McGill University
School of Computer Science
COMP 523

Project Report: Efficient Java bytecode verification by the means of proof-carrying code

Eric Bodden

April 11, 2006

Contents

List of Figures

Abstract

Bytecode verification is known to be a crucial component in the overall security model of Java programs, in particular applets, a term often used for mobile code serving on the Web, on embedded devices or smart cards. In those environments executable code is often sent over untrusted channels or even downloaded from completely untrusted sources. Hence it is critical, that every piece of code to be executed by a Java Virtual Machine (JVM) is proven to not be able to cause any harm. This is what bytecode verification tries to achieve. In this article we comment on the different approaches to Java bytecode verifications and mention some of the shortcomings of the current reference implementation by Sun. In particular we discuss the problem of complexity-based denial-of-service attacks and explain how it is solved in the upcoming release of Java 6. This new version of Java uses techniques from proof-carrying code to exchange bytecode verification for faster bytecode checking with the same safety properties. Not only do we discuss in detail how this technique works, also we give pointers to how other problems of similar flavor could be addressed with the same technique.

1 Introduction and motivation

The Java security model consists of two major components:

1. The bytecode verifier.
2. The Security Manager.

The former has always been part of the official Java specification, simply because it serves the purpose of ensuring both, *security* and *safety*, where with security we mean the ability to prevent a system from failing due to intentional attacks and with safety we mean the ability to avoid data loss or program termination due to unintentional programming errors. The bytecode verifier is the crucial component in assuring type safety of Java bytecode and consequently is essential in order to prove that no unintentional data loss or control flow can occur, for example due to ill-typed programs. With the growing number of different Java virtual machines, various compilers which compile even from programming languages entirely different from Java to Java bytecode[1] and new execution environment such as Java Card[2] , this verification is today even more important than ever. Bytecode verification must be reliable and - as we argue in this article - must also be fast in order to not open a Java application to attacks from an untrusted source.

The second component, the Security Manager was added around 1996 in a very ad-hoc manner, when it became clear that the pure verifiability of a piece of bytecode is not strong enough to give all the guarantees one would like to have with respect to security (not safety). Even well-typed programs can access APIs which they not ought to. Sometimes Java's language features are too weak to prevent an applet from accessing certain forbidden APIs. Hence the main feature of the Security Manager is to allow or restrict access to APIs and resources or to allow code being loaded only from specific trusted sources. Further treatment of those mechanisms are however out of the scope of this work. Here we rather focus on the task of the bytecode verifier and explore its internal workings and its shortcomings. The remainder of this paper is organized as follows.

After a short introduction in Section 1, we first investigate some shortcoming with respect to completeness (Section 2.2). Then in Section 3, we investigate a problem of another flavor which does not concern effectiveness but efficiency of the verification. It is the problem of complexity based denial-of-service attacks which was present in all versions of the Sun bytecode verifier up to Java version 5.

We explain in detail what such an attack could look like and why it works, respectively what negative properties of the current verification algorithm it exploits. In Java 6, bytecode verification was replaced by bytecode checking, a relatively new approach based on techniques of proof-carrying code. Using those concepts, it is possible to check Java bytecode in linear time and space and hence all vulnerability to the aforementioned attack is eliminated. In Section 4 we explain how this approach works and give some examples to ease the understanding of this new method. In Section 4.3, we explain how the same techniques could be useful for addressing other problems in the field, such as the ones we mention in Section 2.2.

We conclude our survey in Section 5.

[1] See `http://www.robert-tolksdorf.de/vmlanguages.html` for a list of such programming languages.
[2] See `http://java.sun.com/products/javacard/` for information on the Java Card specification.

Phase	Characterization	Time
1	Basic class-file representation	load
2	Class-component integrity except for code-array	link
3	Static type reconstruction (bytecode verification)	link
4	Dynamic checks	run

Table 1: The four phases of Java class-file verification

2 Java bytecode verification

2.1 The current approach

In a Java environment, executable bytecode is always dynamically loaded and linked. Consequently, all bytecode verification has to be deferred at least to the load time of a class. In the general setting, a Java class can be loaded at any time in a given program. Hence, the verification takes in fact place while the application is already running.

In order to validate dynamically loaded code, this code goes through a sequence of four phases when being loaded into the Java Virtual Machine (JVM) which are shown in Table 1.

Phases 1 and 2 check the static integrity of the class file, i.e. verify that all necessary components exist and that they do not contradict each other. Those checks are fully static and cheap in computation time.

Phase 3 is the verification phase which is most involved: It focuses on the code-array, i.e. the part of the class file holding the actual executable code. In particular it checks the correct usage of the explicit computation stack which is present in the Java Virtual Machine and which is used by the bytecode during runtime to perform its computations. Amongst the checks that are performed are the following:

- Nothing should be popped from the empty stack.
- Maximal stack size may not be exceeded.
- Operands must conform to the required type of each operator.
- All local variables (= stack positions) must be consistently typed.

The last step is crucial. With consistent typing we mean that the verifier has to prove that regardless which path is taken during runtime, the computation stack is always of the same size and holds values of "compatible types". In some situations such as explicit casts this cannot be proven statically. Consequently, explicit runtime checks are inserted in order to check the typing constraints during runtime (step 4).

Since Java bytecode might encode arbitrary loop structures, the typing information has to be computed in an iterative manner, and in fact it is computed by an iterative dataflow analysis interpreting all Java bytecode operations over the abstract domain of types.

For example, when investigating the bytecodes `iconst_1, iconst_2` (pushing the integer constants 1 and 2 on the stack), we can infer that the two topmost stack locations must in fact be of type `int`. Consequently, a subsequent execution of `imul` (multiplying both) would be well-typed. If at any point, the verifier encounters a type incompatibility, verification fails and the current method (and hence class) is rejected, i.e. will never be executed.

Difficulties may occur whenever control flow splices and merges in a program. Since the control flow graph of a Java application is a static overapproximation of all possible computation paths, there might be several possible control flows leading to the same statement. Whenever this is the case, the verifier has to make sure that for any two possible types τ_1 and τ_2 for a local variable or stack position v, those are both taken into account. It does so by computing the most general common type of τ_1, τ_2 and relating this to v. In the following, we call this a *(control flow) merge*.

2.2 Shortcomings with respect to completeness

In his 2001 survey paper [6], Xavier Leroy notes several shortcomings of the current[3] implementation of Sun's bytecode verifier, which we briefly want to discuss in this section.

[3]With "current" we here always refer to Java version 5.

2.2.1 Interface types

Each Java class can only extend one single superclass. This is commonly being referred to as single inheritance. Nevertheless, each class type can have several supertypes by implementing multiple interfaces. The normal class hierarchy forms a tree structure and hence trivially a semi-lattice where each two classes have exactly one closest common supertype. When performing a merge as described above, it is straightforward to find this supertype and assign it in the merge operation.

With the introduction of interfaces, things get more complicated: The type system does not form a semi-lattice any more. (See [6] for details.) Consequently, some types may not be comparable and a merge would always result in the assignment of `java.lang.Object` (the root of all reference types) which would certainly not be a satisfactory approximation. As Leroy states [6], this problem can easily be overcome by introducing dummy types: When a class implements interfaces `I` and `J`, it is made a subtype of the artificial type `IandJ`. It could of course be necessary to generate a large number of such dummy types, however, still the very same fast algorithms could be used as without the presence of interface types, since the resulting systems forms a semi-lattice again.

Sun however opted for a cheaper approach: They simply ignore the interface types completely, treating all interface types as `java.lang.Object`. As a consequence, bytecode verification does not decide whether a bytecode `invokeinterface` $I.m$ will really receive a class type implementing I and consequently the method m at runtime. Hence, the VM performs dynamic checks at each such statement, obviously paying a runtime penalty on each such invocation.

2.2.2 Constructor calls

A constructor call in Java source code is on the bytecode level broken into two distinct components:

1. The creation of the new object (bytecode `new`).
2. The invocation of the constructor, i.e. one of the `<init>` methods, just in the same way as any private method (bytecode `invokespecial`).

Due to this separation, the verifier has to assure that any newly created object is initialized before it is used. This problem is complicated by the fact that references to uninitialized objects are usually aliased (see [6] for details). Consequently, in order to mark all those aliases as "initialized" after a constructor has been executed, the verifier would have to do a full alias analysis, which can become quite expensive. Sun again takes a shortcut here, rejecting some programs which actually adhere to the rule, i.e. do on every path call a constructor before using the object in question.

2.2.3 Java subroutines

Java subroutines (JSR) are used in mobile versions of Java in order to encode code which can be commonly executed coming *from different places*. This is for example the case in try/finally constructs where code has to be executed regardless of whether an exception was thrown or not (i.e. one has multiple entry points). A JSR is jumped to by the bytecode `jsr` $< linenumber >$ and is exited by the bytecode `ret` which *returns to the statement behind the* `jsr` *call*.

This means that the typing information in the subroutine itself not only depends on the context from which it was called but that in turn it also affects the typing information of the calling context. Consequently, it would actually be necessary to implement a fully polyvariant (i.e. context-sensitive) type reconstruction, where the JSR body is analyzed separately in each different calling context. Sun however opted for a faster monovariant analysis instead, again rejecting bytecode which is according to the semantics actually well-typed. A better approach to this problem will be discussed in Section 4.3.

2.2.4 Summary

We have seen three different shortcomings of the Java bytecode verification as it is currently implemented in Sun's JVM. It stands out that in all three situations, Sun opted for an easy and *fast* solution. So why is performance of the analysis so important? This question is easy to answer. As explained in Section 1, bytecode verification takes place at class load time, i.e. while the application is already partially running. So if there were parts of the verification algorithm with a too high complexity, this would make the system vulnerable to denial-of-service attacks. An attacker could have well-typed but still malicious code loaded into the VM, whose only purpose would be to keep the bytecode verifier busy for as long

as possible. This is even made more critical by the fact that verification takes place deep inside the VM and cannot be intercepted or aborted at all without shutting down the VM as a whole.

It turns out that the current implementation of Sun's bytecode verifier indeed exposes such a vulnerability, and indeed it turns out to be even an intrinsic problem of the verification algorithm employed. Gal et al. [3] discuss the problem in detail and so do we in the following section.

3 A complexity-based denial-of-service attack

3.1 The idea

The problem discussed in the following scenario is *not* the possible acceptance of ill-typed programs, nor to the possible rejection of well-typed ones. The problem is rather that even when a program is well-typed and it is verifiable, it can impose such a burden on the verifier that its verification algorithm does not terminate in a reasonable amount of time even on fast machines.

The problem arises with the iterative type reconstruction algorithm for stack locations as mentioned in Section 1. When loops within the bytecode are arranged in a certain inconvenient order in the bytecode, this can lead to a chain of basic blocks[4] where each block depends on the analysis information of its *successor*. An iterative dataflow analysis however, is in each iteration able to only propagate information to each *immediate predecessor*. Consequently, the full verification of such a system with n basic blocks would take n iterations over those blocks, resulting in an overall complexity of n^2.

3.2 Example

An example program that exposes such complexity is shown in Figure 1. At the beginning, we load an integer constant on the stack and store it into local variable 1 (LV1). This means that at the `goto`[5] statement, the typing information for LV1 is `int`. When reaching the `goto` statement, the verifier does not actually follow this jump but resumes verification in sequential order, continuing with the block labeled with `Ln`. The following code up to label `L0` is all type checked under the assumption that the type of LV1 is `int`. When reaching `L0`, we first see a conditional branch: We load 0 on the stack and then branch to `L1` if the top value on the stack is equal to 0. Although this can be statically determined, the verifier does not evaluate the condition and neither follows the jump. It simply remembers that now there is an additional control flow edge to `L1`.

In the following statements, a null pointer is loaded onto the stack and stored into LV1, altering its typing information to `null_type`. Due to the jump `goto L0`, this invalidates now the typing information for the block at `L0`. Hence another iteration is necessary.

In the next iteration, when reaching the block at `L0` the verifier will now have two typing informations, the former $type(LV1) =$`int` and now also $type(LV1) =$`null_type`. The join of both types is the artificial type $\top$, which indicates that the variable is currently not accessible (because it could hold a value of both, an integer or a reference type). The block at `L0` is now reevaluated with this new typing information, which invalidates the typing information for the block at `L1`, forcing another iteration to be necessary. The verification continues that way and in the end successfully terminates after n iterations.

3.3 Experiments

Gal et al. [3] conducted experiments showing that there is indeed an increase in verification time in the order of n^2 for a class with n such basic blocks.

Moreover, bytecode of such a form can be very well compressed. Hence by using compressed JAR files, one can send a class file with an almost arbitrarily large number of such basic blocks to a Java VM. Experiments we conducted ourselves on a Pentium M 1.4 GhZ with Sun's Hotspot VM Version 1.5.0_06 showed that a JAR file of only 2.7 kilobytes can already keep the verifier busy for around 3 seconds. The experiment can be tried out on the following website: `http://www.sable.mcgill.ca/~ebodde/javados/`

We also gave the same class file to the Soot [9] bytecode analysis and optimization framework. Soot implements the same type reconstruction in its `TypeAssigner`. It took about 20 minutes to have the local types assigned on one of our compute servers. This proofs that the problem is really due to an intrinsic

[4] A basic block is a piece of straightline code without any jumps. Any well-structured program can hence be decomposed into a graph over basic blocks.

[5] The `goto_w` statement is just a normal `goto` statement with the only exception that a 32 bit register is being used for the offset instead of a 16 bit one.

```
  iconst_0
  istore_1
  goto_w L0

Ln:
  return

  ...

L1:
  iconst_0
  ifeq L2
  aconst_null
  astore_1
  goto L1

L0:
  iconst_0
  ifeq L1
  aconst_null
  astore_1
  goto L0
```

Figure 1: Bytecode exposing the worst-case complexity of n^2

complexity of the algorithm involved. Today almost any Java bytecode verifier and bytecode analysis tool is based on Sun's reference implementation [7]. Consequently, all those tools are vulnerable to such attacks. However, admittedly, only in the setting of a Java virtual machine, this kind of vulnerability exposes a security risk.

3.3.1 Trying to reconstruct the source code

In a further experiment, we tried to see if it was possible to find well-structured Java code that would induce the same complexity. As it turned out, the abovementioned bytecode is *not* equivalent to any well-structured Java program. Hence one can conclude that such malicious code would really have to be generated by purpose and hence, we are talking about a security, not a safety issue — at least not in the most common setting where Java bytecode is generated from Java source code.

In a first experiment we converted the bytecode to *Jimple* the principal internal three-address representation of Java bytecode in Soot. The code, when being produced with the local packer `jb.lp` enabled, looks as shown in Figure 2.

We see the same basic block structure as before with the same control flow dependencies. The type assigner inferred that the register No. Current versions of Dava perform constant folding and hence recognize the code as being dead, eliminating the whole body completely.

In order to circumvent this problem of generating obviously dead code, we changed the Jimple code, exchanging each comparison `0 == 0` for `0 == z0`. For the verifier, this makes no difference, since the expression is not evaluated. The decompiled source however becomes the one shown in Figure 4, which compiles due to the lack of constant folding in the `javac` compiler.

However, as we expected, those loops are now compiled in *sequential* order, and not in the inverse order that caused the quadratic number of iterations in the first place. The generated bytecode has the form shown in Figure 5. As one can see, it holds only backward jumps to each same basic block. Hence, two iterations suffice to type check this program, opposed to the n^2 we encountered before.

3.3.2 Summary

As a consequence, we conclude that bytecode exploiting the worst-time complexity of the verification algorithm does not naturally occur when compiling from Java source. However, the threat of an attack against the verifier by directly generating malicious bytecode remains.

```
  boolean z0;
  null_type n0;

  z0 = 0;
  goto label3000;

label0:
  return;

...

label2999:
  if 0 == 0 goto label2998;
  n0 = null;
  goto label2999;

label3000:
  if 0 == 0 goto label2999;
  n0 = null;
  goto label3000;
```

Figure 2: Jimple representation of the offending bytecode

```
while (0 != 0)
{
}

...

while (0 != 0)
{
}
```

Figure 3: Java code decompiled from the code in Figure 1

```
boolean z0;
z0 = false;
while (z0)
{
}

...

while (z0)
{
}
```

Figure 4: Java code decompiled from the modified Jimple code

```
  iconst_0
  istore_0
  iload_0
  ifeq L0
  goto L1

L1:
  iload_0
  ifeq L2
  goto L1

L2:
  iload_0
  ifeq L3
  goto L2

L3:
  iload_0
  ifeq L4
  goto L3

  ...

Ln:
  return
```

Figure 5: Bytecode compiled from the modified source code in Figure 4

Unlike the shortcomings described in Section 2.2, this problem cannot easily be mitigated by adding runtime checks or by rejecting a set of actually well-typed programs. Adding runtime checks would mean doing so for each local variable use, which is would induce a far too large overhead. Rejecting potentially well-typed programs in this setting would mean to reject any program with a backwards dependency which is also infeasible.

So how can one overcome this serious problem in practice? This is what we discuss in the next section.

4 Bytecode checking and Proof-carrying code

4.1 The idea

The solution to the problem of complexity-based denial-of-service attacks as it will implemented in the upcoming Java release version 6 [4], is based on the ideas of Proof-carrying code [8]. Bytecode verification is exchanged with bytecode checking. The verifier now only *checks* a given certificate (i.e. typing information) encoded in the bytecode for correctness with respect to a given program.

The official specification document from the Java Community Process [5] states:

> Class files whose version number is greater than or equal to 50.0, must be verified using the type checking rules given in this section. If, and only if, the class files major version number equals 50.0, then if the type checking fails, the virtual machine may choose to attempt to perform verification by type inference.
>
> This is a pragmatic adjustment, designed to ease the transition to the new verification discipline.

This means that each Java 6 compliant compiler, i.e. each compiler generating bytecode with a version number 50.0 or higher, must generate so so-called `StackMapTable` attributes for each target of a backwards jump in the program. Currently, there are at least two compilers with that compliance level, namely Sun's prototype [4] of javac (we tested 1.6.0-beta2) and Eclipse 3.2.0 [2].

For security purposes, the abovementioned fallback mode can be disabled in Sun's JVM for Java 6, so that in order for a class to pass the the check it has to contain those attributes. When the attributes are present, the type checking algorithm can run in linear time and virtually constant space, hence eliminating the potential for complexity-based denial-of-service attacks as mentioned in Section 3.

The `StackMapTable` attribute holds a number of `StackMapFrame` attributes, each of which encodes a pair $(delta_{offset}, types)$. Here $delta_{offset}$ encodes the bytecode offset this frame belongs to via the delta with respect to the last such encoded frame state plus 1. *types* stands for an ordered tuple of types representing the types of locals and stack positions at this bytecode offset.

In fact, Sun implements some basic form of compression, so that not at all positions the full `StackMapFrame` is given. A `StackMapFrame` can be of different subtypes, replacing previous typing information with new ones, hence encoding a typing "delta" with respect to the last frame.

4.2 Example

As an example, consider the code shown in Figure 6. The code declares one single local variable of type `int` and then an empty loop (which is actually never executed but that is irrelevant here).

```
int i=0;
while(i>0){ }
```

Figure 6: Some basic example source code

Figure 7 shows the corresponding bytecode along with the `StackMapTable` attribute. As one can see, the attribute first encodes the type `int` at position 2 (in the first frame, the delta value actually encodes the absolute offset). This information is correct because the integer constant 0 was stored into local variable 1 at line 1 and is not altered before the jump at line 6, which gives the only alternative control flow path to instruction 2.

The second `StackMapFrame` attribute is of type `same_frame`. It says that that at position 9 (old offset 2 plus `frame_type` value of 6 plus 1) the same typing information holds.

The checker can now exploit this information on order to check the code in linear time. First it reads the first two instructions and infers the type `int` for the local variable 1. When it reaches instruction 2, it compares this information with the one in the attribute. This check passes because the types coincide. Then, in the loop body, all the checker has to do is make sure that this typing constraint is never violated, i.e. that local variable 1 is never assigned an incompatible type or that it is never used in an incompatible instruction. The crucial difference to the type reconstruction in bytecode verification is here that at each merge point (i.e. target of a jump), the most general type of each variable is already known. Since any use of this variable must be compatible with this most general type, all which remains to do is checking against it, which can be done in one single linear pass.

```
0:   iconst_0          StackMapTable: number_of_entries = 2
1:   istore_1
                          frame_type = 252 /* append */
2:   iload_1                offset_delta = 2
3:   ifle 9                 locals = [ int ]
6:   goto 2
                          frame_type = 6 /* same */
9:   return
```

Figure 7: Bytecode and StackMapTable attribute for source from Figure 6

4.3 Applicability to the subroutine problem

In Section 2.2.3 we briefly discussed the problem of type checking Java bytecode in the presence of Java subroutines (JSR). The conclusion in this section was that actually a fully-fledged polyvariant type reconstruction would be desirable but that in practice such algorithms often proof too slow - in particular when taking into account complexity-based denial-of-service attacks as we did in Section 3.

As proposed by Mathieu Corbeil [1], it is however possible to exploit the same idea of proof-carrying code also for assuring correct typing of JSRs. One simply encodes the polyvariant type information at each target of a `ret` statement. Resolving those targets is not a trivial task because unlike `goto` statements, the former do not have an explicit target given. The target is rather always the statement after the corresponding `jsr` call. The compiler however has all information it needs to generate those attributes and again for the checker it is easy to proof such information correct.

5 Evaluation and conclusion

Ensuring the safety and security of a Java application is a non-trivial task. It is further complicated by the dynamic structure of Java where executable code can be loaded dynamically any time and from any potentially untrusted source.

In order to assure perfect safety, proper verification of such potentially malicious code is crucial. We have seen that not all potential safety issues are properly solved by the current implementation of the Java bytecode verifier. Some are resolved by performing runtime checks. Others are solved by rejecting an overapproximated set of actually valid Java programs.

One problem however could not easily be overcome using the old verification technique: The problem of complexity-based denial-of-service attacks. We showed what such an attack could look like by examining a Java class in bytecode format which exploits the quadratic worst-case complexity of the verification algorithm.

We further showed that such bytecode does not directly correspond to Java source code and in particular that any equivalent Java source code being compiled by a usual Java compiler would not yield this malicious bytecode structure. Hence we concluded that such code certainly imposes a security risk but that it is unlikely to be generated accidentally by writing ordinary Java source code.

We explained how the latest version 6 of Java employs techniques of proof-carrying code to eliminate this complexity problem. Any Java 6 compliant compiler has to encode a "proof" of type safety in the bytecode by the means to special attributes. Those attributes encode the most general type for each local variable and stack location at each target of a jump. Java bytecode can henceforth simply be checked

instead of verified, by checking this typing information against the bytecode instructions in linear time and space.

We further on commented on how the same technique can be of use in order to solve another known problem with bytecode verification, namely the one of type checking code in the presence of Java subroutines. Generally, we believe that the technique of encoding such information in the bytecode should allow to solve many related problems in performance-critical systems, because it should be generally possible to move expensive analysis from the client side to the compiler by encoding analysis information directly within the bytecode.

References

[1] Mathieu Corbeil. Efficient verification of bytecode with JSRs, 04 2006. at the gradutate course COMP 621 on compiler optimization, `http://www.sable.mcgill.ca/~hendren/621/`.

[2] Eclipse Integrated Development Environment. `http://www.eclipse.org`.

[3] Andreas Gal, Christian W. Probst, and Michael Franz. Complexity-Based Denial-of-Service Attacks on Mobile Code Systems. Technical Report 04-09, University of California, Irvine, School of Information and Computer Science, April 2004.

[4] Java 6 prerelease. `https://mustang.dev.java.net/`.

[5] Java Specification Request 202 (JSR-202). `http://www.jcp.org/en/jsr/detail?id=202`.

[6] Xavier Leroy. Java bytecode verification: An overview. *Lecture Notes in Computer Science*, 2102:265+, 2001.

[7] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[8] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

[9] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.