JFlow: Practical Mostly-Static Information Flow Control

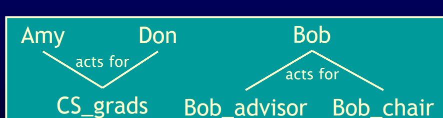
Andrew Myers
POPL'99

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: $reader_1$, $reader_2$,...
  - 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 z)
- y ⊑ x ("x protects y") means

  For every policy in y, there is a policy in x that is at least as restrictive

  o:r, r' ⊑ o:r
  o:r ⊑ o':r (if o' acts for o)
  o:r ⊑ o:r' (if r' acts for r)

Assignment example

```
int {Bob: Bob, Preparer} y;
int {Bob: Bob; Preparer: Preparer} x;
x = y;
```

y ⊑ x ?

{Bob: Bob, Preparer} ⊑ {Bob: Bob; Preparer: Preparer}

- Binary label relation ⊑ defines legal relabelings
- Label semantics: relation on owners and readers o → r
  - Takes into account acts-for (trust) relationships
- Proven sound and complete assuming addition of principals, acts-for relationships

Computation

- Combining values → preserve input labels

$$y + z \quad \rightarrow \quad \underline{y} \sqcup \underline{z}$$

- New label is the *join* (⊔) of the input labels

$$\underline{y}, \underline{z} \quad \sqsubseteq \quad \underline{y} \sqcup \underline{z} \;=\; \underline{y} \cup \underline{z}$$

- Label on result protects all source labels
- preorder ⊑ defines a lattice of equivalence classes

Selective downgrading

- Declassification = downgrading confidentiality
- A principal can rewrite its part of the label

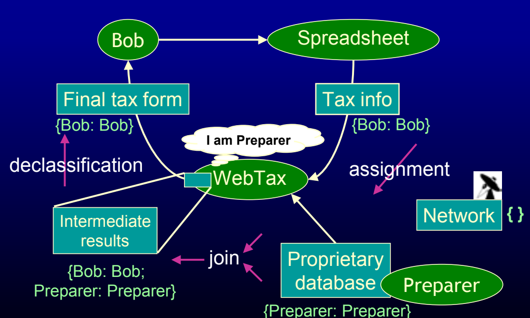
{O1: R1, R2; O2: R2}

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

```
int {L} x;
```

```
float {Bob: Bob} cos (float {Bob: Bob} x) {
  float {Bob: Bob} y = x — 2*PI*(int)(x/(2*PI));
  return 1 — y*y/2 + …;
}
```


Authority

- Each program point has the authority of some set of principals
- Authority is needed only for declassification but can be used as an access control mechanism

```
T m() where authority(p) { … }
T m() where caller(p) { … }
actsFor(p1, p2) { … }
```


Labeled Types

- Variables, expressions have *labeled type* *T{L}*
- Labels express privacy constraints
- Assignment rule:
- Expressions incorporate pc label *A[pc]*:

$$\frac{v : T\{L_v\} \in A \qquad A \vdash E : L_e \qquad L_e \sqsubseteq L_v}{A \vdash v = E : L_e}$$

Annotated Class Example

```
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

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

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


actsFor & declassify

```
class passwordFile authority(root) {
  String [ ] names;
  public  String {root: root} [ ] passwords;

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


Implicit Label Polymorphism

- Method signatures contain labeled types

```
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*

```
float{x} cos (float x) {
   float y = x - 2*PI*(int)(x/(2*PI));
   return 1 - y*y/2 + …;
}
```


Explicit Parameterization

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

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

Static Authority

- Authority of code is tracked statically

```
class C authority(root) {
     void m() where authority(p) { … }
}
```

- but can be propagated dynamically:

```
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

{b}⊑{x}

```
x = b;
```

```
x = false;
if (b) {
   x = true;
}
```

```
x = false;
try {
   if (b) throw new Foo ();
} catch (Foo f) {
   x = true;
}
```


Methods and Implicit Flows

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

begin-label = pc

implicit begin-label

- Begin-label constrains calling pc : pc {L}
- Prevents implicit flow into method
- Omitted begin-label: implicit parameter, prevents mutation

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 Label Discrimination

- switch label statement tests a run-time label dynamically:

```
final label a = ... ;
int{*a} b;
int { C: D } x;
switch label(b) {
   case ( int { C: D } b2 )  x = b2;
   else throw new BadLabelCast();
}
```

tests a ⊑ { C : D } at run time

Run-time Labels and Implicit Flows

```
final label{b} a = b ? new label {L1} : new label {L2};
int{*a} dummy;
switch label(dummy) {
  case ({L1}) : x = true;
  case ({L2}) : x = false;
}
```

=

```
x = b;
```

- Proper check is {b}⊑{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

  int{x} f(int{L} x) ≠ int{x} f(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