JFlow: Practical Mostly-Static Information Flow Control

By Andrew C. Myers (POPL ’99)
Presented by Daryl Zuniga

Overview

- Information-flow: what and why
- JFlow: Intro
- JFlow: How it works
- JFlow: Characteristics and limitations
- Discussion

Overview

- **Information-flow: what and why**
- JFlow: Intro
- JFlow: How it works
- JFlow: Characteristics and limitations
- Discussion

Information-flow

- Goal: ensure programs satisfy security policies
- Example: ensure secret data isn't leaked
- Information-flow control is a *mechanism* for enforcing *policies*
- Non-goal(s): program optimization

Information-flow

- Security concepts:
  - Confidentiality: don't leak important data (e.g. passwords)
    - Formally: given two arbitrary executions of a program, if you only changed the secret inputs, only the secret outputs can change (aka "non-interference")
  - Integrity: don't corrupt important data (e.g. votes)
    - Formally: given two arbitrary executions of a program, if you only changed the public inputs, only the public outputs can change (also non-interference)

Information-flow

- Security concepts:
  - Channels: mechanisms for signaling information through a computing system.
  - Covert channels: channels that exploit a mechanism whose primary purpose is not information transfer.
    - Timing channels
    - Termination channels

Information-flow

- Other security mechanisms:
  - Access control
  - Firewalls
  - Encryption
  - Antivirus

Information-flow

- Access control
  - Example: permissions in a file system. Only authorized readers can access certain files.
- **“Access control does not control how the data is used after it is read from the file.”**

Information-flow

- Firewalls
  - Works by preventing communication with the outside world.
  - **“Firewalls permit some communication in both directions; whether this communication violates confidentiality lies outside the scope of the firewall mechanism.”**

Information-flow

- Encryption
  - Secures an information channel so only the endpoints have access.
  - **“Encryption provides no assurance that once the data is decrypted, the computation at the receiver respects the confidentiality of the transmitted data.”**

Information-flow

- Antivirus
  - Detects patterns of previously known malicious software.
  - **Limited protection against new attacks.**

Information-flow

- Information-flow control lets you reason about how programs that have access to sensitive data, handle that sensitive data.
- None of these other approaches can do that.

Overview

- Information-flow: what and why
- **JFlow: Intro**
- JFlow: How it works
- JFlow: Characteristics and limitations
- Discussion

JFlow: Intro

- Information-flow control mechanism
- By Andrew Myers (Cornell)
  - > 40 papers
  - Badass
- JFlow's successor "Jif" is still active

JFlow: Intro

- "JFlow: Practical Mostly-Static Information Flow Control"
  - JFlow: Java language extension
  - Practical: expressiveness, easy-of-use, and run-time performance are important goals for JFlow
  - Mostly-static: most policy checking is done statically; great runtime performance

Overview

- Information-flow: what and why
- JFlow: Intro
- **JFlow: How it works**
- JFlow: Characteristics and limitations
- Discussion

JFlow: How it works

- Type annotations
- Assignment
- Definitions
- Implicit Flow
- Runtime labels
- Runtime principles
- Authority
- Declassification
- passwordFile example
- Parameterization
- Vector example
- Method labels
- [SKIPPING] Static checking
- Translation

JFlow: Type Annotations

- JFlow works by adding policies as type annotations
- Checked statically (mostly)
- Example:
  `int{o1:r1, r2; o2:r2, r3} x;`
- Only r2 can read x
- Every object/value has a label
  - most are inferred or have sensible defaults
- `{}` is the least-restrictive / most-public label
  - (no owner has expressed an interest in restarting the data)

JFlow: Assignment

- Example:
  ```
  int{o1:r1, r2; o2:r2, r3} x;
  x = v;
  ```
- Legal only if x's label is at least as restrictive as v's label

JFlow: Definitions

- Principle: user, role, group, …
- Policy: {owner: [readers…]}
  - Owners and readers are principles
- Label: {policy1; policy2; var1; …}
  - Copies(?) all policies from var1's label

JFlow: Implicit Flow

- Example:

```
int{public} x;
boolean{secret} b;
…
int x = 0;
if (b) {
    x = 1;
}
```

- Secret information has leaked! (`x = b ? 1 : 0`).

- Solution? Program-counter (**pc**) labels.

JFlow: Implicit Flow

- Example:

```
{}   int{public} x;
{}   boolean{secret} b;
…
{}   int x = 0;
{}   if (b) {
{b}      x = 1;
{}   }
```

- The literal “1” actually has the label {b}. (All literals do this.)

- Compiler error because 1’s label is more restrictive than x’s

JFlow: Runtime labels

- Labels are also first-class values
- Examples:
  - File systems: each file has its own permissions.
  - Bank accounts: each account has its own privacy requirements.
- Necessary also if you want to compute labels.
- Label variables are always immutable (aka final).

JFlow: Runtime labels

- Example:
  `static float{*lb} compute(int x{*lb}, label lb)`

- `lb` is both a value and a label for other types

- `*lb` means the label inside `lb`.

- Note: JFlow function arguments are immutable (aka final).

JFlow: Runtime labels

- "`switch label`" construct lets you branch on labels at runtime
- Example:

```
label{L} lb;
int{*lb} x;
int{p:} y;
switch label(x) {
    case (int{y} z)
        y = z;
    else throw new UnsafeTransfer();
}
```

- Note: PC label at "`y = z`" includes `L`
- Only legal if `{L}` is less restrictive than `{y}`
- (`switch label` is evaluated at run-time)

JFlow: Runtime principles

- Principles are also first-class values
- Examples:
    - Bank accounts: each account is a different customer; each customer is a different principle.
- Necessary also if you want to compute principles.
- Principle variables are always immutable (aka final).

JFlow: Runtime principles

- Example:

```
class Account {
        final principle customer;
        String{customer:} name;
        float{customer:} balance;
}
```

JFlow: Authority

- Each principle has some "**authority**".
- Authority grants the ability to **act for** some set of principles.
  - This creates a **principal hierarchy**.
- Authority also grants the ability to **declassify** data.
  - Declassification reduces the strictness of a label.

JFlow: Authority

- Each code location also has some authority.
- Classes are given authority by an "**authority clause**"
    - Restricts who is allowed to create instances
    - (Note: It is not possible to obtain authority by inheriting from a superclass.)
- Methods are given authority by an "**authority constraint**"
    - Authority constraints are a subset of class authorities
        - **principle of least privilege**: not all the methods of a class need to possess the full authority of the class.
- Or by "**caller constraint**"
    - Caller grants authority to method (works for dynamic principles too)

JFlow: Authority

- Authority can be tested dynamically using the "`actsFor`" construct

- Example:
  `actsFor(p1, p2) S;`

- S is a statement.

- S only executes if p1 can act for p2

- If S's authority includes p1, then it is augmented with p2

- (`actsFor` is evaluated at run-time)

JFlow: Authority

- Authority can be also be tested at method call-sites using the "**actsFor constraint**"

- (evaluated statically)

JFlow: Declassification

- `declassify(e, L)`
- Relabels the result of expression `e` with label `L`
- `declassify` is checked statically.
- Legal only if the static authority at the code location can act for all the principles in the policies being relaxed.
  - Doesn't need authority to act for ALL principles mentioned in `e`'s policies.

JFlow: passwordFile Ex.

```
• class passwordFile authority(root) {
    public boolean
      check (String user, String password)
      where authority(root) {
        boolean match = false;
        try {
          for (int i = 0; i < names.length; i++) {
            if (names[i] == user && passwords[i] == password) {
              //PC: {user; password; root:}
              match = true;
              break;
            }
          }
        }
          catch (NullPointerException e) {}
          catch (IndexOutOfBoundsException e) {}
        return declassify(match, {user; password});
    }
    private String[] names;
    private String{root:}[] passwords;
}
```

JFlow: Parameterization

- Classes may be **generic** with respect to some set of labels and/or principles

- Necessary for general purpose data structures
  - Otherwise, you'd need to reimplement "Vector" for every possible label that elements might have.

- Note: parameterization makes JFlow classes simple "**dependent types**" (types contain values)

JFlow: Parameterization

- Sub-typing is generally **invariant** in label parameters
  - Unless a parameter is declared "**covariant**" (this places additional restrictions.)
- A class always has an implicit {this} label parameters which is covariant.

JFlow: Vector Ex.

```
• public class Vector[label L] extends AbstractList[L] {
    private int{L} length;
    private Object{L}[]{L} elements;

    public Vector() …
    public Object elementAt(int i):{L; i}
      throws(ArrayIndexOutOfBoundsException){
        return elements[i];
      }
    public void setElementAt{L}(Object{} o, int{} i) …
    public int{L} size() { return length; }
    public void clear{L}() …
}
```

JFlow: Parameterization

- Methods may also be generic with respect to some set of labels and/or principles

- Necessary for general purpose library functions

- Otherwise, you'd need to reimplement "Math.Add" for every possible label that inputs might have.

JFlow: Parameterization

- ```
  static int{x;y} add(int x, int y) { return x+y; }
  boolean compare_str(String name, String pwd)
     :{name; pwd}
     throws(NullPointerException){…}
  boolean store{L}(int{} x)
     throws(NotFound){…}
  ```

- "**implicit label polymorphism**": When an argument label is omitted, the method is generic with respect to the label of the argument
  ```

JFlow: Method labels

- Methods may optionally specify a “begin-label” and “end-label”

- **begin-label**: restricts the pc label at the call-site

- **end-label**: specifies information that may be learned by observing normal termination

- **termination**: Normal termination, return values, and exceptions all have labels

JFlow: Method labels

- ```
  static int{x;y} add(int x, int y) { return x+y; }
  boolean compare_str(String name, String pwd)
     :{name; pwd}
     throws(NullPointerException){…}
  boolean store{L}(int{} x)
     throws(NotFound){…}
  ```

- The default **end-label** is the PC label at the end of the method.
  ```

JFlow: Method labels

- ```
  static int{x;y} add(int x, int y) { return x+y; }
  boolean compare_str(String name, String pwd)
     :{name; pwd}
     throws(NullPointerException){…}
  boolean store{L}(int{} x)
     throws(NotFound){…}
  ```

- The default label for a **return value** is the end-label joined with the labels of all arguments
  ```

JFlow: Method labels

- ```
  static int{x;y} add(int x, int y) { return x+y; }
  boolean compare_str(String name, String pwd)
     :{name; pwd}
     throws(NullPointerException){…}
  boolean store{L}(int{} x)
     throws(NotFound){…}
  ```

- The default label for an **exception** is the end-label.
  ```

JFlow: Static checking

- **SKIPPING**: (most of section 3)
  - Exceptions
    - "**Path labels**" (n, r, nv, nr, <goto l>, <goto e>, …)
  - Type checking vs. label checking
  - Subtype rules
  - Label-checking rules
  - Throwing and catching exceptions
  - Run-time label checking
  - Checking method calls
  - Constraint solving
    - O(nh) and O(nd)
      - h: max height of lattice
      - d: max back-edges in depth-first traversal of constraint dependency graph)

JFlow: Translation

- JFlow is compiled to Java
- All type labels are erased
- All class parameters are erased
- declassify expressions are replaced by their contained statement
- label goes to `jflow.lang.Label`
- principal goes to `jflow.lang.Principal`
- `actsFor` and `switch label` become dynamic tests

Overview

- Information-flow: what and why
- JFlow: Intro
- JFlow: How it works
- **JFlow: Characteristics and limitations**
- Discussion

JFlow: Characteristics

- "**Decentralized** label model"
  - Allows safe, statically-checked declassification even with **mutual distrust**
- Access control (code privilege can be controlled statically or dynamically)
- Label polymorphism (parameterization/generics)
- Label & parameters inference & defaults (makes it easier for the developer)
- Exception & termination precision (adds expressiveness)
- Runtime support (can compute with labels and principles)
- Mostly-static (low run-time costs; immediate validation)
- Fast compilation (O(hn); h = height of lattice)
- Java extension (uses Java infrastructure)
- Dependent types (neat)

JFlow: Limitations

- Java language extension
    - JFlow can’t verify programs not written in JFlow
    - Limited use of libraries not written in JFlow (e.g. the entire Java standard library)
- Mostly-static
    - Most policies only checked at compile time (doesn’t carry proof)
    - Output is frozen
- Policy specification: {owner: [readers, …]}
    - Is it a natural way to express all desired policies?
- Allows declassification (feature and liability)
    - Lazy programmer might declassify something inappropriately to shut up the compiler.
- Other Java feature limitations: HashCode, static variables, finalizers, casts & instanceof, immutable arguments
- Mostly sound

JFlow: Limitations

- Mostly-sound
  - Soundness: only correct programs are admitted
- Completeness: only incorrect programs are rejected
  - JFlow is also incomplete
  - (But so is every type system)

JFlow: Limitations

- Mostly-sound
  - System clock (more generally: *timing channels*)
  - Multiple threads
  - Resource exhaustion
  - Power channels

Overview

- Information-flow: what and why
- JFlow: Intro
- JFlow: How it works
- JFlow: Characteristics and limitations
- **Discussion**

Discussion

- Limitations. How big of an issue are they, and what can we do about them?
  - How expressive are JFlow policies?
    - What about write-only permissions?
    - Incompleteness: What programs satisfy our policies that JFlow rejects?
- What are some broader applications of information-flow?
  - Program optimization?
  - Other forms of correctness?
  - Can we ensure integrity?
- Is JFlow provably sound?

Appendix

JFlow: Protected Ex.

```
• class Protected {
    final label{this} lb;
    Object{*lb} content;

    public Protected{LL}(Object{*LL} x, label LL) {
      lb = LL; //must occur before all to super()
      super();
      content = x; //checked assuming lb == LL
    }
    public Object{*L} get(label L):{L}
      throws (IllegalAccess) {
        switch label(content) {
          case (Object{*L} unwrapped) return unwrapped;
          else throw new IllegalAccess();
        }
      }
      public label get_label() {
        return lb;
      }
}
```