Practical Mostly-Static Information Flow Control

Andrew Myers
MIT Lab for Computer Science
Privacy

• Old problem (secrecy, confidentiality): prevent programs from leaking data
• Untrusted, downloaded code: more important
• Standard security mechanisms not effective (e.g., access control)
Privacy with Mutual Distrust
Static Information Flow

- Denning & Denning ’77
- Programs must follow rules
- Annotations added for tractability
- Static analysis = type checking
- Security property composes

\[ A + B = AB \]
Jif Language

• Jif = Java + information flow annotations (Java Information Flow)

• More practical than previous work
  – **Real language:** supports Java features
  – **Convenience:** automatic label inference
  – **Genericity:** label polymorphism
  – **Decentralized declassification** mechanism
  – **Run-time** label checking
Architecture

- Source to source translator (mostly erasure)
- Modification to the **PolyJ** compiler (Java + parametric polymorphism)
Jif Features

- Labeled types
- Convenience: automatic label inference
- Genericity: label polymorphism
- Static, decentralized declassification
- Safe run-time label checking (first-class labels)
- First-class principals
- Object-oriented features
  - Subtyping rules
  - Inheritance
  - Constructors
  - Method constraints
- Exceptions
- Arrays
- Described by formal inference rules
Labeled Types

- Variables, expressions have *labeled type* $T\{L\}$
- Labels express privacy constraints
- $L_2$ is at least as restrictive as $L_1$: $L_1 \sqsubseteq L_2$
- Assignment rule (simplified)

\[
\begin{align*}
v : T\{L_v\} & \in A \\
A \vdash E : L_e \\
L_e & \sqsubseteq L_v \\
A \vdash v = E : L_e
\end{align*}
\]
Decentralized Label Model

• 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 owner’s policy is obeyed
• Relation $\sqsubseteq$ is pre-order w/lattice properties [ML98]
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 + ...;
  }
  ```

- Omitted argument labels: *implicit label polymorphism*
  
  ```
  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
Declassification

• A principal can rewrite its part of the label

\{O1: R1, R2; O2: R2\}

\{O1: R1, R2\} \quad \{O1: R1, R2; O2: R2, R3\}

• Other owners’ policies still respected
• Must know authority of running process
• Potentially dangerous: explicit operation

\texttt{declassify}(E, L)
Static Authority

• Authority of code is tracked statically

   class C authority(root) {
   ...
   }

• Authority 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

\[
\{b\} \sqsubseteq \{x\}
\]

\[
x = b;
\]

\[
x = false; \\
if (b) \{ \\
    x = true; \\
\}
\]

\[
x = false; \\
try \{ \\
    if (b) throw new Foo (); \\
} catch (Foo f) \{ \\
    x = true; \\
\}
\]
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 constrains calling \( \text{pc} : \text{pc} \sqsubseteq \{L\} \)
• 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 dependent type
Run-time Label Discrimination

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

```java
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 $a \sqsubseteq \{ C : D \}$ at run time
Run-time Labels and Implicit Flows

Proper check is $\{b\} \subseteq \{x\}$

In case clause, `pc` augmented with label of label `a` (which is `\{b\}`)

Therefore: $x = true$ results in proper check
Implementation

- Translates to efficient Java, mostly by erasure
  - Labeled types become unlabeled types
  - Label parameters erased
- First-class label, principal values remain
- `switch label, actsFor` translated simply
Is it Practical yet?

- Addresses limitations of earlier approaches to checking information flow statically
  - allows run-time checking
  - infers annotations
  - limited declassification mechanism
  - genericity: implicit & explicit polymorphism
- Greater expressiveness and convenience
- Only small programs so far
- Can reuse existing Java code
- Only sequential programs, no timing channels
Related Work

Denning, Denning. CACM 1977
Palsberg, Ørbæk. ISSA 1995
Volpano, Smith, Irvine. JCS 1996
Heintze, Riecke. POPL 1998
Smith, Volpano. POPL 1998
Abadi, Banerjee, Heintze, Riecke. POPL 1999
Conclusions

• Most practical language yet for static enforcement of privacy
• Promising; more experience needed to understand limitations
• Why not 20 years ago?
Inheritance/Subtyping

• Subclass signature (1) constrained by superclass signature (2)
• Argument, begin-label \( a : \{ a_2 \} \sqsubseteq \{ a_1 \} \)
• Return value, exception \( r : \{ r_1 \} \sqsubseteq \{ r_2 \} \)
• Class authority (set of principals) can only increase with inheritance: \( A_1 \supseteq A_2 \)