JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 4, No. 8, 2005

Checking Access to Protected Members in the Java Virtual Machine

Alessandro Coglio, Kestrel Institute, Palo Alto, California, USA

This paper studies in detail how to correctly and efficiently check access to protected members in the Java Virtual Machine. This aspect of type safety is not explained in the official specification and, to the author's knowledge, has been completely neglected in the research literature. Nonetheless, it is a subtle aspect that is not straightforward to implement correctly, as also evidenced by the presence of a bug in earlier versions of Sun's Java 2 SDK. This paper presents example programs that expose the bug, along with a conjectural explanation for it. This paper also presents some corpus measurements of the number of checks that can be performed using various techniques.

1 INTRODUCTION

Java [5] is normally compiled to a platform-independent bytecode language, which is executed by the Java Virtual Machine (JVM) [7]. For security reasons [4], the JVM must ensure type-safe execution (e.g. object references must not be forged from integers), without relying on the type checking performed by Java compilers.

This paper studies in detail a narrow but interestingly subtle aspect of ensuring type safety in the JVM: checking access to protected members (fields and methods). Java compilers can easily check access to protected members and constructors[1] by looking them up in their declaring classes, which must be available in source or bytecode form. Checking access to protected members in the JVM is more difficult because of dynamic class loading. This aspect of type safety in the JVM is not explained in [7] and, to the author's knowledge, has been completely neglected in the research literature.

Section 2 reviews some relevant features of the JVM. Sections 3 and 4 describe the requirements on protected member access and how they can be checked. Section 5 reports on the incorrect checking of protected member access by earlier versions of Sun's Java 2 SDK, including example programs that expose the bug. Concluding remarks are given in Section 6.

[1] In Java, constructors are not members. In the JVM, constructors are realized by instance initialization methods, i.e. methods with the special name <init>, which are members.

Cite this article as follows: citation

2 BACKGROUND

Some of the JVM features described in this section are slightly simplified for brevity and ease of understanding; for instance, we only consider classes and not interfaces. However, the results of the paper do not depend on the simplifications and apply to the unsimplified JVM. For full details on the JVM features described here, see [7, 6], as well as [10] for a formal treatment.

Class Loading

The JVM supports dynamic, lazy loading of classes. Lazy loading improves the response time of a Java applet or application and reduces memory usage: execution starts after loading only a few classes and the other classes are loaded on demand if and when needed.

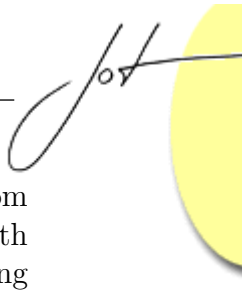
Classes are loaded into the JVM by means of class loaders, which can be user-defined to realize customized loading policies. To load a class, the JVM supplies a (fully qualified) class name C (e.g. $\mathsf{edu.kestrel.Specware}$) to a loader $\mathcal{L}$. $\mathcal{L}$ may itself create a class c with name C, e.g. from a binary representation stored in the local file system or fetched through a network connection. Alternatively, $\mathcal{L}$ may delegate loading to another loader $\mathcal{L}'$ by supplying C to $\mathcal{L}'$. $\mathcal{L}'$ may itself create c or delegate loading to yet another loader $\mathcal{L}''$, and so on until some loader $\mathcal{L}^*$ eventually creates c.

The initiating loader of c is $\mathcal{L}$, the first one in the delegation chain, which the JVM supplies C to. The defining loader of c is $\mathcal{L}^*$, the last loader in the delegation chain, which creates c. Initiating and defining loader coincide when no delegation takes place.

The JVM prohibits a loader from creating two classes with the same name, but two different loaders may well create two classes with the same name. Classes created by different loaders are always distinct (even though they may have the same binary representation, including name), i.e. each loader has its own name space. Thus, while at compile time a class is uniquely identified by its name, in the JVM a class is uniquely identified by its name plus its defining loader, e.g. $\langle \mathsf{edu.kestrel.Specware}, \mathcal{L}^* \rangle$.

Accordingly, a (run time) package is uniquely identified by a name plus a loader, e.g. $\langle \mathsf{edu.kestrel}, \mathcal{L}^* \rangle$, which includes $\langle \mathsf{edu.kestrel.Specware}, \mathcal{L}^* \rangle$ among its classes; as a special case, an unnamed package is uniquely identified by just a loader. All the classes in a package have the same defining loader, which is the loader that, together with the name (unless the package is unnamed), uniquely identifies the package.

When a class is loaded, all its superclasses are loaded. So, the JVM satisfies the invariant that the classes present in the machine are closed w.r.t. the superclass relation.



The JVM maintains a cache of loaded classes in the form of a finite map from names and loaders to classes. When a class c is loaded, the cache is extended with an entry that associates c to its name C and its initiating loader $\mathcal{L}$; if the defining loader is $\mathcal{L}^* \neq \mathcal{L}$, the cache is also extended with an entry that associates c to C and $\mathcal{L}^*$.

Resolution

Some bytecode instructions embed symbolic references to classes and members, e.g. to invoke methods. These symbolic references are resolved to actual classes and members before the embedding instructions are executed for the first time.

A class reference C is a class name. A member reference $\mathsf{C}\,.\,\mathsf{n}\,{:}\,\mathsf{d}$ consists of a class reference C, a member name n, and a member descriptor d. A field descriptor is (a textual representation of) a type, while a method descriptor consists of (a textual representation of) zero or more argument types and a result type (possibly void); a class type occurring in a descriptor is only a class name, without any defining loader because descriptors are generated at compile time when loaders, which are run time entities, are unknown.

The JVM resolves a class reference C that occurs in a class $\langle \mathsf{X}, \mathcal{L} \rangle$ by supplying C to $\mathcal{L}$, i.e. the defining loader of the class in which the reference occurs; the resulting class c is the result of resolution. More precisely, before supplying C to $\mathcal{L}$ for loading, the JVM looks up the loaded class cache. If there is an entry that associates a class c to C and $\mathcal{L}$, c is the result of resolution and no loading takes place. Otherwise, loading takes place as described. This cache look up mechanism ensures that resolving a name C via a loader $\mathcal{L}$ (i.e. from a class $\langle \mathsf{X}, \mathcal{L} \rangle$) consistently yields a unique class, denoted by $\mathsf{C}^{\mathcal{L}}$.

Both notations $\langle \mathsf{C}, \mathcal{L} \rangle$ and $\mathsf{C}^{\mathcal{L}}$ [7, 6] denote classes. The notation $\langle \mathsf{C}, \mathcal{L} \rangle$ denotes the class with name C and defining loader $\mathcal{L}$; in this notation, $\mathcal{L}$ is always the defining loader of the class. The notation $\mathsf{C}^{\mathcal{L}}$ denotes the class associated to name C and loader $\mathcal{L}$ in the loaded class cache; in this notation, $\mathcal{L}$ is either the initiating or the defining loader of the class. If $\langle \mathsf{C}, \mathcal{L} \rangle$ is defined, then also $\mathsf{C}^{\mathcal{L}}$ is defined and it is the case that $\langle \mathsf{C}, \mathcal{L} \rangle = \mathsf{C}^{\mathcal{L}}$.

A member reference $\mathsf{C}\,.\,\mathsf{n}\,{:}\,\mathsf{d}$ is resolved by first resolving the class reference C to a class c and then searching for a member with name n and descriptor d declared in c. If such a member m is found, that is the result of resolution. Otherwise, the member is searched in the direct superclass of c, and so on until the member is found in some superclass of c.

Bytecode Verification

After a class is loaded and before any of its code is executed, its methods go through bytecode verification, one at a time. Bytecode verification is a (forward) data flow

analysis [9] whose purpose is to statically establish that certain type safety properties will be satisfied when instructions are executed at run time, so that the interpreter or just-in-time compiler can safely omit checks of those properties for better performance.

For example, the bytecode verifier checks that the iadd instruction, which adds two integers together, will always operate on two integers, and not on other data values such as floating point numbers or object references. As another example, the bytecode verifier checks that the getfield instruction, which retrieves the value stored in a field, will always operate on a reference to an object whose class is the one referenced in the field reference embedded in the instruction or a subclass of it.

As previously explained in the description of resolution, a class reference C occurring in a class $\langle \mathsf{X}, \mathcal{L} \rangle$ stands for the class $\mathsf{C}^{\mathcal{L}}$. The same holds for a class type C inferred by the bytecode verifier during the data flow analysis. At the time $\langle \mathsf{X}, \mathcal{L} \rangle$ is verified, $\mathsf{C}^{\mathcal{L}}$ may or may not have been loaded yet. The bytecode verifier generally avoids resolving (and loading) classes, thus supporting lazy loading. For example, if it infers a type C as the target of a getfield whose embedded field reference is $\mathsf{C}.\mathsf{n}:\mathsf{d}$, C is not resolved because no matter what class $\mathsf{C}^{\mathcal{L}}$ will be, the referenced field will be declared in $\mathsf{C}^{\mathcal{L}}$ or one of its superclasses and so the field access will be type-safe. However, if the inferred target type is $\mathsf{D} \neq \mathsf{C}$, the bytecode verifier resolves D and C to check that $\mathsf{D}^{\mathcal{L}}$ is a subclass of $\mathsf{C}^{\mathcal{L}}$, written $\mathsf{D}^{\mathcal{L}} < \mathsf{C}^{\mathcal{L}}$.

Loading Constraints

When (instructions in) different classes exchange objects, they must agree on the exchanged objects' classes (i.e. names plus loaders) and not just class names (agreement on names is automatically ensured by resolution). For example, consider a class $\langle \mathsf{X}, \mathcal{L} \rangle$ creating an instance of a class named D and passing it to a method, with argument type D, of a class $\langle \mathsf{Y}, \mathcal{L}' \rangle$. The instance belongs to $\mathsf{D}^{\mathcal{L}}$ but the method will use it as an instance of $\mathsf{D}^{\mathcal{L}'}$. So, if $\mathcal{L} \neq \mathcal{L}'$, it must be $\mathsf{D}^{\mathcal{L}} = \mathsf{D}^{\mathcal{L}'}$ for type safety to be maintained.

These constraints are checked when members are resolved. When a class $\langle \mathsf{X}, \mathcal{L} \rangle$ resolves a member $\mathsf{C}.\mathsf{n}:\mathsf{d}$, if $\mathsf{C}^{\mathcal{L}} = \langle \mathsf{C}, \mathcal{L}' \rangle$, for every class name D appearing in the descriptor d the constraint $\mathsf{D}^{\mathcal{L}} = \mathsf{D}^{\mathcal{L}'}$ must hold. If any of $\mathsf{D}^{\mathcal{L}}$ and $\mathsf{D}^{\mathcal{L}'}$ has not been loaded yet, the JVM generates "$\mathsf{D}^{\mathcal{L}} = \mathsf{D}^{\mathcal{L}'}$" as a syntactic constraint, adding it to a set of pending constraints that are part of the state of the machine. The loaded class cache and the loading constraints are maintained consistent by re-checking the pending loading constraints every time the loaded class cache is updated.

The transitive closure of the pending constraints must be considered, e.g. if $\mathsf{D}^{\mathcal{L}}$ and $\mathsf{D}^{\mathcal{L}''}$ are loaded but $\mathsf{D}^{\mathcal{L}'}$ is not, the constraints $\mathsf{D}^{\mathcal{L}} = \mathsf{D}^{\mathcal{L}'}$ and $\mathsf{D}^{\mathcal{L}'} = \mathsf{D}^{\mathcal{L}''}$ are (jointly, though not individually) violated if $\mathsf{D}^{\mathcal{L}} \neq \mathsf{D}^{\mathcal{L}''}$. The need to consider the transitive closure is illustrated by the following example. A class $\langle \mathsf{X}, \mathcal{L} \rangle$ creates an object of class $\mathsf{D}^{\mathcal{L}}$ and passes it to a method of a class $\langle \mathsf{Y}, \mathcal{L}' \rangle$; when the method

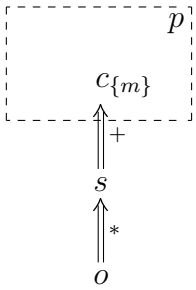


Figure 1: Restriction on protected access

is resolved, the constraint $\mathsf{D}^{\mathcal{L}} = \mathsf{D}^{\mathcal{L}'}$ is generated. The method in $\langle \mathsf{Y}, \mathcal{L}' \rangle$ does not access the object (so that $\mathsf{D}^{\mathcal{L}'}$ does not get loaded) but passes it to a method of a class $\langle \mathsf{Z}, \mathcal{L}'' \rangle$; when the method is resolved, the constraint $\mathsf{D}^{\mathcal{L}'} = \mathsf{D}^{\mathcal{L}''}$ is generated. Now $\langle \mathsf{Z}, \mathcal{L}'' \rangle$ accesses the object as if it had class $\mathsf{D}^{\mathcal{L}''}$. Thus, it must be $\mathsf{D}^{\mathcal{L}} = \mathsf{D}^{\mathcal{L}''}$ for type safety to be maintained.

It has been proposed in [10, 3] to use subtype constraints, next to the equality constraints just described, to support lazier loading during bytecode verification. When a class $\langle \mathsf{X}, \mathcal{L} \rangle$ is verified, instead of resolving D and C to check whether $\mathsf{D}^{\mathcal{L}} < \mathsf{C}^{\mathcal{L}}$, the constraint "$\mathsf{D}^{\mathcal{L}} < \mathsf{C}^{\mathcal{L}}$" is generated. The transitive closure of equality and subtype constraints must be considered. Subtype constraints are not part of [7], but they are presented here because a similar idea will be described for protected members in Section 4.

3 REQUIREMENTS ON PROTECTED MEMBERS

In the JVM, like in Java, every member has an associated access attribute: public, private, protected, or default. The attribute determines the circumstances in which the member can be accessed.

Let m be a member declared in a class c that belongs to a package p. If m is public, it can be accessed by (code in) any class. If m is private, it can be accessed only by c. If m has default access, it can be accessed only by any class that belongs to p.

If m is protected, things are slightly more complicated. First, m can be accessed by any class belonging to p, as if it had default access. In addition, it can be accessed by any subclass s of c that belongs to a package different from p, with the following restriction: if m is not static, then the class o of the object whose member is being accessed must be s or a subclass of s, written $o \leq s$ (if m is static, the restriction does not apply: m can be always accessed by s). The relationship among c, s, o, m, and p is depicted in Figure 1, where the double-line arrow labeled by $+$ denotes one or more direct superclasses and the double-line arrow labeled by $*$ denotes zero or more direct superclasses.

The restriction on protected access (i.e. that $o \leq s$ if $s < c$ and s does not belong to p) ensures that a protected member m declared in c is accessible, outside the package p of c, only (on objects whose classes are) inside the part of the class hierarchy rooted at the accessing (sub)class s [1, Sect. 3.2] [2]. In particular, s cannot access m in a part of the class hierarchy rooted at a (sub)class $s' \neq s$ in a different branch. See [1, Sect. 3.2] [2] [5, Sect. 6.6.7] for examples.

The restriction on protected access prevents almost arbitrary access to protected instance members of objects [12]. Suppose that m is a protected instance field declared in c. Without the restriction, any class x could read the content of the field m of any object of class c, using the following trick: define a subclass s of c (the trick works only if c is not final, hence the "almost" adverb above); declare a method in s that takes an object of class c as argument and returns the content of its m field; and have x call this method. The restriction on protected access prevents this situation, because s can access the field only if $c \leq s$, which is impossible since $s < c$.

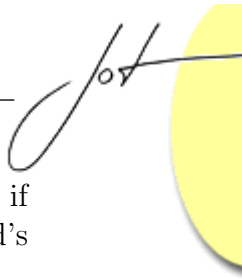
The access rules are described in [7, Sect. 5.4.4]. The restriction on protected access is described in the specification of the getfield, putfield, invokevirtual, and invokespecial instructions in [7, Chapt. 6]. The access rules for members in the JVM correspond to the access rules for members and constructors in Java [5, Sect. 6.6].

The access rules do not include any requirement on the accessibility of c. One might expect, for instance, that if m is public then it can be accessed by a class x not belonging to p only if c is also public. However, c may not be the class directly referenced by x: x may reference a class c' that inherits m from c (i.e. m is found in c, starting the search from c'). It is sufficient that c' can be accessed by x (i.e. c' and x are in the same package or c' is public), regardless of whether c can be accessed by x or not.

Because of dynamic dispatch, the method actually invoked on an object may differ from the method that the reference resolves to. However, access control checks only apply to the resolved method, not the dynamically invoked one. While Java compilers check that an overriding method does not restrict the access attribute [5, Sect. 8.4.6.3], the JVM does not check that: [7] does not explicitly require the check and it is easy to verify that, for instance, SDK 1.4 does not perform the check. Anyhow, this restriction on overriding methods does not always ensure that access to the resolved method implies access to the overriding method [11]: if both methods are protected and are declared in different packages, then classes in the resolved method's package that are not subclasses of the overriding method's declaring class can access the resolved method but not the overriding method.

4 CHECKING THE REQUIREMENTS

Most access rules are checked during resolution. For example, when a getfield is executed for the first time, the embedded field reference is resolved and the access



attribute of the resulting field is checked against the class where getfield occurs: if access is disallowed, an exception is thrown; otherwise, getfield retrieves the field's content.

The restriction on protected access requires additional checking. Suppose that the getfield occurs in a class s, that the field reference resolves to a non-static, protected field declared in a superclass c of s, and that c and s belong to different packages, as in Figure 1. Whether the field access is allowed depends on the class o of the target object, which may be different each time the getfield is executed. The following subsections describe various strategies to check whether $o \leq s$. While [7] states the requirements on protected member access, it does not explain how they can be checked.

Run Time Checking

The class of an object is always available through a reference to the object, in order to support the semantics of the instanceof and checkcast instructions. In typical implementations of the JVM, the storage for an object includes a reference to the object's class.

The simplest strategy to check the restriction on protected access is to have getfield, putfield, invokevirtual, and invokespecial do it each time they are executed. This run time check is performed only if the referenced member is protected and declared in a superclass of the current class (i.e. the one where the instruction occurs) that belongs to a different package.

The first edition of [7] describes, in Chapter 9, an optimization for JVM implementations: certain instructions are replaced by quick, internal pseudo-instructions the first time they are executed. For instance, the first time a getfield is executed, it is replaced by the getfield_quick pseudo-instruction: instead of a field reference, getfield_quick embeds implementation-dependent information (typically, an offset) that is determined when the field is resolved and that is used to access the target object's field more quickly than going through the (resolved) field reference.

A similar rewriting approach could be used in a JVM implementation to more efficiently check protected member access at run time. The first time a getfield is executed and the field reference is resolved, there are two cases: if the resulting field is not protected or is not declared in a superclass of the current class that belongs to a different package, getfield is replaced by getfield_quick; otherwise, it is replaced by getfield_quick_prot. This new pseudo-instruction embeds the same implementation-dependent information as getfield_quick; in addition, its execution includes the run time check that $o \leq s$, where o is the class of the target object and s is the current class. An analogous strategy can be introduced for putfield, invokevirtual, and invokespecial.

However, getfield_quick_prot would not be very quick. Even though the run time check is only performed for protected fields declared in superclasses belonging to

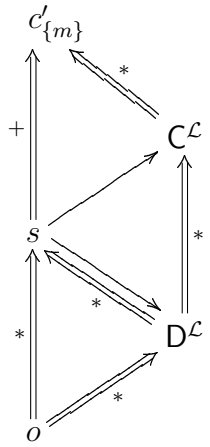


Figure 2: Relationship among all the involved classes

different packages, the overhead could be significant in certain programs.

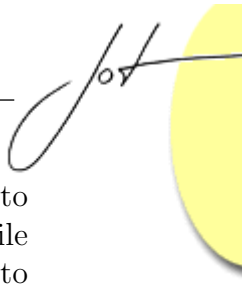
In addition, [7, Sect. 5.4.4] states explicitly that the restriction on protected access should be checked as part of bytecode verification. In general, type safety in the JVM could be completely ensured via run time checks, but execution would be painfully slow. The purpose of static checking in the JVM is to increase performance. The following strategies check the restriction on protected access statically.

Eager Resolution

When analyzing a getfield, putfield, invokevirtual, or invokespecial, the bytecode verifier could resolve the embedded member reference and, if needed, check the restriction on protected access using the statically inferred target type.

Suppose that the class under verification is $s = \langle \mathsf{S}, \mathcal{L} \rangle$, the embedded member reference is $\mathsf{C}\,.\,\mathsf{n}\!:\!\mathsf{d}$, and the inferred target type is D. If the member reference resolves to a protected member m declared in a class $c' \geq \mathsf{C}^{\mathcal{L}}$ that belongs to a package different from s and such that $c' > s$, the bytecode verifier checks that $\mathsf{D}^{\mathcal{L}} \leq s$. If $\mathsf{D} \neq \mathsf{S}$, this subtype check can be performed by resolving D or, according to the proposal described at the end of Section 2, by generating the subtype constraint $\mathsf{D}^{\mathcal{L}} < \mathsf{S}^{\mathcal{L}}$; if $\mathsf{D} = \mathsf{S}$, no resolution or constraint is necessary because $\mathsf{S}^{\mathcal{L}} = s$. The check that $\mathsf{D}^{\mathcal{L}} \leq s$ is in addition to the check that $\mathsf{D}^{\mathcal{L}} \leq \mathsf{C}^{\mathcal{L}}$, which is always needed, regardless of whether m is protected and of where it is declared. Since at run time the class o of the target object always satisfies $o \leq \mathsf{D}^{\mathcal{L}}$, it is always the case that $o \leq s$ if $\mathsf{D}^{\mathcal{L}} \leq s$. The relationship among all these classes is depicted in Figure 2, where the single-line arrows denote resolution (i.e. the target class is the result of resolving the name of the target class from the source class).

This static check is less precise than a run time check, because it might always be



$o \leq s$ even if $\mathsf{D}^{\mathcal{L}} \leq s$ does not hold. However, less precision is the inevitable price to pay for better performance. In addition, the static check is as precise as the compile time checking in Java: the type statically inferred at compile time corresponds to the type statically inferred by the bytecode verifier. Besides soundness, the only requirement on bytecode verification is that code generated by Java compilers is accepted; if the restriction on protected access is satisfied in some Java code, it is also satisfied in the bytecode generated from that Java code.

The problem with this strategy is that *every* member reference from getfield, putfield, invokevirtual, and invokespecial is eagerly resolved in order to determine whether the member is protected and declared in a superclass of s that belongs to a different package. The additional check that $\mathsf{D}^{\mathcal{L}} \leq s$ is performed if and only if that is the case. Resolution may involve class loading and thus be a costly operation. Furthermore, eager resolution counters lazy loading.

As mentioned in Section 1, Java compilers check access to protected members and constructors by looking them up in their declaring classes. This effectively corresponds to the eager resolution strategy just described for the JVM. At compile time all classes are available and no dynamic loading is involved; thus, the strategy is adequate for compilation.

Limited Resolution

There are cases in which the restriction on protected access can be checked by the bytecode verifier without eagerly resolving the referenced member. In addition, in certain cases member resolution is guaranteed to cause no loading. For the remaining cases, the member reference can be resolved as described in the previous subsection.

Current Class as Target Type

Resolving the class name S from a class $s = \langle \mathsf{S}, \mathcal{L} \rangle$ results in s itself because the loaded class cache associates s to S and $\mathcal{L}$ when s is created, i.e. $\mathsf{S}^{\mathcal{L}} = s$.

If the bytecode verifier infers the class name S as the target of a getfield, putfield, invokevirtual, or invokespecial that occurs in s, there is no need to resolve the embedded member reference because $s \leq s$, regardless of whether the member is protected and of where it is declared. In other words, resolution can be soundly avoided if the inferred target type coincides with the name of the class under verification.

Necessary Condition for Subtype Check

Consider a getfield, putfield, invokevirtual, or invokespecial occurring in a class s, with embedded member reference $\mathsf{C}.\mathsf{n}:\mathsf{d}$. Since the member resulting from resolution must have name n and descriptor d, a necessary condition for m to be protected and declared in a superclass of s that belongs to a different package is that *some*

superclass of s, belonging to a different package, declares a protected field with name n and descriptor d.

This necessary condition can be checked by inspecting the superclasses of s, without resolving the member reference. Since the classes loaded into the JVM are closed under the superclass relation, inspecting the superclasses of s does not cause any loading. If no superclass of s declares a protected member with name n and descriptor d or every superclass that declares it belongs to the same package as s, it is impossible that the member resulting from resolution will be protected and declared in a superclass of s that belongs to a different package. Thus, the additional subtype check (i.e. that $\mathsf{D}^{\mathcal{L}} \leq s$) needs not be performed.

Resolution not Causing Loading

As mentioned before, resolving the class name S from a class $s = \langle \mathsf{S}, \mathcal{L} \rangle$ results in s itself without any loading taking place. In addition, resolving the direct superclass name R of s results in the direct superclass $\mathsf{R}^{\mathcal{L}}$ without any loading taking place, because when s is loaded $\mathsf{R}^{\mathcal{L}}$ is loaded too.

So, if the class name referenced by a getfield, putfield, invokevirtual, or invokespecial is S or R, the resolution of the member will cause no loading. This is a property that is satisfied because of the way the JVM is designed; no special action by the bytecode verifier is required.

Conditional Subtype Constraints

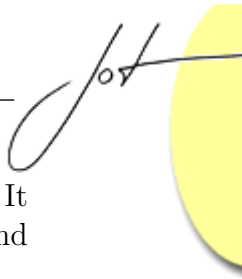
Instead of eagerly resolving a member reference and then checking the restriction on protected access if needed, the bytecode verified could generate a *conditional* subtype constraint. The condition of the constraint expresses that the referenced member is protected and declared in a superclass of the current class that belongs to a different package.

For example, consider a member reference $\mathsf{C}\,.\,\mathsf{n}\!:\!\mathsf{d}$ embedded in a getfield, putfield, invokevirtual, or invokespecial that occurs in a class $s = \langle \mathsf{S}, \mathcal{L} \rangle$. Suppose that D is the inferred target type of that instruction, with $\mathsf{D} \neq \mathsf{S}$. The bytecode verifier generates the conditional subtype constraint

$$\text{“}\textbf{if } \mathit{ProtCond}(\mathsf{C}\,.\,\mathsf{n}\!:\!\mathsf{d}^{\mathcal{L}}, \mathsf{S}^{\mathcal{L}}) \textbf{ then } \mathsf{D}^{\mathcal{L}} < \mathsf{S}^{\mathcal{L}}\text{”}$$

The condition $\mathit{ProtCond}(\mathsf{C}\,.\,\mathsf{n}\!:\!\mathsf{d}^{\mathcal{L}}, \mathsf{S}^{\mathcal{L}})$ holds if and only if the member to which $\mathsf{C}\,.\,\mathsf{n}\!:\!\mathsf{d}$ resolves via $\mathcal{L}$ (i.e. the member with name n and descriptor d found searching from $\mathsf{C}^{\mathcal{L}}$) is protected and declared in a superclass of $\mathsf{S}^{\mathcal{L}}$ that belongs to a different package.

Conditional subtype constraints are integrated with the unconditional (equality [7, 6] and subtype [10, 3]) constraints in the JVM. The satisfaction of pending



constraints is re-checked whenever classes are loaded and members are resolved. It is necessary to consider the transitive closure of all the constraints, conditional and unconditional.

Even though conditional subtype constraints avoid premature loading altogether, their generation, maintenance, checking, and integration with the other loading constraints require additional machinery in the JVM that would not be needed otherwise. Thus, there is a trade-off between eager resolution and conditional subtype constraints.

Corpus Measurements

The 3,915 classes (and interfaces) in the java and javax packages of SDK 1.4 require 68,621 static checks for protected members. That is the total number of triples $(s, \mathsf{C}.\mathsf{n}:\mathsf{d}, \mathsf{D})$ where s is a class in the java or javax package, $\mathsf{C}.\mathsf{n}:\mathsf{d}$ is a reference embedded in a getfield, putfield, invokevirtual, or invokespecial occurring in s, and D is one of the types inferred at that instruction by the bytecode verifier[2]. Other counts are possible, e.g. the total number of pairs (i, D) where i is an occurrence of getfield, putfield, invokevirtual, or invokespecial in some class in java or javax and D is one of the types inferred at i; however, what seems important in the data below is the relative percentages, which are probably largely invariant to the exact count used.

Of the 68,621 checks:

- 30,598 (45%) can be checked as explained in Subsection "Current Class as Target Type" because the target class coincides with the current class;
- 31,571 (46%) do not cause any loading because the class referenced in the member reference is the current class or its direct superclass, as described in Subsection "Resolution not Causing Loading";
- *all* 68,621 can be checked as explained in Subsection "Necessary Condition for Subtype Check" because the necessary condition for the restriction on protected access does not hold.

Of course, there is overlap among these three sets of checks.

A reasonable implementation of the bytecode verifier could perform those checks as follows:

- 30,598 of the total 68,621 (45%) by recognizing that the target class coincides with the current class;

[2]While many bytecode verifiers, including the one in SDK, infer only one type for every instruction (by merging types from converging paths), the measurements reported here were taken with a bytecode verifier that infers multiple types, for increased precision.

- 1,085 of the remaining 38,023 (1% of the total, 3% of the remaining) by recognizing that the referenced class is the current one or its direct superclass, so that resolution does not cause any loading;
- *all* the remaining 36,938 (54% of the total) by inspecting the superclasses and discovering that the necessary condition does not hold.

Thus, no eager resolution or conditional subtype constraint is necessary.

5 A BUG IN EARLIER VERSIONS OF SDK

The checking of protected member access in earlier versions of SDK is incorrect; see [8], which includes a list of affected versions. There are programs where the access to a protected member is illegal but allowed, and programs where the access is legal but disallowed.

Illegal Programs Accepted

Consider the program in Figure 3, consisting of three classes $c = \langle \mathsf{p.C}, \mathcal{L} \rangle$, $s = \langle \mathsf{q.S}, \mathcal{L} \rangle$, and $d = \langle \mathsf{q.D}, \mathcal{L} \rangle$, where $\mathcal{L}$ is some loader whose attributes are irrelevant to this example; c belongs to the package $p = \langle \mathsf{p}, \mathcal{L} \rangle$, while s and d belong to $q = \langle \mathsf{q}, \mathcal{L} \rangle$. The classes are declared in three files (compilation units) C.java, S.java, and D.java. The relation among these three classes is depicted in Figure 4, where the unlabeled double-line arrows indicate direct superclasses and, as in Figure 2, the single-line arrow denotes resolution.

The program must be compiled in two steps. First, the field declaration in D.java is uncommented and the three files are compiled, via "javac -d . *.java" (the option "-d ." puts the class files in subdirectories p and q, as needed for execution). Then, the field declaration in D.java is commented out and D.java is re-compiled, via "javac -d . D.java". An attempt to compile the three files together with the field declaration in D.java commented out would cause an error because the field accessed by q.S is protected and declared in the superclass p.C that belongs to a different package, but the type q.D of the target object is neither q.S nor a subclass of it.

When the program is run, via "java q.S", the number 3 is printed on screen. However, the access is illegal: the field, inherited by d, is protected and declared in the superclass c of s in a different package, but the class d of the target object does not satisfy $d \leq s$.

Another example of illegal program that gets accepted is in Figures 5, 6, and 7, where there are two classes named C, one in C.java in the current directory and the other in a homonymous file in a subdirectory dir1. The class Main creates a class loader $\mathcal{L}_1$ of class DelegatingLoader1 and a class loader $\mathcal{L}_2$ of class DelegatingLoader2 and uses the latter to load $\mathsf{S}^{\mathcal{L}_2}$. $\mathcal{L}_2$ delegates the loading of $\mathsf{I}^{\mathcal{L}_2}$ to $\mathcal{L}_1$ and the loading

```
***** C.java:
package p;
public class C {
  protected int f = 3;
}

***** S.java:
package q;
public class S extends p.C {
  public static void main(String[] args) {
    System.out.println((new D()).f);
  }
}

***** D.java:
package q;
public class D extends p.C {
  // public int f;
}
```

Figure 3: Illegal program accepted

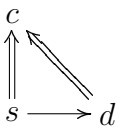


Figure 4: Classes in Figure 3

of any class whose name differs from S and I to the system class loader $\mathcal{L}_0$, while it loads $\mathsf{S}^{\mathcal{L}_2}$ from the subdirectory dir2. $\mathcal{L}_1$ delegates the loading of any class whose name differs from I and C to $\mathcal{L}_0$, while it loads $\mathsf{I}^{\mathcal{L}_1}$ and $\mathsf{C}^{\mathcal{L}_1}$ from the subdirectory dir1. $\mathcal{L}_0$ always loads classes from the current directory. See the SDK API documentation for details.

This program must be compiled in multiple steps. First, extends C in J.java is commented out and the files in the current directory are compiled, via "javac *.java". Then, the files in the subdirectories are compiled, via "javac */*.java". Finally, extends C in J.java is uncommented and the file is re-compiled along with C.java, via "javac J.java C.java". An attempt to compile all the files together would cause an error because of the cyclic inheritance between C and J and/or because of the duplicate class declaration for C. The reason to re-compile also C.java, besides J.java, after uncommenting extends C, is to prevent the compiler from using dir1/C.java instead and causing an cyclic inheritance error.

The relation among the first five classes in Figure 5 is depicted in Figure 8 (the other classes comprising the program are not shown because they do not contribute to the illustration of the problem). Since $\mathcal{L}_2$ delegates to $\mathcal{L}_1$ the loading of $\mathsf{I}^{\mathcal{L}_2}$, the direct superclass of $\langle \mathsf{S}, \mathcal{L}_2 \rangle$ is $\langle \mathsf{I}, \mathcal{L}_1 \rangle$. The direct superclass of $\langle \mathsf{I}, \mathcal{L}_1 \rangle$ is $\langle \mathsf{C}, \mathcal{L}_1 \rangle$, because $\mathcal{L}_1$ does not delegate the loading of $\mathsf{C}^{\mathcal{L}_1}$. Since $\mathcal{L}_1$ delegates to $\mathcal{L}_0$ the loading of $\mathsf{J}^{\mathcal{L}_1}$, the direct superclass of $\langle \mathsf{C}, \mathcal{L}_1 \rangle$ is $\langle \mathsf{J}, \mathcal{L}_0 \rangle$. The direct superclass of $\langle \mathsf{J}, \mathcal{L}_0 \rangle$ is $\langle \mathsf{C}, \mathcal{L}_0 \rangle$, because $\mathcal{L}_0$ does not delegate loading. The reference C from $\langle \mathsf{S}, \mathcal{L}_2 \rangle$ is resolved to $\langle \mathsf{C}, \mathcal{L}_0 \rangle$, because $\mathcal{L}_2$ delegates to $\mathcal{L}_0$ the loading of $\mathsf{C}^{\mathcal{L}_2}$. Note that the intermediate class $\langle \mathsf{I}, \mathcal{L}_1 \rangle$ between $\langle \mathsf{S}, \mathcal{L}_2 \rangle$ and $\langle \mathsf{C}, \mathcal{L}_1 \rangle$ is necessary to create the situation in which $\langle \mathsf{S}, \mathcal{L}_2 \rangle$ has a superclass named C that differs from the class that the reference C resolves to; without the intermediate class, the reference C in $\langle \mathsf{S}, \mathcal{L}_2 \rangle$ would resolve to the direct superclass $\langle \mathsf{C}, \mathcal{L}_1 \rangle$. The intermediate class $\langle \mathsf{J}, \mathcal{L}_0 \rangle$ between $\langle \mathsf{C}, \mathcal{L}_1 \rangle$ and $\langle \mathsf{C}, \mathcal{L}_0 \rangle$ is necessary to create the situation in which $\langle \mathsf{S}, \mathcal{L}_2 \rangle$ has two distinct superclasses both named C; without the intermediate class, the direct superclass reference C in $\langle \mathsf{C}, \mathcal{L}_1 \rangle$ would resolve to the same class, causing a cyclic inheritance error.

When the program is run, via "java Main", the number 3 is printed on screen. However, the access is illegal: the field is protected and declared in the superclass $\langle \mathsf{C}, \mathcal{L}_0 \rangle$ of $\langle \mathsf{S}, \mathcal{L}_2 \rangle$ in a different package[3], but the class $\langle \mathsf{C}, \mathcal{L}_0 \rangle$ of the target object does not satisfy $\langle \mathsf{C}, \mathcal{L}_0 \rangle \leq \langle \mathsf{S}, \mathcal{L}_2 \rangle$. The homonymous public field declared in $\langle \mathsf{C}, \mathcal{L}_1 \rangle$ is not accessed.

Legal Programs Rejected

Consider the program in Figures 9 and 10, where there are two classes named C, one in C.java in the current directory and the other in a homonymous file in a

[3] As explained in Section 2, at run time an unnamed package is uniquely identified by a loader and all the classes in a package have the same defining loader. Since $\mathcal{L}_0 \neq \mathcal{L}_2$, the classes $\langle \mathsf{C}, \mathcal{L}_0 \rangle$ and $\langle \mathsf{S}, \mathcal{L}_2 \rangle$ belong to different (unnamed) packages.

```
***** C.java:
public class C {
  protected int f = 3;
}

***** J.java:
public class J extends C {
}

***** dir1/C.java:
public class C extends J {
  public int f;
}

***** dir1/I.java:
public class I extends C {
}

***** dir2/S.java:
public class S extends I {
  public static void m() {
    System.out.println((new C()).f);
  }
}

***** Main.java:
public class Main {
  public static void main(String args[])
    throws Exception {
    DelegatingLoader1 loader1 = new DelegatingLoader1();
    DelegatingLoader2 loader2 = new DelegatingLoader2(loader1);
    Class s = loader2.loadClass("S");
    Object[] arg = {};
    Class[] argClass = {};
    s.getMethod("m",argClass).invoke(null,arg);
  }
}
```

Figure 5: Another illegal program accepted (part 1 of 3)

```
***** Loader.java:
import java.io.*;

public abstract class Loader extends ClassLoader {
  private String directory;
  protected Loader(String dir) {
    directory = dir;
  }
  protected Class loadClassFromFile(String name)
    throws IOException {
    File target = new File(directory + "/" + name + ".class");
    int bytecount = (int) target.length();
    byte[] buffer = new byte[bytecount];
    FileInputStream f = new FileInputStream(target);
    f.read(buffer);
    f.close();
    return (defineClass(name, buffer, 0, bytecount));
  }
}

***** DelegatingLoader1.java:
import java.io.*;

public class DelegatingLoader1 extends Loader {
  public DelegatingLoader1() {
    super("dir1");
  }
  public Class loadClass(String name)
    throws ClassNotFoundException {
    try {
      Class prevLoaded = this.findLoadedClass(name);
      if (prevLoaded != null)
        return prevLoaded;
      else if (name.equals("I") || name.equals("C"))
        return this.loadClassFromFile(name);
      else
        return this.findSystemClass(name);
    } catch (IOException e) {
      throw new ClassNotFoundException();
    }
  }
}
```

Figure 6: Another illegal program accepted (part 2 of 3)

```
***** DelegatingLoader2.java:
import java.io.*;

public class DelegatingLoader2 extends Loader {
  private DelegatingLoader1 loader1;
  public DelegatingLoader2(DelegatingLoader1 loader1) {
    super("dir2");
    this.loader1 = loader1;
  }
  public Class loadClass(String name)
    throws ClassNotFoundException {
    try {
      Class prevLoaded = this.findLoadedClass(name);
      if (prevLoaded != null)
        return prevLoaded;
      else if (name.equals("S"))
        return this.loadClassFromFile(name);
      else if (name.equals("I"))
        return loader1.loadClass(name);
      else
        return this.findSystemClass(name);
    } catch (IOException e) {
      throw new ClassNotFoundException();
    }
  }
}
```

Figure 7: Another illegal program accepted (part 3 of 3)

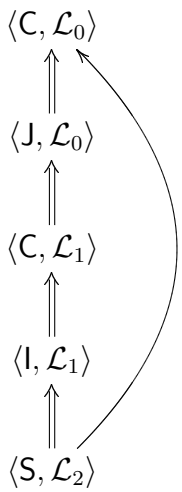


Figure 8: First five classes in Figure 5

```
***** C.java:
public class C {
  protected int f;
}

***** I.java:
public class I extends C {
}

***** dir/S.java:
public class S extends I {
  public static void m() {
    System.out.println((new C()).f);
  }
}

***** dir/C.java:
public class C {
  public int f = 5;
}

***** Main.java:
public class Main {
  public static void main(String args[])
    throws Exception {
    DelegatingLoader loader = new DelegatingLoader();
    Class s = loader.loadClass("S");
    Object[] arg = {};
    Class[] argClass = {};
    s.getMethod("m", argClass).invoke(null, arg);
  }
}
```

Figure 9: Legal program rejected (part 1 of 2)

subdirectory dir. The class Main creates a class loader $\mathcal{L}$ of class DelegatingLoader (class Loader is the same as in Figure 6) and uses it to load $\mathsf{S}^{\mathcal{L}}$. $\mathcal{L}$ delegates the loading of any class whose name differs from S and C to the system class loader $\mathcal{L}_0$, while it loads $\mathsf{S}^{\mathcal{L}}$ and $\mathsf{C}^{\mathcal{L}}$ from the subdirectory dir. $\mathcal{L}_0$ always loads classes from the current directory.

Since there are two classes with the same name, this program must be compiled in two pieces, one with C.java and the other with dir/C.java. For example, the two commands "javac *.java" and "javac */*.java" accomplish the task.

The relation among the first four classes in Figure 9 is depicted in Figure 11 (the other classes comprising the program are not shown because they do not contribute to the illustration of the problem). Since $\mathcal{L}$ delegates to $\mathcal{L}_0$ the loading of $\mathsf{I}^{\mathcal{L}}$, the direct superclass of $\langle \mathsf{S}, \mathcal{L} \rangle$ is $\langle \mathsf{I}, \mathcal{L}_0 \rangle$. The direct superclass of $\langle \mathsf{I}, \mathcal{L}_0 \rangle$ is $\langle \mathsf{C}, \mathcal{L}_0 \rangle$, because

```
***** DelegatingLoader.java:
import java.io.*;

public class DelegatingLoader extends Loader {
  public DelegatingLoader() {
    super("dir");
  }
  public Class loadClass(String name)
    throws ClassNotFoundException {
    try {
      Class prevLoaded = this.findLoadedClass(name);
      if (prevLoaded != null)
        return prevLoaded;
      else if (name.equals("S") || name.equals("C"))
        return this.loadClassFromFile(name);
      else
        return this.findSystemClass(name);
    } catch (IOException e) {
      throw new ClassNotFoundException();
    }
  }
}
```

Figure 10: Legal program rejected (part 2 of 2)

$\mathcal{L}_0$ does not delegate loading. Even though $\langle \mathsf{C}, \mathcal{L}_0 \rangle$ is an (indirect) superclass of $\langle \mathsf{S}, \mathcal{L} \rangle$, the reference C from $\langle \mathsf{S}, \mathcal{L} \rangle$ is resolved to $\langle \mathsf{C}, \mathcal{L} \rangle$, because $\mathcal{L}$ does not delegate the loading of $\mathsf{C}^{\mathcal{L}}$. Note that the intermediate class $\langle \mathsf{I}, \mathcal{L}_0 \rangle$ between $\langle \mathsf{S}, \mathcal{L} \rangle$ and $\langle \mathsf{C}, \mathcal{L}_0 \rangle$ is necessary to create the situation where $\langle \mathsf{S}, \mathcal{L} \rangle$ has a superclass named C that differs from the class that the reference C resolves to.

The class $\langle \mathsf{C}, \mathcal{L}_0 \rangle$ is a superclass of $\langle \mathsf{S}, \mathcal{L} \rangle$ that belongs to a different package and declares a protected field. However, the program is legal because the field accessed by $\langle \mathsf{S}, \mathcal{L} \rangle$ is the public one in $\langle \mathsf{C}, \mathcal{L} \rangle$. So, the number 5 should be printed on screen. Instead, when the program is run, via "java Main", it is rejected with a verification error for $\langle \mathsf{S}, \mathcal{L} \rangle$.

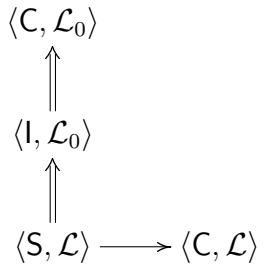


Figure 11: First four classes in Figure 9

Likely Cause of the Bug

The buggy versions of SDK handle other cases correctly. For example, if new D() is replaced with new C() in Figure 3, a verification error correctly signals that the field cannot be accessed, because it is protected and declared in a superclass in a different package, but $\mathsf{p.C}^{\mathcal{L}} \leq s$ does not hold. (In order to compile S.java with new C() in place of new D() it is first necessary to make the field in C.java public, then compile the two files, then change the field back to protected, and finally re-compile C.java.)

It is likely that the bytecode verifier in the buggy versions of SDK uses the following strategy to check accessibility of protected members. Suppose that the member reference in the current class s with defining loader $\mathcal{L}$ is $\mathsf{C.n:d}$. The bytecode verifier inspects the superclasses of s to see if any of them has name C. If none of them has name C, access is considered legal. Otherwise, a member with name n and descriptor d is searched in that class and its superclasses. If one is found but it is not protected, access is considered legal. If it is protected, the package of the class where the member is declared is compared with the package of s. If they are the same, access is considered legal. Otherwise, the inferred target type D is checked to satisfy $\mathsf{D}^{\mathcal{L}} \leq s$.

In the bytecode compiled from Figure 3, the getfield in class s references q.D.f: int, because during the first step of compilation (when the field declaration in D.java is uncommented) the compiler resolves the field reference to the field declared in D.java. During the bytecode verification of s, the strategy conjectured above finds that no superclass of s has name q.D, thus erroneously concluding that the access is legal.

In the bytecode compiled from Figure 5, the getfield in class $\langle \mathsf{S}, \mathcal{L}_2 \rangle$ references C.f:int. During the bytecode verification of $\langle \mathsf{S}, \mathcal{L}_2 \rangle$, the strategy conjectured above finds that the superclass $\langle \mathsf{C}, \mathcal{L}_1 \rangle$ of $\langle \mathsf{S}, \mathcal{L}_2 \rangle$ has name C and that it contains a field with name f and descriptor int. Since the field is not protected, the strategy erroneously concludes that the access is legal. Apparently, the strategy iterates through superclasses upwards (i.e. from subclass to superclass); otherwise, it would find the protected field in class $\langle \mathsf{C}, \mathcal{L}_0 \rangle$ first and correctly conclude that the access is illegal.

In the bytecode compiled from Figure 9, the getfield in class $\langle \mathsf{S}, \mathcal{L} \rangle$ references C.f:int. During the bytecode verification of $\langle \mathsf{S}, \mathcal{L} \rangle$, the strategy conjectured above finds that the superclass $\langle \mathsf{C}, \mathcal{L}_0 \rangle$ of $\langle \mathsf{S}, \mathcal{L} \rangle$ has name C and contains a protected field with name f and descriptor int. Since $\langle \mathsf{S}, \mathcal{L} \rangle$ and $\langle \mathsf{C}, \mathcal{L}_0 \rangle$ belong to different (unnamed) packages (because $\mathcal{L} \neq \mathcal{L}_0$), the strategy erroneously concludes that the access if illegal.

On the other hand, if new D() is replaced with new C() in Figure 3, the getfield in class s references p.C.f:int. During the bytecode verification of s, the strategy conjectured above finds that the superclass c of s has name p.C, has a field with name f and descriptor int, and belongs to a different package from s. Thus, the strategy checks whether $c \leq s$ (the inferred target type of the getfield is clearly p.C,

which resolves to c, because the direct superclass of s has name p.C), which is not the case, thus correctly concluding that the access is illegal.

6 CONCLUSIONS

This paper illustrates how to check protected member access in the JVM, which is not explained in [7] and, to the author's knowledge, has been completely neglected in the research literature. This aspect of enforcing type safety in the JVM is rather subtle and its correct implementation is not straightforward, as also evidenced by the bug in earlier versions of SDK exposed in this paper.

The best way to correctly and efficiently check protected member access seems to be the following: first, use the strategies described in Subsection "Limited Resolution" of Section 4; then, deal with the remaining checks by eagerly resolving member references or by generating conditional subtype constraints.

If the classes in the java and javax packages of SDK 1.4 are representative of typical Java programs in the usage of protected members, the results reported in Subsection "Corpus Measurements" of Section 4 suggest that all or most checks can be performed without any eager resolution or conditional subtype constraints. Anyhow, there is a trade-off between eagerly resolving member references and generating conditional subtype constraints: the former is simpler and requires less machinery in the JVM, while the latter supports lazier class loading.

REFERENCES

[1] Ken Arnold, James Gosling, and David Holmes. *The Java™ Programming Language*. Addison-Wesley, third edition, 2000.

[2] Lawrence Brown. Protected access. In Data Structures Course Notes, available at http://www.apl.jhu.edu/Classes/Notes/LMBrown/resource/ProtectedAccess.pdf.

[3] Alessandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency and Computation: Practice and Experience*, 13(13):1153–1171, November 2001.

[4] Li Gong. *Inside Java™ 2 Platform Security*. Addison-Wesley, 1999.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley, second edition, 2000.

[6] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java™ virtual machine. In *Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, volume 33, number 10 of *ACM SIGPLAN Notices*, pages 36–44, October 1998.

[7] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[8] Sun Microsystems. Sun alert notification #50083, January 2003. Available at http://sunsolve.sun.com/search/document.do?assetkey=1-26-50083-1.

[9] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, 1998.

[10] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *Proc. 15th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, volume 35, number 10 of *ACM SIGPLAN Notices*, pages 325–336, October 2000.

[11] Anonymous Reviewer. Private communication, March 2005.

[12] Frank Yellin. Private communication, August 2002.

ABOUT THE AUTHORS

Alessandro Coglio has been a Computer Scientist at Kestrel Institute since 1998, working on formal methods and their application to Java. He has also been a Computer Scientist at Kestrel Technology LLC since 2001, working on technology transfer. Prior to joining Kestrel, Mr. Coglio was a Consulting Researcher for the Department of Informatics, Systems, and Telecommunications of University of Genoa (Italy), working on theorem proving, Petri nets, discrete event systems, and artificial emotions. Mr. Coglio received a Master degree in Computer Science Engineering from University of Genoa in 1996. His thesis, in the field of theorem proving, was in collaboration with Stanford University (Palo Alto, California), where he spent two months in 1996. During his middle- and high-school years, one of Mr. Coglio's pastimes was programming videogames in Assembly.