Java Security Model and Bytecode Verification[1]

Haiping Xu
EECS department
University of Illinois at Chicago
Email: hxu1@eecs.uic.edu

1. Introduction

Security is the practice by which individuals and organizations protect their physical and intellectual property from all forms of attack and pillage. Although security concerns are not new, there is revived interest in the entire area of security in computing systems. This is because today's information systems have become the repositories for both personal and corporate assets and computer networks are providing new levels of access for users. Consequently, new opportunities for unauthorized interaction and possible abuse may occur. In order to combat potential security threats, users need programs they can rely on. Moreover, developers are looking for a development platform that has been designed with built-in security capabilities. This is where the Java platform comes in. As a matter of fact, Java is designed from the ground up for network-based computing, and security measures are an integral part of Java's design.

The business end of the Java security model is conveniently described by using the metaphor of the Sandbox [1][2]. The sandbox comprises a number of cooperating system components, ranging from security managers that execute as part of the application, to security measures designed into the Java Virtual Machine* (JVM) and the language itself. Dr. Li Gong has classified the Java security model into four layers, which are [1]:

1. The language is designed to be type-safe, and easy to use. Language features such as automatic memory management, garbage collection and range checking on strings and arrays are examples of how the language helps the programmer to write safer code.
2. Compilers and a bytecode verifier ensure that only legitimate Java code is executed. The bytecode verifier, together with the Java virtual machine, guarantees language type safety at run time.
3. A class loader defines a local name space, which is used to ensure that an untrusted applet cannot interfere with the running of other Java programs.

4. Access to crucial system resources is mediated by the Java virtual machine and is checked in advance by a SecurityManager class that restricts to the minimum the actions of untrusted code.

In the next section, we will briefly introduce the Java security model with respect to the above security layers. In Section 3, we will concentrate on Java bytecode verification, and describe how to formally specify and verify Java bytecode by using model checking.

2. Java Security Architecture

Java security model can be illustrated in Figure 1 [5]. Note that both local Java bytecode and applets (viewed as untrusted bytecode) must pass the bytecode verifier. After that, the class loader is invoked to determine how and when applets can load classes. A class loader also enforces namespace partition, and it ensures that one applet cannot affect the rest of the runtime environment. Finally the security manager is used to perform run-time verification of all so-called "dangerous methods", which are those methods that request file I/O, network access, or those that want to define a new class loader.

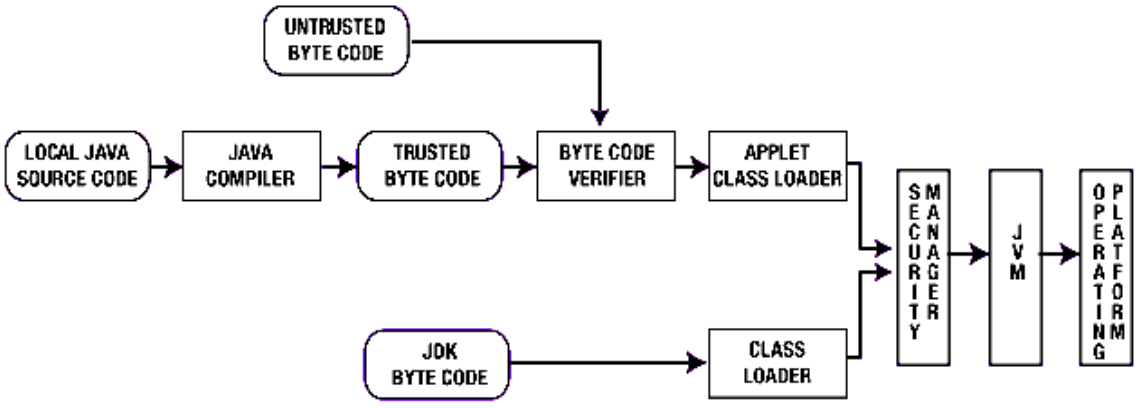

Figure 1. Java Security Model

In the rest of this section, we will briefly describe three parts of the Java security model, which are bytecode verifier, class loader and security manager.

[1] This report serves as a part of background research on security modeling.

2.1 Java Bytecode Verifier

Java compiler compiles source programs into bytecodes, 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 bytecode verifier.

The bytecode verifier is a mini theorem prover, which verifies that the language ground rules are respected. It checks the code to ensure that [5]:

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

The bytecode 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.

2.2 Java Class Loader

The class loader is defined in Java by an abstract class, ClassLoader. As an interface, it can be used to define a policy for loading Java classes into the runtime environment. The major functions of the class loader are [5]:

- It fetches the applet's code from the remote machine.
- It creates and enforces a namespace hierarchy. One of its more important functions is to ensure that running applet do not replace system-level components within the runtime environment. In particular, it prevents applets from creating their own class loader.
- It prevents applets from invoking method, that are part of the system's class loader.

An executing Java environment (i.e., a running JVM), permits multiple Class Loaders, each with its own namespace, to be active simultaneously, and namespaces allow the JVM to group classes based on where they originates (e.g., local or remote). This delineates and controls what other portions of the runtime environment the applet can access and modify. In addition, by placing restrictions on the namespace, the class loader prevents untrusted applets from accessing other machine resources (e.g., local files).

2.3 Java Security Manager

The SecurityManager contains a number of methods which are intended to be called to check specific types of actions. The SecurityManager class itself is not intended to be used directly, instead it is intended to be subclassed and installed as the System SecurityManager. The subclassed SecurityManager can be used to instantiate the desired security policy.

The SecurityManager provides an extremely flexible and powerful mechanism for conditionally allowing access to resources. The SecurityManager methods which check access are passed arguments which are necessary to implement conditional access policies, as well as having the ability to check the execution stack to determine if the code has been called by local or downloaded code. Some of the Security Manager's duties include [5]:

- Managing all socket operations.
- Guarding access to protected resources including files, personal data, etc.
- Controlling the creation of, and all access to, operating system programs and processes.
- Preventing the installation of new Class Loaders.
- Maintaining thread integrity.
- Controlling access to Java packages (i.e., groups of classes).

To ensure compliance, all methods that are part of the basic Java libraries (i.e., those supplied by Sun) consult the Security Manager prior to executing any dangerous operations, such as network access and file I/O request.

Our goal of trusting execution of untrusted programs on a JVM requires solutions to a number of problems, such as defining the behavior of JVM execution, defining safe execution on the JVM, and proving that recognized programs execute safely on JVM. In the next section, we will concentrate on Java bytecode verification by using model checking, which is one the fundamental problem we need to solve. The referenced paper was written by J. Posegga and H. Vogt [6]. In their paper, the authors described the basic idea for Java bytecode verification by using model checking, and they tried to achieve the goal of proving

CTL formulas [12] automatically. However, J. Posegga's method was based on unlabeled state transition graph, and the CTL formulas they described were not written in terms of atomic propositions. In this case, the CTL formulas cannot be proved directly by using existing methods, such as the algorithms introduced in paper [12]. In the next section, we first introduce the idea proposed by J. Posegga and H. Vogt, after that, we introduce a labeled finite state transition graph and define a set of atomic propositions. Finally we show that some interesting security properties written in CTL formula can be automatically proved by using the algorithms introduced in paper [12].

3. Java Bytecode Verification Using Model Checking

Java bytecode verification ensures that bytecode can be trusted to avoid various dynamic runtime errors, and it proves that a given bytecode program conforms to certain security requirements intended to protect the executing platform from malicious code. There are many related works in this research area [7][8][9][10], however each of them concentrates either on formalization of the process of bytecode verifier or implementation of the bytecode verifier. Examples for formalization of the process of bytecode verifier are listed as follows:

1. Stata and Abadi [7] proposed a type system for subroutines, provided lengthy proofs for the soundness of the system and clarified several key semantic issues about subroutines.
2. Qian [8] presented a constraint-based typing system for objects, primitive values, methods and subroutines and proved the soundness.
3. Goldberg [9] directly used dataflow analysis to formally specify bytecode verification focusing on type-correctness and global type consistency for dynamic class loading. He successfully formalized a way to relate bytecode verification and class loading.

And an example for the implementation of the bytecode verifier is the Kimera project [10], which is quite effective in detecting flaws in commercial bytecode verifier. Using a comparative testing approach, they wrote a reference bytecode verifier and tested commercial bytecode verifier against it. Their code is well structured and organized, and derived from the English JVM specification [13]. It achieves a higher level of assurance than commercial implementations. However, since there is no formal specification, it is not possible to reason about it, or to establish its formal correctness.

Through all these works, there seems to be no implementation that is actually built upon a rigid formal description. Based on this fact, J. Posegga and H. Vogt tried to show how Java bytecode verification can be implemented as a model checking problem. As we can see from their paper, although they aimed at using a standard model tool like SMV [11], and being able to describe bytecode verification at a very high level in terms of CTL formulas [12], an important part, which is how to translate ASM (Abstract State Machine)

rules [6] into a SMV program, is missing. In this report, we try to use another method, which was based on labeled state transition graphs. By using the model checker introduced in paper [12], we may verify CTL formulas efficiently.

3.1 Formal Semantics of Java Bytecode Instructions

The formal semantic of most Java bytecode instructions has been described in Qian's work [8]. Based on the formalism of Java byte code instructions, a Java bytecode program can be described as an ASM (Abstract State Machine). All variables in a bytecode program plus some additional variables, such as *pc* (program counter) become the state variables in the ASM, and each instruction is mapped to a state transition in ASM, which could be described as an ASM rule as follows:

```
If pc == i then S_I_i;
```

where *pc* is the program counter, and R_{Ii} is the instantiation of rule schemes for it's corresponding instruction I_i. Thus a bytecode program $I_1; I_2; \ldots; I_n$ can be mapped to a set of ASM rules:

```
If pc==1 then S_I_1,
If pc==2 then S_I_2,
…
If pc==n then S_I_n
```

An ASM rule (S_{I_i} part) also serves as an interpretation for its corresponding instruction. An example of the ASM rule (S_{I_i} part) for instruction *"if_acmpeq L"* could be:

```
S if_acmpeq L =
   x1 := top(opd)
   opd := pop(opd)
   x2 := top(opd)
   opd := pop(opd)
   pc :=
       L  if (x1==x2)
       pc+1   otherwise
```

Here *x1* and *x2* represent local variables (or registers), *opd* is the operand stack, and *top(), pop()* are regular stack operations. The interpretation S_I says, the instruction *"if_acmpeq L"* first restores the value of local

variable *x1* and *x2* from the operand stack, then compares them. If they equal, the program branches to the target address L; otherwise, it continues to execute the next instruction.

In paper [6], the authors did not treat all bytecode instructions in every detail, but picked up the *jsr/ret* instructions as an illustration of building the ASM (Abstract State Machine) model. This is because *jsr/ret* instructions, which are used for implementing subroutine calls, impose certain difficulties on bytecode verification, and the reasons are as follows:

- The variables used by the subroutine must have appropriate types for the subroutine, this limits the use of these variables in the code before the subroutine call.
- The ret instruction must be used in a well-structured way such that the variable used by ret contains a valid return address.
- Subroutines may not be called recursively, and several subroutines can be completed by returning from an inner subroutine call.

Actually, according to the official JVM specification by Lindholm and Yellin [13], *jsr/ret* instructions are control transfer instructions typically used to implement *finally* clauses in Java . A *"jsr L"* instruction pushes the return address *pc+1* onto the operand stack and transfers control to the *jsr* target at address *L*; while an *"ret x"* instruction uses a local variable *x* to restore the return address from the operand stack and transfers control to the returning program point.

Similarly as before, the formal semantics of *jsr/ret* instructions can be described as follows:

```
S jsr L =
   opd := push(opd, pc+1)
   pc := L

S ret x =
   pc := x

S astore x =
   x := top(opd)
   opd := pop (opd)
   pc := pc +1
```

To meet our requirements for model checking, the above description is not enough. In addition, we use a list of 2-tuples *<target_addr, return_addr>*, we call it *sr*, to record nested subroutine calls. By doing so, it

would be possible for us to check, when an instruction *"ret x"* is executed, the return address *x* is valid. The additional description is given by the following rules (S'_{I_i} part):

```
S' jsr L =
    sr := <L, pc+1> • sr

S' ret x =
    let (-, sr') = split (sr, <-, x>)
    in sr:= sr'
```

where the dot (•) operator is used to add a 2-tuple to the list *sr*; the dash (-) in the *let* statement denotes a "*don't care*" part, and the function *split()* is defined as follows:

```
split (u<-,a>v, <-,a>) = (u<-,a>, v), where <-,a> ∉ u
```

Now we can build an ASM for a Java bytecode program. However, the domain of the state variables defined above can be unbounded, thus the resulting ASM could be an infinite state machine. This is unrealistic for a real bytecode program, and also it would be impossible for us to do the model checking with infinite states. Our next step is to give constraints to the state variables and try to build a finite ASM for a Java bytecode program.

3.2 Building a Finite Abstract State Machine

In order to build a finite state machine, we need to redefine the state variables which could only have a finite number of values. Meanwhile, since for a concrete program, the number of variables is fixed, also we assume that we add finite number of additional variables into ASM, our resulting ASM will be a finite state machine. Now suppose the number of the bytecode instructions in a given program is *codelength* and the instructions are given in an array *code* with the length *codelength*; the local variables are given in an array *variable* with the length *varlength*; also we assume that the maximum height of operand stack *opd* is *H* and the maximal integer value for each local variable *variable(i)* is N. Then we can define the state variables as follows:

```
pc : {1, …, codelength} ∪ {undef}
variable(i): {0, 1, …, N} ∪ {undef}  for 1≤ i ≤varlength
opdsize: {0, 1, …, H}
sr: {<L, i+1> : ∃ i, code(i) = "jsr L"}*
```

Here *pc* is explicitly restricted to the possible address of instructions in the *code* array, extended by a special value *undef* denoting an invalid address; *variable(i)* is a local variable whose value must be an integer between [0,N] and a value of *undef* denotes an underflow or overflow; *opd* is the operand stack with the maximum height of *H*; *opdsize* is the current size of the operand stack *opd*; and *sr* is a list of 2-tuples of *<target_addr,return_addr>*.

When we rewrite the ASM rules, the structure of the ASM rules are not changed, however, the functions must be substituted with finite abstractions. We give the definitions of the abstracted functions as follows:

```
push(u, a) =
         {au}      if |u| < H
         {undef} otherwise
pop(u) =
         {v}  for u = av
         {undef} if u ∈ {ε, undef}
top(u) =
         {a} for u = av
         {undef} if u ∈ {ε, undef}
n + i =
         {m}  with m = n+i, if (n+i) ≤codelength
         {undef} otherwise
<a, b> • u =
         {<a,b>u}   if <a,b> ∉ u
         {undef}  otherwise
split (u, <-,a>) =
         {w<-,a>, v}  if u = w<-,a>v and <-,a> ∉ w
         {undef} otherwise
```

In J. Posegga and H. Vogt's paper, the finite ASM corresponds to an unlabeled state transition graph, and they defined the finite state transition graph as a 3-tuple M = (S, R, I), where S is a finite set of states; $R \in S \times S$ is the transition relation, and $I \in S$ is the initial state. This method limited their ability to use CTL formulas to express security properties. As we can see from their paper, the CTL formulas they described are not in terms of atomic propositions. In the next section, we treat the finite state transition graph in another way: we first define a set of atomic propositions P, and then we map each state to a subset of 2^P.

3.3 Labeling the State Transition Graph with Atomic Propositions

A labeled state transition graph is a 5-tuple M = (P, S, L, N, S_0) where P is a set of atomic propositions, S is a finite set of states, L: S-> 2^P is a function labeling each state with a set of atomic propositions, N $\subseteq$ S x S is a transition relation, and S_0 is an initial state. To meet the requirements for CTL formulas expressiveness, we define a set of atomic propositions P as follows (of course, the set P can be extended to meet certain requirements of a particular CTL formula):

p_1: 0 ≤ opdsize ≤H

p_2: code(pc) == "jsr L"

p_3: code(pc) == "ret x"

p_4: code(pc) == "jsr L" such that ∃ x, <L, x> ∈ sr

p_5: code(pc) == "ret x" such that ∃ L, <L, x> ∈ sr

p_6: sr = ε

For the rest of elements in the 5-tuple M, we define them as follows:

- S is a finite set of states, and each state is a mapping from the set of state variables to a set of values. The set of state variables is defined in Section 3.2. Since the number of values for each state variable is finite, the number of states must be finite too, although most of the states are unreachable.
- L is a mapping from each state to a set of atomic propositions. For the given set of propositions P, the value of each proposition p_i can be determined trivially for each state. We label each state with the set of propositions those are valued as True.
- N stands for the transition relation, and it can be determined by ASM rules. For *jsr/ret* instructions, each corresponding state has a unique successor. However, for conditional branching instructions such *as "if_acmeq L"*, each corresponding state has two successors, which state will be reached next in our abstract model is a nondeterministic choice.
- S_0 is the initial state, for our model, the initial state corresponds to the following mappings:

```
pc = 1
variable(i) = undef  for 1≤ i ≤varlength
opdsize = 0
sr = ε
```

With the labeled state transition graph M, we can write CTL formulas in terms of atomic propositions now. By using the model checker introduced by E. M. Clarke, E. A. Emerson and A. P. Sistla [12], we may verify that a Java bytecode program meets a certain security specification expressed in a temporal logic in linear time.

3.4 Security Properties Verification

To illustrate that some interesting security properties can be expressed in CTL formulas and could actually be verified, we rewrite some of the CTL formulas shown in paper [6] in terms of atomic propositions. The last security property is used to show that not only safety properties can be specified, liveness properties can be specified and verified in the same way. However the semantic of the last property might not be true in real life.

1) No underflow or overflow on the operand stack

$$AG\ p_1$$

This property can simply be stated as above. The preceding quantifier AG says that p_1 must hold at every state in every execution path.

2) No recursive call allowed

$$AG\ p_2 \rightarrow \neg\ p_4$$

This CTL formula says that, at every state in every execution path, whenever we meet a *"jsr L"* instruction, it can not be the one that has been recorded in list *sr.* In other words, the same subroutine call can not be made until it is returned.

3) Returning from subroutine

$$AG\ p_3 \rightarrow p_5$$

This formula simply says that, at every state in every execution path, any return address must be recorded before. In other words, the return address must be a valid one.

4) every subroutine call must finally be returned

$$AG\ (p_2 \rightarrow AF\ p_6)$$

This formula says that, at every state in every execution path, whenever we meet a *"jsr L"* instruction, it will finally be returned either from the same subroutine call or from one of it's inner subroutine call. In other words, finally, the list *sr* must become empty.

4. Conclusion and Future Works

Java security model provides us an excellent test bed for security formal modeling and verification. Currently, researches are concentrating on describing the formal semantics of Java bytecode instructions, and trying to prove their soundness. Verifying Java bytecode by model checking is one of those works and it's different from the traditional theorem proving approach. Because there are many existing model checking tools, such as SMV, it gives us a chance to concentrate ourselves on creating the model for Java bytecode, and let the model checker do the rest of the work, such as security verification. There are few works on formal modeling the class loader, such as Goldberg's work [9], and the formal modeling of Java security manager remains untouched. However, those research areas are interesting and have realistic significance. Our future work could be modeling Java security manager and verifying its soundness. This work could be hard because security manager is a concurrent system, rather than a sequence of bytecode instructions.

References

1. Li Gong. Java security: present and near future, *IEEE Micro*, 17(3):14-19, May/June 1997.
2. Li Gong. Going beyond Sandbox: an overview of the new security architecture in the Java Development kit 1.2, In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.
3. Joseph A. Bank. Java security, PMG group at MIT LCS, December, 1995, http://swissnet.ai.mit.edu/~jbank/javapaper/javapaper.html
4. Rich Levin. Security grows up: the Java 2 platform security model, October 1998, http://www.javasoft.com/features/1998/11/jdk.security.html
5. Executive Summary. Secure computing with Java: now and the future, 1998, http://www.javasoft.com/marketing/collateral/security.html
6. J. Posegga and H. Vogt. Java bytecode verification using model checking, October, 1998, http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html
7. Raymie Stata and Martin Abadi. A type system for Java bytecode subroutines. In *Proc. 25st ACM symp*. Principles of Programming Languages, 1998.

8. Zhenyu Qian. A formal specification of Java Virtual Machine Instructions for objects, methods and subroutines. In Jim Alves-Foss(ed.), *Formal Syntax and Semantics of Java*, Springer Verlag LNCS 1523, 1998.
9. Allen Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security*, 1998, http://www.kestrel.edu/HTML/people/goldberg/
10. Emin G. Sirer, Sean McDirmid and Brian Bershad. A Java system architecture, 1997, http://kimera.cs.washington.edu/
11. E. M. Clark, Sergey Berezin and K. L. McMillan. Model checking guided tour, 1998, http://www.cs.cmu.edu/~modelcheck/tour.html
12. E. M. Clarke, E. A. Emerson and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *In ACM Transactions on Programming Languages and Systems*, 8(2):244--263, 1986.
13. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, Addison-Wesley, 1996.