Type systems for computationally secure information flow in Jif

Liisi Haav

Joint work with
Peeter Laud

5.Oct, 2008

Outline of the talk

- Jif
  - Extension of Java
  - Provides secure information flow
  - http://www.cs.cornell.edu/jif/
- Laud-Vene type system
  - Guarantees computationally secure information flow
  - Two special operations:
    - key generation
    - encryption
- Modeling those two operations in Jif

Jif

- An example

  `int {o -> r; p <- w} x;`
- Label
- Policy
  - Confidentiality policy
    - o -> r
  - Integrity policy
    - p <- w
- Principal
  - o, r, p, w

Jif

- Delegating authority
  - q acts for p
  - ⊤ and ⊥ principals
- Conjunctions and disjunctions
  - p&q acts for both p and q
  - both p and q can act for p,q
- Information with more restrictive label may not be stored in a variable with a less restrictive label
- Downgrading security of information
  - declassification
  - endorsement

Laud-Vene type system

- Designed for a simple imperative language
- Two operations:
  - key generation
  - encryption
- Security level for each variable
  - information does not flow from higher level variables to lower level variables

Laud-Vene type system

- Type of a variable is a pair:
  - information type - gives potential dependencies of sensitive data
    - h — secret information
    - G — the set of all key generation points
    - $T_0 = \{h\} \cup G$ - basic secrets
    - $T_1 = \{t_N \mid t \in T_0, N \subseteq G\}$ - encrypted secrets
    - $T_2 = P(T_1)$ - information type
  - usage type
    - Key_N for $N \subseteq G$
    - Data
- The least upper bound of information types of all public variables must not be $\geq h$

Implementation

- Two operations
  - Key generation
  - Encryption
- Important to take into account
  - keys versus other types of data
  - in which program point a key was generated
  - outputing information
  - principal H
    - only principal that is allowed to read secret data

Implementation

- A special *Key*-class
    - instances of this class can be used for encryption
- Principals P and NotP for each key generation g
    - P is allowed to read keys generated at g
    - NotP surely does not know keys generated at g
    - P&NotP is considered equivalent to T
- Method value() - for using the value of the key

Key-class

```
Class Key [covariant label l1, covariant label l2] {
    final byte[ ]{this} key;
    Key( ) {
        this.key = real_keygen( );
    }
    String{pt meet l2} encrypt{this}(principal p, String pt)
          where {pt , this} <= {p → T; p ← T}, caller(p) {
        String r = real_encrypt(key,pt);
        return declassify(r, {pt meet l2; p ← T});
    }
    String{this ; l1} value( ) {
        return new String(key);
    }
}
```

Key-class

```
Class Key [covariant label l1, covariant label l2] {
    final byte[ ]{this} key;
    Key( ) {
        this.key = real_keygen( );
    }
    String{pt meet l2} encrypt{this}(principal p, String pt)
          where {pt , this} <= {p → T; p ← T}, caller(p) {
        String r = real_encrypt(key,pt);
        return declassify(r, {pt meet l2; p ← T});
    }
    String{this ; l1} value( ) {
        return new String(key);
    }
}
```

l1 ≡ {p → P1 & ... & Pn; p ← T}

l2 ≡ {p → NotP1 & ... & NotPn; p ← T}

Example 1

```
public static void main{p ← T}(principal p, String args[ ])
        where caller(p) {
   PrintStream[{p → NotP1; p ← T}] out = . . . ;
   Key[{p → P1; p ← T},{p → NotP1; p ← T}] k =
        new Key[{p → P1; p ← T},{p → NotP1; p ← T}]( );

   String{p → H; p ← T} pt = . . . ;
   String x = k.encrypt(p,pt );
   out.println(”x: ” + x);
}
```

Example 1

```
public static void main{p ← T}(principal p, String args[ ])
          where caller(p) {
    PrintStream[{p → NotP1; p ← T}] out = . . . ;
    Key[{p → P1; p ← T},{p → NotP1; p ← T}] k =
          new Key[{p → P1; p ← T},{p → NotP1; p ← T}]( );

    String{p → H; p ← T} pt = . . . ;
    String x = k.encrypt(p,pt );
    out.println(”x: ” + x);
}
```

L ≡ {p → Pi & ... & Pk & NotPj & ... & NotPl; p ← T}

Example 1

```
public static void main{p ← T}(principal p, String args[ ])
        where caller(p) {
   PrintStream[{p → NotP1; p ← T}] out = . . . ;
   Key[{p → P1; p ← T},{p → NotP1; p ← T}] k =
        new Key[{p → P1; p ← T},{p → NotP1; p ← T}]( );

   String{p → H; p ← T} pt = . . . ;
   String x = k.encrypt(p,pt );
   out.println(”x: ” + x);
}
```

x: principals H and NotP1

L ≡ {p → Pi & ... & Pk & NotPj & ... & NotPl; p ← T}

Example 1

```
public static void main{p ← T}(principal p, String args[ ])
        where caller(p) {
   PrintStream[{p → NotP1; p ← T}] out = . . . ;
   Key[{p → P1; p ← T},{p → NotP1; p ← T}] k =
        new Key[{p → P1; p ← T},{p → NotP1; p ← T}]( );

   String{p → H; p ← T} pt = . . . ;
   String x = k.encrypt(p,pt );
   out.println(”x: ” + x);
}
```

x: principals H and NotP1

L ≡ {p → Pi & ... & Pk & NotPj & ... & NotPl; p ← T}

Denote {p → P1; p ← T} with **P1**

Example 2

```
{
    PrintStream[{NotP1&NotP2}] out = . . . ;
    Key[{P1},{NotP1}] k1 = new Key[{P1},{NotP1}]( );
    Key[{P2},{NotP2}] k2 = new Key[{P2},{NotP2}]( );

    String{H} pt = . . . ;
    String x1 = k1.encrypt(p,pt );
    String x2 = k2.encrypt(p, k1.value() );
    out.println(”x1: ” + x1 + ”, x2: ” + x2);
}
```

Example 2

```
{
    PrintStream[{NotP1&NotP2}] out = . . . ;
    Key[{P1},{NotP1}] k1 = new Key[{P1},{NotP1}]( );
    Key[{P2},{NotP2}] k2 = new Key[{P2},{NotP2}]( );

    String{H} pt = . . . ;
    String x1 = k1.encrypt(p,pt );
    String x2 = k2.encrypt(p, k1.value() );
    out.println(”x1: ” + x1 + ”, x2: ” + x2);
}
```

x1: principals H and NotP1

x2: principals P1 and NotP2

Example 3

```
{
    PrintStream[{P1&NotP2}] out = . . . ;
    Key[{P1},{NotP1}] k1 = new Key[{P1},{NotP1}]( );
    Key[{P2},{NotP2}] k2 = new Key[{P2},{NotP2}]( );

    String{H} pt = . . . ;
    String x1 = k1.encrypt(p,pt );
    String x2 = k2.encrypt(p, x1);
    out.println(”x2: ” + x2 + ”, k1: ” + k1.value());
}
```

Example 3

```
{
    PrintStream[{P1&NotP2}] out = . . . ;
    Key[{P1},{NotP1}] k1 = new Key[{P1},{NotP1}]( );
    Key[{P2},{NotP2}] k2 = new Key[{P2},{NotP2}]( );

    String{H} pt = . . . ;
    String x1 = k1.encrypt(p,pt );
    String x2 = k2.encrypt(p, x1);
    out.println(”x2: ” + x2 + ”, k1: ” + k1.value());
}
```

k1: principal P1

x2: principals H, NotP1 and NotP2

Example 4

```
{
    PrintStream[{NotP1&NotP2}] out = . . . ;
    Key[{P1},{NotP1}] k1 = new Key[{P1},{NotP1}]( );
    Key[{P2},{NotP2}] k2 = new Key[{P2},{NotP2}]( );
    Key[{P1&P2},{NotP1&NotP2 }] k3 = low ? k1 : k2;

    String{H} pt = . . . ;
    String x = k3.encrypt(p,pt );
    out.println(”x: ” + x);
}
```

Example 4

```
{
    PrintStream[{NotP1&NotP2}] out = . . . ;
    Key[{P1 },{NotP1}] k1 = new Key[{P1},{NotP1}]( );
    Key[{P2 },{NotP2}] k2 = new Key[{P2},{NotP2}]( );
    Key[{P1&P2},{NotP1&NotP2 }] k3 = low ? k1 : k2;

    String{H} pt = . . . ;
    String x = k3.encrypt(p,pt );
    out.println(”x: ” + x);
}
```

x: principal NotP1&NotP2

Example 5 - Failing example

```
{
    PrintStream[{NotP1&NotP2}] out = . . . ;
    Key[{P1 },{NotP1}] k1 = new Key[{P1},{NotP1}]( );
    Key[{P2 },{NotP2}] k2 = new Key[{P2},{NotP2}]( );
    Key[{P3 },{NotP3}] key = new Key[{P3},{NotP3}]( );
    Key[{P1&P2},{NotP1&NotP2 }] k3 = high ? k1 : k2;

    String{H} pt = . . . ;
    String x = k3.encrypt(p,pt );
    out.println(”x: ” + x);
}
```

Example 5 - Failing example

```
{
    PrintStream[{NotP1&NotP2}] out = . . . ;
    Key[{P1 },{NotP1}] k1 = new Key[{P1},{NotP1}]( );
    Key[{P2 },{NotP2}] k2 = new Key[{P2},{NotP2}]( );
    Key[{P3 },{NotP3}] key = new Key[{P3},{NotP3}]( );
    Key[{P1&P2},{NotP1&NotP2 }] k3 = key.value() ? k1 : k2;

    String{H} pt = . . . ;
    String x = k3.encrypt(p,pt );
    out.println(”x: ” + x);
}
```

x: should be principals H&P3 or NotP1&NotP2

Example 5 - Failing example

```
{
    PrintStream[{NotP1&NotP2}] out = . . . ;
    Key[{P1 },{NotP1}] k1 = new Key[{P1},{NotP1}]( );
    Key[{P2 },{NotP2}] k2 = new Key[{P2},{NotP2}]( );
    Key[{P3 },{NotP3}] key = new Key[{P3},{NotP3}]( );
    Key[{P1&P2},{NotP1&NotP2 }] k3 = key.value() ? k1 : k2;

    String{H} pt = . . . ;
    String x = k3.encrypt(p,pt );
    out.println(”x: ” + x);
}
```

x: principals H&P3 or NotP1&NotP2&P3

Instance encryption method

- Advantages
  - Correctly typed according to Laud-Vene type system
- Disadvantages
  - Too strong restrictions where implicit information flow is concerned

Static encryption method

```
Class Key [covariant label l1, covariant label l2] {
    final byte[ ]{this} key;
    String{pt meet l2} encrypt{this}(principal p, String pt)
                where {pt , this} ≤ {p → T; p ← T}, caller(p) {
        String r = real encrypt(key,pt);
        return declassify(r, {pt meet l2; p ← T});
    }
    static String{pt meet l2; k meet l2} s_encrypt{this}
                (principal p, Key[l1,l2] k, String pt) throws NullPointerException
                      where {pt , k} ≤ {p → T; p ← T}, caller(p) {
        byte[ ] kv = declassify(k, k meet l2).key;
        String r = real encrypt(kv,pt );
        return declassify(r, {pt meet l2; k meet l2; p ← T});
    }
}
```

Static encryption method

- Advantages
  - Typed almost like in Laud-Vene type system
- Disadvantages
  - NullPointerException
    - Handling the exception is not correct
    - Handling the exception might be cumbersome

Thank you!