**JFlow: Practical Mostly-Static Information Flow Control**

Andrew Myers

POPL'99

---

**Goal: Expressiveness, practicality**

- Support expected language features
  - Mutable objects
  - Inheritance and subtyping
  - Exceptions
- Explore new security features
  - Explicit security policy annotations (labels)
  - Principals
  - Intentional information release (declassification)
  - Static and dynamic reasoning about information flow and access control
- Support/resolve interactions
  - Label inference, polymorphism, parameterization

---

**Principals**

- Users, groups, and roles: principals
- Principal (or role) hierarchy generated by the acts-for relation
- Policies mention more abstract entities

---

**Labels**

- Every data item has an attached label
- Label is a set of policies
- Each policy is owner: reader1, reader2, ...
  - owner (principal)
  - set of readers (principals)

{Bob: Bob, Preparer ; Preparer: Preparer}

- Every policy is enforced simultaneously

---

**Assignment**

- Assignment relabels a value
  \[ x = y; \]
- Okay if \( x \) is at least as restrictive as \( y \)
  (label of \( z \) is \( \geq \))
- \( y \subseteq x \) ("\( x \) protects \( y \)")
  - For every policy in \( y \), there is a policy in \( x \) that is at least as restrictive

\[
\begin{align*}
\text{or}, \ r' & \equiv \text{or: r} \\
\text{or} & \equiv \text{or: r} \quad (\text{if o' acts for o}) \\
\text{or} & \equiv \text{or: r} \quad (\text{if r' acts for r})
\end{align*}
\]

---

**Assignment example**

\[
\begin{align*}
\text{int } \{\text{Bob: Bob, Preparer}\} \ y; \\
\text{int } \{\text{Bob: Bob; Preparer: Preparer}\} \ x; \\
x = y;
\end{align*}
\]

\[\begin{align*}
\text{Bob: Bob, Preparer} \sqsubseteq \{\text{Bob: Bob; Preparer: Preparer}\}
\end{align*}\]

- Binary label relation \( \sqsubseteq \) defines legal relabelings
- Label semantics: relation on owners and readers
  - Takes into account acts-for (trust) relationships
- Proven sound and complete assuming addition of principals, acts-for relationships
Computation
- Combining values → preserve input labels
  \[ y + z \rightarrow y \sqcup z \]
- New label is the join (\( \sqcup \)) of the input labels
  \[ y \sqcup z = y \sqcup z \]
- Label on result protects all source labels
- preorder \( \sqsubseteq \) defines a lattice of equivalence classes

Selective downgrading
- Declassification = downgrading confidentiality
- A principal can rewrite its part of the label
  \[
  [O1: R1, R2; O2: R2] \\
  O2 \\
  [O1: R1, R2] \\
  [O1: R1, R2; O2: R2, R3]
  \]
- Potentially dangerous: explicit operation
- Other owners’ policies still respected
- Must test authority [and integrity] of running process

Tax Preparer example

Java + Information Flow
- Annotate (Java) programs with labels
- Variables have type + label
  \[ \text{int [L] x;} \]

Labeled Types
- Variables, expressions have labeled type \( T(L) \)
- Labels express privacy constraints
- Assignment rule:
  - Expressions incorporate pc label \( A[pc] \):
  \[
  v : T[L_v] \in A \\
  A \vdash E : L_e \\
  L_v \sqsubseteq L_y \\
  A \vdash v = E : L_e
  \]
Annotated Class Example

```java
class PasswordFile {
    boolean check(String user, String password);
    // Return whether the password is correct
}
```

A password file that store passwords securely but allows them to be checked.

Labeling the Program

```java
class PasswordFile {
    String[] names;
    public String {root: root}[] passwords;

    public boolean {user; password} check(String user, String password) {
        // Return whether the password is correct
        ...
    }
}
```

Implicit Label Polymorphism

- Method signatures contain labeled types
  ```java
  float {Bob: Bob} cos(float {Bob: Bob} x) {
      float {Bob: Bob} y = x - 2*PI*(int)(x/(2*PI));
      return 1 - y*y/2 + ...;
  }
  ```
- Omit argument labels: `label polymorphism`
- Omit variable labels: `label inference`
  ```java
  float{x} cos(float x) {
      float y = x - 2*PI*(int)(x/(2*PI));
      return 1 - y*y/2 + ...;
  }
  ```

Explicit Parameterization

```java
class Cell[label L] {
    private Object{L} y;
    public void store{L} (Object{L} x) { y = x; }
    public Object{L} fetch() { return y; }
}
```

- Straightforward analogy with type parameterization
- Allows generic collection classes
- Parameters not represented at run time

Static Authority

- Authority of code is tracked statically
  ```java
  class C authority(root) {
      void m() where authority(p) { ...
  }
  ```
- but can be propagated dynamically:
  ```java
  void m(principal p, int {root:} x) where caller(p) {
      actsFor(p, root) {
          int{} y = declassify(x, {})
          // checked statically
      } else {
          // can't declassify x here
      }
  }
  ```
Implicit Flows and Exceptions

- Implicit flow: information transferred through control structure
- Static program counter label (pc) that expression label always includes
- Fine-grained exception handling: pc transfers via exceptions, break, continue

Run-time Labels

- Labels may be first-class values, label other values:
  final label a = ...;
  int{*a} b;
- Run-time label treated statically like label parameter: unknown fixed label
- Exists at run time (Jif.lang.Label)
- int{*a} is a (simple) dependent type

Run-time Labels and Implicit Flows

- Proper check is \([b] \sqsubseteq [x]\)
- In case clause, pc augmented with label of label a (which is [b])
- Therefore: \(x = true\) results in proper check

Current and future work

- Current version of language is Jif
- Better constraint solving
- Implicit polymorphism now bounded polymorphism
  \(\text{int}[x] f(\text{int}[L] x) = \text{int}[x] f(\text{int}[L] x)\)
- Integrity extension for distributed systems security (Jif/split)
- Better reasoning about dynamic labels and principals
- Concurrent programming
  www.cs.cornell.edu/jif