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

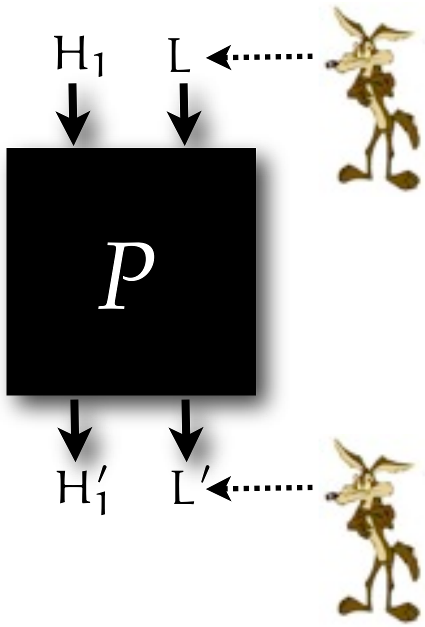
- We propose an information flow type system for a sequential JVM-like language, including classes, objects, arrays, exceptions and method calls.
- We prove in Coq that it guarantees the semantical non-interference property on method input/output.

Non-Interference

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

High = secret

Low = public

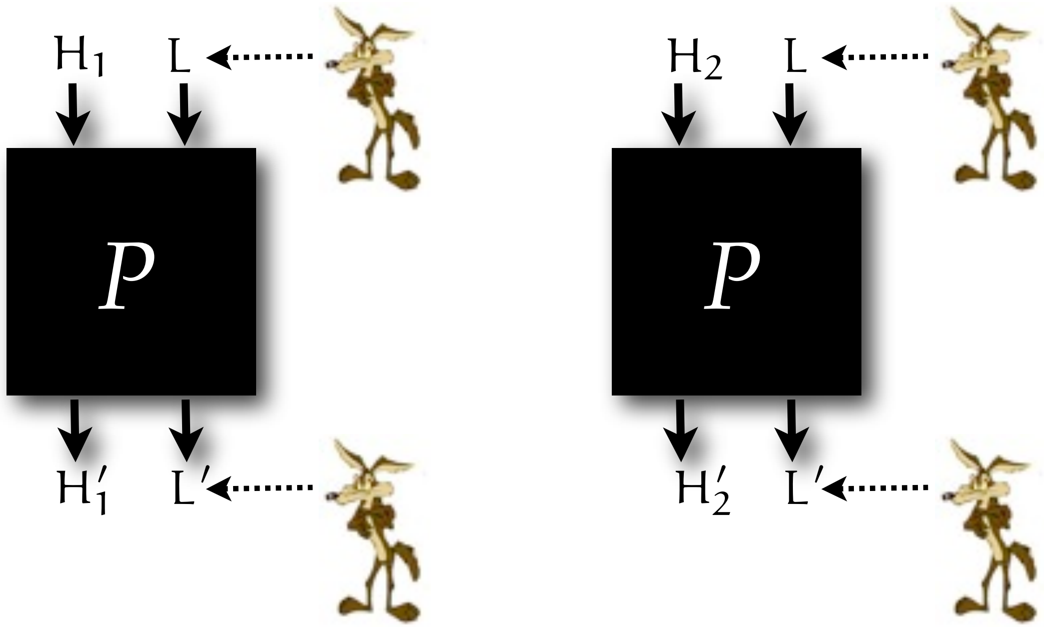


Non-Interference

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

High = secret

Low = public

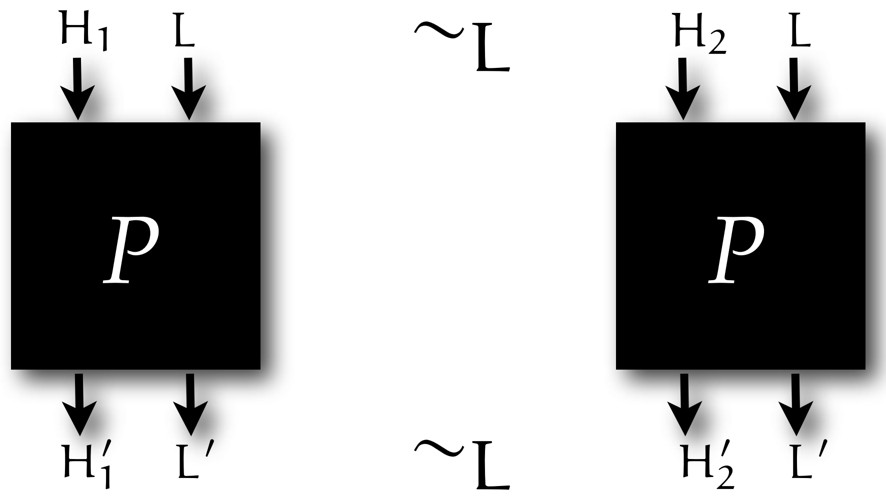


Non-Interference

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

High = secret

Low = public



$$\forall s_1\ s_2,\ s_1 \sim_L s_2 \implies [\![P]\!](s_1) \sim_L [\![P]\!](s_2)$$

Example of information leaks

Explicit flow:

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

Implicit flow:

```
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):

$$\vec{k_v} \xrightarrow{k_h} \vec{k_r}$$

  - $\vec{k_v}$ provides the security level of the method parameters (and local variables),
  - k_h is the effect of the method on the heap,
  - $\vec{k_r}$ 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

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

```
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,
- $\vec{k}_r[n]$= 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

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

$$\frac{\begin{array}{c} m[i] = \mathtt{putfield}\ f_k \\ k_1 \sqcup \mathrm{se}(i) \sqcup k_2 \leq k \qquad k_h \leq k \qquad k_2 \leq \vec{k_r}[\mathbf{np}] \\ \forall j \in \mathrm{region}(i, \emptyset) \cup \mathrm{region}(i, \mathbf{np}),\ k_2 \leq \mathrm{se}(j) \end{array}}{\Gamma, \mathrm{region}, \mathrm{se}, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, i \vdash k_1 :: k_2 :: \mathrm{st} \Rightarrow \mathrm{lift}_{k_2}\mathrm{st}}$$

Typing judgment

General form:

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

Example: `putfield` without handler for `NullPointer` exceptions

$$\frac{\begin{array}{c} m[i] = \mathtt{putfield}\ f_k \\ k_1 \sqcup \mathrm{se}(i) \sqcup k_2 \leq k \qquad k_h \leq k \qquad k_2 \leq \vec{k_r}[\mathbf{np}] \\ \forall j \in \mathrm{region}(i, \emptyset) \cup \mathrm{region}(i, \mathbf{np}),\ k_2 \leq \mathrm{se}(j) \end{array}}{\Gamma, \mathrm{region}, \mathrm{se}, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, i \vdash k_1 :: k_2 :: \mathrm{st} \Rightarrow \mathrm{lift}_{k_2} \mathrm{st}}$$

See the Coq development for 63 others typing rules...

The `putfield` rule on an example

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

```
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

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

```
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
```

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

The `putfield` rule on an example

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

```
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
```

$\mathrm{region}(2, \emptyset)$

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

The `putfield` rule on an example

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

```
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.

```
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
```

$\mathrm{region}(7, \emptyset)$

$\mathrm{region}(7, \mathbf{np})$

The `putfield` rule on an example

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

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

		se
1	load x	L
2	ifeq 5	L
3	new C	L
4	throw	L
5	load y	L
6	push 3	L
7	putfield f:H	L
8	push 1	H
9	return	H

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

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

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

		se
1	load x	L
2	ifeq 5	L
3	new C	L
4	throw	L
5	load y	L
6	push 3	L
7	putfield f:H	L
8	push 1	H
9	return	H

$$\frac{\begin{array}{c} m[i] = \texttt{putfield}\ f_k \\ k_1 \sqcup \mathrm{se}(i) \sqcup k_2 \leq k \qquad k_h \leq k \qquad k_2 \leq \vec{k_r}[\mathbf{np}] \\ \forall j \in \mathrm{region}(i, \emptyset) \cup \mathrm{region}(i, \mathbf{np}),\ k_2 \leq \mathrm{se}(j) \end{array}}{\Gamma, \mathrm{region}, \mathrm{se}, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, i \vdash k_1 :: k_2 :: \mathrm{st} \Rightarrow \mathrm{lift}_{k_2}\mathrm{st}}$$

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

L. Hubert, T. Jensen, and D. Pichardie. *Semantic foundations and inference of non-null annotations*. FMOODS'08.

- 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