CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2001; **???**:1–21 *Prepared using* cpeauth.cls *[Version: 2000/05/12 v2.0]*

Type safety in the JVM: some problems in Java 2 SDK 1.2 and proposed solutions

Alessandro Coglio[1,*,†] and Allen Goldberg[2,‡]

[1] *Kestrel Institute*[§], *3260 Hillview Avenue, Palo Alto, CA 94304, USA*
[2] *Shoulders Corporation, 800 El Camino Real, Mountain View, CA 94040, USA*

SUMMARY

In the course of our work in developing formal specifications for components of the JVM, we have uncovered subtle bugs in the bytecode verifier of Sun's Java 2 SDK 1.2. These bugs, which lead to type safety violations, relate to the naming of reference types. Under certain circumstances, these names can be spoofed through delegating class loaders. These flaws expose some inaccuracies and ambiguities in the JVM specification.

We propose several solutions to all of these bugs. In particular, we propose a general solution that makes use of subtype loading constraints. Such constraints complement the equality loading constraints introduced in the Java 2 Platform, and are posted by the bytecode verifier when checking assignment compatibility of class types. By posting constraints instead of resolving and loading classes, the bytecode verifier in our solution has a cleaner interface with the rest of the JVM, and allows lazier loading. We sketch some excerpts of our mathematical formalization of this approach and of its type safety results.

KEY WORDS: Java; JVM; type safety; bugs

1. INTRODUCTION

We are currently developing mathematical specifications for critical components of the JVM, including the bytecode verifier [1, 3, 7, 8] and the class loading mechanisms [9]. A major goal of such efforts is to formally analyze the JVM in order to increase confidence in its correctness.

*Correspondence to: Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304, USA
†E-mail: `coglio@kestrel.edu`
‡E-mail: `agoldberg@shoulderscorp.com`
§URL: `http://www.kestrel.edu`

Received ???
Revised ???
Accepted ???

That involves verifying that the existing mechanisms exhibit desired properties, or identifying flaws and proposing fixes. Another major goal of ours is to derive implementations of some JVM components using Specware [11], a system developed at Kestrel Institute, that supports provably correct, compositional development of software from formal specifications through refinement. We have used Specware to develop a complete bytecode verifier. These formally-derived components can serve, among other uses, as high-assurance reference implementations against which hand-written implementations can be tested (following the approach of the Kimera Project [4]).

In the course of our formalization efforts, we have uncovered subtle bugs in Sun's Java 2 SDK 1.2 [12] that lead to type safety violations. These bugs are in the bytecode verifier and relate to the naming of reference types. We found that in certain circumstances these names can be spoofed by suitable use of delegating class loaders. Since the JVM specification [6] is written in informal English prose, we cannot crisply characterize these bugs as errors in the specification or errors in one or more implementations. However, some of these bugs are consistent with a reasonable interpretation of the specification. We have verified that the bugs exist in SDK 1.2, on both Solaris and Windows NT. Some are fixed in SDK 1.3 [13] in an "indirect" way, by restricting access to system packages. We have also verified these bugs in Symantec's Java version 1.2.2.Symc. We propose several fixes for all the bugs, including a more general approach that has additional advantages, such as lazier class loading and cleaner interface between the bytecode verifier and the rest of the JVM.

This paper shows that formal studies can help find and fix bugs in real-world, existing systems. Our experience, in this and other projects, is that flaws, inadequacies, etc. are often uncovered during the formalization process itself, before attempting to verify any property. The reason is that constructing mathematical models encourages a deeper understanding of the entities under consideration, enabling the detection of problems. Attempting to state and prove desired properties of the constructed mathematical models often unveils further problems and subtleties.

The remainder of this paper is organized as follows. The next section provides background about the JVM, in particular class loading and bytecode verification. Section 3 describes the bugs we found, including the source code that exhibits them. Section 4 proposes solutions to the bugs. Section 5 presents related work, while Section 6 concludes.

2. BACKGROUND

The JVM supports dynamic, lazy loading of classes. Lazy loading is desirable because it improves the response time of an applet or application and reduces memory usage. In fact, execution can start after loading just a few classes. The other classes are loaded on demand if and when they are needed.

Classes in the JVM are loaded by means of *class loaders*. A class loader is an instance of the abstract class `java.lang.ClassLoader`, which can be subclassed to implement arbitrary loading policies. The JVM also includes a built-in *system class loader*, used to start up the machine and to load system classes.

Each class has a *fully qualified name* (*FQN*), constituted by a package name followed by a simple name (e.g., `java.util.HashSet`). A class references other classes (e.g., to invoke methods) symbolically, using their FQNs. Eventually, symbolic references are *resolved* to actual classes, i.e., instances of class `java.lang.Class`. When the JVM needs to resolve a FQN `N`, it invokes the `loadClass` methods of `java.lang.ClassLoader` upon a class loader L (chosen as explained later), with `N` as argument. If the method does not throw an exception it returns a class C, the result of resolution. L is called an *initiating loader* of class C. The JVM maintains a *loaded class cache* recording initiating loaders for all the classes loaded by the machine. The cache is looked up each time the JVM needs to resolve a FQN `N` through an initiating loader L; if the cache records a class C with FQN `N` and initiating loader L, then C is the result of resolution. Otherwise, `loadClass` is invoked and the cache updated, as described above. This enforces the constraint that resolving a FQN `N` through a loader L always results in the same class C. Therefore, as in [6] and [5], we use the notation $\mathtt{N}^L$ to denote the unique class with FQN `N` and initiating loader L.

Loading a class into the JVM consists of two steps: (1) fetching a byte array that represents the class in `classfile` format; and (2) creating an instance of `java.lang.Class` from the byte array. The second step can only be carried out by calling the `defineClass` method of `java.lang.ClassLoader` upon a class loader L. This method calls internal JVM code that checks the format of the byte array and creates the proper internal representation for the class, returning a (new) class C (if an exception is not thrown). L is called a *defining loader* of class C. The internal JVM code called by `defineClass` enforces the constraint that a loader L cannot be used to create two classes with the same FQN. Therefore, as in [6] and [5], we use the notation $\langle \mathtt{N}, L\rangle$ to denote the unique class with FQN `N` and defining loader L. Note that, while $\mathtt{N}^L$ and $\mathtt{N}^{L'}$, with $L \neq L'$, may denote the same class, $\langle \mathtt{N}, L\rangle$ and $\langle \mathtt{N}, L'\rangle$ always denote different classes (created by different invocations of `defineClass`).

The process of fetching a byte array in `classfile` format is carried out by the code in `loadClass`. By overriding this method, arbitrary loading policies can be realized, including fetching byte arrays over network connections, caching them, and even constructing or instrumenting them on the fly. Eventually, `loadClass` may invoke `defineClass`, in which case the resulting class has the same loader as both initiating and defining loader. Alternatively, the code in `loadClass` may *delegate* the loading of some FQN to some other loader, e.g., the system class loader. In this case, the initiating and defining loader differ.

When a FQN referenced in a class C needs to be resolved, the defining loader of C is chosen as an initiating loader for the class to be resolved.

The bytecode verifier is in charge of checking, prior to executing the code in a class, that the code satisfies certain properties. These properties, combined with certain run-time checks, are intended to guarantee that the code is type-safe. Bytecode instructions operate on an operand stack and some local variables. The bytecode verifier attempts to assign, via a data flow analysis, types to local variables and stack positions, for each instruction position in the code. The type assignment must be consistent with the types required by the instructions and with the operations they perform.

As explained above, a class in the JVM is identified by a FQN plus a defining loader. In particular, there can be distinct classes with the same FQN. However, the bytecode verifier, when verifying a class, essentially uses just FQNs. In a few cases, it resolves names and makes use of the `java.lang.Class` instances they resolve to. In particular, the bytecode verifier sometimes must *merge*, that is find the first common superclass of, two class names. The two class names are resolved, thus loading the classes and their superclasses, and their ancestry searched to find their first common superclass. The bytecode verifier also resolves names to check assignment compatibility (i.e., subtype relationship) between two class names.

The use of FQNs and occasional use of actual classes guarantee type safety only under certain assumptions. Examples of these assumptions are the loading constraints introduced in the Java 2 Platform [6, 5] to avoid the type safety problems arising precisely because of the violation of the assumptions they enforce [10]. Simply stated, loading constraints ensure that classes exchanging objects (through their methods and fields) agree on the actual types (i.e., not only on the FQNs but also the loaders) of such objects.

As it turns out, the loading constraints introduced in [6, 5] do not cover all the assumptions needed to guarantee type safety. For example, when checking a stack position that contains the type (FQN) that results from merging two classes, the bytecode verifier assumes that the FQN in the stack position resolves to the actual first common superclass. Another example occurs when checking type constraints for the `invokespecial` instruction: the bytecode verifier assumes that a FQN of any superclass of the current class resolves to the actual superclass. Furthermore the bytecode verifier assumes that the FQNs `java.lang.Object` and `java.lang.String` resolve to the "usual" system classes. We will now show how it is possible to construct programs where these assumptions are violated, thus causing name spoofing and type safety failures.

3. DESCRIPTION OF THE BUGS

3.1. Spoofing the first common superclass

In [6, Sect. 4.9.2] it is described how, during data flow analysis of a method's code, the types assigned to stack positions and local variables along different control paths are merged. In order to merge two distinct class names `Sub1` and `Sub2`, the corresponding classes are loaded by invoking the `loadClass` method of the defining loader L of the class whose method is being verified, with argument `Sub1` first, then `Sub2`. Loading a class recursively causes all of its superclasses to be loaded, if they have not already been loaded. The ancestry of the classes $\texttt{Sub1}^L$ and $\texttt{Sub2}^L$ is then searched to find the first common superclass. If the first common superclass found is $\langle \texttt{Sup}, L_0 \rangle$ then the bytecode verifier writes the FQN `Sup` in the merged stack position. Suppose that, after the merging point, an instruction accesses a field or method of a class named `Sup` in a field or method reference. Since the bytecode verifier has deduced that the stack position indeed contains a class with FQN `Sup`, the check for assignment compatibility will succeed, as described in [6, Sect. 4.9.1]. This is correct only assuming that $\texttt{Sup}^L = \langle \texttt{Sup}, L_0 \rangle$, i.e., that loading of FQN `Sup` initiated by L results in the actual superclass of $\texttt{Sub1}^L$ and $\texttt{Sub2}^L$.

However, such an assumption can be violated, as shown in the code in Figure 1. In the code listings in this paper, when there is more than one class with the same FQN, we indicate their defining loaders to disambiguate them.

```
public class <Sup,L0> {
    public float f = 123.456f;
}

public class Sub1 extends Sup {}

public class Sub2 extends Sup {}

public class Merger {
    static public void merge() {
        Sub1 s1 = new Sub1();
        Sub2 s2 = new Sub2();
        Sup s;
        if (s1 != null)  s = s1;  // this test just serves to
        else  s = s2;             // create two merging paths
        int i = s.f;
        System.out.println("Field is " + i);
    }
}

public class <Sup,L> {
    public int f = 1;
}

public class Main {
    public static void main(String argv[]) {
        String[] undelNames = {"Merger", "Sup"};
        Starter.go(undelNames, "Merger", "merge");
    }
}
```

Figure 1. Program to spoof the first common superclass

The code for `Starter` is in the Appendix. Its method `go` creates an instance L of class `DelegatingLoader` (also in the Appendix). A loader belonging to this class delegates the loading of certain FQNs to the system class loader. The system class loader loads classes from the current directory. Classes not delegated to the system are loaded from a subdirectory. This delegating loader is a simple mechanism to load distinct classes with the same FQN, and thus set up a circumstance where classes can be spoofed. The name of the subdirectory and the array of undelegated FQNs (i.e., those loaded by the loader itself) are set when the loader is constructed. The array of undelegated FQNs is passed as the first argument of `go`. Class

$$\begin{tikzcd}
& \langle \texttt{Sup}, L_0 \rangle & \\
\langle \texttt{Sub1}, L_0 \rangle \arrow[ur, Rightarrow] & \langle \texttt{Merger}, L \rangle \arrow[l] \arrow[r] \arrow[d] & \langle \texttt{Sub2}, L_0 \rangle \arrow[ul, Rightarrow] \\
& \langle \texttt{Sup}, L \rangle &
\end{tikzcd}$$

Figure 2. Situation arranged by the program in Figure 1

`Starter` is used in the following examples as well, with its `go` method called with different arguments.

In this example, L loads `Merger` and `Sup` from `dir`. The situation depicted in Figure 2 is thus arranged. L_0 is the system class loader. An arrow from a class (identified by its name and defining loader) to another indicates that the source of the arrow resolves the FQN of the target class to the target class. A double arrow indicates that the source is a direct subclass of the target. This graphical representation makes it easy to understand the relationship among classes and the delegation paths.

The code in class `Starter`, after creating L, uses it to load `Merger` and then uses the reflection APIs to invoke the `merge` method. The name of the class and the name of the method are passed as the second and third argument of `go`. The use of reflection is necessary here because class `Merger` has been loaded by a user-defined class loader, thus it is "accessible" as an instance of class `java.lang.Class` and not through a textual reference in the program.

The `merge` method creates two instances of $\langle \texttt{Sub1}, L_0 \rangle$ and $\langle \texttt{Sub2}, L_0 \rangle$, whose loading is initiated by L but carried out by L_0. When the bytecode verifier verifies this method, it finds their first common superclass $\langle \texttt{Sup}, L_0 \rangle$. The integer field access to class `Sup` passes bytecode verification because `Sup` is the name of the first common superclass. However, when `merge` is executed, `Sup` resolves to $\langle \texttt{Sup}, L \rangle$, which is different from $\langle \texttt{Sup}, L_0 \rangle$.

On SDK 1.2, we obtain the output shown in Figure 3. Apparently, the implementation chooses the same field layout for classes $\langle \texttt{Sup}, L \rangle$ and $\langle \texttt{Sup}, L_0 \rangle$. Therefore, the 32-bit representation of the single-precision floating point value 123.456 is accessed as the 32-bit representation of the integer value 1,123,477,881. This is shown in the last line of output, that is type safety has been violated: a floating point value has been accessed as an integer value. The other lines of output show the loading requests received by L, and whether they are delegated to L_0 or handled by L itself.

3.2. Spoofing a superclass using `invokespecial`

One of the uses of the bytecode instruction `invokespecial` is to invoke superclass methods, e.g., as the result of compiling Java code such as `super.m(x,y)`. The difference between

```
[Loaded java.lang.Object from system]
[Loaded Merger from dir/Merger.class (645 bytes)]
[Loaded java.lang.Throwable from system]
[Loaded Sub1 from system]
[Loaded Sub2 from system]
[Loaded Sup from dir/Sup.class (219 bytes)]
[Loaded java.lang.System from system]
[Loaded java.lang.StringBuffer from system]
[Loaded java.io.PrintStream from system]
Field is 1123477881
```

Figure 3. Output of the program in Figure 1

`invokevirtual`, which is the "normal" instruction for method invocation, and `invokespecial` is that `invokespecial` uses a modified dynamic dispatching strategy (see [6] for more information). When verifying an `invokespecial` instruction in a class $\langle \texttt{Sub}, L\rangle$ that references a method in a non-immediate superclass $\langle \texttt{SupSup}, L_0\rangle$ with the current class as the `this` reference, the bytecode verifier checks that the types (FQNs) assigned to stack positions conform to the descriptor of the method. In particular, it checks that name `Sub` (assigned to the `this` reference) is assignment-compatible with `SupSup`. Since $\langle \texttt{SupSup}, L_0\rangle$ is a superclass of $\langle \texttt{Sub}, L\rangle$, the bytecode verifier passes the check. This is correct only assuming that $\texttt{SupSup}^L = \langle \texttt{SupSup}, L_0\rangle$, i.e., that $\langle \texttt{Sub}, L\rangle$ resolves name `SupSup` to $\langle \texttt{SupSup}, L_0\rangle$.

This assumption can indeed be violated, as shown in the code in Figure 4. Class `Main` uses the same class `Starter` of the previous example to create a loader L of class `DelegatingLoader` that loads `Sub` and `SupSup` from `dir`, thus arranging the situation depicted in Figure 5.

The actual superclasses of $\langle \texttt{Sub}, L\rangle$ are $\langle \texttt{Sup}, L_0\rangle$ and $\langle \texttt{SupSup}, L_0\rangle$. However, resolving `SupSup` directly from $\langle \texttt{Sub}, L\rangle$ results in $\langle \texttt{SupSup}, L\rangle$, which is different from $\langle \texttt{SupSup}, L_0\rangle$. Note that the "intermediate" superclass $\langle \texttt{Sup}, L_0\rangle$ is needed to change loader from L to L_0 along the path. When executing the `invokespecial` instruction, the reference to `m` is resolved to method `m` in class $\langle \texttt{SupSup}, L\rangle$, and, as specified in [6, page 284], that method is invoked, because $\langle \texttt{SupSup}, L\rangle$ is not a superclass of $\langle \texttt{Sub}, L\rangle$. On SDK 1.2, we obtain the output shown in Figure 6, demonstrating a type safety violation.

3.3. Spoofing `java.lang.Object`

As previously mentioned, the bytecode verifier loads classes to check assignment compatibility between reference types with different names. However, if the name of the assignment target is `java.lang.Object` then the verifier assumes the assignment is valid, since any class is assignable to `java.lang.Object`. This is correct only assuming that the FQN `java.lang.Object` resolves to the "usual" system class $\langle \texttt{java.lang.Object}, L_0\rangle$, and not to some other class unrelated to the assignment source.

```
public class <SupSup,L0> {
    float f = 123.456f;
    public void m() {}
}

public class Sup extends SupSup {}

public class Sub extends Sup {
    static public void doInvSpec () {
        Sub s = new Sub();
        s.invSpec();
    }
    public void invSpec() {
        super.m();
    }
}

public class <SupSup,L> {
    int f = 1;
    public void m() {
        System.out.println("Field is " + f);
    }
}

public class Main {
    public static void main(String argv[]) {
        String[] undelNames = {"Sub", "SupSup"};
        Starter.go(undelNames, "Sub", "doInvSpec");
    }
}
```

Figure 4. Program to spoof a superclass using `invokespecial`

This assumption can be violated, as shown in the code in Figure 7. Class `Main` uses `Starter` to create a loader L of class `DelegatingLoader`, which loads `Sub` and `java.lang.Object` from `dir`, thus arranging the situation depicted in Figure 8.

When `assign` is executed, the newly created instance of `Sub` is assigned to $\langle$`java.lang.Object`$, L\rangle$, thus producing, on SDK 1.2, the output shown in Figure 9.

Even though $\langle$`java.lang.Object`$, L_0\rangle$ is an indirect superclass of $\langle$`java.lang.Object`$, L\rangle$, there is no circularity since they are distinct classes. The intermediate superclass `Sup` is needed because resolving `java.lang.Object` from $\langle$`java.lang.Object`$, L\rangle$ would result in itself.

$$\begin{tikzcd}
 & \langle \texttt{SupSup}, L_0 \rangle \\
 & \langle \texttt{Sup}, L_0 \rangle \arrow[u, Rightarrow] \\
\langle \texttt{SupSup}, L \rangle & \langle \texttt{Sub}, L \rangle \arrow[l] \arrow[u, Rightarrow]
\end{tikzcd}$$

Figure 5. Situation arranged by the program in Figure 4

```
[Loaded Sup from system]
[Loaded Sub from dir/Sub.class (336 bytes)]
[Loaded java.lang.Throwable from system]
[Loaded java.lang.Object from system]
[Loaded SupSup from dir/SupSup.class (574 bytes)]
[Loaded java.lang.System from system]
[Loaded java.lang.StringBuffer from system]
[Loaded java.io.PrintStream from system]
Field is 1123477881
```

Figure 6. Output of the program in Figure 4

3.4. Spoofing `java.lang.String`

The bytecode instruction `ldc` is used to load constants onto the operand stack. The instruction's operand is an index into the class's constant pool that references an integer, floating point, or string constant. In the case of a string constant, the symbolic reference is resolved to a reference to the unique instance of the system's `java.lang.String` class whose value is the string constant (the instance is unique because string constants from the constant pool are interned). During data flow analysis, the bytecode verifier infers a type with FQN `java.lang.String` for the top of the stack after execution of `ldc`. This is correct only if this FQN resolves to the "usual" system class $\langle \texttt{java.lang.String}, L_0 \rangle$, and not to some other unrelated class.

Again, this assumption can be violated, as shown in the code in Figure 10. Class `Main` uses class `Starter` to create a loader L of class `DelegatingLoader`, which loads `StrLoader` and `java.lang.String` from `dir`, thus arranging the situation depicted in Figure 11.

When `loadStr` is executed, the field access gathers some content of the `java.lang.String` object created for string `"abc"`. The output on SDK 1.2 is shown in Figure 12.

```
public class Sup {}

public class Sub extends Sup {
    public float f = 123.456f;
    static public void assign() {
        Sub s = new Sub();
        java.lang.Object o = s;
        int i = o.f;
        System.out.println("Field is " + i);
    }
}

public class <java.lang.Object,L> extends Sup {
    public int f = 1;
}

public class Main {
    public static void main(String argv[]) {
        String[] undelNames = {"Sub", "java.lang.Object"};
        Starter.go(undelNames, "Sub", "assign");
    }
}
```

Figure 7. Program to spoof `java.lang.Object`

$$\begin{tikzcd}
 & \langle \texttt{Sup}, L_0 \rangle \\
\langle \texttt{java.lang.Object}, L \rangle \arrow[ur, Rightarrow] & \langle \texttt{Sub}, L \rangle \arrow[u, Rightarrow] \arrow[l]
\end{tikzcd}$$

Figure 8. Situation arranged by the program in Figure 7

```
 [Loaded Sup from system]
 [Loaded Sub from dir/Sub.class (631 bytes)]
 [Loaded java.lang.Throwable from system]
 [Loaded java.lang.Object from dir/java/lang/Object.class (185 bytes)]
 [Loaded java.lang.System from system]
 [Loaded java.lang.StringBuffer from system]
 [Loaded java.io.PrintStream from system]
Field is 1123477881
```

Figure 9. Output of the program in Figure 7

```
public class StrLoader {
    public static void loadStr(){
        java.lang.String s = "abc";
        Printer.showSomeContent(s.f);
    }
}

public class <java.lang.String,L> {
    public int f = 1;
}

public class Printer {
    static public void showSomeContent (int i) {
        System.out.println("Some content is " + i);
    }
}

public class Main {
    public static void main(String argv[]) {
        String[] undelNames = {"StrLoader", "java.lang.String"};
        Starter.go(undelNames, "StrLoader", "loadStr");
    }
}
```

Figure 10. Program to spoof `java.lang.String`

$$\begin{tikzcd}\langle\texttt{java.lang.String}, L\rangle & \langle\texttt{StrLoader}, L\rangle \arrow[l] \arrow[r] & \langle\texttt{Printer}, L_0\rangle\end{tikzcd}$$

Figure 11. Situation arranged by the program in Figure 10

```
[Loaded java.lang.Object from system]
[Loaded StrLoader from dir/StrLoader.class (356 bytes)]
[Loaded java.lang.Throwable from system]
[Loaded java.lang.String from dir/java/lang/String.class (235 bytes)]
[Loaded Printer from system]
Some content is -339094920
```

Figure 12. Output of the program in Figure 10

Note that without the class `Printer` the JVM would detect a loading constraint violation involving the FQN `java.lang.String` when the method reference to `println` is resolved. However, loading constraints do not catch the type violation exhibited by the program above, because the string `"abc"` is not obtained from any other class's field or method, but internally from the constant pool.

4. PROPOSED SOLUTIONS

4.1. Preventing spoofing of the first common superclass

The bug that allows spoofing the first common superclass may be interpreted as a bug in the JVM specification, rather than the implementation. Although [6] does not crisply state that types are denoted by FQNs in the bytecode verifier (it typically just talks about "reference types"), that seems to be the intended meaning, or at least the most reasonable interpretation. In any case, future editions of [6] should clarify this point. This bug also exists in SDK 1.3.

A possible solution to the problem is to keep information, when merging two FQNs `Sub1` and `Sub2`, about the actual first common superclass $\langle\texttt{Sup}, L_0\rangle$ (not only its FQN `Sup`). So, when checking assignment compatibility there can be no confusion between the class that is the result of merging and the class $\texttt{Sup}^L$. Interestingly, inspection of the bytecode verifier code in SDK 1.2 shows that information about the actual first common superclass is indeed maintained and accessible. However, it is not used to prevent this problem.

An alternative solution that avoids early loading of `Sup` by L is to introduce a loading constraint $\texttt{Sup}^L = \texttt{Sup}^{L_0}$, added by the bytecode verifier to the set of globally maintained loading constraints. Indeed as we shall see all of these bugs can be avoided by using such constraints.

4.2. Preventing spoofing of a superclass using `invokespecial`

The spoofing by means of `invokespecial` is an implementation bug. According to [6], assignment compatibility should be checked by loading the actual classes and checking subtype relationship. If this were done, the bytecode verifier would detect the unsoundness of the invocation of method `m` of $\langle \texttt{SupSup}, L\rangle$ over an instance of $\langle \texttt{Sub}, L\rangle$ (which has $\langle \texttt{SupSup}, L_0\rangle$ as superclass). The obvious fix for the problem is to resolve $\texttt{SupSup}^L$ and use loaded classes, not FQNs. A better alternative is to generate a loading constraint between the loaded superclass and the reference to the superclass, namely $\texttt{SupSup}^L = \texttt{SupSup}^{L_0}$.

In SDK 1.3, running the example causes the JVM to terminate abnormally with a segmentation fault on Solaris, and with an internal error on Windows NT. This happens after the line about `SupSup` being loaded is printed on screen. The bytecode verifier in SDK 1.3 is essentially the same of SDK 1.2. The abnormal termination appears to be originated by some other code, probably some class loading code. In any case, this still exposes a bug in SDK 1.3. However, we have not fully investigated its source.

4.3. Preventing spoofing of `java.lang.Object`

Similar to the solution above, the spoofing of `java.lang.Object` can be avoided by resolving `java.lang.Object` and checking that it is a superclass of $\langle \texttt{Sub}, L\rangle$.

However, it is worth noting that [6] implies, more or less clearly, that `java.lang.Object` denotes the class that is the root of the class hierarchy. On the other hand, [6] describes no mechanisms (and no requirements) to ensure that this is the case. In SDK 1.3, the problem is not fixed in the bytecode verifier, but within the security components of the JVM. With SDK 1.3, a security exception is thrown, because of restricted access to the `java.lang` package. This choice may be reasonable if a unique hierarchy and unique system classes are meant for any JVM. There are arguments in favor of a more liberal notion (as mentioned in [10]) in which system classes and multiple class hierarchies may be user-definable. In any case, the issue of system classes certainly deserves further explanation and clarification in future editions of [6].

If we assume a single class hierarchy (and therefore a single `java.lang.Object` class), an obvious measure to avoid the bug is to ensure that the FQN `java.lang.Object` resolves to the usual, unique system class. A better alternative is to generate a loading constraint $\texttt{java.lang.Object}^L = \texttt{java.lang.Object}^{L_0}$, where L is the defining loader of the class being verified and L_0 is the system loader.

4.4. Preventing spoofing of `java.lang.String`

The confusion relating to `java.lang.String` can be prevented by resolving `java.lang.String` and ensuring that it is indeed the "usual" system class, or by generating the loading constraint $\texttt{java.lang.String}^L = \texttt{java.lang.String}^{L_0}$, where L is the defining loader of the class being verified and L_0 is the system loader.

This bug brings to light inadequacies in the specification [6], which does not explicitly prohibit user-defined classes with name `java.lang.String`. As previously mentioned, since the string constant is not passed to the methods as a parameter, no loading constraint is generated. The spoofing of `java.lang.String` is disallowed in SDK 1.3 through the same security mechanism that disallows spoofing of `java.lang.Object`. The above discussion about unique hierarchy and system classes applies here as well.

4.5. A more general solution

4.5.1. A new design for the bytecode verifier

We have seen that the bytecode verifier uses FQNs to denote classes (types) under the assumption that a FQN resolves to a unique class. The bugs described in this paper are based on violations of that assumption. In each case a loading constraint can be introduced that ensures this assumption is correct. In the current SDK, the class loading mechanisms generate constraints but the bytecode verifier does not. The bytecode verifier sometimes needs to load classes to check assignment compatibility or to obtain the FQN of the first common superclass of two classes. Since it is generally desirable to avoid loading classes until required by execution, eliminating class loading by the bytecode verifier is advantageous.

We now propose a design for the bytecode verifier (and related parts of the JVM) that (1) avoids premature loading and (2) provides a cleaner separation between bytecode verification and loading. This cleaner separation also promotes a better understanding of how bytecode verification and other mechanisms (such as loading constraints) cooperate to ensure type safety in the JVM.

In the design we propose, the bytecode verifier uniformly uses FQNs, never actual classes. The intended disambiguation is that FQN `N` stands for class $\mathtt{N}^L$, where L is the defining loader of the class under verification (note that, at verification time, class $\mathtt{N}^L$ might not be present in the JVM yet).

The bytecode verifier never causes resolution (and loading) of any class. The result of merging two FQNs is a set containing the two FQNs. More precisely, the bytecode verifier uses (finite) sets of FQNs (and not just FQNs) as types for stack positions and local variables containing reference types [1, 3, 7]. Initially (e.g., in the local variables containing method invocation arguments) sets are singletons. Merging is set union. The meaning of a set of FQNs assigned as the type of a local variable or stack position is that the local variable or stack position may contain an instance of a class whose FQN is in the set. No relationship among the elements of the set is intended.

When a set of FQNs is checked for assignment compatibility with a given FQN `N`, for each element `M` of the set different from `N`, a *subtype loading constraint* $\mathtt{M}^L < \mathtt{N}^L$ is generated. The meaning of such constraint is that class $\mathtt{M}^L$ must be a subclass of class $\mathtt{N}^L$. The constraint is added to the global state of the JVM, and checked for consistency with the loaded class cache. If either class has not been loaded yet, the constraint is just recorded. Whenever the loaded class cache is updated, it is checked for consistency with the current subtype loading constraints.

This is very similar to the equality loading constraints of the form $\mathtt{N}^L = \mathtt{N}^{L'}$ introduced in the Java 2 Platform. In fact, subtype constraints complement equality constraints. Checking the consistency of the loaded class cache and loading constraints that include both subtype constraints and equality constraints is neither difficult nor inefficient. A naïve algorithm will close transitively both subtype and equality constraints and then check that when the loaded class cache is updated none of the constraints in the transitive closure is violated. An efficient algorithm will use a union-find data structure to store equivalence classes of classes asserted to be the same and track the asserted subtype dependencies of the equivalence classes.

In this design, the result of bytecode verification of a class is therefore not simply a yes/no answer, but also a set of subtype constraints that explicitly and clearly express the assumptions made by the bytecode verifier for certification of the class. Furthermore, the bytecode verifier is a well-defined, purely functional component of the JVM that does not depend on the current state of JVM data structures.

Our approach also allows a cleaner treatment of interface types in the bytecode verifier. Since an interface can have more than one superinterface, two given interfaces may not have a unique first common superinterface. According to [6], the result of merging two interface FQNs is therefore `java.lang.Object`, which is indeed a superclass of any interface. However, this requires a special treatment of `java.lang.Object` when checking its assignment compatibility with an interface FQN: the bytecode verifier just passes the check because `java.lang.Object` might derive from merging interfaces, even though `java.lang.Object` itself is not assignment-compatible with an interface. This "looseness" does not cause type unsafety because the bytecode instruction `invokeinterface` performs a search of the methods declared in the runtime class of the object on which it is executed. If no method matching the referenced descriptor is found, an exception is thrown. This runtime check does not impose any additional runtime penalty. Our scheme is cleaner in that it provides a uniform treatment of classes and interfaces.

4.5.2. How the new design avoids the spoofing bugs

Let us now see how this design prevents spoofing the first common superclass. When verifying the method `merge`, the creation (and initialization) of the two instances of class `Sub1` and `Sub2` has the effect of typing the local variables as $\{\texttt{Sub1}\}$ and $\{\texttt{Sub2}\}$. After the merging point, the type on top of the stack is $\{\texttt{Sub1},\texttt{Sub2}\}$. Since the bytecode instruction `getfield` references class `Sup`, subtype constraints $\texttt{Sub1}^L < \texttt{Sup}^L$ and $\texttt{Sub2}^L < \texttt{Sup}^L$ are generated. When `merge` is eventually executed, before the field is accessed all of $\texttt{Sub1}^L$, $\texttt{Sub2}^L$, and $\texttt{Sup}^L$ will have been loaded: since subtype constraints are violated, the JVM will throw an exception preventing field resolution (and therefore access).

Let us now consider the spoofing of a superclass by means of `invokespecial`. When verifying the method `invSpec`, the `invokespecial` instruction references class `SupSup`, and the top of the stack is typed as $\{\texttt{Sub}\}$. So, the subtype constraint $\texttt{Sub}^L < \texttt{SupSup}^L$ is generated. When `invSpec` is eventually executed, before `invokespecial` is executed both $\texttt{Sub}^L$ and $\texttt{SupSup}^L$ will have been loaded: the inconsistency would make the JVM raise an exception and prevent the method call.

Concerning the spoofing of `java.lang.Object`, during verification of `assign` the subtype constraint $\texttt{Sub}^L < \texttt{java.lang.Object}^L$ is generated. Its violation will raise an exception.

As already discussed the spoofing of `java.lang.String` is prevented by generating an equality constraint $\texttt{java.lang.String}^{L_0} = \texttt{java.lang.String}^L$, which would cause an exception to be raised if it is not satisfied. Note that subtype constraints do not directly address the spoofing of `java.lang.String`: equality constraints are needed.

4.5.3. *Formalization of the new design*

The approach described above has been formalized in [9]. Such a paper provides a formal specification of an abstraction of the JVM and proves type safety results about it. Our JVM specification, while making several simplifications (e.g., only a few bytecode instructions are considered and error exceptions cause termination), captures the essence of class loading and bytecode verification.

Concretely, we specify the JVM as a state machine. The state includes (models of) the method call stack for one thread, the heap, the loaded class cache, as well as equality and subtype loading constraints. Transitions in the machine include execution of some instructions (e.g., method invocation and field access), resolution of classes, fields, and methods, and calls to user code to load classes (through the `loadClass` method). Our state machine is a defensive one, i.e., transitions include type safety checks about objects having certain classes. According to [6] the behavior of the JVM is undefined when such checks are not satisfied. Therefore, our main type safety result states that our state machine never halts because of the failure of any of these type safety checks.

The following is the transition rule (GF) in our specification, which formalizes the execution of a `getfield` instruction:

$$\frac{\begin{array}{l} cd(m)|_p = \mathsf{getfield}(cn, fn, cn_0) \\ \langle hp, lc, ct\rangle \overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle \\ cl(o) \preceq_{lc'} lc'(ld(c), cn) \end{array}}{\begin{array}{l} \langle stk :: \langle c, m, p, os :: o, lv\rangle, hp, lc, ct\rangle \Longrightarrow \\ \langle stk :: \langle c, m, p+1, os :: hp'(o)(fn), lv\rangle, hp', lc', ct'\rangle \end{array}} \quad \text{(GF)}$$

Without going into details, the rule asserts that if (1) the current instruction is a `getfield`, (2) field resolution succeeds, and (3) the runtime class of the object target of the `getfield` instruction is a subclass of the result of resolving cn from the current class (this is the type safety check), then the object (reference) is removed from the top of the operand stack and replaced by the value of the field. The tuple $\langle c, m, p, os :: o, lv\rangle$ constitutes the top frame of the method call stack (the rest being stk), where c, m, and p are the current class, method, and program counter. $os :: o$ and lv are the current operand stack (with o at the top) and local variables. Furthermore, hp, lc, and ct constitute the current heap, loaded class cache, and loading constraints (both equality and subtype). The field resolution process results in updated heap hp', loaded class cache lc', and loading constraints ct'. The updating is specified by other transition rules, which capture that field resolution requires class resolution first, which may require loading (through a call to user code in a `loadClass` method). In fact, the inherent mutual recursion between user code and internal JVM processing such as resolution, is reflected in our specification by nested rules that link internal transitions with user code transitions.

In order to prove our type safety results, we proceed as follows. First, we define a notion of *valid* states. Validity includes simple requirements such as closure of the heap with respect to referenced objects, as well as a more interesting *conformance* property. Conformance means that each value in the operand stack and local variables "matches" the type statically assigned by the bytecode verifier to the corresponding stack position or local variable (at the current program point), and that each value in an object field "matches" the type of the field. However, both the class types of fields and the class types assigned by the bytecode verifier consist of FQNs only. Therefore, equality and subtype loading constraints are taken into account: for example, an object with runtime class $\langle \mathtt{N}, L\rangle$ conforms to a class $\langle \mathtt{M}, L'\rangle$ if there are loading constraints $\mathtt{N}^L = \mathtt{N}^{L'}$ and $\mathtt{N}^{L'} < \mathtt{M}^{L'}$, even in case $\mathtt{N}^{L'}$ has not been loaded yet.

Next, we prove a theorem stating that each transition from a valid state leads to a valid state. This means that, starting from a valid initial state, validity is preserved throughout the whole execution of our state machine. Finally, we prove that in valid states the type safety conditions (such as the third condition in rule (GF) above) are always satisfied. This means that the only reason why our state machine can halt is either that there is no more code to execute or that some other condition is not satisfied. Each of these other conditions indeed corresponds to checks actually performed by the JVM. An example is the second condition in rule (GF): if field resolution in the JVM fails, an exception is thrown. In our model, the state machine just halts.

5. RELATED WORK

Saraswat [10] reported bugs in the class loading mechanisms of Sun's JDK 1.1 [14], which caused type safety violations using delegating class loaders. In order to fix those problems, equality loading constraints were introduced [5]. Tozawa and Hagiya [15] reported type-safety bugs in the bytecode verifier of Sun's SDK 1.2, which partially overlap with the bugs reported in this paper. The common bugs were discovered and reported independently.

Our idea of having a self-contained bytecode verifier that generates subtype constraints finds its roots in earlier work by Goldberg [3] and is similar in spirit to Fong and Cameron's work [2]. However, both papers do not consider multiple class loaders. Another difference is that in those formalizations the bytecode verifier generates, besides subtype constraints, additional constraints, e.g. for fields and methods referenced in the code being verified. In the design proposed in this paper only subtype constraints are generated because the others are checked at runtime, as described in [6], without performance penalty or premature loading.

In [9] we provide a formal specification of Java class loading. The formalization includes equality and subtype constraints, as well as theorems stating that the constraints plus bytecode verification guarantee type safety. The focus of that paper is to provide formal results. The focus of this paper is to expose bugs and propose solutions; subtype constraints constitute one of the solutions to three of the bugs.

6. CONCLUSION

We have presented some bugs in the bytecode verifier of SDK 1.2 that lead to type safety violations. Some of these bugs expose inaccuracies or ambiguities in the JVM specification [6]. We have proposed fixes for all such bugs, including a general approach with subtype constraints that results in additional benefits, namely lazier loading and a cleaner interface between the bytecode verifier and the rest of the JVM.

These problems and solutions have been found in the course of our efforts towards specifying various components of the JVM. Therefore, this paper also shows how formal studies can enable the discovery of problems and solutions in real-world systems.

Java and the JVM are perhaps the first language and platform in widespread use sufficiently well-designed to be amenable to precise analysis. This paper, similarly to other work in the field, aims at improving the understanding, assurance, and usability of the Java paradigm.

REFERENCES

1. Coglio A, Goldberg A, Qian Z. Towards a provably-correct implementation of the JVM bytecode verifier. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java.* 1998.
2. Fong PWL, Cameron R. Proof linking: an architecture for modular verification of dynamically-linked mobile code. In *Proc. ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering.* 1998; 222–230.
3. Goldberg A. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security.* 1998.
4. Gün Sirer E, Grimm R, Bershad B. The Kimera Project page. `http://kimera.cs.washington.edu`
5. Liang S, Bracha G. Dynamic class loading in the Java$^{\text{TM}}$ Virtual Machine. In *Proc. 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM Press, 1998; 36–44.
6. Lindholm T, Yellin F. *The Java^{TM} Virtual Machine Specification* (2nd edn). Addison-Wesley, 1999.
7. Qian Z. A formal specification of Java$^{\text{TM}}$ Virtual Machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java^{TM},* Alves-Foss J (ed.). LNCS 1523, Springer-Verlag, 1998.
8. Qian Z. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems* 2000; **22**(4):638–672.
9. Qian Z, Goldberg A, Coglio A. A formal specification of Java class loading. In *Proc. 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM Press, 2000; 325–336. Long version available at `http://www.kestrel.edu/java`.
10. Saraswat V. Java is not type-safe. 1997. Available at `http://www.research.att.com/vj/bug.html`.
11. Srinivas Y, Jüllig R. Specware: formal support for composing software. In *Proc. 3rd Conference on Mathematics of Program Construction,* Moeller B (ed.). LNCS 947, Springer-Verlag, 1995; 399–422.
12. Sun Microsystems. Java 2 SDK, Standard Edition, vers. 1.2. Available at `http://java.sun.com`.
13. Sun Microsystems. Java 2 SDK, Standard Edition, vers. 1.3. Available at `http://java.sun.com`.
14. Sun Microsystems. Java Development Kit, vers. 1.1. Available at `http://java.sun.com`.
15. Tozawa A, Hagiya M. Careful Analysis of Type Spoofing. In *JIT'99 Java-Informations-Tage 1999.* Springer Verlag, 1999; 290–296.

APPENDIX: CODE SHARED BY THE EXAMPLES

The code for class `Starter` is shown in Figure 13. The code for class `DelegatingLoader` and its (abstract) superclass `LocalClassLoader`, largely borrowed from [10], is shown in Figures 14 and 15.

```
class Starter {
    public static void go (String[] undelNames,
                           String className,
                           String methodName) {
        try {
            DelegatingLoader loader =
                new DelegatingLoader("dir/", undelNames);
            Class c = loader.loadClass(className);
            Object[] arg = {};
            Class[] argClass = {};
            c.getMethod(methodName, argClass).invoke(null, arg);
        } catch (Exception e) {
            System.out.println("Error " + e.toString() +
                               " in Main.main.");
            e.printStackTrace();
        }
    }
}
```

Figure 13. Code for class `Starter`

```
class DelegatingLoader extends LocalClassLoader {

    private String[] undelegatedNames;

    public DelegatingLoader (String dir, String[] undelNames) {
        super(dir);
        this.undelegatedNames = undelNames;
    }

    private boolean isUndelegatedName (String name) {
        boolean found = false;
        for (int i=0; i < undelegatedNames.length && !found; i++)
            if (undelegatedNames[i].equals(name))  found = true;
        return found;
    }

    public Class loadClass(String name)
        throws ClassNotFoundException {
        try {
            Class prevLoaded = this.findLoadedClass(name);
            if (prevLoaded != null)
                return prevLoaded;
            else if (isUndelegatedName(name))
                return this.loadClassFromFile(name);
            else {
                System.out.println
                    ("[Loaded " + name + " from system]");
                return this.findSystemClass(name);
            }
        } catch (Exception e) {
            System.out.println
                ("Exception " + e.toString() +
                 " while loading " + name + " in DelegatingLoader.");
            throw new ClassNotFoundException();
        }
    }

}
```

Figure 14. Code for class `DelegatingLoader`

```
abstract class LocalClassLoader extends ClassLoader {

    private String directory;

    public LocalClassLoader (String dir) {
        directory = dir;
    }

    protected Class loadClassFromFile(String name)
        throws ClassNotFoundException, FileNotFoundException {
        File target =
            new File(directory + name.replace('.', '/') + ".class");
        if (! target.exists()) throw new FileNotFoundException();
        long bytecount = target.length();
        byte [] buffer = new byte[(int) bytecount];
        try {
            FileInputStream f = new FileInputStream(target);
            int readCount = f.read(buffer);
            f.close();
            Class c = defineClass(name, buffer, 0, (int) bytecount);
            System.out.println
                ("[Loaded " + name + " from " +
                 target + " ("+ bytecount + " bytes)]");
            return c;
        } catch (Exception e) {
            System.out.println("Aborting read: " + e.toString() +
                                   " in LocalClassLoader.");
            throw new ClassNotFoundException();
        };
    }
}
```

Figure 15. Code for class `LocalClassLoader`