Careful Analysis of Type Spoofing

Akihiko Tozawa Masami Hagiya

Department of Information Science,
Graduate School of Science,
University of Tokyo, Japan
{miles, hagiya}@is.s.u-tokyo.ac.jp

Abstract. Saraswat's type spoofing was one of the most long-lasting bugs in the JVM. Recently, its solution was proposed and implemented in JDK1.2. The correctness of this new feature, however, is non-trivial and required the formal soundness proof. Actually, during our work on it, two flaws inside the new JVM implementation were found. This paper briefly reports our work and results.

1 Introduction

The most popular but most serious attack to the Java security is the so-called type confusion or type spoofing. The attack destroys the fundamental type system of the JVM, and can modify any authorized system property. To implement a reliable system against these kinds of attacks, type theoretical approach as in our recent work [1] is required. This report briefly summarizes our achievements in the work.

Last year, a new version of Sun's official Java Development Kit, JDK1.2, was released. With respect to the type spoofing attack, one of its variations originally reported by Saraswat [3] was prevented. His problem had been remaining long time unfixed, since its root existed deeply in the basic design of class loaders. The new loader constraint scheme introduced by Sieng and Bracha [4] fixed the problem by imposing the minimum runtime checks on the JVM.

We also studied these type spoofing issues in our formal approach and discovered the same fix independently from Sieng and Bracha. We also worked on the soundness proof of our formal model of the JVM. This is the first achievement in our work [1]. We proved the soundness of our formalization including the new loader constraint scheme, i.e., we proved that any code never violates its type system.

However, JDK1.2, in particular its bytecode verification, slightly differs from our idealized model. There are at least two inconsistencies of JDK1.2 against ours, which are small and easily overlooked, but during the proof work we discovered that each of these was really a serious flaw. Our second achievement is the discovery of the new flaws in JDK1.2, which could only be found through the proof work. In this short report, we mainly focus on these new flaws, and discuss some issues in our formalization that are crucial to these flaws.

In the next section, we briefly explain Saraswat's problem and its solution by the loader constraint scheme. In Section 3, we introduce the requirements for the solution to be work properly, which are based on our formalization and proof work. In Section 4, the new flaws are described and explained in the light of the requirements.

2 Problem and Solution

This section briefly explain Saraswat's problem and its solution by the loader constraint scheme.

2.1 Saraswat's Type Spoofing

We use the term *context* instead of a class loader. It is well-known that each class c has its own class loader c.cl, which performs any class resolution related to c. Each class c is said to belong to its own context c.cl. In practice, each method m also has its own context m.cl, since it belongs to a certain class that defines it.

```
class RT { ....
    RR rr = new RR();
    R v = rr.getR();
    v.speakUp();
.... }
```

Look at the above erroneous code, originally written by Saraswat himself. The code by itself does nothing. But suppose there are two distinct loaders, one of which is the current context *cl* of the processed code. The other loader is `getR`.cl, which is the same context as that of its definer class `RR`. What happens is the following.

- At first, the JVM invokes `rr.getR()`, and returns a certain value `v` that has its type `R` brought from `getR`.cl.
- The JVM then invokes `v.speakUp()`, assuming value `v` has type `R` in context *cl*. Since in fact it was locally typed in context `getR`.cl, the JVM coredumps.

Note that each context has its own map *associating* static class names to dynamic class objects. Therefore, two distinct contexts — the current context *cl* and the method's context `getR`.cl — might associate the same name `R` to different dynamic classes.

This type spoofing was an essential flaw in JDK1.1, since, in general, the JVM can invoke any method in an arbitrary context from another absolutely unrelated context.

2.2 Loader Constraint Scheme

Sieng and Bracha introduced the loader constraint scheme into JDK1.2 in order to solve Saraswat's problem.

A loader constraint is of the form, $E \vdash cl \overset{\texttt{R}}{\sim} \texttt{getR.cl}$ where E denotes the current environment of the JVM.

This constraint requires,

- Nothing, if either cl or `getR.cl` has not yet associated `R` to a certain class, i.e., the class is not yet loaded.
- Otherwise, both of them should associate `R` to the same class.

In the above Saraswat's example, this requirement will be unsatisfied at the invocation of `speakUp()`. In this case, we say that the loading of `R` *violates* the loader constraint. It should be noted that the above constraint does not force any class loading.

In our work [1], loader constraints are defined as follows.

- For any classes c, c' where c' is a parent class of c, and any class name n appearing in descriptors of some method of c', the following constraint is introduced.
$$E \vdash c.\mathsf{cl} \overset{n}{\sim} c'.\mathsf{cl}$$
- For any occurrence of invokevirtual in class c, if its method is resolved in class sym(symbolically referenced class[2]), then for any class name n appearing in the descriptor of invokevirtual, the following constraint is introduced.
$$E \vdash c.\mathsf{cl} \overset{n}{\sim} sym.\mathsf{cl}$$
- Relation $\overset{n}{\sim}$ is defined as the smallest transitive and reflexive relation satisfying the above conditions.

JDK1.2 allows neither any class loading nor any method resolution that violates loader constraints. Therefore, a loader constraint, as its name suggests, constrains class loadings. In the next section, it will be shown that the relation also has another meaning, i.e., a record keeping track of flow of values across contexts.

3 Requirements

The largest difficulty of the JVM lies in its mixture of two distinct features — the bytecode verification and the dynamic class loading. The former concerns with static typing, while the latter with runtime typing. In practice, once the JVM verifies the correctness of static typing in its bytecodes to be processed, it generally performs no more runtime checks on the dynamic types of internal values.

It is generally believed that static typing is enough to insure any runtime well-typedness. This is also what our soundness theorem states. More precisely:

– Any internal execution of the JVM preserves the invariant called *dynamic conformity.*

This invariant, defined in the next section, means that a runtime value is consistent with its static typing.

3.1 Dynamic Conformity

The dynamic conformity, denoted as $E \vdash v ::^{+}_{cl} n$, means that value v belongs to class n in context cl. It is formally defined as follows.

– There is another context cl', in which v statically typed with n (instantiated from n exactly in cl'), and the following constraint has been introduced.

$$E \vdash cl \overset{n}{\sim} cl'$$

3.2 Two Lemmas

Below are the descriptions of the two lemmas required to derive the soundness theorem. Each of these lemmas concerns with a method invocation of the JVM.

– The lemma, **Correctness of** invokevirtual, states that if the JVM is in a safe state, any method invocation peacefully succeeds.
– The lemma, **Existence of constraints**, states that if the JVM is in a safe state and about to invoke method m, there have already been introduced loader constraints between the current context cl and the invoked method's context $m.\mathsf{cl}$

$$E \vdash cl \overset{n}{\sim} m.\mathsf{cl}$$

for each class name n appearing in the descriptor of the method.

The first lemma will not be described in detail here. It is derived from the requirement on *method table compatibility* of the JVM and guarantees the safety of any next method invocation.

The second lemma implies that the JVM will be still in a safe state after the invocation of a method. As we noted, the root of Saraswat's bug lies in the fact that JDK1.1 does not relate the current context cl to $m.\mathsf{cl}$ at the invocation of m. The lemma states that in JDK1.2, for any invocation, there have already been introduced such inter-context relations, namely, loader constraints.

Carefully examine the lemma. It will be understood that any value transferred across contexts by a method invocation satisfies the dynamic conformity invariant, if it is initially satisfied before the invocation.

3.3 Theorem (Subsumption)

For the proof of above lemmas, and also for the safety of any flow of values in a single context, the next theorem is very important.

– If the *implicit widening* relation between two names, $n \leq_{cl} n'$, holds in context cl, then,

$$\forall v.\ E \vdash v ::^{+}_{cl} n \implies v ::^{+}_{cl} n'.$$

The implicit widening $n \leq_{cl} n'$ means that either n is equal to n' as names, or the loading of n with cl results in a subtype of the loaded class of n' with cl. In particular, the proof of the theorem necessarily requires the above loading of n. The bytecode verification of the JVM is responsible for this loading, whose absence leads to the flaw in 4.2.

Just as we regard the last lemma as what, for inter-context flow of values, preserves the invariant, the subsumption theorem preserves it for intra-context flow. They are complementary to each other and either of them cannot be ignored. The next section describes what will happen if our requirement, in particular, this theorem is ignored.

4 New Flaws

During the proof work, we examined the implementation of JDK1.2 with respect to our idealized model, and found the following new flaws.

4.1 Type-Spoofing without invokevirtual

JDK1.2 has not implemented its bytecode verification, as our relation $n \leq_{cl} n'$ specifies.

```
public class D {
  static boolean t = true;
  public D() {
    A a;
    if (t) a = new B(); else a = new C();
    a.speakUp();
  }
}
```

Look at the above code declaring a class with single method `D`. When our model verifies the method, it will check the following widenings.

$$E \vdash \texttt{B} \leq_{cl} \texttt{A} \ \wedge\ \texttt{C} \leq_{cl} \texttt{A}$$

Remember that the first widening requires both class names `B` and `A` to be loaded. The verifier of JDK1.2, however, does not seem to load class name `A` in this case (though the JVM Specification [2] requires to do so). It merely checks that classes loaded for `B` and `C` have their least common superclass with its name `A`. Therefore, the value newly created by `new B()` is possibly not typed in context cl. If so, what results is the same as Saraswat's example, i.e., the JVM either coredumps or exposes a serious security hole[1].

[1] Gilad Bracha, the author of [4], agreed that this is a bug and promised that it will be fixed in future JDK releases.

4.2 System Class Verification

The above bug is not actually a variation of Saraswat's bug, because it has nothing to do with invokevirtual, which transfers values, and the JVM cannot utilize constraints. However, we have found another bug, considered to be a variation of Saraswat's bug. This bug is more subtle, but similarly (or even more, for its subtlety) important. It escapes loader constraints where they exist.

In practice, JDK1.2 does not verify its system classes. This obviously contradicts with our model, and we actually found some examples where this contradiction leads to bugs. The following code conceptually defines the system classes exloitable by the type-spoofing. There are several actual classes in the system library (Sun's JDK1.2.1), which have the similar functions and allow us to write the bug examples.

```
package java.lang;
public class A {
  public java.lang.C foo(java.lang.B x) { return x; }
}
```

Consider an invocation of method `foo` in system class `java.lang.A`, in which `java.lang.C` is assumed to be the parent class of `java.lang.B`. Just as in the last example, the method's verification had required `java.lang.B` to be loaded in the null context, i.e., by the bootstrapping loader. Suppose that this verification and therefore the loading of `java.lang.B` have not occurred here.

To invoke method `foo` from another context cl and with its argument v locally typed in cl, the following constraint should be checked.

$$E \vdash cl \overset{\texttt{java.lang.B}}{\sim} \mathsf{null}$$

Whether it is violated or not is unknown at this moment, since `java.lang.B` is not loaded inside context null. The constraint will be judged at its loading which may never occur, and regardless of it, method `foo` will be invoked. Successively the method will return v as if it were of another type `java.lang.C` though v is not actually of that type.

In our model verifying any method, the above invocation with wrong argument v will immediately violate constraints.

5 Conclusion

If it is not implemented merely to check accidental mistakes, the loader constraint scheme should perfectly remove any possibility, including incidental attacks, of the type-spoofing by untrusted loaders. This means that its implementation hardly makes significance until any flaws are fixed. Furthermore, it is generally difficult to trust class loaders (*cf.* our work[1]), particularly, those in various recent applications of class loaders. All these problems suggest the requirement of the perfect scheme. We hope that our work would be its basis.

With respect to the formalization of th JVM, there have been many related work.

– Its bytecode verification and operational semantics are discussed[5][6][8][11].
– Its dynamic class loading is discussed[7][8][9].
– Its object initialization is discussed in detail[10].

Our formalization is built on these previous discussions.

6 Acknowledgment

We would like to thank Gilad Bracha, the author of the OOPSLA'98 paper[4], for giving us insightful comments and suggestions from the designer's viewpoint.

References

1. A. Tozawa and M.Hagiya, New Formalization of the JVM, *in preparation. Draft available from*
`http://nicosia.is.s.u-tokyo.ac.jp/members/miles/papers/cl-99.ps`
2. T.Lindholm and F.Yellin, *The Java Virtual Machine Specification*, The Java Series, Addison-Wesley Longman, 1996.
`http://java.sun.com/docs/books/vmspec`
3. V. Saraswat, Java is not type-safe, 1997.
`http://www.research.att.com/~vj/bug.html`
4. Sheng Liang and Gilad Bracha, Dynamic Class Loading in the Java Virtual Machine, *OOPSLA'98, proc.*, pp.36-43, 1998.
5. M. Hagiya and A. Tozawa, On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines, *SAS'98, Proc.*, LNCS 1503, Springer-Verlag, pp.17-32, 1998.
`ftp://nicosia.is.s.u-tokyo.ac.jp/pub/staff/hagiya/pro98/jvm-pro.ps`
6. R.Stata and M.Abadi, A Type System for Java Bytecode Subroutines, *POPL'98, proc.*, pp.149-160, 1998
7. D.Dean, The Security of Static Typing with Dynamic Linking, *4th Conference on Computer and Communications Security, ACM, proc.*, 1997
8. A.Goldberg, A specification of Java Loading and Bytecode Verification, 1998
`http://www.kestrel.edu/HTML/people/goldberg/index.html`
9. T. Jensen, D. Le Metayer and T. Thorn, Security and Dynamic Class Loading in Java: A Formalisation, *Proceedings of IEEE International Conference on Computer Languages*, pp.4-15, 1998
10. S.Freund and J.C.Mitchell, A Type System for Object Initialization in the Java Bytecode Language, *ACM Symp. OOPSLA'98, Proceedings*, pp.310-327, 1998
11. Z.Qian, A Formal Specification of Java Virtual Machine Instruction, 1997
`http://www.infomatik.uni-bremen.de/~gian/abs-fsjvm.html`