Careful Analysis of Type Spoofing

Akihiko Tozawa
Masami Hagiya

University of Tokyo, Japan

Java Security(1)

- Type-Safe VM
  - ⇒ Safe Memory Management
    Guarantees that the JVM never coredumps.
- Class Libraries
  - ⇒ Access Control
    Security/Privilege Managers
  - ⇒ Network Security
    Encryption, etc.

Java Security(2)

- Requirement ↙What is protected?

  e.g. A Java program never coredumps.
  Applets never get access to local files.

- Specification

  Java Virtual Machine Specification(Lindholm & Yellin,1996)
  Inside Java 2 Platform Security(Gong,1999)

- Our Project

  ⇒ Does the spec really satisfy its requirement?

  ⇒ Formalization ↙Verification

Our Project

- "On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines" *SAS'98, Pisa, Italy, Proc.* pp.17-32, 1998
- "Fomalization and Solution of Saraswat's Problem of the Java Virtual Machine" In Japanese, 1999
- "New Formalization of the JVM", 1999
  http://nicosia.is.s.u-tokyo.ac.jp/members/miles/papers/cl-99.ps

Some Related Works

- Correctness of Bytecode Verification (Stata & Abadi,1998) (Qien, 1997)
- Are objects used only after initialization? (Mitchell & Freund, 1998)
- Are classes inside package scope invisible from outside? (Kojima & Maruyama, 1998)
- Is the JVM with class loaders type-safe? (Saraswat,1997)

Type-Spoofing Attack

- The problem

  A java program coredumps where there exist class loaders.

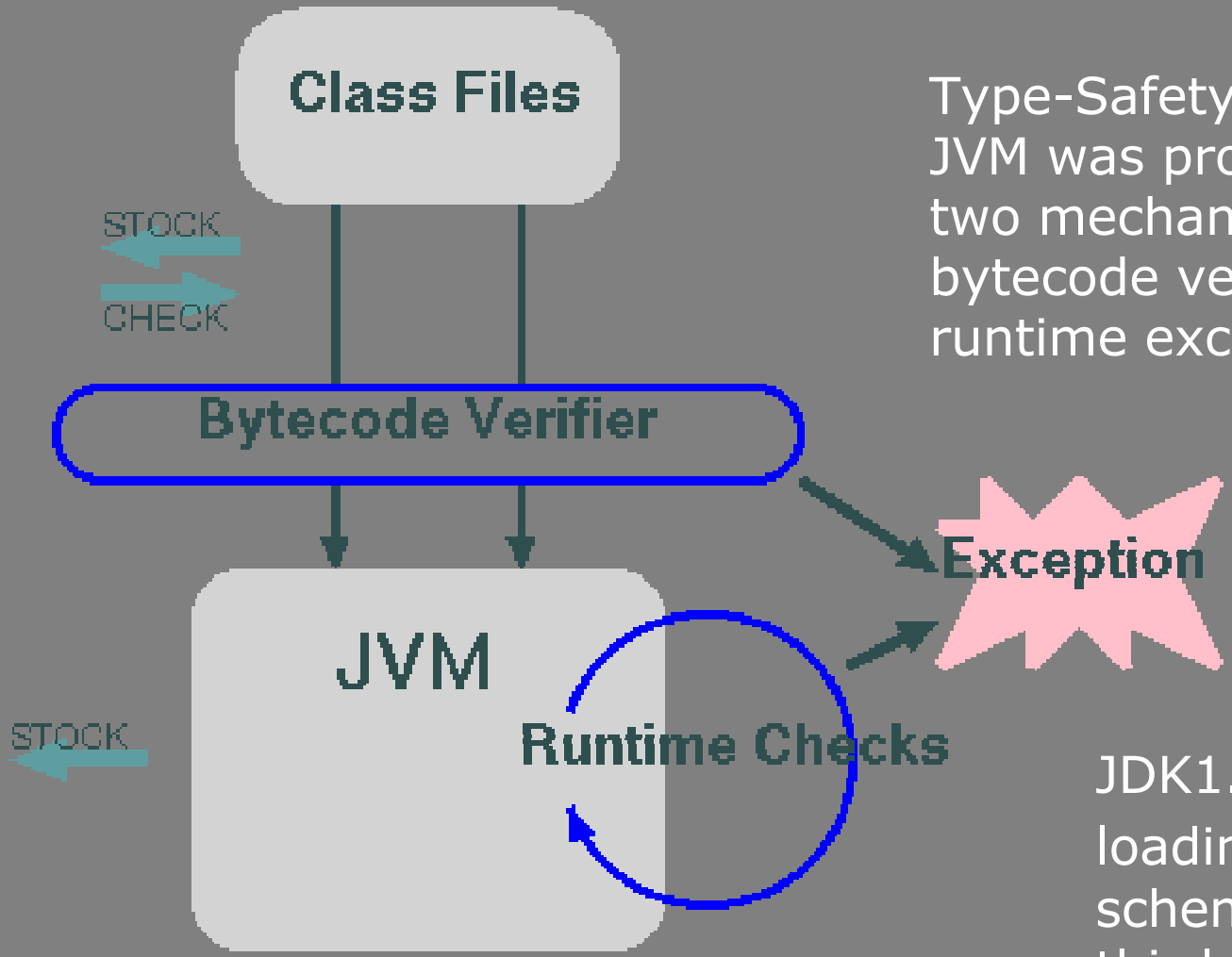
- Loading Constraint Scheme (Liang & Bracha, 1998)

  $\Rightarrow$ implemented in JDK1.2
  requirement: Preventing type spoofing.
  specification: In the 2nd edition of JVM spec.

- Careful Analysis of Type Spoofing

  $\Rightarrow$ The scheme is NOT trivial.

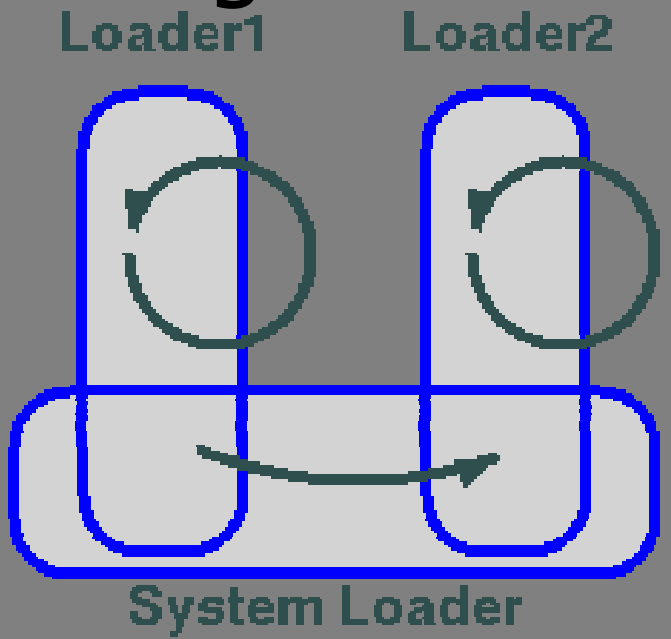
Loading Constraint Scheme(1)



Type-Safety of the JVM was protected by two mechanisms, bytecode verification & runtime exceptions.

JDK1.2 employs the loading constraint scheme  as the third mechanism.

Loading Constraint Scheme(2)



Intra–context execution

local execution

Inter–context execution

method invocation,
field access

A class loader

⇒ a map from class names to class objects

Two distinct classes with the same name

⇒ type spoofing attack

Loading Constraint Scheme(3)



The JVM judges the possibility of inter-context operation, and remembers it in the form of loading constraints.

$$E| -_{java.lang.XXX} Loader1 \sim Loader2$$



At any class loading, it is checked whether the loaded class does not violate any constraint.

Lazy class loading - Lazy checking!

Our Formal Model

Base model (Stata & Abadi, 1998)

- ✓ Bytecode Verifier

New Features

- ✓ Method invocation
- ✓ Lazy Class Loading
- ✓ Loading Constraints

Results

- Proof of the Soundness Theorem with our model.
- Discovery of two flaws in the bytecode verifier of JDK1.2.
- One more flaw with invokeinterface.
- Other results including the light-weight implementation of the loading constraint scheme.

⇒ cf. “New Formalization of the JVM” (our recent paper)

Soundness Theorem

- Lemma: Subsumption
  The implicit widening conversion preserves the well-typedness invariant.
- Lemma: Correctness of invokevirtual
  An invokevirtual correctly invokes an expected method.
- Lemma: Existence of Constraints
  At any invocation of a method, there have already been introduced the corresponding constraints.

Verifier's Problem(1)

Key definition of formalization!

nt

newly [illegible]rined.

$$E\left| -v ::^{+}_{cl} \; n \right. \overset{\text{def}}{\Leftrightarrow} \exists cl'.\, E\left| -v ::_{cl'} \; n \right. \wedge cl \sim cl'$$

$\Rightarrow$ Value v will have type n, when the JVM uses v.

- The implicit widening, an intra-context operation, sho[illegible]

Implicit Widening Relation

- Lemma: Subsumptio[illegible]

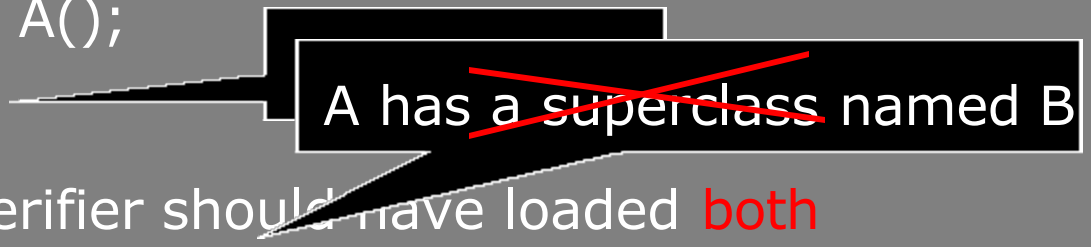
$$E\left| -v ::^{+}_{cl} \; n \wedge n \leq_{cl} n' \right.$$

$$\Rightarrow E\left| -v ::^{+}_{cl} \; n' \right.$$

Verifier's Problem(2)

- Implicit Widening Conversion

```
A a = new A();
B b = a;
```

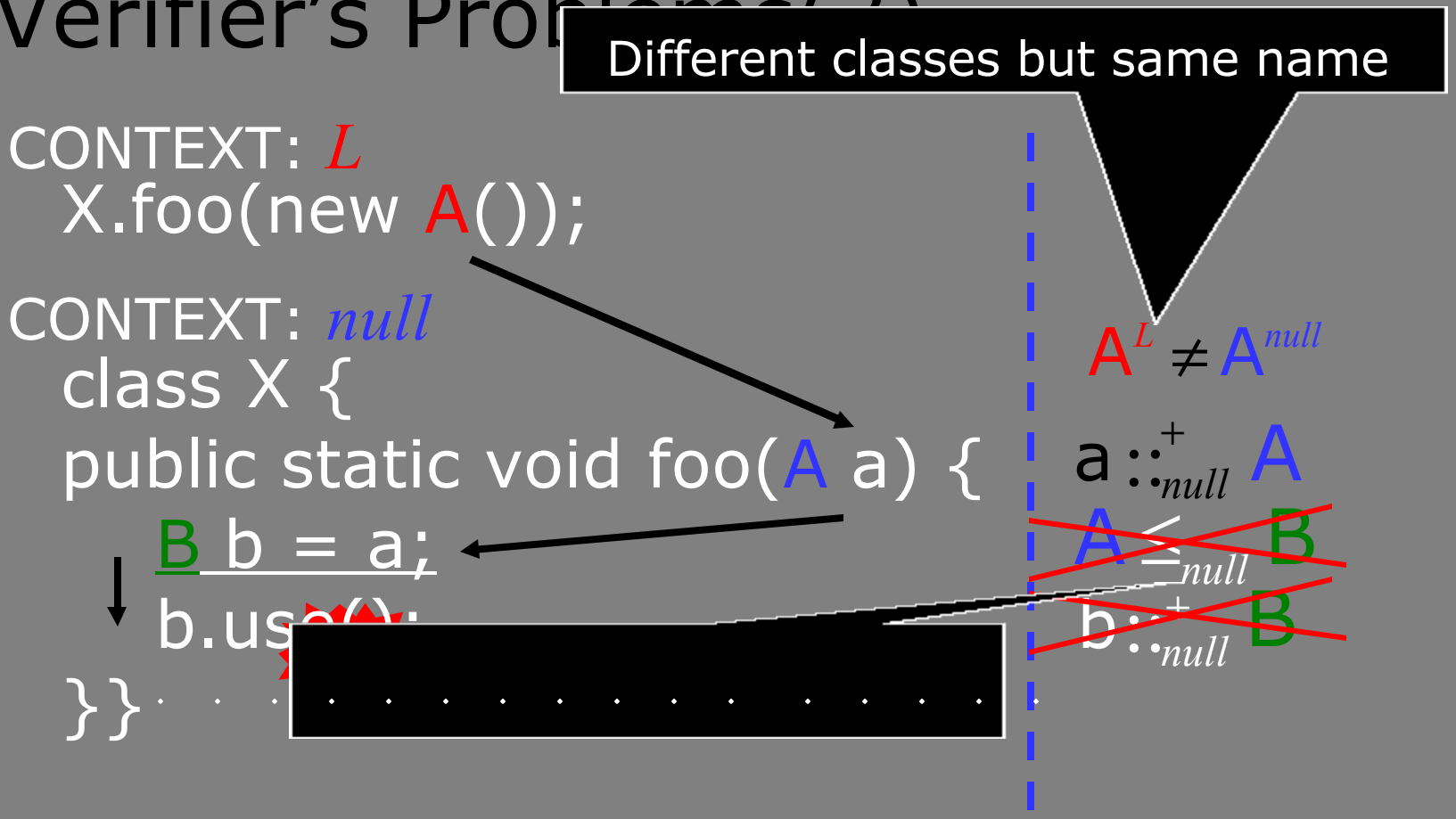


Bytecode verifier should have loaded both classes A and B to check if B is a superclass of A.

- Two problems of JDK1.2's verifier

⇒ The verifier does NOT always load both A and B.
⇒ It does not verify System Libraries.

Verifier's Problems(2)



The JVM cannot detect this erroneous use of value b (= value a)

Invokeinterface's Problem

```
class C {
    void foo(..) {
    …. }}
```

```
interface I {
    void foo(..);
}
```

```
class D extends C implements I {}
```

```
I i = new D();
i.foo(..);
```

⇒ With this invocation, there is possibly no constraint to check the value transfer across different loaders.

Summary

- Loopholes of the loading constraint scheme

  They will be fixed in the near future.

- Formalization ↙Verification

  Significance of theoretical approaches to security issues.