ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 11, November 2014

Security Architecture and Verification of Java Bytecode

Ankit Tyagi, Abhishek Anand, Archana Bharti, Rashi Kohli

Scholar, Department of Computer Science and Engineering, Amity University, Greater Noida, India

Scholar, Department of Computer Science and Engineering, Amity University, Greater Noida, India

Scholar, Department of Computer Science and Engineering, Amity University, Greater Noida, India

Lecturer, Department of Computer Science and Engineering, Amity University, Greater Noida, India

ABSTRACT: Bytecode is a stack based java virtual machine instruction set that contains 202 instructions, not only it provides architectural independence to java but also ensures that the code is secure and portable. It has widely evolved into a topic of high importance , for the industrial organizations as well as academic institutions. The industry derive it's interest because of the fact that bytecode is used specifically for websites and mobile device applications namely: cell phones, metro cards, credit cards, internet banking etc where security is of prime importance. Moreover it allows classes to load dynamically which results into an additional dare for those applications which contains formal methods. Also, the poor structuredness of the code and the operand stack being pervasively present results in furthermore challenges for the analysis of bytecode. This paper reviews the theoretical aspects of analysis, transformation, verification and security of Java bytecode by presenting the existing ways of java bytecode verification and proposing some optimizations.

KEYWORDS: JVM, Bytecode, JVML, ClassLoader, SecurityManager, JRE.

I. INTRODUCTION

Nowadays, Dynamic language applications that are written in java and other object oriented languages come alongside a high level intermediate byte code. There are two clear cut advantages over other programs which produces direct machine code. Firstly, it provides platform independency and secondly, we can execute it over target systems using different set of local instructions to perform specific tasks without using much of the system resources. This unique attribute provides byte code the ability to be ported across targeted platforms. Since, frequently executed areas of code are quite inefficiently interpreted, the bytecode programs are dynamically compiled to translate different parts of byte code program into codes that are capable enough to be executed on the local host machine. This bytecode is also verified before execution keeping in mind certain boundary conditions which are laid well before by the code consumer. Whether the code is typed well and fit for running on the java virtual machine or not is the duty of code receiver as per the java virtual machine language(JVML) specification. This verification step assures that the code written is safe and can be run without the breach of security which may result in buffer overflow. Another major concern for Bytecode verifier is to prevent viruses from entering the system and making suspicious system calls in order to compromise the system. These viruses are often introduced by foreign bytecode programs which are analyzed for safety by the inbuilt java bytecode analyzer. This bytecode analyzer performs a check on the bytecode before it is executed on the system. Prior studies have been done and experiments been performed by taking certain test cases in order to check the dynamic compilation and verification of Byte code to ensure the safety and correctness of this verification step and how well does the compilation produces local machine code for a particular bytecode fragment.

II. RELATED WORK

The Java Virtual Machine (JVM) is based on the canonical stack format. In stack, majority of the instructions pop out the operators and operands from the stack, calculate and push back the result into the stack. Additionally, it is furnished with a set of registers also known as local variables, which can be approached using “load” and “store” instructions that

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 11, November 2014

push the value of a given register on the stack or store the top of the stack in the present register. Majority of Java compilers require registers to retain the values of source-level local variables and function parameters and also uses temporary stack to hold the evaluation results of the expressions whereas the architecture does not support it. Stack and registers are held upon through method calls and are a part of the activation record. Both the number of registers and the stack space in the memory is used by the method which in particular is specified at the starting point, as a result on method entry, activation record of right size is allocated. For the JVM to work properly the code should meet the following conditions :-

- Correctness perspective: The argument types which are required by the instruction fulfill the expectations by the instruction.
- No stack underflow or overflow: In stack underflow, no parameter can be poped out of the stack which is empty and in stack overflow, no parameter can be pushed back into the stack which is full i.e. the size of the stack declared and the parameters in the stack are equal.
- Code containment: In the method, the instruction counter should perpetually point at the intial starting instruction code, it should neither point at the end nor in the middle of the method code.
- Initialization of the registers: Any method parameters cannot be loaded from an uninitialized registers so atleast one load from a register should perpetually follow at least one store in this register; we can also say, that on method entrance the registers which do not correspond to method parameters are not initialized.
- Initialization of Object: when an object of a class *A* is designed, any one of the initialization methods for class *A* should be executed before the class object can be used.

Source Java code:

```
static int fact(int x)
{
        int res;
        for (res = 1; x > 0; x--) res = res * x;
        return res;
}
```

Corresponding JVM bytecode:

```
method static int fact(int), 2 stack slots, 2 registers
0:  iconst_      // push the integer constant 1
1:  istore_1     // store it in register 1 (the res variable)
2:  iload_0      // push register 0 (the x parameter)
3:  ifle 14      // if negative or null, go to step 14
6:  iload_1      // push register 1 (res)
7:  iload_0      // push register 0 (x)
8:  imul         // multiply the two integers at top of stack
9:  istore_1     // pop result and store it in register 1
10: iinc 0, -1   // decrement register 0 (x) by 1
11: goto 2       // go to step 2
14: iload_1      // load register 1 (res)
15: ireturn      // return its value to caller
```

Figure 1: An example of JVM bytecode

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 11, November 2014

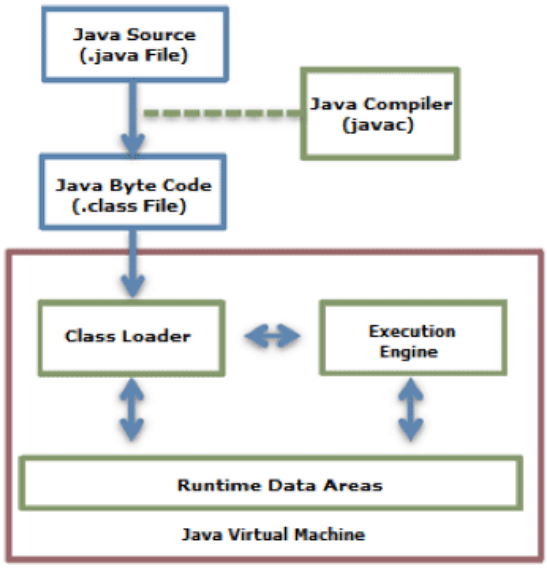
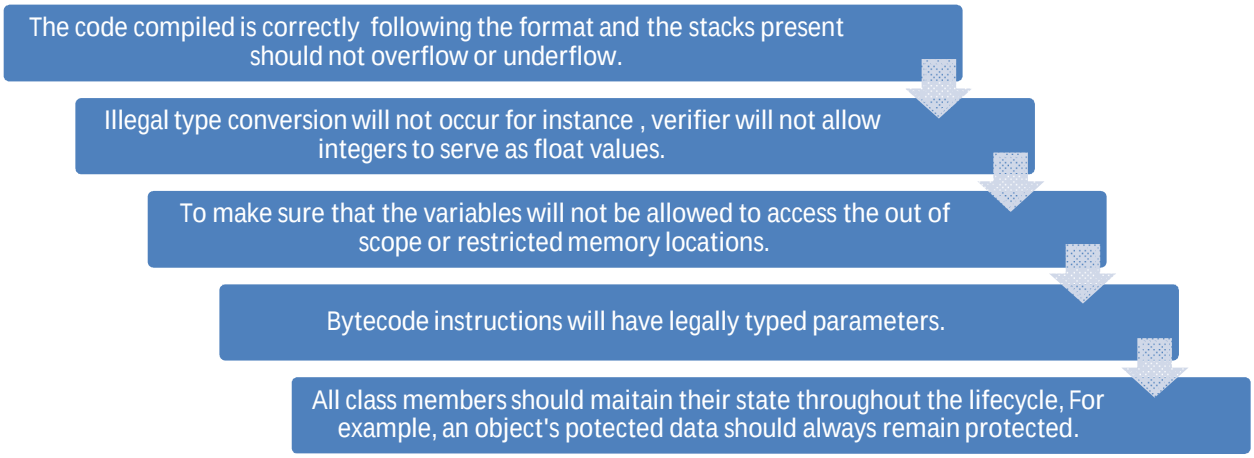


Figure 2: Java code execution process

III. VERIFICATION PROCESS FOR JVM

Java source program written by a programmer is initially compiled and converted into its corresponding bytecode (.class file) representation. If the program is compiled using a different compiler other than the one stocked within java, the JRE declares such code as "hostile" meaning, the code should be verified before execution by the Bytecode verifier. This step ensures that every program that has been compiled from a third party compiler does not violate the safety conditions set by the JVM. So, practically JVM does not even see the code till the time it has passed through a series of tests conducted by the bytecode verifier. The byte code verifier is nothing but a mini theorem prover, which verifies that the language ground rules are respected.

It checks the code by following certain steps:



By following the above mentioned steps, Java bytecode verifier makes sure that the code which is being passed to the interpreter is in a fit state so as to be executed without producing a breach in security.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 11, November 2014

IV. JAVA SECURITY ARCHITECTURE

A model for Java security has been illustrated through the fig.3 Observe that both the bytecode in Java i.e., local bytecode and unreliable bytecode (applet) must pass the verifier. Later, the invoking of class loader takes place that determines when and in which manner applets can load the classes. In addition to this, class loader also creates partition for namespace and it makes sure that remaining runtime environment should not be affected by the single applet. At last the verification of runtime is performed by Security Manager to make a check on all the methods which defines a new class loader or seeks permission for I/O or access for network etc.(dangerous method).

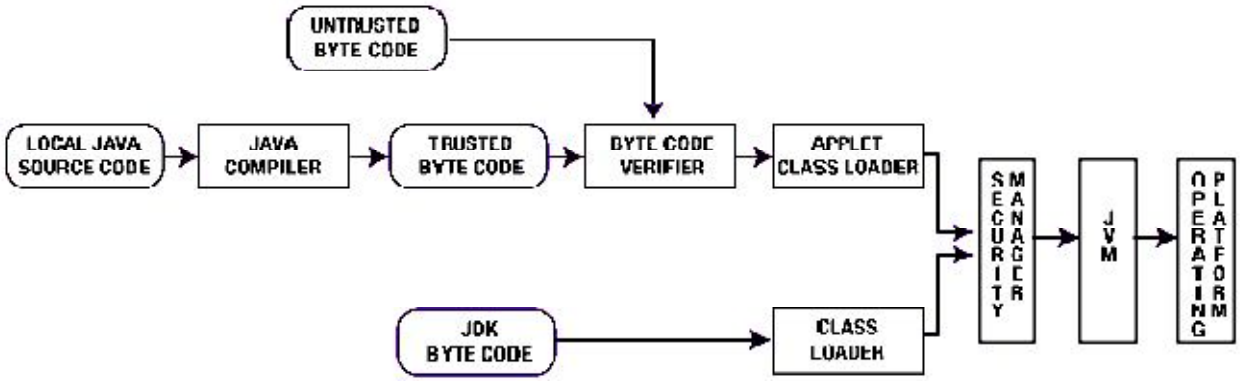


Figure 3: Java Security Model

Java Class Loader

An abstract class called ClassLoader defines the class loader in java programming language. The policies for loading java classes into the runtime environment can be done by using the Class Loader as an interface.

The overall use of class loader are:

- Class loader dynamically loads the java classes into Java Virtual Machine as class loader is the part of Java Runtime Environment.
- In a simple manner, a flat namespace of body of class is created by class loader, which is referenced by the name of a string.
- A class loader locates the libraries ,reads the contents of library and then loads the classes which are contained inside the libraries
- The parent class loader avoids the applets from invoking methods ,which are part of other class loaders.

The running JVM(i.e., the java Runtime environment(JRE) in execution),allows more than one Class Loaders with their own namespace, to get activated at one time, and then JVM groups the classes(e.g. local or global) on the basis of their origin by the allowance of the namespace. This figures out that what are the divisions of Java Runtime Environment (JRE),that can be accessed or can be changed by the applet. Moreover by restricting the namespace the unreliable applets can be prevented from getting access of additional machine resources (e.g. topical files)

Java Security Manager

The methods present in SecurityManager(basic part of Java Security Model) are called to check various operations performed by different codes to distinguish between reliable and unreliable code and thereafter the SecurityManager disallows most of the tasks requested by unreliable code. The work of exemplifying desired policies for security can be done by using the subclass of SecurityManager

The SecurityManager provides an extremely flexible and powerful mechanism for conditionally allowing access to resources Some of the Security Manager's duties include :

- All Socket transformations are managed by SecurityManager.
- Guarding secured resources (e.g. files, personal data etc) from getting accessed in unauthorized manner
- Preventing the new class loader from getting installed.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 11, November 2014

- To maintain the integrity of the Thread.
- Controlling access to group of classes i.e. java packages.

Before the execution of any kind of dangerous operation, all methods which are part of basic Java libraries (i.e., Given by Oracle corporation) consult the Security Manager for ensuring compliance .
The security manager also looks after the potential suspicious calls to the native operating system which works below JVM. Figure 4 illustrates this feature of Java Security Manager.

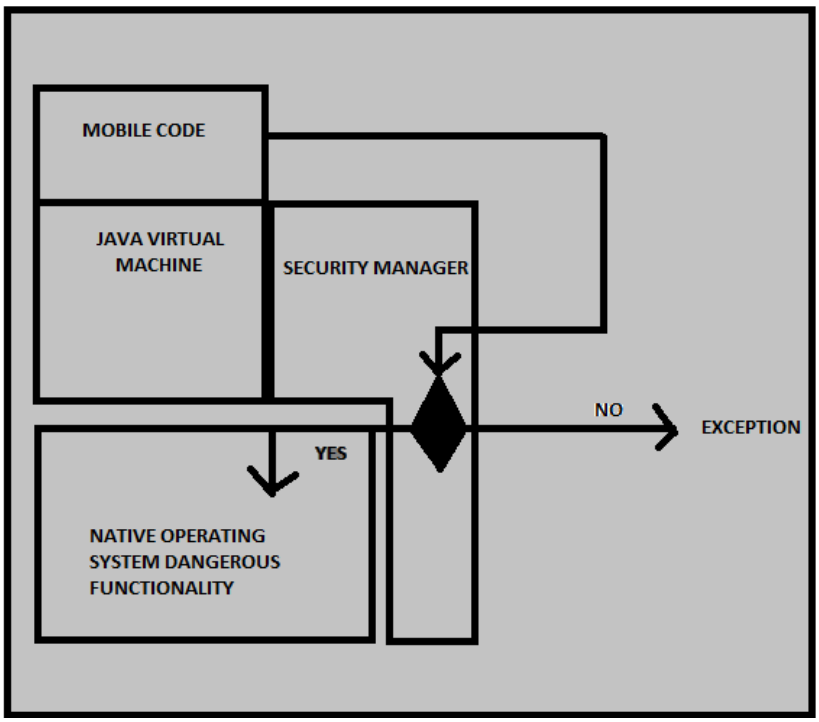


Figure 4: Security Manager keeping tab on suspicious calls to OS.

V. PROPOSED JAVA BYTECODE VERIFICATION ALGORITHM

Keeping in view the existing java virtual machine bytecode verification algorithm, we propose the following algorithm that tries to solve the performance issues faced by the existing algorithm.

```
start←1
While start=1
{
start←0
for each ‘k’ in all the steps of a function
{
if ‘k’ is changing then
{
start←1
verify if local variable's states and stack matches with state of ‘k’
calculate new state after checking ‘k’
for every ‘m’ in all the steps prior to ‘k’
{

if present state of ‘m’≠ new state out from’k’
```

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 11, November 2014

{
Let the state successor of ‘k’ as the new entry for ‘m’
Mark ‘k’ as it is changed
}}}}
The algorithm contains a number of optimizations in order to reduce the number of iterations which intern reduces the time taken for the verification process of the java bytecode.

VI. CONCLUSION AND FUTURE WORK

Though a lot of prior work exists in the technique of Java bytecode verification yet majority of the work surveyed in this paper lead us to the exact enlightenment of what bytecode and it's verification is and how it is performed to curb security breaches. Upon the analysis of the algorithm we found out that for shorter programs the overall effort seems to be the quickest while for average case programs this effort seems to increase a bit but for larger program size this effort seems to take the longest. This observation clearly states that the verification time is directly proportional to the size of the java program. Taking due note of the observations and characteristic behaviors that are closely analyzed, we conclude this paper by highlighting a largely open question which remains, whether bytecode verification can go beyond basic type safety and initialization properties, and statically establish more advanced properties of applets, such as resource usage and reactiveness which can be defined as bounding the running time of an applet between two interactions with the environment.

ACKNOWLDGEMENT

We would like to express our special thanks to our guide Ms. Rashi Kohli for her constant support and guidance without whom this work wouldn't have seen the light.

REFERENCES

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *26th symposium Principles of Programming Languages*, pages 147–160. ACM Press, 1999.
2. David Basin, Stefan Friedrich, and Marek Gawkowski. Bytecode verification by model checking.
3. *Journal of Automated Reasoning.* Special issue on bytecode verification (this issue).
4. Yves Bertot. Formalizing a JVML verifier for initialization in a theorem prover. In *Proc. Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 14–24. Springer-Verlag, 2001.
5. Pascal Brisset. Vers un vérifieur de bytecode Java certifié. Seminar given at Ecole Normale Supérieure, Paris, October 2nd 1998, 1998.
6. Klaus Brunnstein. Hostile ActiveX control demonstrated. *RISKS Forum*, 18(82), February 1997.
7. Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*
8. Gennady Chugunov, Lars Åke Fredlund, and Dilian Gurov. Model checking multi-applet Java Card applications. In *Smart Card Research and Advanced Applications Conference (CARDIS'02)*, 2002.
9. Alessandro Coglio. Simple verification technique for complex Java bytecode subroutines. In *4th ECOOP Workshop on Formal Techniques for Java-like Programs*, 2002. Extended version available as Kestrel Institute technical report.
10. Alessandro Coglio. Improving the official specification of Java bytecode verification. *Concur- rency and Computation: Practice and Experience*, 15(2):155–179, 2003.
11. Richard Cohen. The defensive Java virtual machine specification. Technical report, Computational Logic Inc., 1997.
12. Stephen N. Freund and John C. Mitchell. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning*. Special issue on bytecode verification (this issue).
13. Stephen N. Freund and John C. Mitchell. A formal framework for the java bytecode language and verifier. In *Object-Oriented Programming Systems, Languages and Applications 1999*, pages 147–166. ACM Press, 1999.
14. Koushal Kumar, Ashwani Kumar Verification of Bytecode in a Virtual machine, IJARCSS, Volume 3,Issue 3, pp. 127-130, March 2013.
15. Xavier Leroy Java bytecode verification: algorithms and formalizations, Journal of Automated Reasoning, special issue on bytecode verification (this issue).
16. Executive Summary. Secure computing with Java: now and the future, 1998.
17. Li Gong. Java security: present and near future, *IEEE Micro*, 17(3):14-19, May/June 1997.
18. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, Addison-Wesley, 1996

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 11, November 2014

BIOGRAPHY

Ankit Tyagi is a B.Tech student in the department of computer science & engineering, Amity Greater Noida. His research interest includes machine learning, Big Data, Java and encryption techniques.

Abhishek Anand is a B.Tech student in the department of computer science & engineering, Amity Greater Noida . His research interest includes programming languages, data mining, operating systems.

Archana Bharti is a B.Tech student in the department of computer science & engineering, Amity Greater Noida . Her research interest includes Operating systems, Distributed systems, cloud computing, java.

Rashi Kohli is a lecturer in the Department of Computer science & Engineering, Amity Greater Noida, Uttar Pradesh. She received her M.Tech degree in Computer science & Engineering from Amity University, India. Her research interests include Cryptography, network security and software engineering domain.