JFlow*

CS 6301-002: Language-based Security

Kevin W. Hamlen

November 15, 2017

Goal: Write software that avoids divulging secrets

Information flow policies
- programmer labels secret sources (input parameters, input channels, variables)
- everything else is assumed attacker-viewable
- enforcement detects/blocks flows from secret to public

Traditional approach: taint tracking
- label all values with runtime confidentiality level
- detect policy-violating flows at runtime
- disadvantage: very high runtime overhead (space & time)
- disadvantage: few pre-deployment guarantees
Language-based Information Flow Control

- Additional Goals
  - detect as many flow violations as possible at compile-time ("mostly static")
  - support large, versatile policies (distributed label model)
  - use a normal(ish) programming language (Java)
  - support common language features (objects, subclassing, dynamic typing, exceptions, ...)
  - support controlled declassification

- Approach: Encode classifications and flow tracking as types!
  - type-checker detects most policy violations
  - compiler inserts dynamic tests for remaining flows
  - type-checker proves integrity of dynamic tests (!)
Decentralized Label Model

- Principal – an entity with a security interest
  - Examples: customer, shipping department, billing department
  - Each principal “owns” certain data
- Principals specify may-read policies for owned data.

\[
\text{int\{o_1: r_1, r_2\} x; // o_1 says only r_1 and r_2 (and o_1) may read x}
\]

\[
\text{int\{o_2: r_2, r_3\} y; // o_2 says only r_2 and r_3 (and o_2) may read y}
\]

- Binary operations *join* (⊔) operand policies

\[
z = x+y; // \text{now z has policy \{o_1: r_1, r_2; o_2: r_2, r_3\}}
\]

- *Effective reader set* – intersection of permitted readers
  - Only \( r_2 \) may read \( z \)
Label Comparisons

- \( L_1 \sqsubseteq L_2 \)
  - \( L_1 \) is “less restrictive than (or equivalent to)” \( L_2 \)
  - \( L_1 \) is constrained by a subset of the restrictions upon \( L_2 \)

- Examples
  - \( \{ o_1 : r_1, r_2 \} \sqsubseteq \{ o_1 : r_1 \} \)
  - \( \{ o_1 : r_1 \} \sqsubseteq \{ o_1 : r_1; o_2 : r_2 \} \)

- defines lattice of security labels (\( \sqsubseteq \) is a partial order)
  - reflexive, anti-symmetric, transitive

- basis for defining \( \sqcup \): \( L_1 \sqcup L_2 = \min_{\sqsubseteq}\{ L \mid L_1 \sqsubseteq L, \ L_2 \sqsubseteq L \} \)

- label-checking and inference based on linear constraint solving

- relative labels: \( \text{int}\{y\} x \)
  - \( x \) is at least as confidential as \( y \)
  - \( label(y) \sqsubseteq label(x) \)

- sub-classing / sub-typing
  - can safely up-cast “(\( L \)) \( x \)” if \( label(x) \sqsubseteq L \)
  - downcasts must be guarded by runtime type-checks
Implicit Flow Protection

```java
int{} x;
boolean{o1 : r1} b;

x = 0;
if (b) {
    x = 1;
}
```
Implicit Flow Protection

```java
int{} x;
boolean{o_1 : r_1} b;
:

x = 0;
if (b) { // label(pc) ← label(pc) △ label(b)
    x = 1;  // compile-time type-error: label(pc) ∉ label(x)
}
```
Label Polymorphism

```java
class Account {
    final principal customer;
    final label{customer:} customer_policy;
    float{∗customer_policy} balance;
}
```

- Principals and labels are *first-class*.
  - Statically tracked as *type variables* ($\alpha$)
  - Dynamically tracked as *values* ($L$)
- Static is conservative: $L \subseteq \alpha$
  - Programmer must insert dynamic type-checks to avoid compilation errors.
void print(principal user, String{user:} s) { ... }

void show_balance(principal user, Account a) {
    switch label(a.balance) {
        case (Float{user:} z) print(user, z.toString());
        else print(user, “Permission denied”);
    }
}

class passwdFile authority(root){
    private String[] names;
    private String{root:}[] pwds;
    public boolean check(String user, String pwd) where authority(root) {
        boolean match = false;
        try {
            for(int i = 0; i < names.length; i++) {
                if(names[i] == user && pwds[i] == pwd) {
                    match = true;
                    break;
                }
            }
        } catch(NullPointerException e){}
        catch(IndexOutOfBoundsException e){}
        return declassify(match, {user; pwd});
    }
}
Limitations

- **Threads**: confidentiality not guaranteed for multi-threaded code
  - **Finalizers** are threaded in Java, so excluded from JFlow.
- **Timing channels**: not protected
- **hashCode**: default Java implementation divulges info (how?)
  - Every JFlow class must override hashCode.
- **Static vars**: order of initialization can divulge info
  - No dynamic checks possible in static initializers!
  - Static vars illegal in JFlow
- **Resource exhaustion**: uncatchable in JFlow (1 bit of info divulged)
- **Unchecked exceptions**: excluded from JFlow (rare in Java)
- **Dynamic type discrimination**: only in parameterless classes
Discussion Questions

- Introduced in ’99 (actually earlier). Still not standard. Why?
- Can we verify declassification points?
- What about confidentiality for binary legacy code?
- What about covert channels (power, timing, etc.)?