A Certified Non-Interference Java Bytecode Verifier

G. Barthe, D. Pichardie and T. Rezk, A Certified Lightweight Non-Interference Java Bytecode Verifier, ESOP'07
Motivations 1: bytecode verification

Java bytecode verification

- checks that applets are correctly formed and correctly typed,
- using a static analysis of bytecode programs

But Java bytecode verifier (and more generally Java security model)

- only concentrates on who accesses sensitive information,
- not how sensitive information flows through programs

In this work

- We propose an information flow type system for a sequential JVM-like language, including classes, objects, arrays, exceptions and method calls.
- We prove in Coq that it guarantees the semantical non-interference property on method input/output.
Non-Interference

“Low-security behavior of the program is not affected by any high-security data.”  Goguen&Meseguer 1982

High = secret
Low = public
Non-Interference

“Low-security behavior of the program is not affected by any high-security data.”  Goguen&Meseguer 1982

High = secret
Low = public
Non-Interference

“Low-security behavior of the program is not affected by any high-security data.”  Goguen & Meseguer 1982

\[ \forall s_1, s_2, s_1 \sim_L s_2 \implies \llbracket P \rrbracket(s_1) \sim_L \llbracket P \rrbracket(s_2) \]
Example of information leaks

Explicit flow:

```java
public int{L} foo(int{L} l; int{H} h) {
    return h;
}
```

Implicit flow:

```java
public int{L} foo(int{L} l1; int{L} l2; int{H} h) {
    if (h==0) {return l1;} else {return l2;};
}
```

We use here the Jif (http://www.cs.cornell.edu/jif) syntax:

- a security-typed extension of Java (source) with support for information flow.
Information flow type system

Type annotations required on programs:

* one security level attached to each fields,
* one security level for the contents of arrays (given at their creation point),
* each methods posses one (or several) signature(s):

\[ k_v \xrightarrow{k_h} k_r \]

* \( k_v \) provides the security level of the method parameters (and local variables),
* \( k_h \) is the effect of the method on the heap,
* \( k_r \) is a record of security levels of the form \( \{ n : k_n, e_1 : k_{e_1}, \ldots e_n : k_{e_n} \} \)

* \( k_n \) is the security level of the return value (normal termination),
* \( k_i \) is the security level of each exception that might be propagated by the method.
Example

\[ m : (x : L, y : H) \xrightarrow{H} \{ n : H, C : L, np \} \]

```java
int m(boolean x, C y) throws C {
    if (x) {
        throw new C();
    } else {
        y.f = 3;
    }
    return 1;
}
```

- \( k_h = H \): no side effect on low fields,
- \( \tilde{k}_r[n] = H \): result depends on \( y \)
- termination by an exception \( C \) doesn't depend on \( y \),
- but termination by a null pointer exception does.
Typing judgment

\[ m[i] = ins \quad constraints \]
\[ \Gamma, \text{region}, \text{se}, \text{sgn}, i \vdash st \Rightarrow st' \]

\[ m[i] = \text{putfield } f_k \]
\[ k_1 \sqcup \text{se}(i) \sqcup k_2 \leq k \quad k_h \leq k \quad k_2 \leq \vec{k}_r[\text{np}] \]
\[ \forall j \in \text{region}(i, \emptyset) \cup \text{region}(i, \text{np}), \quad k_2 \leq \text{se}(j) \]
\[ \Gamma, \text{region, se}, \vec{k}_v \xrightarrow{k_h} \vec{k}_r, i \vdash k_1 :: k_2 :: st \Rightarrow \text{lift}_{k_2} st \]
Typing judgment

General form:
\[
\begin{align*}
\text{m}[i] &= \text{ins constraints} \\
\Gamma, \text{region}, \text{se}, \text{sgn}, i \vdash st &\Rightarrow st'
\end{align*}
\]

Example: putfield without handler for NullPointerException exceptions

\[
\begin{align*}
\text{m}[i] &= \text{putfield } f_k \\
 k_1 \sqcup \text{se}(i) \sqcup k_2 &\leq k \\
 k_h &\leq k \\
 k_2 &\leq \vec{k}_r[\text{np}] \\
 \forall j \in \text{region}(i, \emptyset) \cup \text{region}(i, \text{np}), k_2 &\leq \text{se}(j)
\end{align*}
\]

\[
\begin{align*}
\Gamma, \text{region}, \text{se}, \vec{k}_v &\rightarrow k_h, i \vdash k_1 :: k_2 :: st \Rightarrow \text{lift}_{k_2} st
\end{align*}
\]

See the Coq development for 63 others typing rules...
The putfield rule on an example

\[ m : (x : L, y : H) \xrightarrow{H} \{ n : H, C : L, np \} \]

```java
int m(boolean x, C y) throws C {
    if (x) { throw new C(); }
    else { y.f = 3; }
    return 1;
}
```

1. load x
2. ifeq 5
3. new C
4. throw
5. load y
6. push 3
7. putfield f:H
8. push 1
9. return
The putfield rule on an example

\[ m : (x : L, y : H) \xrightarrow{H} \{ n : H, C : L, np \} \]

```java
int m(boolean x,C y) throws C {
    if (x) {throw new C();}
    else {y.f = 3;};
    return 1;
}
```

region(i,tau) is a control depend region that contains the scope of a branching point i.
The putfield rule on an example

\( m : (x : L, y : H) \xrightarrow{H} \{n : H, C : L, np\} \)

```java
type m(boolean x, C y) throws C {
    if (x) {throw new C();}
    else {y.f = 3;};
    return 1;
}
```

region(i,tau) is a control depend region that contains the scope of a branching point i.
The putfield rule on an example

\[ m : (x : L, y : H) \xrightarrow{H} \{ n : H, C : L, np \} \]

```java
int m(boolean x, C y) throws C {
    if (x) { throw new C(); }
    else { y.f = 3; };
    return 1;
}
```

region(i, tau) is a control depend region that contains the scope of a branching point i.
The putfield rule on an example

\[ m : (x : L, y : H) \xrightarrow{H} \{n : H, C : L, np\} \]

```java
int m(boolean x, C y) throws C {
    if (x) {throw new C();}
    else {y.f = 3;};
    return 1;
}
```

region(i,tau) is a control depend region that contains the scope of a branching point i.

se(i) : program point security level
The putfield rule on an example

\[ m : (x : L, y : H) \xrightarrow{H} \{n : H, C : L, np\} \]

```java
int m(boolean x, C y) throws C {
    if (x) {throw new C();}
    else {y.f = 3;};
    return 1;
}
```

\[ m[i] = \text{putfield } f_k \]

\[
k_1 \sqcup \text{se}(i) \sqcup k_2 \leq k \quad k_h \leq k_2 \leq k_r[\text{np}] \\
\forall j \in \text{region}(i, \emptyset) \cup \text{region}(i, \text{np}), k_2 \leq \text{se}(j)
\]

\[
\Gamma, \text{region}, \text{se}, k_v \xrightarrow{k_h} k_r, i + k_1 :: k_2 :: \text{st} \Rightarrow \text{lift}_{k_2} \text{st}
\]
Machine-checked proof

Motivations

- Implementing an information flow type checker for real Java is a non-trivial task.
- A non-interference paper proof is already a big achievement but how is it related to what is implemented at the end?

Using a proof assistant like Coq allows

- to formally define non-interference definition,
- to formally define an information type system,
- to mechanically proved that typability enforces non-interference, (20,000 lines of Coq...),
- to program a type checker and prove it enforces typability,
- to extract an Ocaml implementation of this type checker.
Information flow in practice

Information flow analysis is impossible without a minimum of precise information about potential exceptions that might be raised.

Two kind of complementary analysis are specially useful:

- Null pointer analysis
- Array bound analysis
Null pointer analysis

- We have defined a null pointer analysis that infer non-null field.
- It is based on the type system proposed by [Fahndrich&Leino, OOPSLA’03]
- The analysis is proved correct in Coq (for an idealized OO language)

L. Hubert. *A Non-Null annotation inferencer for Java bytecode*. PASTE'08.
- a tool has been developed on top of the previous work
- available at: http://nit.gforge.inria.fr
- efficient: around 2min for the 20.000 methods of Soot
- quite precise: 80% of the dereferences are proved safe