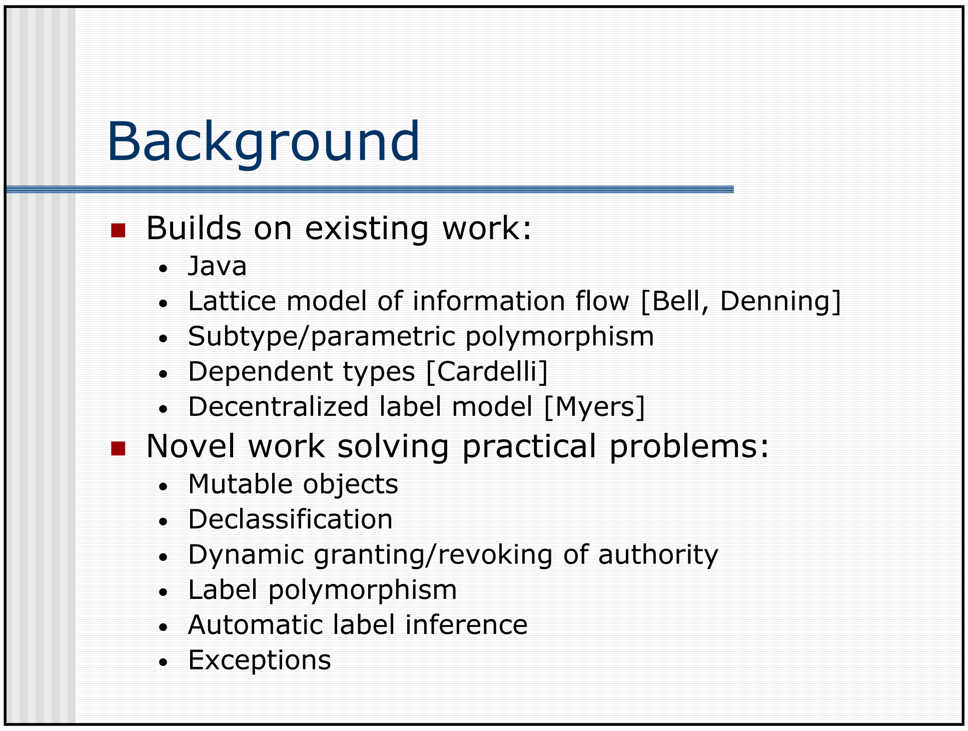
JFlow: Practical Mostly-Static Information Flow Control

A.Myers and B.Liskov. **A Decentralized Model for Information Flow Control** (SOSP 1997).

Andrew C. Myers and Barbara Liskov. **Protecting privacy using the decentralized label model.** *ACM Transactions on Software Engineering and Methodology, 9(4): 410–442, October* 2000.

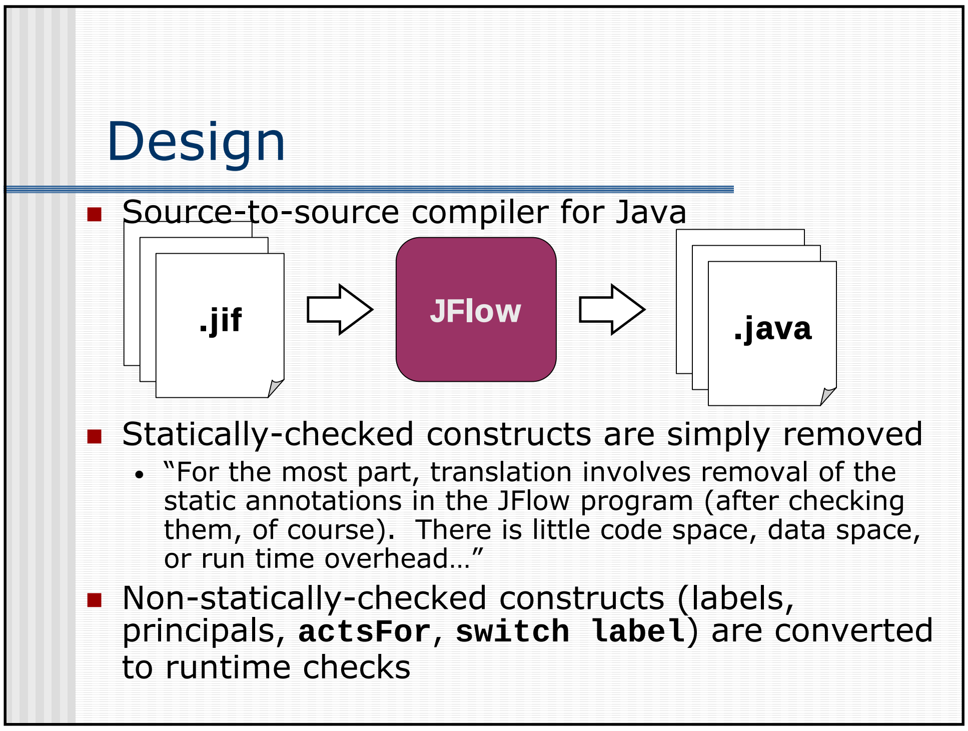
Motivation

- Privacy protection is increasingly critical
- Static information-flow checking is a good solution
  - Compromise between security and performance
- Several languages exist on paper that allow static information-flow checking, but…
  - None are practical; too limited and/or restrictive
- Goal of JFlow: support static information-flow checking *and be practical*



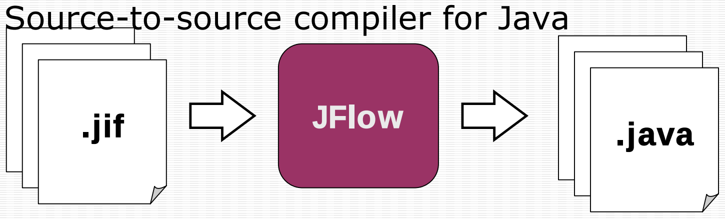
Background

- Builds on existing work:
  - Java
  - Lattice model of information flow [Bell, Denning]
  - Subtype/parametric polymorphism
  - Dependent types [Cardelli]
  - Decentralized label model [Myers]
- Novel work solving practical problems:
  - Mutable objects
  - Declassification
  - Dynamic granting/revoking of authority
  - Label polymorphism
  - Automatic label inference
  - Exceptions

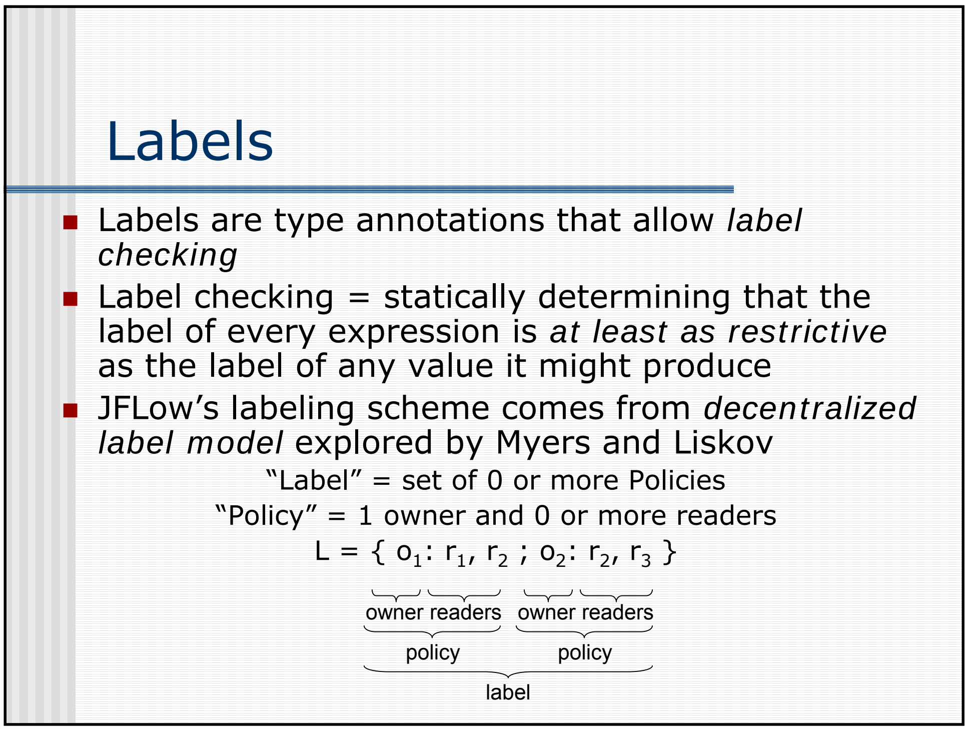


Design

- Source-to-source compiler for Java



- Statically-checked constructs are simply removed
  - "For the most part, translation involves removal of the static annotations in the JFlow program (after checking them, of course). There is little code space, data space, or run time overhead…"
- Non-statically-checked constructs (labels, principals, **`actsFor`**, **`switch label`**) are converted to runtime checks



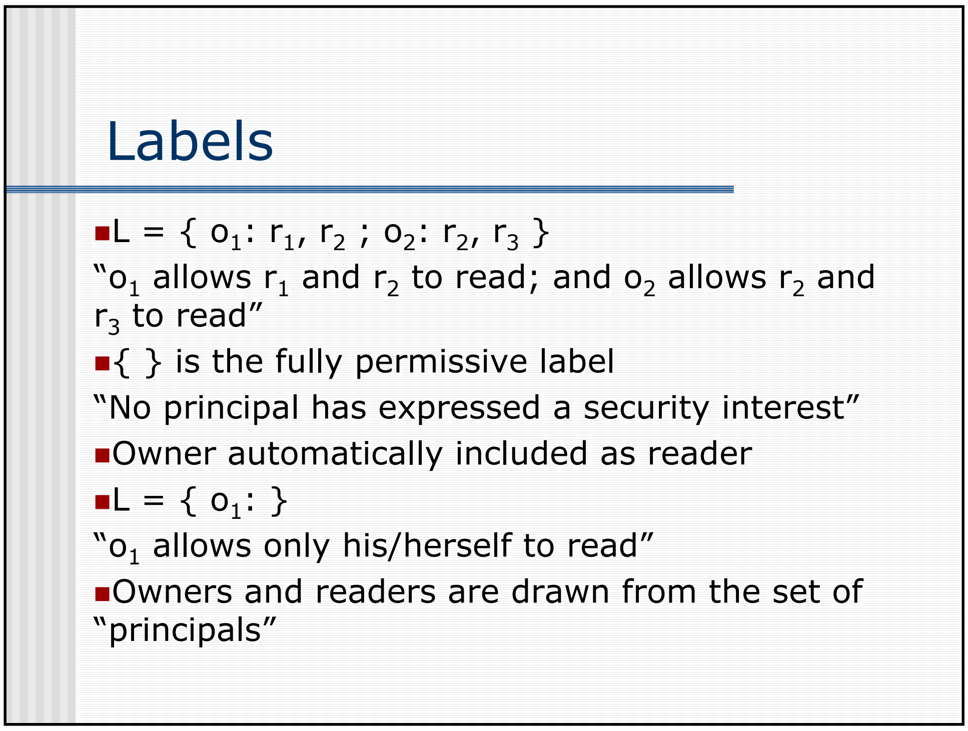
Labels

- Labels are type annotations that allow *label checking*
- Label checking = statically determining that the label of every expression is *at least as restrictive* as the label of any value it might produce
- JFLow's labeling scheme comes from *decentralized label model* explored by Myers and Liskov

"Label" = set of 0 or more Policies

"Policy" = 1 owner and 0 or more readers

$$L = \{ \underbrace{\underbrace{\underbrace{o_1}_{\text{owner}}: \underbrace{r_1, r_2}_{\text{readers}}}_{\text{policy}} ; \underbrace{\underbrace{o_2}_{\text{owner}}: \underbrace{r_2, r_3}_{\text{readers}}}_{\text{policy}}}_{\text{label}} \}$$



Labels

- $L = \{ o_1: r_1, r_2 ; o_2: r_2, r_3 \}$

"o_1 allows r_1 and r_2 to read; and o_2 allows r_2 and r_3 to read"

- { } is the fully permissive label

"No principal has expressed a security interest"

- Owner automatically included as reader
- $L = \{ o_1: \}$

"o_1 allows only his/herself to read"

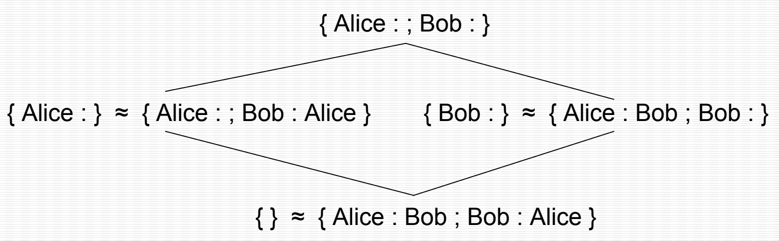
- Owners and readers are drawn from the set of "principals"

Labels

- Data may only be read by a principal if all of the policies in its label list that principal as a reader
  - The "effective policy" of a label is the *intersection* of all its policies
- Labels form lattices
  - Let A ⊔ B be join/LUB of A and B
  - Let A ≤ B be A "can be relabeled as" B
    - For each policy in A, there is a policy at least as restrictive in B
  - Let ≈ be shorthand for A ≤ B and B ≤ A

Labels

- Example lattice where only principals are Alice and Bob


{ Alice : ; Bob : }
{ Alice : } ≈ { Alice : ; Bob : Alice }
{ Bob : } ≈ { Alice : Bob ; Bob : }
{ } ≈ { Alice : Bob ; Bob : Alice }

Labels

- A principal may choose to relax (add readers to) a policy that it owns; this is declassification
  - "Safe" because other policies are not affected
- Some principals are allowed to *act for* other principles
  - There is a "principal hierarchy" that can be updated dynamically
  - Not a key detail

Labeled Types

- Every variable is statically bound to a static label
- A label is denoted by a label expression, which is a set of component expressions
  - However, a component expression may take other forms; e.g., it may be a variable name:

```
int { Alice : } x;
int { x } y;
int { Bob : ; y } z;
```

  - Policy of "x" means "copy variable x's policies here"
  - Effective readers for x, y, z are Alice, Alice, Nobody

Labeled Types

- The programmer may omit labels, in which case JFlow will either infer the label or assign a default
  - "If omitted, the label of a local variable is inferred automatically based on its uses. In other contexts where a label is omitted, a context-dependent default label is generated. For example, for default label of an instance variable is the public label { }."
- Other cases of default label assignment will be noted later

Implicit Flows

- All guarded expressions' labels are forced to be at least as restrictive as the guard's label
- Type system uses variable pc to hold the join-of-all-guard-labels at all points

```
int { public } x;     // pc = {}
boolean { secret } b; // pc = {}
…
x = 0;                // pc = {}
if (b) {              // pc = {}
    x = 1;            // pc = { secret }
}
```

- This example will fail label checking: secret $\not\leq$ public

Runtime Labels

- New primitive type **`label`**
- Needed when a label is relevant but is not known a priori
- Only thing you can do with a **`label`** is **`switch`** on it:

```
label { L } lb;
int { *lb } x;
int { p: } y;
switch label(x) {
    case ( int { y } z ) y = z;
    else throw new UnsafeTransfer();  }
```

- Example is an attempt to transfer value of x to y
- "The statement executed is the first whose associated label is at least as restrictive as the expression label."

Runtime Labels

- **`label`**s also allow methods with dependent type signatures:

```
static float {*lb} compute(int x {*lb}, label lb)
```

- **`label`**s may be used in label expressions only if they are immutable (final) after initialization
  - (method args are implicitly final)

Runtime Principals

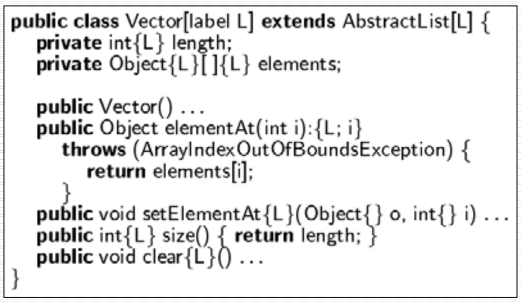
- New primitive type **`principal`**
- Needed if a principal is relevant but not known a priori
- "Run-time principals are needed in order to model systems that are heterogeneous with respect to the principals in the system, without resorting to declassification."

Authority and Declassification

- A principal may declassify (weaken) policies that he or she owns – but where's the principal?
- At a given point, the program is operating on behalf of some set of principals (called the *static authority*)
- Static authority at a given point depends on annotations made by the programmer on the class and method levels
- Only purpose of static authority is to statically determine whether declassifications are legal
- Declassification syntax:
  - **`declassify(e, L)`**: relabels expression e with L

Classes

- Classes may be parameterized (generic with respect to some labels and/or principals)
- "To ensure that these types have a well-defined meaning only immutable (final) variables may be used as parameters"



```
public class Vector[label L] extends AbstractList[L] {
    private int{L} length;
    private Object{L}[ ]{L} elements;

    public Vector() ...
    public Object elementAt(int i):{L; i}
        throws (ArrayIndexOutOfBoundsException) {
            return elements[i];
        }
    public void setElementAt{L}(Object{} o, int{} i) ...
    public int{L} size() { return length; }
    public void clear{L}() ...
}
```

Classes

- If { secret } ≤ { public }, does it follow that Vector[{secret}] ≤ Vector[{public}]?
  - No!
- Programmer may allow this in cases where it is sound by declaring label parameter **`covariant label`**
- **`covariant`** imposes additional constraints: no method argument or mutable instance variable may be labeled using the parameter

Classes

- A class always has one implicit label parameter: the label {this}, representing the label on an object of the class
- In the case of {this}, L1 ≤ L2 should imply that C{L1} acts as a subtype of C{L2}, so {this} must be a **`covariant label`**

Classes

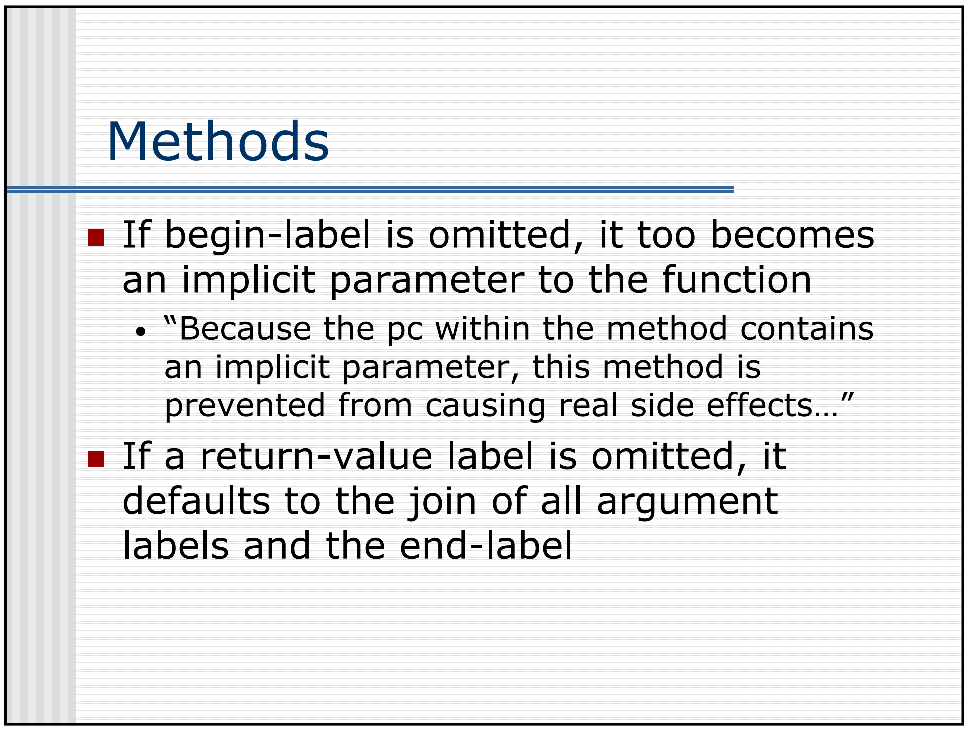
- A class may have some authority granted to its objects by adding an authority clause to the class header
  - **`class passwordFile authority(root) { … }`**
- If the authority clause names "external principals," the process that installs the class into the system must have the authority of the named principals
- "If the authority clause names principals that are parameters of the class, the code that creates an object of the class must have the authority of the actual principal parameters used in the call to the constructor."

Methods

- The return value, arguments, and exceptions may each be individually labeled
- Arguments are always implicitly final
- There is also an optional *begin-label* and *end-label*
  - If *begin-label* is specified then **pc** must be at least as restrictive as *begin-label* at time of call
  - If *end-label* is specified then no termination of the method may leak more information than *end-label* specifies (*end-label* is at least as sensitive as the leaked information)

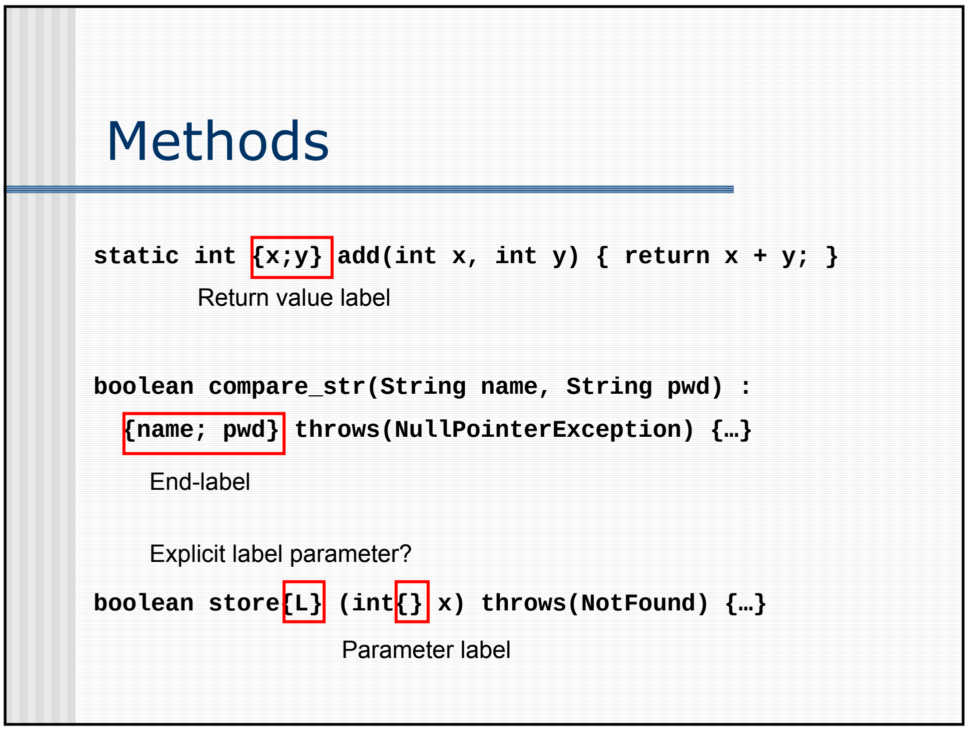
Methods

- When labels are omitted from parameters, those parameters use *implicit label polymorphism*
  - The argument labels become implicit parameters to the function
  - Without label polymorphism, libraries are intractable (need one method for every possible labeling of the parameters)



Methods

- If begin-label is omitted, it too becomes an implicit parameter to the function
  - "Because the pc within the method contains an implicit parameter, this method is prevented from causing real side effects…"
- If a return-value label is omitted, it defaults to the join of all argument labels and the end-label



Methods

```
static int {x;y} add(int x, int y) { return x + y; }
```

Return value label

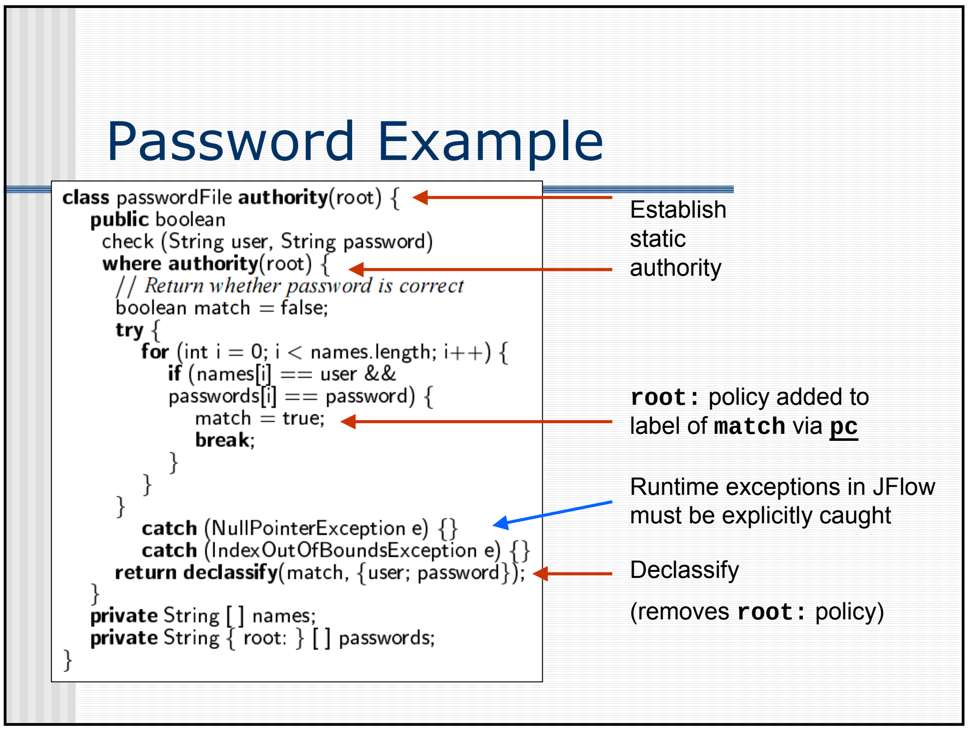
```
boolean compare_str(String name, String pwd) :
  {name; pwd} throws(NullPointerException) {…}
```

End-label

Explicit label parameter?

```
boolean store{L} (int{} x) throws(NotFound) {…}
```

Parameter label



Password Example

```
class passwordFile authority(root) {
  public boolean
   check (String user, String password)
   where authority(root) {
     // Return whether password is correct
     boolean match = false;
     try {
       for (int i = 0; i < names.length; i++) {
         if (names[i] == user &&
         passwords[i] == password) {
           match = true;
           break;
         }
       }
     }
       catch (NullPointerException e) {}
       catch (IndexOutOfBoundsException e) {}
     return declassify(match, {user; password});
  }
  private String [ ] names;
  private String { root: } [ ] passwords;
}
```

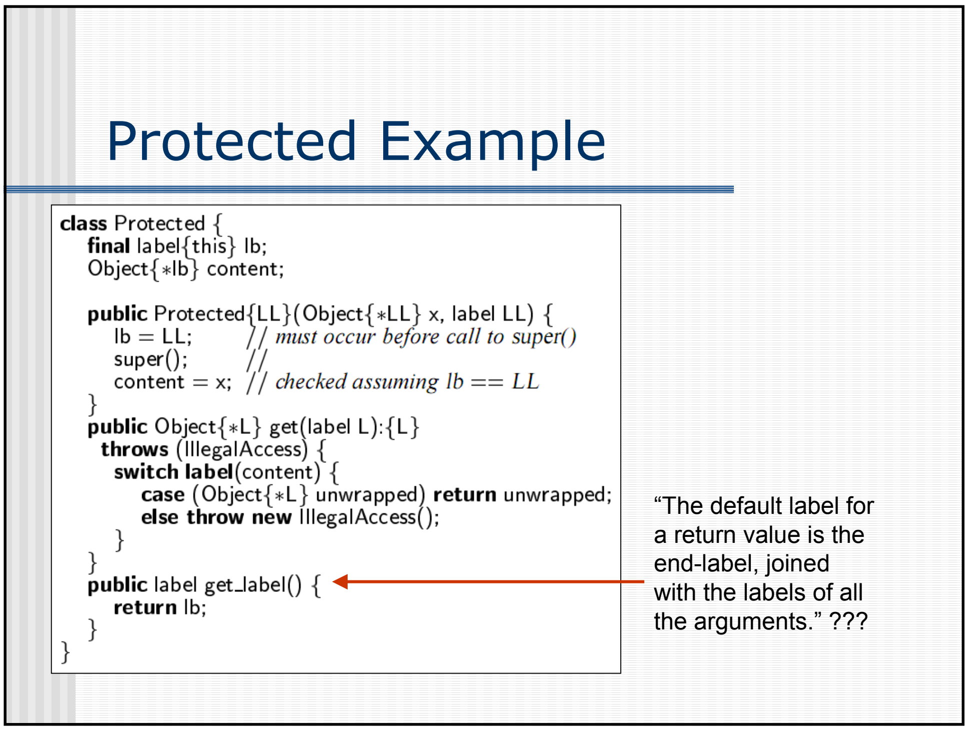
Establish static authority

`root:` policy added to label of **`match`** via **pc**

Runtime exceptions in JFlow must be explicitly caught

Declassify

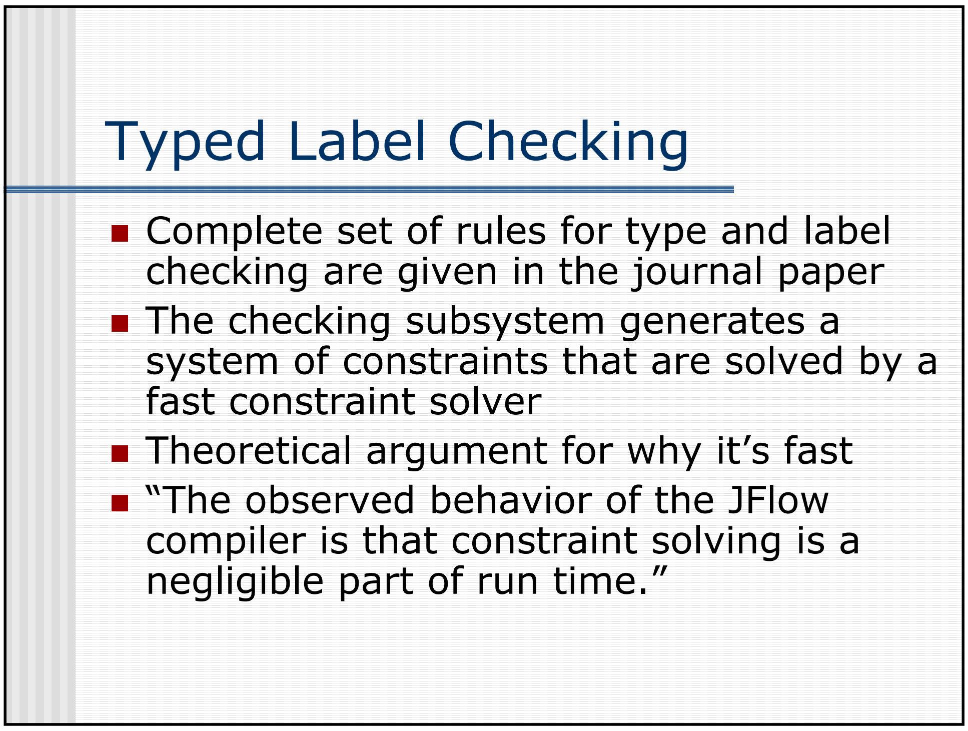
(removes **`root:`** policy)



Protected Example

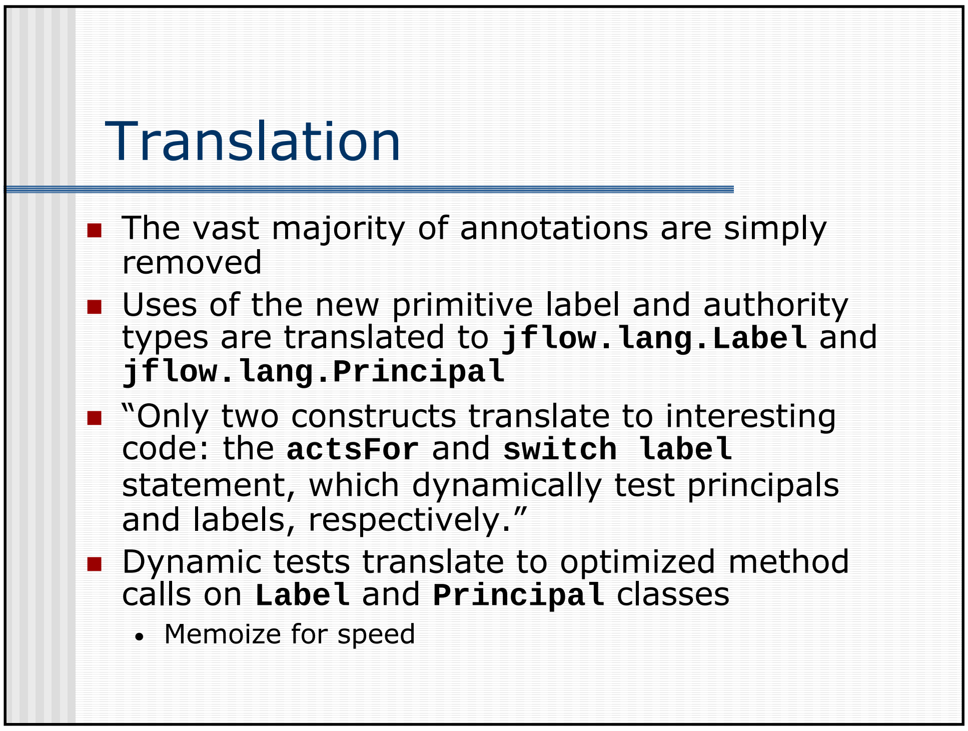
```
class Protected {
  final label{this} lb;
  Object{*lb} content;

  public Protected{LL}(Object{*LL} x, label LL) {
    lb = LL;        // must occur before call to super()
    super();        //
    content = x;  // checked assuming lb == LL
  }
  public Object{*L} get(label L):{L}
   throws (IllegalAccess) {
    switch label(content) {
      case (Object{*L} unwrapped) return unwrapped;
      else throw new IllegalAccess();
    }
  }
  public label get_label() {
    return lb;
  }
}
```

"The default label for a return value is the end-label, joined with the labels of all the arguments." ???



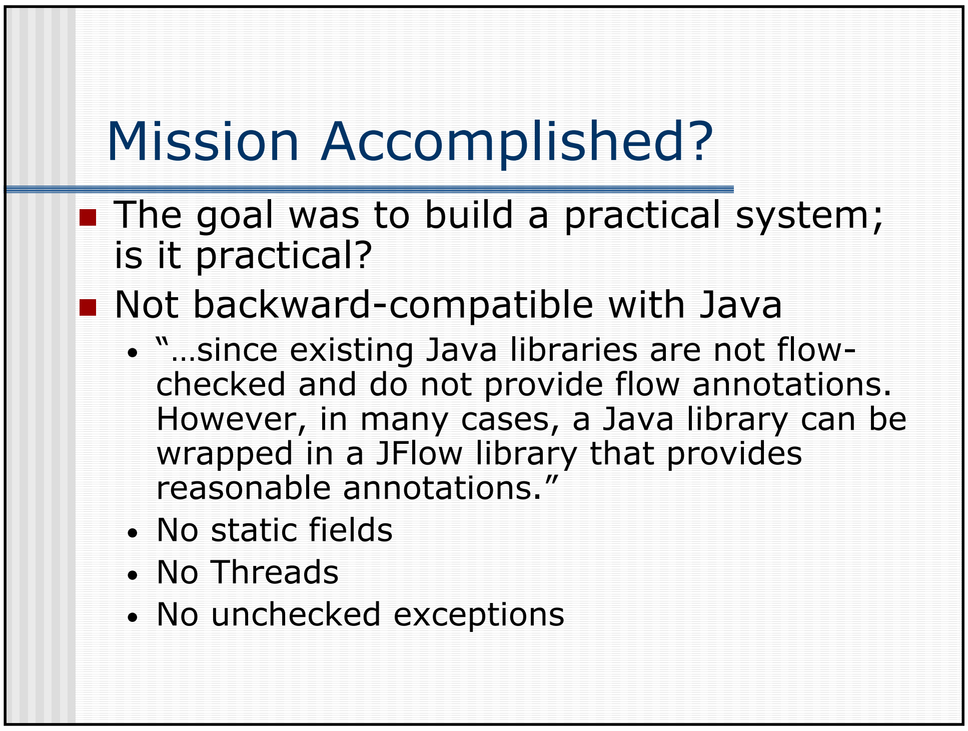
Typed Label Checking

- Complete set of rules for type and label checking are given in the journal paper
- The checking subsystem generates a system of constraints that are solved by a fast constraint solver
- Theoretical argument for why it's fast
- "The observed behavior of the JFlow compiler is that constraint solving is a negligible part of run time."



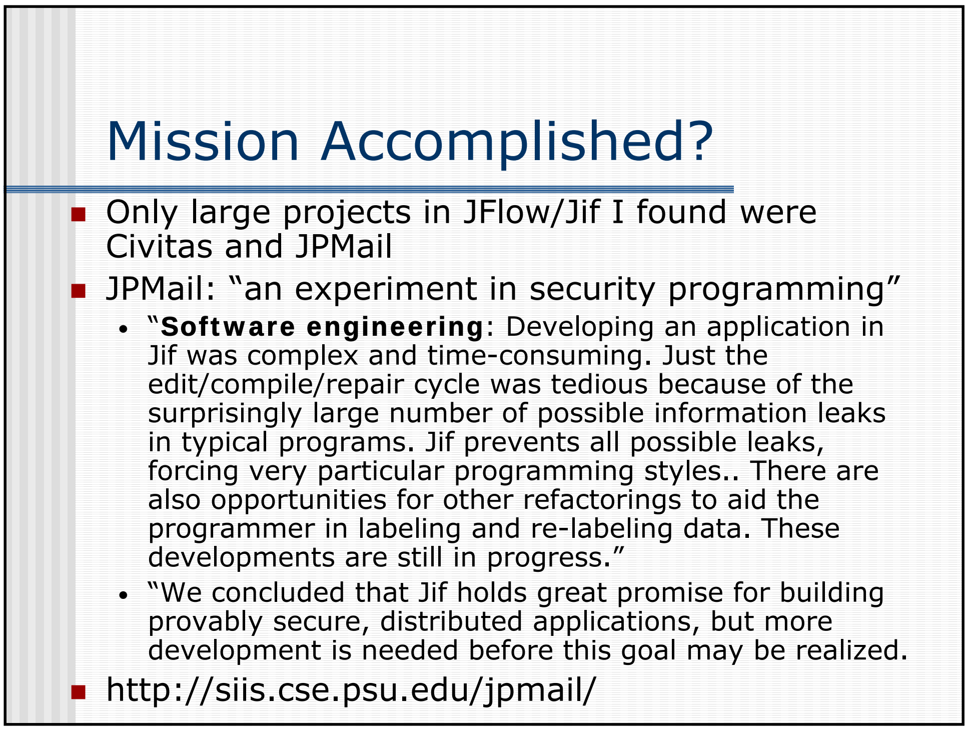
Translation

- The vast majority of annotations are simply removed
- Uses of the new primitive label and authority types are translated to **`jflow.lang.Label`** and **`jflow.lang.Principal`**
- "Only two constructs translate to interesting code: the **`actsFor`** and **`switch label`** statement, which dynamically test principals and labels, respectively."
- Dynamic tests translate to optimized method calls on **`Label`** and **`Principal`** classes
  - Memoize for speed



Mission Accomplished?

- The goal was to build a practical system; is it practical?
- Not backward-compatible with Java
  - "…since existing Java libraries are not flow-checked and do not provide flow annotations. However, in many cases, a Java library can be wrapped in a JFlow library that provides reasonable annotations."
  - No static fields
  - No Threads
  - No unchecked exceptions



Mission Accomplished?

- Only large projects in JFlow/Jif I found were Civitas and JPMail
- JPMail: "an experiment in security programming"
  - "**Software engineering**: Developing an application in Jif was complex and time-consuming. Just the edit/compile/repair cycle was tedious because of the surprisingly large number of possible information leaks in typical programs. Jif prevents all possible leaks, forcing very particular programming styles.. There are also opportunities for other refactorings to aid the programmer in labeling and re-labeling data. These developments are still in progress."
  - "We concluded that Jif holds great promise for building provably secure, distributed applications, but more development is needed before this goal may be realized.
- http://siis.cse.psu.edu/jpmail/