ICS

TECHNICAL REPORT

A Denial of Service Attack on the Java Bytecode Verifier

Andreas Gal *Christian W. Probst* *Michael Franz*

Technical Report 03-23
School of Information and Computer Science
University of California, Irvine, CA 92697-3425

November 17, 2003

Abstract

Java Bytecode Verification was so far mostly approached from a correctness perspective. Security vulnerabilities have been found repeatedly and were corrected shortly thereafter. However, correctness is not the only potential point of failure in the verifier idea. In this paper we construct Java code, which is *correct*, but requires an excessive amount of time to prove safety. In contrast to previous flaws in the bytecode verifier, the enabling property for this exploit lies in the verification algorithm itself and not in the implementation and is thus not easily fixable. We explain how this architectural weakness could be exploited for denial-of-service attacks on JVM-based services and devices.

Information and Computer Science
University of California, Irvine

A Denial of Service Attack on the Java Bytecode Verifier*

Andreas Gal gal@uci.edu
Christian W. Probst cprobst@uci.edu
Michael Franz franz@uci.edu

Department of Computer Science
University of California, Irvine
Irvine, CA 92697-3425

October 17, 2003

Abstract

Java Bytecode Verification was so far mostly approached from a correctness perspective. Security vulnerabilities have been found repeatedly and were corrected shortly thereafter. However, correctness is not the only potential point of failure in the verifier idea. In this paper we construct Java code, which is *correct*, but requires an excessive amount of time to prove safety. In contrast to previous flaws in the bytecode verifier, the enabling property for this exploit lies in the verification algorithm itself and not in the implementation and is thus not easily fixable. We explain how this architectural weakness could be exploited for denial-of-service attacks on JVM-based services and devices.

1 Motivation

The bytecode verifier is an integral component of the Java Virtual Machine (JVM) [7]. Java bytecode verification is based on the idea of data flow analysis (DFA). The abstractions of values and their types are tracked along the edges of the control flow graph and the verifier checks that no rules of the type system are violated. Most JVM vendors use the verifier implementation provided by Sun Microsystems, because the verifier is a key factor in terms of security. Even minor engineering mistakes can compromise safety. Instead of having to worry about such engineering mistakes, most JVM implementors prefer to trust the original Sun code. However, even those JVMs that do not use the code from Sun verbatim, e.g. [1, 5], use in principle the same data flow algorithm to perform the verification.

*Parts of this effort are sponsored by the Office of Naval Research under grant N00014-01-1-0854. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the Office of Naval Research (ONR) or any other agency of the U.S. Government.

```
1: todo ← true
2: while todo = true do
3:   todo ← false
4:   for all i in all instructions of a method do
5:     if i was changed then
6:       todo ← true
7:       check whether stack and local variable types match definition of i
8:       calculate new state after i
9:       for all s in all successor instructions of i do
10:        if current state for s ≠ new state derived from i then
11:          assume state after i as new entry state for s
12:          mark s as changed
13:        end if
14:      end for
15:    end if
16:  end for
17: end while
```

Figure 1: The standard verification algorithm found in Sun Microsystem's JVM implementations.

Bytecode verification by data flow analysis is an established and widely deployed approach. For average Java programs, verification seems to be a minor factor in terms of CPU time. For shorter programs it seems to be somewhat quicker; for larger programs it seems to take somewhat longer. Java developers and users just take for granted that the verifier scales in some acceptable fashion with the program size. This assumption, as we will show in the course of this paper, is completely unsubstantiated. We will demonstrate that the perception of linear scaling of the JVM verifier is not only wrong, but malicious programs can be constructed which can keep the verifier busy for so long as to constitute a denial-of-service attack. This has grave consequences when using Java in mobile code environments like applets and agent-based systems. The effect may be even worse if a server virtual machine is attacked, where there is no user who may kill and restart the VM.

The remainder of this paper is organized as follows. In Section 2 we will analyze the Java bytecode verification algorithm and attempt to construct worst-case scenarios as far as completion time and resource consumption is concerned. Section 3 contains verification time benchmarks for the code examples constructed in the previous section, and in Section 4 we discuss existing and possible future countermeasures to the described denial-of-service attack scenario. Section 5 contains our conclusions and Section 6 outlines our plans for future work in this area.

2 Java Bytecode Verification

This section gives an overview of the Java bytecode verification algorithm. As already pointed out, the whole security concept of the JVM is centered around verifi-

cation. The bytecode verifier [11, 4] checks a piece of code for type consistency and some other properties. Leroy [6] lists the least conditions for bytecode to be accepted by the verifier:

- Type correctness. Bytecode instructions are typed and must receive arguments of corresponding types.
- No stack overflow or underflow. A method must never pop a value from the empty stack or push a value onto the maximal stack specified for that method.
- Code containment. The program counter must always stay within the code limits of the currently active method and must always point to the beginning of a valid instruction.
- Local variable initialization. No variable may be loaded that has not been initialized first.
- Object initialization. Whenever an object of a class C is created, one of the class' constructors must be called.

One possibility to ensure that bytecode obeys to all these restrictions is to check them dynamically at runtime [3]. However, this is expensive and is not advisable since it slows down execution significantly.

In order to eliminate runtime checks, Gosling and Yellin introduced *bytecode verification* [13, 7], where properties are checked statically before execution. The verifier is actually a data flow analyzer that verifies that the code satisfies the requirements listed above.

In order to check type correctness, the bytecode verifier contains an abstract interpreter, which executes the instructions on *types* instead of *values*. Therefore, all actual values in the program are abstracted by their type. Usually, the program's class types are presented as a graph, where the nodes are types and the edges represent sub-typing relations. Figure 2 and Figure 3 show an example program and the associated type graph.

The data flow algorithm used in Sun's implementation of bytecode verification is shown in Figure 1. This algorithm can be found, among others, in the CVM [9] and the Hotspot virtual machine [8]. The algorithm is performed separately for every method in the Java program. For each method, it iterates over all instructions of that method until no more operand type changes are observed. For each instruction i, the verifier checks whether the abstract data associated with i has changed. If so, it checks whether the current abstract local variable and stack content allows the execution of i and computes the new local variable and stack content. Finally, this new abstract state is propagated to all successors of i.

The analysis of straight-line code is inexpensive, since the abstract interpreter only needs to propagate type information through the instructions and to compute the abstract stack state after each instruction. Figure 4 shows the computed stack and local variable states for method `main` in the example program after 7 selected instructions. The first part of the example code in Figure 2 is simple to verify. From (1) to (4) the code is simply executed at the symbolic level using types instead of values.

```
class A {
  void m() {
    ...
  }
}
class B extends A {
  void m() {
    ...
  }
}
class C extends A {
  void m() {
    ...
  }
}
```

```
class App {
  void main() {
/*1*/ iconst 1
/*2*/ istore 1
      iload 1
      iconst 1
      iadd
/*3*/ istore 2
      iload 2
/*4*/ iconst 2
      if_icmple goto L1
      new B
      dup
      invokespecial B/B()
/*5*/ astore 3
      goto L2
   L1:
      new C
      dup
      invokespecial C/C()
/*6*/ astore 3
   L2:
      aload 3
/*7*/ invokevirtual A/m()
  }
}
```

Figure 2: Example Java program in bytecode form. The *main* method first stores some *integer* values into local variables and then instantiates one object of type B and one object of type C. In two different control flow paths, B and C are stored in the same local variable location.

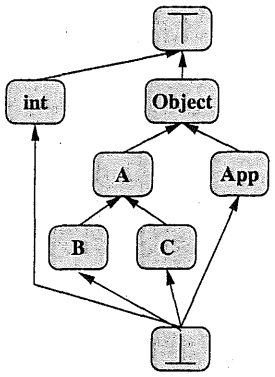

Figure 3: The example type graph

The runtime of such a data flow analysis is significantly increased if the code contains jumps, exception handlers, and subroutines, which introduce forks and joins in the control-flow graph. When separate control flows are merged together, an instruction may have several predecessors with different abstract stack or variable types. In the example in Figure 2, the call to the method `m` based on local variable 3 may be called on either a `B` or a `C` object. The verifier adapts the abstract type of the variable slot containing either `B` or `C` to be the smallest common ancestor of the two classes in the type graph (`A`). After merging the state information, the data flow analysis has to be repeated for all instructions which are reachable from this point in the control flow of the method. For simplicity, the Java verifier repeats the entire data flow analysis for every instruction of a method every time there is a change.

For average Java programs, the verifier algorithm quickly reaches a fixed point after only a few iterations. It is obvious, that *–in theory–*, the Java verifier could need up to n iterations over the method, with n being the number of instructions in the method. As for each iteration all instructions have to be visited, the overall complexity is $O(n^2)$.

However, such quadratic runtime behavior does not only exist in theory. We will show in the remainder of this section, how simple Java programs can be constructed which expose the worst case scenario in practice.

Studying the pseudo code of the verifier algorithm in Figure 1 reveals that newly learned type information is immediately available for *downstream* instructions, but can be propagated *upstream* only during the next iteration of the DFA. This property is given by the order in which the algorithm iterates over the instructions in each method. Once an instruction was visited for a particular iteration, it will not be visited again, even if new information about the operand types of that instruction was learned.

Thus, if we manage to order N instructions in such a way that each depends on the completion of the verification of the *successor* instruction, we effectively force the verifier to repeat the data flow analysis N times.

Consider the Java bytecode in Figure 5. The right hand part of Figure 5 shows several stages during verification. Column S depicts the computed type for the topmost stack cell, column S shows the computed type for the local variable with index 1

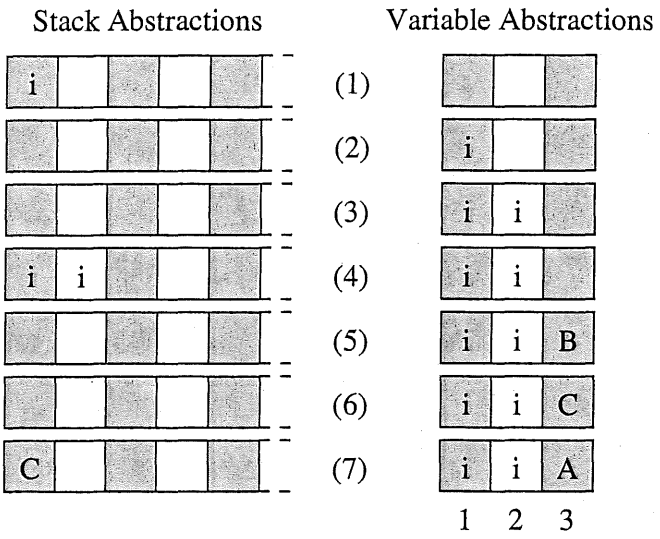

Figure 4: The stack and variable states for method `main`

$(LV1)$.

At the entry point of the method, an integer constant is loaded into $(LV1)$ and the control is transfered to $L0$. The verifier will actually not follow the branch instruction to the target, but continue to check instructions in sequence. Once reaching $L0$, the verifier remembers that $L0$ was already the target of a jump instruction with $type(LV1) = integer$. The first two instruction after $L0$ constitute a conditional branch. While the branch is actually statically predictable in this example, the verifier does not perform value folding and thus considers the *ifeq* instruction as conditional. The verifier records that this *ifeq* could transfer the control flow to $L1$ with $type(LV1) = integer$. The *aconstnull* instruction following the unconditional branch loads a *null* value onto the stack, which is then stored in $LV1$. Thus, $LV1$ holds a value of type *reference*. The unconditional *goto* at the end of this code block transfers control back to $L0$.

During the next iteration the verifier will verify $L1$ assuming $type(LV1) = integer$. As the verifier hits $L0$ again, it will invalidate the previous assumption $type(LV1) = integer$, because it now knows that $L0$ can be reached from two points in the program. $L0$ is actually a merge point for control flows. It can be reached from the unconditional goto at the method entry with $type(LV1) = integer$, but also from the end of the $L0$ basic block with $type(LV1) = reference$. Thus, the correct type for $LV1$ is *nil*, which indicates that the value is not accessible at this point because its type depends on which path was taken to get to $L0$.

This discovery also affects the $L1$ basic block, which was previously assumed to be entered with $type(LV1) = integer$ only. Now it can be entered with $LV = nil$ as well. The verifier has thus to iterate over the code again to correct the wrong assumptions previously made for $L1$. This process is repeated until all basic blocks are verified and the fixed point is reached.

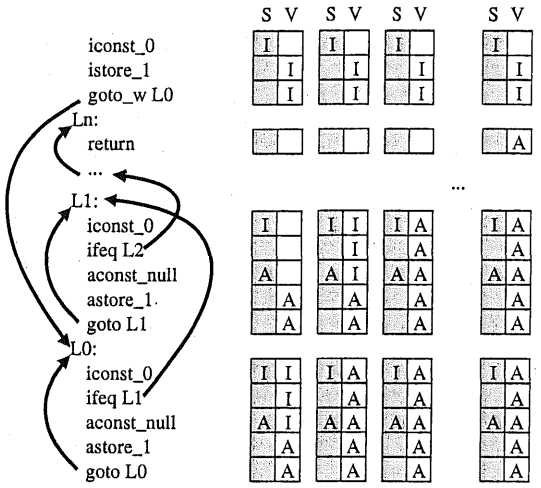

Figure 5: Java bytecode program, which takes n iterations to be verified using the standard DFA verifier approach. The entry state for each basic block depends on the successor basic block. To the right the stack and local variable states are displayed for each iteration of the DFA.

The number of basic blocks arranged in this fashion determines how often the verifier has to iterate over the code. For N basic blocks the verifier will have to iterate at least N times over the code, because the length of the longest path information has to flow along backwards is N.

To achieve an even greater slowdown, the loading of the *null* constant into $LV1$ can be performed in a subroutine which is called from every basic block using the `jsr` instruction (Figure 6). While not increasing the theoretical complexity of the verification, the practical verification time is indeed significantly higher as we will see in the benchmarks in the following section.

To evaluate the worst case efficiency of the Java bytecode verifier, in the following section we will run the verifier on Java methods containing code constructed using the control flow pattern presented in this section.

3 Benchmarks

We have benchmarked the verification time for the two example programs presented in the previous section using the Sun Microsystems Java 2 HotSpot Client VM[1]. As we have mentioned above, not all JVM implementors are using exactly the verifier implementation offered by Sun. We have repeated our tests with a number of JVMs from other vendors. While slight performance advantages or disadvantages can be

[1]Java 2 SE Runtime Environment (build 1.4.1.02-b06), Java HotSpot Client VM (build 1.4.1.02-b06, mixed mode), running on a Dell Dimension 8250, 2.53GHz P4, 512MB RAM, RedHat Linux 9.

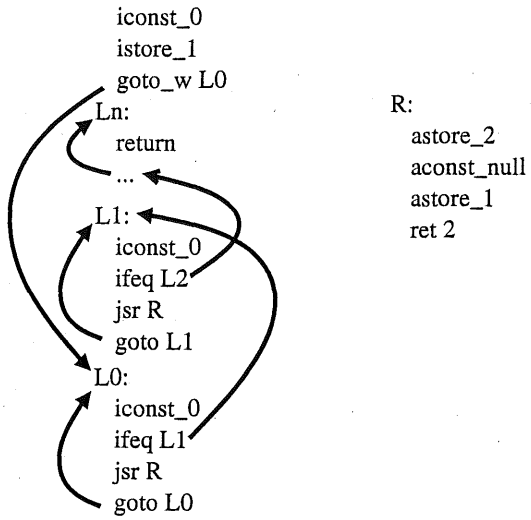

Figure 6: To increase the amount of work the verifier has to perform in each iteration, the access to the local variable is outsourced into a subroutine. Each basic block calls the subroutine using the `jsr` instruction.

observed, we are not aware of any verifier implementation which does not expose quadratic runtime behavior for the discussed test cases.

Figure 7 shows the verification time for a single method containing bytecode with increasing maximum data flow path length N. This time includes only the time it takes the verifier to prove safety. The code is never actually executed or compiled to executable code. The first curve shows the for verification time for the basic example shown in Figure 5. The second curve in the graph shows the maximum flow path problem with an added subroutine call in each code block. Both curves clearly show quadratic growth. The second curve grows much faster than the basic example due to the poor implementation of the verifier. The increased steepness is not caused by a symptomatic problem. We have merely included this second curve to make the quadratic behavior more visible.

All measurements were taken on a 2.53 GHz P4 running Linux and the Sun HotSpot VM 1.41. The maximum verification time we observed on this machine for a single method was approximately 40 seconds. The maximum basic block count N we could reach was $N = 7280$ for the simplified scenario and $N = 5460$ for the test case with subroutine calls. This stems from the 65,536 bytes limit for method code in the JVM. To achieve even longer verification times, an attacker could hide more than just one of these methods in the code. Just including 20 methods instead of one would already increase the verification time to approximately 15 minutes on a 2.54 GHz P4.

The standard JAR archive format used by Java can be used to drastically reduce the apparent size of the malicious code. But not only method repetitions can be compressed well using this approach. The code patterns used in the presented scenarios

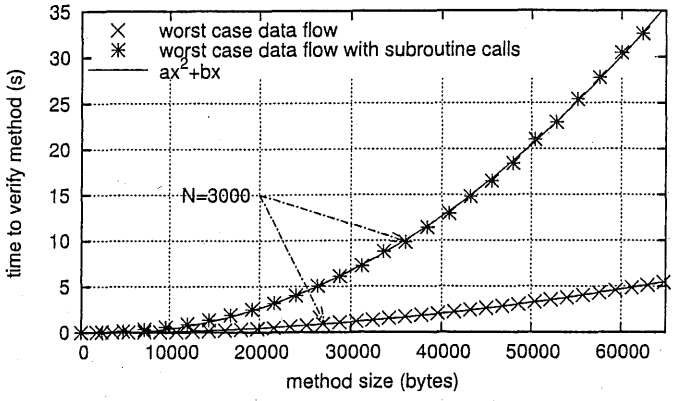

Figure 7: Verification time for verifying a single method containing a worst-case data flow scenario. The x-axis indicates the length of the method bytecode in bytes, which is proportional to the number of basic blocks N used to construct the code. The second curve shows the verification time for the worse-case data flow scenario with one added subroutine call per control flow merge. The arrows indicate for comparison purposes the code size for the maximum path length $N = 3000$ for each of the examples.

lend themselves for compression due to their very regular structure. Figure 8 indicates the compressed size for different problem lengths N for each of the two approaches. The basic scenario can be compressed much denser using the standard JAR algorithm, because each block consists at the bytecode level of exactly the same instructions. This is guaranteed because Java uses relative addressing for jump instructions. The branch instructions in each block thus use the same relative offset over and over again.

For the scenario using subroutine calls, this feature of the JVM is less favorable. The subroutine invocation has to reference the same subroutine location from different program counter locations, creating a different offset for each subroutine call instruction.

To demonstrate the practical impact of the shortcoming of the JVM verification discussed in this paper, we have set up a website containing a Java applet, which is automatically executed when the browser displays the website:

`http://nil.ics.uci.edu/~gal/usenixvm`

The applet (`time.class`) loads a second class file (`paralyse.class`) using the Java `Class.forName()` API call and measures the time it takes the JVM to load the requested class. Both class files reside in a JAR archive. The size of the JAR archive is less than 3kB. On our test machines the execution time for the applet ranged from minutes to hours, depending on the machine and JVM used. Readers are welcome to try this experiment on their own machine.

Short of disabling Java applets, the user cannot prevent or interrupt the loading of this applet. In fact, many browser do not even allow the user to interrupt the verification, because the browser implementor never considered the verification time

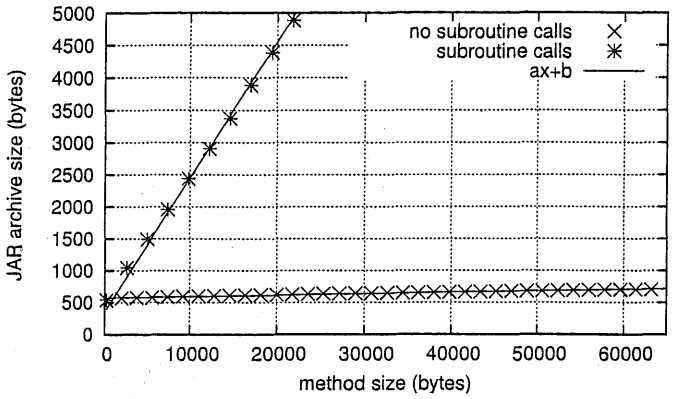

Figure 8: Compression of constructed code examples using the standard JAR archive format. The basic scenario is extremely well compressible as it basically repeats identical code patterns. The example code including a subroutine call per block suffers from the relative addressing used for the subroutine call instruction.

to be long enough for the user to ask for termination of it. Other browsers, including some versions of the Microsoft Internet Explorer, allow the verifier to continue the verification silently and continue to *hog* the CPU in the background if the user leaves a website containing an applet which takes an excessive amount of time to verify.

4 Countermeasures

In contrast to security flaws previously discovered in the JVM [2], the enabling property for this vulnerability of the JVM verifier is an *inherent property* of the algorithm used and not merely some *faulty code*, that could be exchanged.

Rewriting the verifier algorithm to iterate over the code in some other order, or the introduction of a work list algorithm, would not significantly improve the situation. Each of these algorithms would still expose quadratic runtime behavior for some worst case code scenarios.

However, a number of mitigating factors exist. First, current JVMs limit the code size per method to 65,536 bytes. On high-end desktop systems this limits the maximum verification time we were able to achieve using a single method to approximately 40s. This (probably accidental) ceiling prevents the construction of worst case scenarios with near-infinite verification time.

Further shortening the maximum method length of Java methods is not an option, since long Java methods are not uncommon. Some compilers emitting Java bytecode generate even very long methods. This includes some XML transformation tools and parser generators. In contrary, it would be not surprising if Sun decides to remove the current code size limitation in future versions of the Java Virtual Machine.

It seems unlikely that one could establish a clear set of rules to detect this class

of malicious programs. Just rejecting a program because it takes more than a certain number of iterations to verify would be arbitrary. On the other hand, trying to detect patterns such as described in this paper would not eliminate the problem as more complex and less obvious example can be easily constructed.

The verification time can be increased by shipping a large number of malicious methods to the verifier. While this increases the verification time only by a linear factor, in conjunction with compressed archives (JAR) verification times in the magnitude of minutes and hours are achievable. The corresponding JAR archive would still be only a few kilobytes in size. Detecting this exploit is easier than the single method approach exploit. For agent-based systems or applets restricting the overall code volume is probably acceptable. With such code size restrictions the verification time could be limited as well.

Adding resource monitoring to the verification process could be used to counter this attack. However, bytecode verification is deeply embedded into the JVM. Introducing the possibility to abort a running verification from the outside would require invasive changes to JVM implementations. As all approaches previously suggested in this section, resource monitoring introduces arbitrary abort conditions for the verifier and might prevent an important and desirable Java applet or agent to run just because it takes longer than expected to verify the code.

Instead of performing the expensive DFA on the code consumer side, it has been proposed to supply the code consumer already with the fixed point of the DFA. The consumer has then only to check whether the supplied fixed point indeed suffices the data flow equations, which can be done in linear time. The K Virtual Machine (KVM) [12] annotates the JVM code with stack maps for every point reached by a branch or exception to achieve this effect. This annotation can be understood as a very specific case of the more generic proof carrying code [10] approach, where a proof generator performs the computational intensive generation of proof which is transmitted in form a of certificate to the proof checker. The proof checker in turn is able to verify the validity of the code using the certificate in linear time. It is unclear whether Sun will adopt the mechanisms found in the KVM into the general purpose Java VMs. Adding such annotations would break backward compatibility. If not at the class file format level, at least because class files without such an annotation could be rejected in some instances, for example it takes an excessive amount of work to verify them. On the other hand, any such annotation also takes up additional space in the class file, which is not always desirable.

Finally, one could attempt to find a middle course between proof carrying code and the DFA-based Java verifier by changing the Java bytecode representation or the verification algorithm. We outline some of our plans for future work in this area in Section 6.

5 Conclusions

In this paper we have shown that the perceived average case verification times of the Java bytecode verifier do not translate automatically to all correct Java programs. By carefully analyzing the data flow algorithm underlying the Java verification approach,

in Section 2 we successfully constructed a correct Java program, which is very hard to prove correct. Even on high-end desktop systems the verification of a such Java methods can require seconds, minutes, or even hours (Section 3). We have discussed how this vulnerability of the JVM can be exploited for denial-of-service attacks. In Section 4 we addressed some possible countermeasures to such denial-of-service attacks and argued that this exploit is hard to counter without accepting the random rejection of non-malicious programs. From a more general perspective, this vulnerability demonstrates the need for not only correct but also efficient algorithms when dealing with mobile code. This is not necessarily restricted to the verification problem. If inefficient algorithms are used during dynamic code generation or optimization, similar exploits could be constructed.

6 Future Work

The proof carrying code approach has proven itself to be very useful in many domains. However, it requires to ship a certificate with the code, which would enlarge the class files. We are currently exploring a middle course between proof carrying code and the established Java bytecode verification algorithm. We call this approach *well-forming*. *Well-forming* is based on the observation that certain safety requirements in the Java bytecode language are hard to verify. For example, the verifier has to make sure that local variables containing integer values are never read back as references. For this, the verifier has to perform the expensive data flow analysis outlined in previous sections. Instead, our approach introduces multiple local variable planes. Each type has its own set of local variables, numbered from $0..n$. Thus, an integer load instruction could never reference a reference value and vice versa. Our observation is, that Java programs can easily be transformed into *safe* programs using this and similar approaches. However, this also means that our verifier accepts a class of Java programs which are *unsafe*. Instead of rejecting them, they are transformed into *safe* programs. It can be argued whether such a relaxation of the verification algorithm is acceptable or not. While it would still guarantee that only *safe* programs are executed, some programs might get executed which would be rejected without execution by other JVM implementations.

References

[1] B. Alpern, C. R. Attansio, J. J. Baron, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocci, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russel, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, 39(1), Feb. 2000.

[2] CERT Coordination Center, Carnegie Mellon University, http://www.cert.org.

[3] R. M. Cohen. The defensive Java Virtual Machine specification version 0.5. Technical report, Computational Logic, Inc., May 1997.