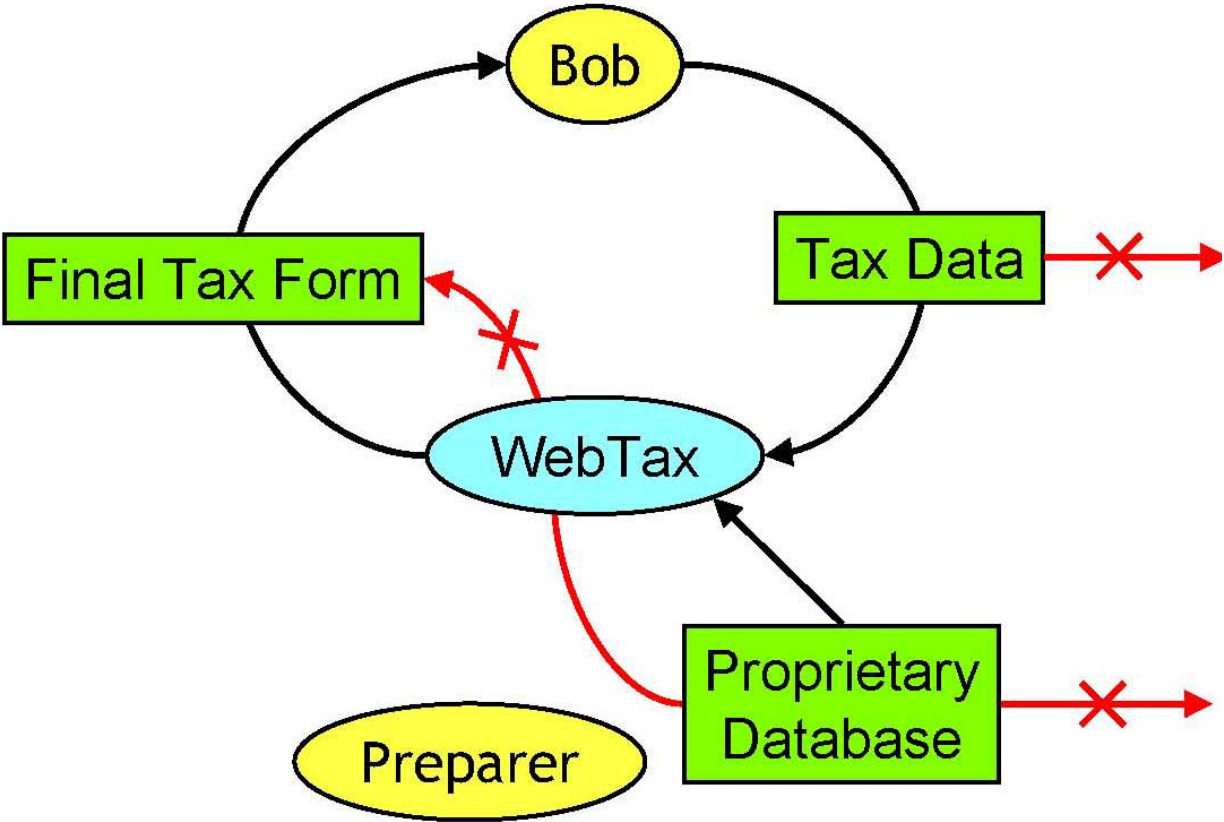
- Only authorized processes can write a file
- When should a process be “authorized?”
- Digital signatures provide end-to-end integrity, but cannot change signed data

◆ End-to-end integrity

- Information should not be updated on the basis of less trustworthy information

Web Tax Example

[Myers]



Information Channels

- ◆ End-to-end security requires controlling information channels [Lampson 1973]
- ◆ **Storage channels:** transmit information explicitly
 - Variable assignment, writing to sockets, files
- ◆ **Covert channels:** transmit by mechanisms not intended for transmitting information
 - System load, locks ...
- ◆ **Timing channels:** transmit information by when something happens (rather than what)

Example of an Implicit Flow

```
boolean b := <secret>
if (b) {
    x := true; f();
}
```

Information flow from b to x

Non-Interference

[Goguen and Meseguer]



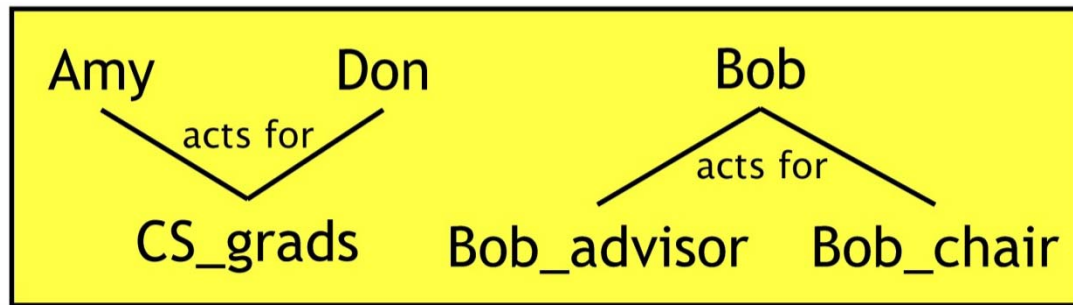
- ◆ Observable behavior of the program should not depend on confidential data
 - Example: private local data should not “interfere” with network communications

Declassification

- ◆ Non-interference is 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

Principals

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



Data Labels

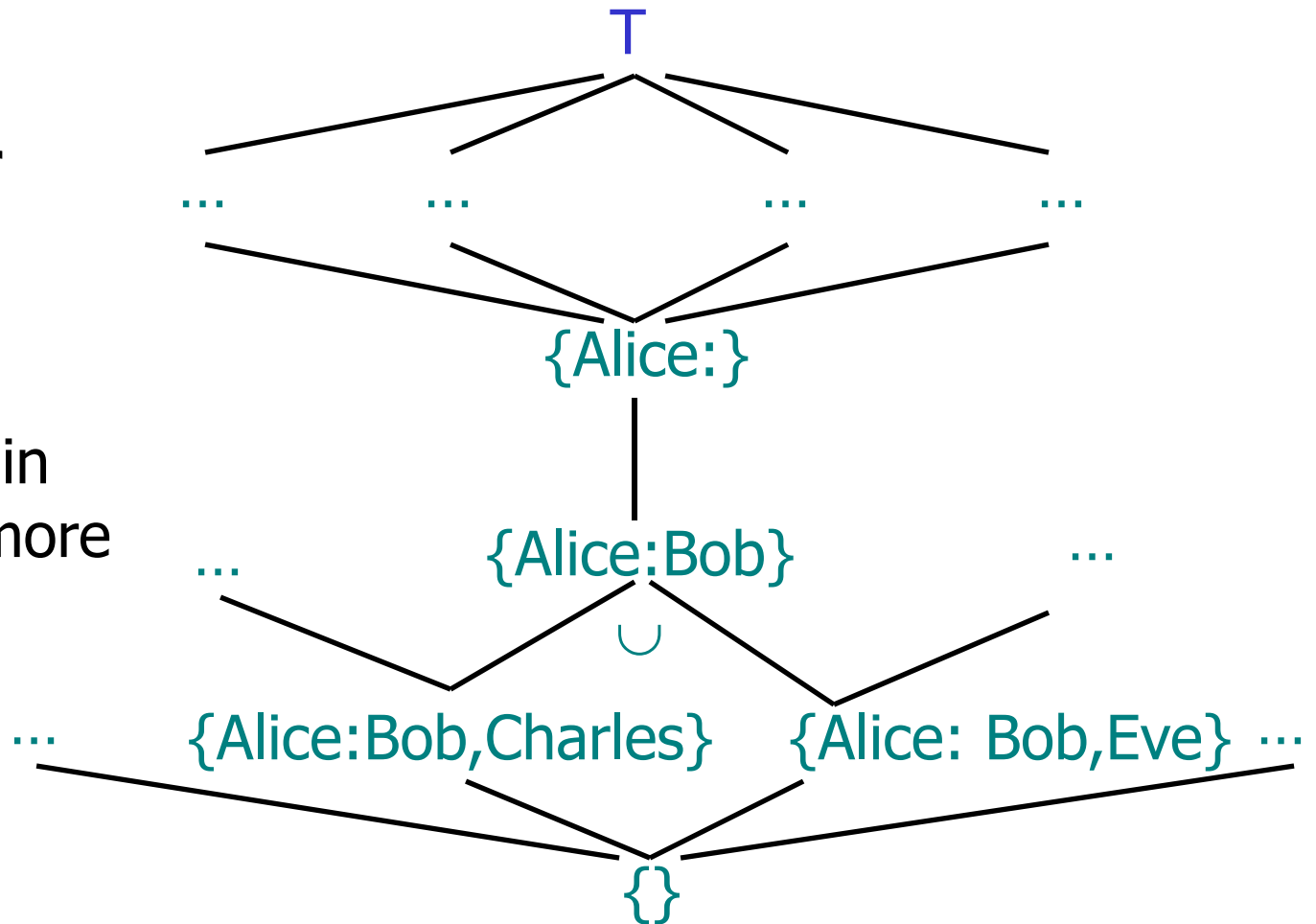
[Myers and Liskov]

- ◆ Label each piece of data to indicate permitted information flows (both 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

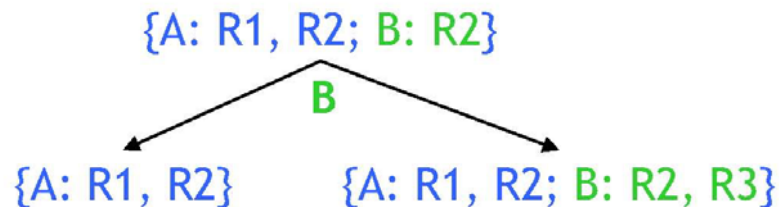
\subseteq order
 \cup join

Labels higher in
the lattice are more
restrictive



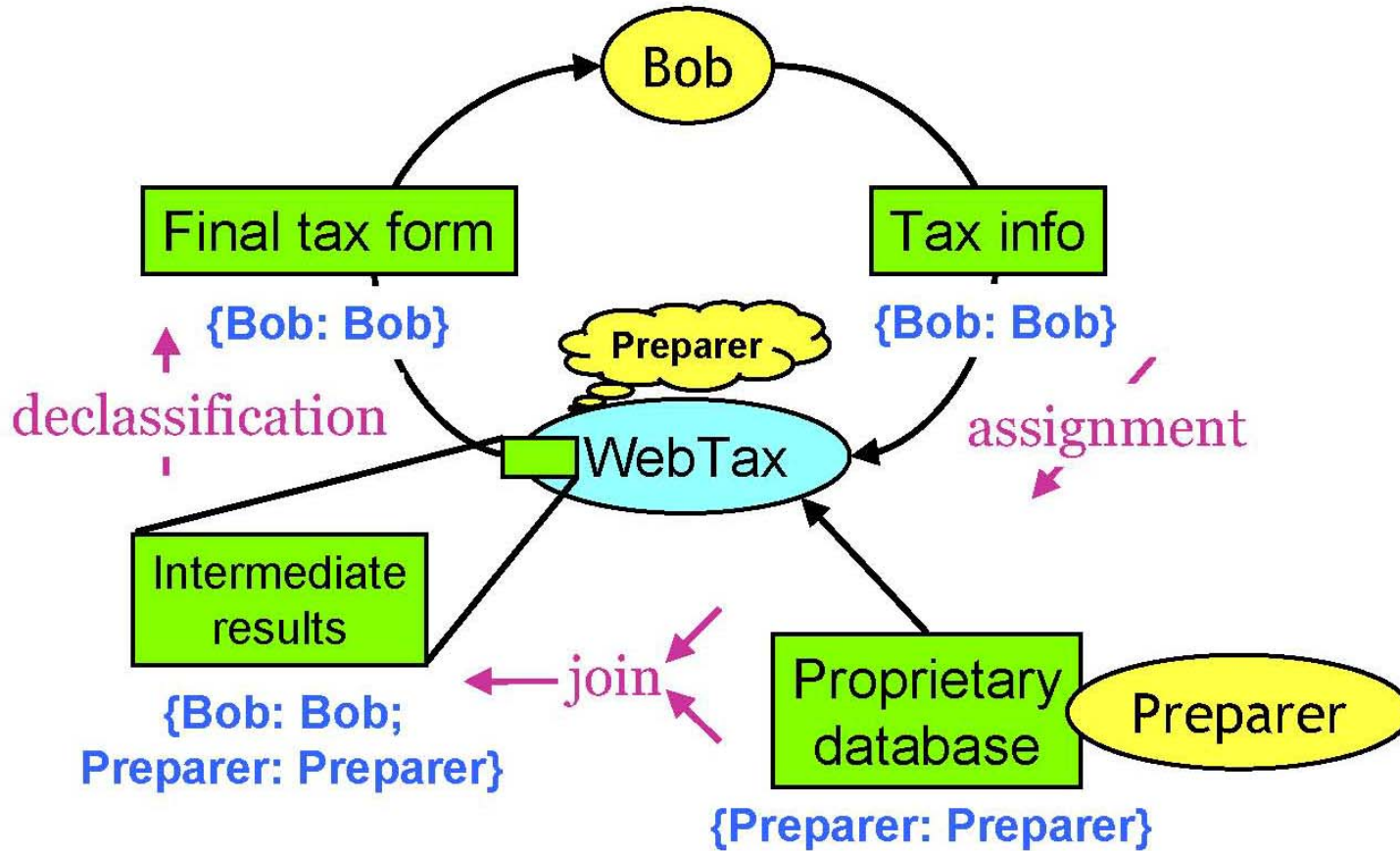
Computation Changes Labels

- ◆ 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 its own part of the label



Web Tax Example

[Myers]



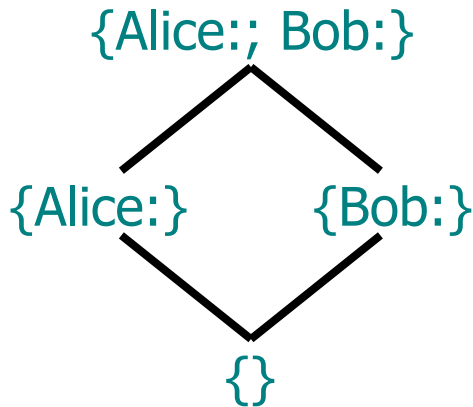
Jif

[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 \subseteq 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 {
```

$\{\} \cup \{Alice:\} = \{Alice:\}$ →

```
  b = 4;
```

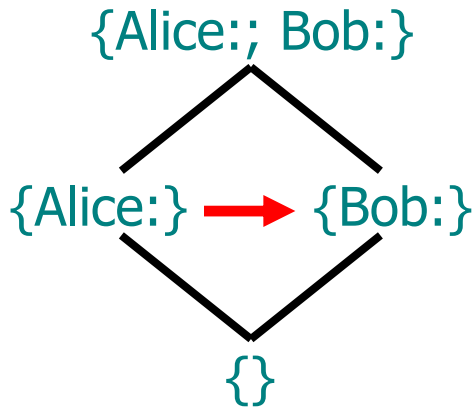
```
}
```

$\{\}$ →

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 {
```

$\{\} \cup \{\text{Alice:}\} = \{\text{Alice:}\}$

```
  b = 4;
```

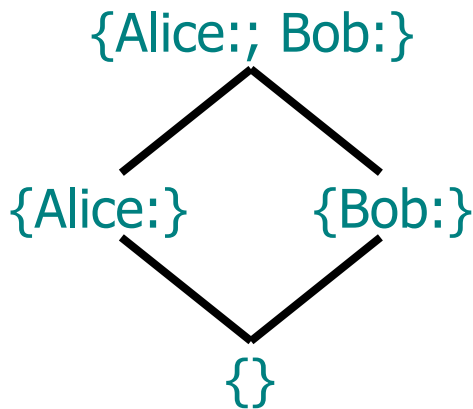
```
}
```

{}

To assign to variable
with label X , must have
 $PC \subseteq X$

Function Calls

[Zdancewic]



PC label

```
int{Alice:} a;  
int{Bob:} b;
```

...



```
if (a > 0) then {
```

$\{\} \cup \{\text{Alice:}\} = \{\text{Alice:}\}$

→ **f(4);**

```
}
```



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 $PC \subseteq B$
 - On return, should have $PC \subseteq E$
- ◆ “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;
```

Alice needs to trust
the contents of Paid

```
... // compute Paid
```

```
if (Paid == 10) {
```

```
    int{Alice:Bob} b = declassify(a, {Alice:Bob});
```

```
    ...
```

```
}
```

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

Jif Caveats

◆ No threads

- Information flow hard to control
- 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