

Language-Based Security

Vitaly Shmatikov

Talk Announcement

- ◆ Matt Wright on P2P anonymity
- ◆ 12 noon, Friday, Dec 1, in TAY 2.122

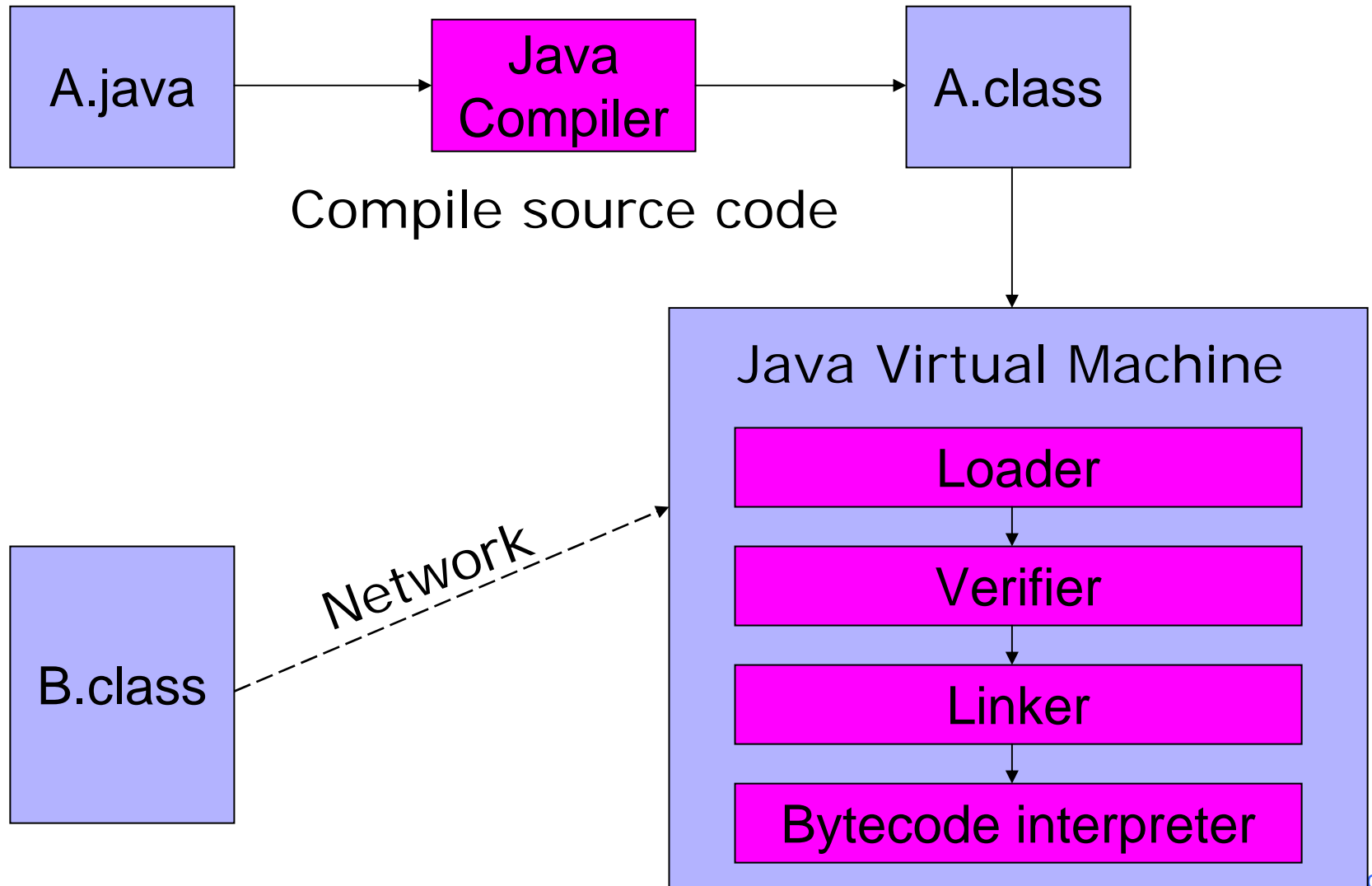
Reading Assignment

- ◆ Sabelfeld and Myers. “Language-Based Information-Flow Security” (JSAC).

Overview of Java Design

- ◆ **Compiler** compiles source code into **bytecode** class
- ◆ **Virtual machine** loads classes on demand, verifies bytecode properties, interprets bytecode
- ◆ **Why this design?**
 - **Bytecode is portable**
 - Can transmit bytecode across network
 - **Minimize machine-dependent part of implementation**
 - Do optimization on bytecode when possible
 - Keep bytecode interpreter simple

JVM Architecture



Java Sandbox

- ◆ Untrusted Java applets run in a sandbox
 - Cannot access local filesystem or devices
 - Network connections only to applet load source
 - Cannot invoke any local program or library
 - “Untrusted” indicator on top-level windows
 - Cannot manipulate basic classes or other threads

... this is too restrictive for many applets
- ◆ Java 2 supports fine-grained security policies
 - Security manager may have several security policies
 - Policy can grant privileges to specific applets based on their source and/or digital signatures on the code

Overview of Sandbox Architecture

- ◆ Several complementary mechanisms
- ◆ Class loader
 - Associates protection domain with each class
- ◆ Bytecode verification and run-time tests
 - NO unchecked casts or other type errors, NO overflow
- ◆ Security manager
 - Library functions call it to decide if request is allowed
 - Uses protection domain associated with code and policy
 - Enforcement relies on stack inspection

Class Loader

- ◆ Runtime system loads classes as needed
 - When class is referenced, loader searches for file of compiled bytecode instructions
 - Namespaces of different applets are kept different
 - Different instances of `ClassLoader`
 - Every loaded class has a reference to loader instance that created it
 - Loader calls bytecode verifier on untrusted classes
- ◆ Default loading mechanism can be replaced
 - Define alternate `ClassLoader` object
 - Extend the abstract `ClassLoader` class and implementation

Bytecode Verifier

- ◆ Checks correctness of bytecode
 - Code has only valid instruction opcodes & register use
 - Code does not overflow/underflow stack
 - Data types are not converted illegally
 - Pointers are not forged
 - Method calls use correct number & types of arguments
 - References to other classes use legal names
 - Every instruction obeys the Java type discipline
 - Type safety is fairly complicated!
- ◆ Goal: prevent access to underlying machine
 - Via forged pointers, overflows, crashes, etc.

Why Is Typing a Security Feature?

- ◆ Java security mechanisms rely on type safety
- ◆ Prevents applet from accessing arbitrary memory
 - Unchecked typecast lets program call any address

```
int (*fp)()    /* variable "fp" is a function pointer    */  
...  
fp = addr;    /* assign address stored in an integer variable */  
(*fp)(n);    /* call the function at this address            */
```
 - Security manager has private fields that store permission information
 - Access to these fields would defeat the security mechanism

Type Safety of JVM

◆ Load-time type checking

◆ Run-time type checking

- All casts are checked to make sure they are type safe
- All array references are checked to be within bounds
- References are tested to be not null before dereference

◆ Memory protection

- Automatic garbage collection
- NO pointer arithmetic

If program accesses memory, the memory is allocated to the program and declared with correct type

Security Manager

- ◆ Java library functions call security manager when they are invoked at runtime
 - For example, `checkRead(String filename)`
 - `checkRead` method is defined by `SecurityManager` class
 - Method throws exception if operation is not allowed
- ◆ Security manager uses the system policy to decide whether calling code is allowed to do operation
 - Examines “protection domain” of calling class
 - Signer: organization that signed code before loading
 - Location: URL where the calling class came from

Sample SecurityManager Methods

checkExec	Checks if the system commands can be executed.
checkRead	Checks if a file can be read from.
checkWrite	Checks if a file can be written to.
checkListen	Checks if a certain network port can be listened to for connections.
checkConnect	Checks if a network connection can be created.
checkCreateClassLoader	Check to prevent the installation of additional ClassLoaders.

Creating a Security Policy

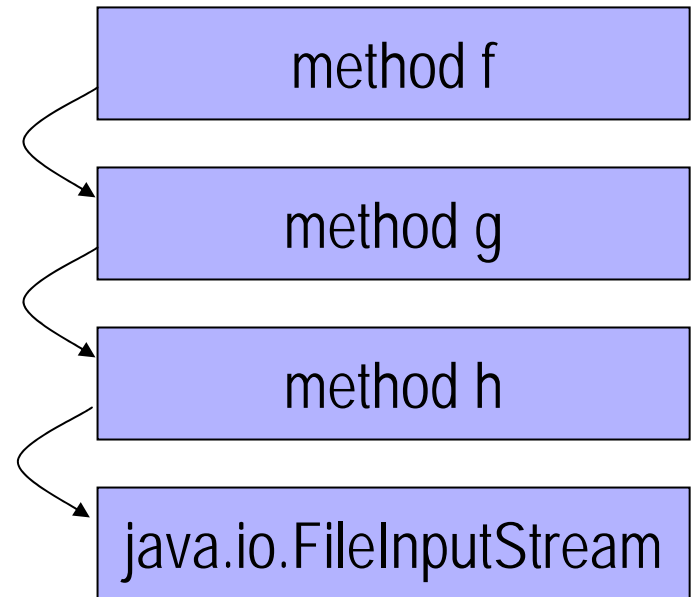
- ◆ Create your own subclass of `SecurityManager` and instantiate
 - Redefine `checkRead`, `checkWrite`, etc. methods to enforce your policy
- ◆ Install using `System.setSecurityManager`
 - `setSecurityManager` cannot be revoked or replaced
- ◆ If no `SecurityManager` installed, all privileges are granted to any applet

Sample Security Policy

CodeSource		Permissions
Base URL	Signature	
<code>http://www.schwab.com/classes/stockeditor.jar</code>	Schwab's signature	<ul style="list-style-type: none">· Read/write file /home/shmat/stocks
<code>http://*.schwab.com/</code>	(not required)	<ul style="list-style-type: none">· Connect/accept bankofamerica.com ports 1-1023· Read file /home/shmat/logo.png

Stack Inspection (Sketch)

- ◆ Permission depends on
 - Permission of calling method
 - Permission of all methods above it on call stack
 - Up to method that is trusted and asserts this trust



Attacks From Within Sandbox

◆ Deny service

- Spawn threads, waste CPU cycles and bandwidth
- Kill other threads

◆ Export confidential information

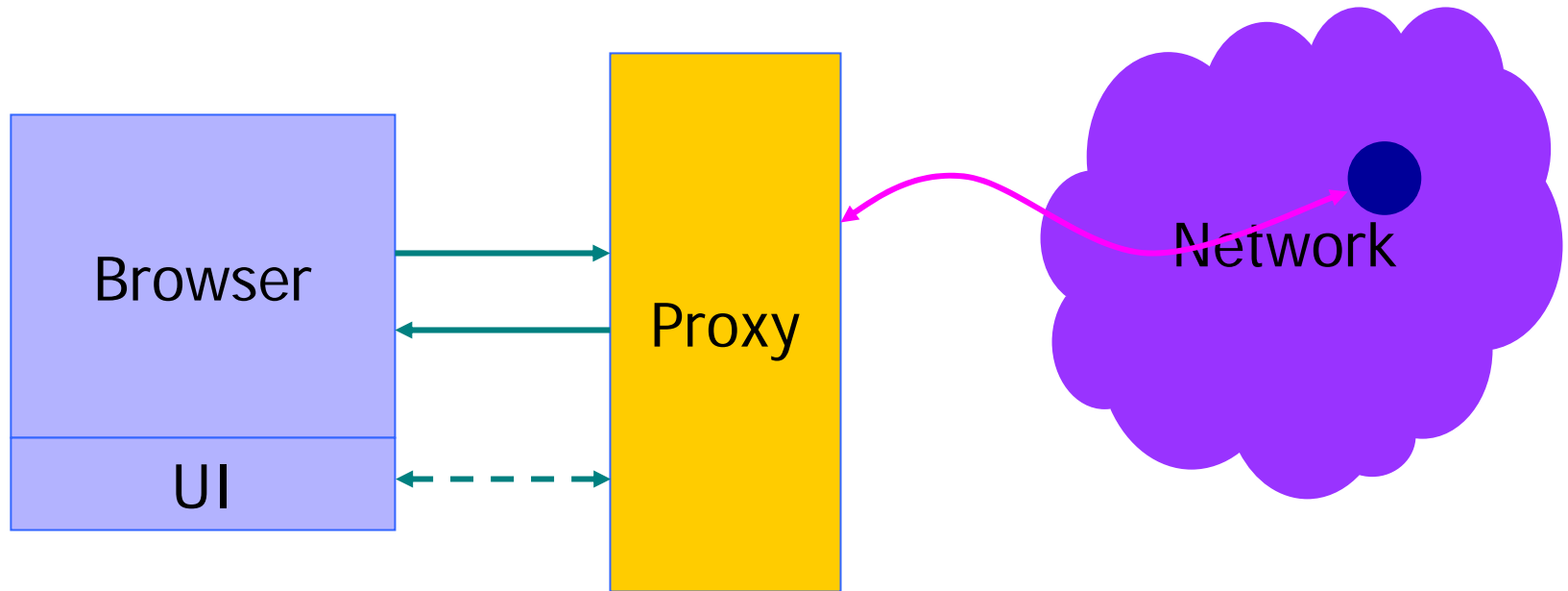
◆ Annoy

- Play irritating sound and don't stop
- Display a large window that ignores mouse input
- Flashing display (causes seizures in some users)

◆ Steal CPU cycles

- For example, help attacker to crack passwords

More Security with Proxies



- ◆ Proxy intercepts request for web page
- ◆ May **modify bytecode** before sending it to browser
- ◆ Can do other checks: filter ads, block sites, etc.

Bytecode Modification Techniques

◆ Class-level replacement

- Define subclass of a library or any other class
- Replace references to original class with subclass
- Works because of subtyping
- Not possible if class has been declared “final”

◆ Method-level replacement

- Change function calls to call new function
- Generally, check or modify arguments and call original function

Sample Bytecode Modification

◆ SafeWindow class

- Subclass of standard Window class
 - Do not allow windows larger than maximum
 - Do not allow more than max number of windows

◆ Restrict network activity

- Replace call to Socket object constructor
 - Do not allow socket connection to port 25 to prevent the applet from forging email

◆ Maintain appearance of browser window

- Replace calls to AppletContext methods
 - Displayed URL must match actual hyperlink

Beyond Access Control

- ◆ Finer-grained data confidentiality policies
 - At the level of principals rather than hosts
- ◆ End-to-end security
 - Control propagation of sensitive data after it has been accessed
- ◆ Make security policies part of the programming language itself
- ◆ Add information flow policies to Java
 - Example: Jif

Non-Interference

[Goguen and Meseguer '82, '84]



- ◆ Confidential local data should not “interfere” with network communications
 - Intuition: network-observable behavior of the program should not depend on private data

Principals in Jif

- ◆ **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
- ◆ Jif runtime represents principals as Java classes

Data Labels

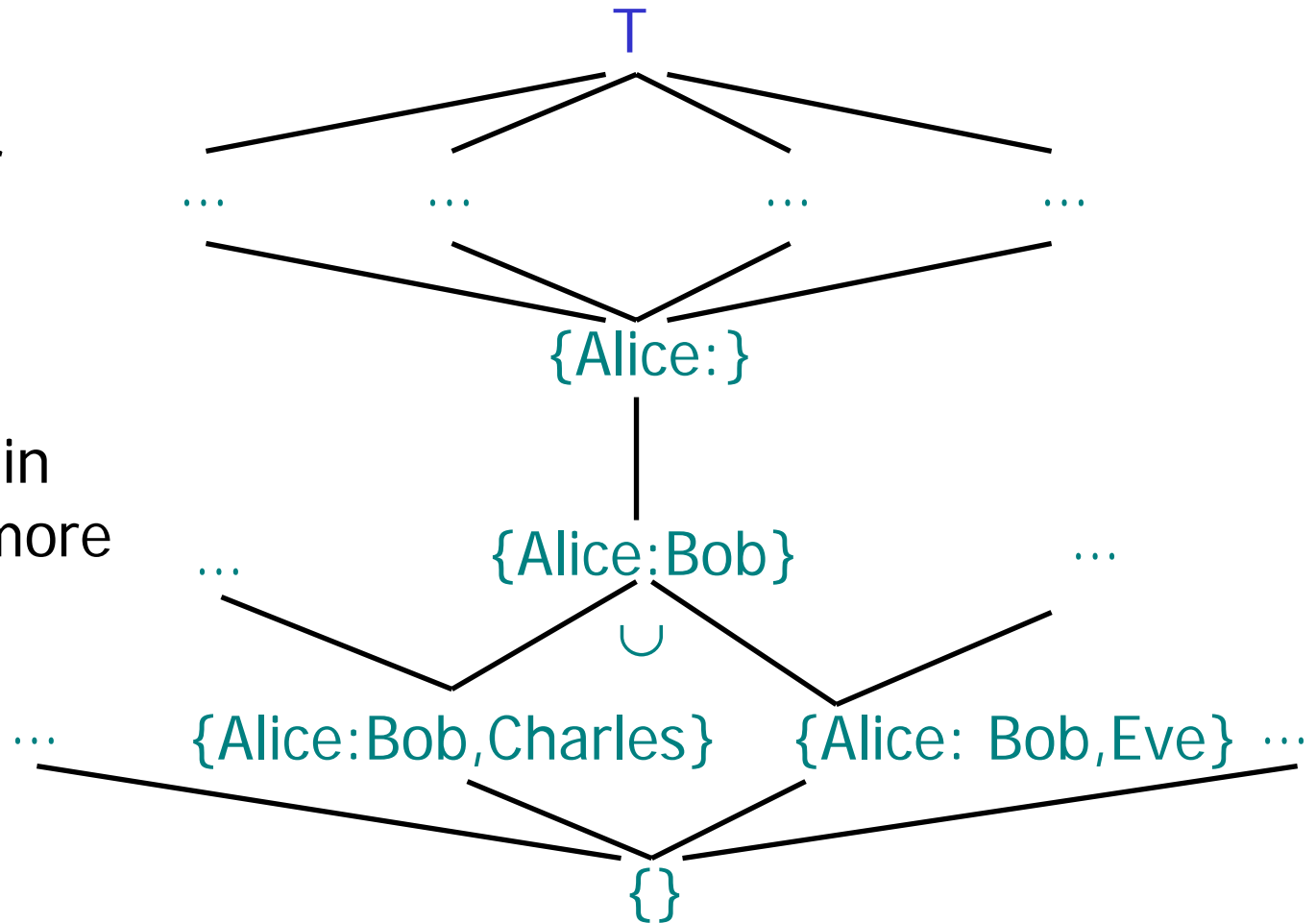
[Myers and Liskov '97, '00]

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

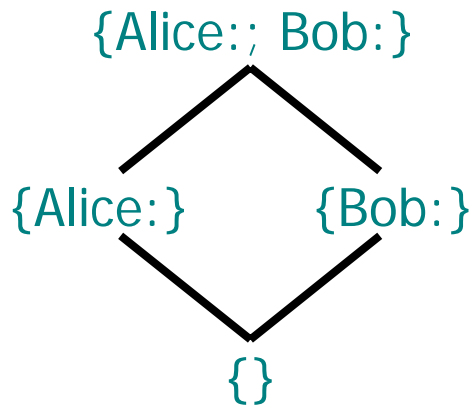


Extended Types in Jif

- ◆ 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)

[slide stolen from Steve Zdancewic]



PC label

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

...

{}

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

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

```
  b = 4;
```

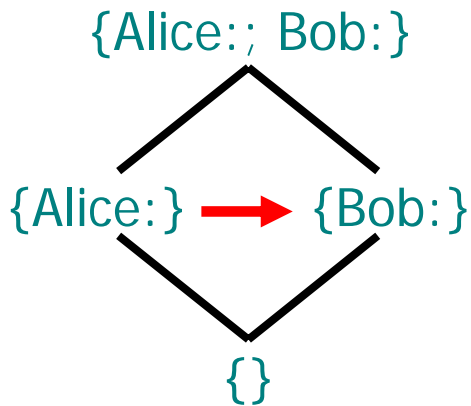
```
}
```

{}

This assignment leaks information contained in program counter (PC)

Implicit Flows (2)

[slide stolen from Steve Zdancewic]



PC label

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

...

{}

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

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

```
  b = 4;
```

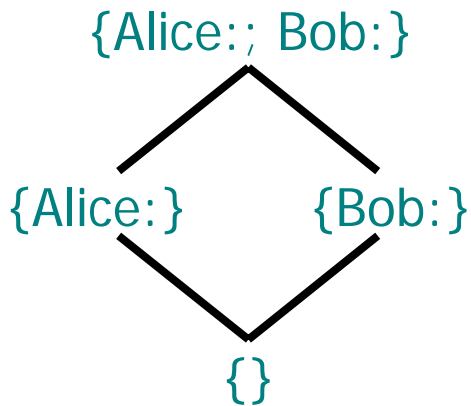
```
}
```

{}

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

Function Calls

[slide stolen from Steve Zdancewic]



PC label

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

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

```
...
```

```
{}
```

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

```
{ } ∪ {Alice:} = {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)

Richer Security Policies

- ◆ Non-interference may be too restrictive for some policies
 - Example: "Alice will release her data to Bob, but only after he has paid \$10"
- ◆ Program may leak some information about private data, but information should be released only in well-defined circumstances

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

Proof-Carrying Code

[Necula et al]

-
- ◆ A code consumer must become convinced that the code supplied by an untrusted code producer has some set of properties
 - ◆ PCC approach:
 - Code consumer publishes a **safety policy**
 - Set of conditions that a foreign program must satisfy for its execution to be considered safe
 - Code producer creates a formal safety proof
 - Proves that his code adheres to the safety policy
 - Code consumer uses a simple and fast proof validator to check that the proof is valid

Certification

- ◆ Code producer compiles source code and verifies that the program satisfies the safety policy
- ◆ A proof of successful verification together with the native code forms the PCC binary
 - Compiler can create the proof automatically
- ◆ Code producer can store the resulting PCC binary for future use, or can deliver it to code consumers for execution

Validation and Execution

- ◆ Code consumer validates the proof part of PCC binary and loads the native code for execution
- ◆ Because proof has already been created by code producer, verification can be done offline and only once for a given program, regardless of how many times it is executed

Advantages of PCC

- ◆ Burden is mostly on the code producer
- ◆ Code consumer only has to perform a fast, simple, easy-to-trust proof checking
 - It's much easier to check an existing proof than to prove that an arbitrary piece of code is correct
- ◆ No cryptography or trusted third parties
 - PCC binaries are "self-certifying"
- ◆ Code is verified before execution
 - Detect dangerous operations early, thus avoiding the need to kill the misbehaving process after it has acquired resources or modified system state