Volume 3, Issue 3, March 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Verification of Bytecode in a Virtual machine

Koushal kumar
M.tech(CSE) dept
IEC college of engineering & Technology
India

Ashwani kumar
Asst. Prof. (CSE) dept
IEC college of Engineering & Technology
India

Abstract: Java Byte code Verification ensures that byte code can be trusted to avoid various dynamic runtime errors. Applications written in modern dynamic languages such as java and .net are shipped in the form of high level intermediate byte code. A virtual machine (VM) is used on the target machine to verify and execute the code. Many Vms use just-in-time compilation to speed up the execution of critical code areas for which interpretation alone is not efficient enough. Byte code verification which is instantiated step by step with increasingly expressive type systems covering all of the interesting and complex properties of Java byte code verification: classes, objects, inheritance, Virtual methods, exception handling, constructors, object initialization, byte code subroutines, and arrays. The instantiation yields two executable verified byte code verifiers: the iterative data flow algorithm of the standard Java platform and also a lightweight byte code verifier for Resource-constrained devices such as smart cards.

General terms- class, objects, Bytecode, virtual machine,
Keywords: Bytecode verifier,opcodes, SSA,JVML

1. INTRODUCTION

Applications written in modern dynamic languages such as java or .net are shipped in the form of high level intermediate byte code. This offers two distinct advantages over shipping programs directly as compiled machine code. On the one hand the byte code is architecture independent and can be executed on different target systems using different native instruction sets and software framework i.e. operating system. This makes byte code programs portable across target platforms. Since interpretation is often too inefficient for frequently executed code areas, dynamic compilation is used to translate parts of the byte code program to directly executable native machine code. On the other hand, high level intermediate byte code can also be verified with respect to certain safety properties by the code consumer. The java virtual machine language (JVML) specification, for example, requires the code receiver to check that code is well typed before it is permitted to execute on this virtual machine. This verification step provides certain safety guarantees and eliminates common security concerns such as buffer overflows by ensuring the type and memory safety of all programs prior to execution. Byte code verification and dynamic compilation have been studied extensively from a correctness perspective , and a large body of prior work exists that discusses how to ensure that byte code verification is actually safe, and dynamic compilation generates correct and efficient native machine code for a byte code fragment. These traditional approaches do not work well in resource-constrained environments such as PDAs and cell phones. It is not sufficient to verify code and to compile it to efficient machine code. Both byte code verification and compilation also have to be efficient themselves. The first JVM byte code verification algorithm is developed by Gosling and Yellin at Sun microsystem. Almost all existing byte code verifiers implements this algorithm. It can be summarized as a dataflow analysis applied to a type-level abstract interpretation of the virtual machine.

II. COMPLEXITY OF JAVA BYTE CODE VERIFICATION

Java byte code verification has been studied extensively from a correctness perspective, and several vulnerabilities have been found and eliminated in this process. However, there are avenues to an attack that do not depend on correctness at all. In this chapter we show how to construct Java code that is *correct*, but that requires an excessive amount of time for verification. We explain how this could be exploited for denial-of-service attacks on JVM-based services and devices. In contrast to previously discovered Flaws in the byte code verifier, the enabling property for our exploit lies in the verification algorithm itself, and not in its implementation. As a consequence, it is much more difficult to correct. The byte code verifier is an integral component of the Java Virtual Machine and is based on the idea of data-flow analysis. 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. The verifier is a key security factor; even minor engineering mistakes can compromise safety. As a consequence, most JVM implementers today use the verifier implementation provided by Sun Microsystems. Even those JVMs that do not use Sun’s code verbatim use in principle the same data-flow algorithm. Hence, byte code verification by data-flow analysis

is an established and widely. deployed approach. For average Java programs, verification effort seems to be negligible:for shorter programs it seems to be somewhat quicker; for larger programs it seems to take somewhat longer.
Standard verification algorithm found in sun microsystem's JVM implementation

```
todo ← true
While todo=true do
    todo ← false
    for all i in all instructions of a method do
        if i was changed then
            todo ← true
            check whether stack and local variable
            types match definition of i
            calculate new state after i
            for all s in all successor instructions of i
            do
                if current state for s ≠ new state derived from i then
                    assume state after i as new entry for s
                    mark s as changed
                end if
            end for
        end if
    end for
end while
```

III. OVERVIEW OF JAVA VIRTUAL MACHINE

The Java Virtual Machine (JVM) is a conventional stack-based abstract machine. Most instructions pop their arguments off the stack, and push back their results on the stack. In addition, a set of registers (also called local variables) is provided; they can be accessed via "load" and "store" instructions that push the value of a given register on the stack or store the top of the stack in the given register, respectively. While the architecture does not mandate it, most Java compilers use registers to store the values of source-level local variables and method parameters, and the stack to hold temporary results during evaluation of expressions. Both the stack and the registers are part of the activation record for a method. Thus, they are preserved across method calls. The entry point for a method specifies the number of registers and stack slots used by the method, thus allowing an activation record of the right size to be allocated on method entry. Control is handled by a variety of intra-method branch instructions: unconditional branch, conditional branches, and multi-way branches (corresponding to the switch Java construct). Exception handlers can be specified as a table of quadruples, meaning that if an exception of class or a subclass is raised by any instruction between locations then control is transferred to the instruction at the exception handler. About 200 instructions are supported, including arithmetic operations, comparisons, object creation, field accesses and method invocations. An important feature of the JVM is that most instructions are typed. For instance, the add instruction requires that the stack initially contains at least two elements, and that these two elements are of type int; it then pushes back a result of type int. More generally, proper operation of the JVM is not guaranteed unless the code meets at least the following conditions:

Type correctness: the arguments of an instruction are always of the types expected by the instruction.

No stack overflow or underflow: an instruction never pops an argument off an empty stack, nor pushes a result on a full stack (whose size is equal to the maximal stack size declared for the method.

Code containment: the program counter must always point within the code for the method, to the beginning of a valid instruction encoding (no falling off the end of the method code; no branches into the middle of an instruction encoding).

Register initialization: a load from a register must always follow at least one store in this register; in other terms, registers that do not correspond to method parameters are not initialized on method entrance, and it is an error to load from an uninitialized register.

Object initialization: when an instance of a class is created, one of the initialization methods for class (corresponding to the constructors for this class) must be invoked before the class instance can be used.

One way to guarantee these conditions is to check them dynamically, while executing the byte code. This is called the "defensive JVM approach" in the literature. However, checking these conditions at run-time is expensive and slows down execution significantly. The purpose of byte code verification is to check these conditions once and for all, by static analysis of the byte code at loading-time. Byte code that passes verification can then be executed faster, omitting the dynamic checks for the conditions above. It must be emphasized that byte code verification by itself does not guarantee secure execution of the code: many crucial properties of the code still need to be checked dynamically, for instance via array bounds checks and null pointer checks in the virtual machine, and access control checks in the API. The purpose of byte code verification is to shift the verifications listed above from run-time to loading-time. Byte code that passes verification can then be executed faster, omitting the dynamic checks for the conditions above. It must be emphasized that byte code verification by itself does not guarantee secure execution of the code: many crucial properties of the code still need to be checked dynamically, for instance via array bounds checks and null pointer checks in the virtual machine, and access control checks in the API. The purpose of byte code verification is to shift the verifications listed above from run-time to loading-time.

IV. JAVA BYTE CODE VERIFIER

Java compiler compiles source programs into byte codes, and a trustworthy compiler ensures that Java source code does not violate the safety rules. At runtime, a compiled code fragment can come from anywhere on the net, and it is unknown if the code fragment comes from a trustworthy compiler or not. So, practically the Java runtime simply does not trust the incoming code, and instead subjects it to a series of tests by byte code verifier. The byte code verifier is a mini theorem prover, which verifies that the language ground rules are respected. It checks the code to ensure that:

a) Compiled code is formatted correctly.
b) Internal stacks will not overflow or underflow.
c) No illegal data conversions will occur (i.e., the verifier will not allow integers to serve as pointers).
d) This ensures that variables will not be granted access to restricted memory areas.
e) Byte-code instructions will have appropriately-typed parameters ·
f) All class member accesses are legal. For instance, an object's private data must always remain private.

The byte code verifier ensures that the code passed to the Java interpreter is in a fit state to be executed and can run without fear of breaking the Java interpreter.

V. CLASS FILES AND BYTE CODE

When Java source code is compiled, the results of the compilation are put into Java class files, whose names typically end with the .class or .cls extension. Java class files are made up of streams of 8-bit bytes. Larger values requiring 16 or 32 bits are composed of multiple 8-bit bytes. Class files contain several pieces of information in a particular format. Included in a class file are:

- The magic constant (0xCAFEBABE)
- Major and minor version information
- heterogeneous array composed of five primitive types
- Information about the classes and interfaces
- Information about the fields and methods in the class
- Debugging information
- The byte code inside a class file is made up of instructions that can be divided into several categories. Among other things, byte code instructions, called opcodes.
- Pushing constants onto the stack
- Accessing and modifying the value of a VM register
- Accessing arrays
- Manipulating the stack
- Arithmetic and Logical instructions
- Conversion instructions
- Control transfer
- Function return
- Manipulating object fields
- Invoking methods
- Creating objects
- Type casting

Since it exists at the level of the VM, Java byte code is very similar to assembly language. Each line of byte code is a one-byte opcode followed by zero or more bytes of operand information. All instructions (with the exception of two table lookup instructions) are of fixed length.

Opcodes and their associated operands represent the fundamental operations of the VM. Every method invocation in Java gets its own stack frame to use as local storage for variable values and intermediate results. As discussed earlier, the intermediate storage area part of a frame is called the operand stack. Opcodes refer to data stored either on the operand stack or in the local variables of a method's frame. The VM uses these values as well as direct operand values from an instruction as execution data.

VI. CONCLUSION

We showed that the worst-case verification effort of the Java byte-code verifier can be problematic. By carefully analyzing the data-flow algorithm underlying the Java verification approach, we were able to construct Java byte codes that, while correct, consume inordinate CPU resources during their verification. From a more general perspective, the vulnerability described in this chapter demonstrates the need, when dealing with mobile code, for algorithms that are not only correct, but also efficient. We introduce the Java Byte code Verification algorithm, and show that the worst-case complexity of this algorithm for certain pathological byte-code arrangements is significant. We conclude that it is not sufficient for a byte code verification algorithm to deliver the correct result. Instead, it has must be efficient, in particular in environments with limited resources such as mobile devices, cell phones, and PDAs. Based on this observation, we develop a more efficient byte code verification algorithm that uses the Static Single Assignment (SSA) form instead of the traditional iterative data-flow analysis. Then it extends the idea of providing novel more-efficient algorithms for implementing virtual machines to the dynamic compilation and code generation pipeline. We present a just-in-time approach that operates on the level of code traces instead of compiling entire methods.

ACKNOWLEDGMENTS

I would like to express my special thanks guide Mr. Ashwani sir who gave me the golden opportunity to write this important paper on the topic verification of bytecode in a virtual machine. He provides me the directions to show my interest in java and also helped me in doing a lot of Research and I came to know about so many new things.

References:

[1.] The Java Hotspot virtual machine v1.4.1
[2.]http://java.sun.com/j2me
[3]Zhenyu Qian, formal specification of JVM instructions for objects , methods and subroutines.
[4]Stephen N. Freund and John C. Mitchell. A formal framework for the Java byte-code language and verifier.
[5] Allen Goldberg. A specification of Java loading and bytecode verification