Byte Code Verification for Java Smart Cards Based on Model Checking*

Joachim Posegga and Harald Vogt

Deutsche Telekom AG
Technologiezentrum
IT Security
D-64307 Darmstadt
Tel. +49 6151 83-7881, Fax -4090
{posegga|vogth}@tzd.telekom.de

Abstract. The paper presents a novel approach to Java byte code verification: The verification process is performed "offline" on a network server, instead of incorporating it in the client. Furthermore, the most critical part of the verification process is based upon a formal model and uses a model checker for checking the verification conditions. The result of the verification process can be securely communicated to the runtime platform with cryptographic means.
The major advantages of our approach are twofold: on the one hand, it offers a higher degree of security, since the verification process is based on a formal framework. Secondly, it saves resources on the client's side, since the process of byte code verification can be replaced by a simple check of a digital signature.
This paper concentrates on Java smart cards, where resource limitations inhibit fully-fledged byte code verification within the client, but the demand for security is very high. However, our approach can also be applied to other variants of Java.

1 Introduction

Java [GS96] is an instance of down-loadable code that many people see as a paradigm shift in computer science. Java is a network-oriented approach with two major advantages:

Firstly, Java is supposed to be *platform-independent.* This is achieved by compiling Java applets or Java applications into Java byte code [LY96], a stack-oriented machine language that is interpreted on the specific platform where Java programs are to be executed (cf Figure 1). The interpreter for this byte code is called the *Java Virtual Machine.*

Secondly, Java applets offer the principle of *downloading executable code over a network on demand*; this is a significant gain in flexibility, since it allows one

* The opinions expressed in this paper are solely those of the authors and do not necessarily reflect the views of Deutsche Telekom AG.

to configure the function of clients in a network as needed; the expensive process of installing software on clients becomes superfluous.

The technical basis for both these specifics of Java have been known in computer science since decades. However, the overall design of the Java scenario is very promising, and the technology became widely accepted and available on various platforms. This makes Java interesting.

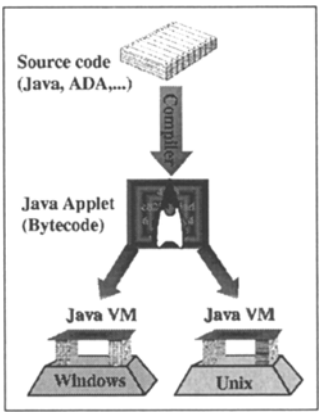

Fig. 1. Java's Architecture

A potential hindrance for the wide-spread use of Java are security concerns. The problem is, essentially, that down-loading executable code from an open network can be dangerous, since it is hard to ensure that such an executable actually does not do any harm to the local system.

Java's answer to this problem is, in essence, a type-safe language, a byte code verifier that checks certain safety properties of the transferred executable code, and a sandbox model that restricts the runtime environment of the down-loaded code. The Java byte code verifier plays a crucial role in this architecture (cf Figure 1): implemented within the user's Java platform, it ensures that the byte code to be executed meets certain properties like type-safety. The sandbox that runs Java applets takes these properties for granted, mainly for reasons of efficiency. Without this verification step, malicious byte code (e.g. code that has been hand-coded, or been manipulated during the transfer) can crash the sandbox and easily take over complete control of the underlying machine.

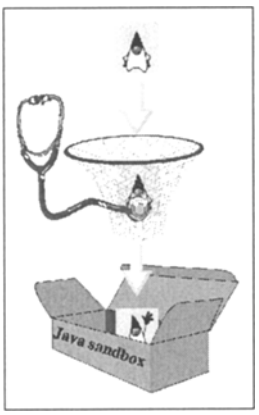

Fig. 2. Byte Code Verification

In this paper we describe a novel security architecture for Java, where the process of byte code verification is carried out using a model checker; this process is assumed to be implemented "offline", i.e. on a server in a network and not within the client. The result of such a verification step can be communicated to a client, e.g. by applying digital signatures to Java byte code programs (applets).

Our approach has two major advantages: firstly, it offers a strong formal basis for an operation that is crucial to security. This helps avoiding bugs in the byte code verifier that can cause security holes. Secondly, the principle of carrying out the byte code verification offline offers much more flexibility than the current approach, since it does not suffer from resource restrictions usually found within the clients, both for size and speed of the Java runtime platform.

The major motivation for our research were the specifics of Java smart cards [Sun97a,Sun97b,KP98], where byte code verification is particularly problematic

because of the extreme space and runtime limitations of smart cards processors. The concept of offline verification fits perfectly into this scenario: it saves resources on the card, and formal approaches like model checking can increase security, which is highly desirable for smart card applications.

Whilst the advantages of our approach are most obvious in the case of smart cards, the approach itself can also be carried forward to the general framework of Java.

The paper is organised as follows. In the next Section (2) we discuss the role of byte code verification within Java's security architecture. Section 3 outlines the idea behind performing byte code verification offline and compares it with the current architecture. In Section 4 we introduce the use of model checking for the semantic checks of Java's byte code verification, Section 5 gives a detailed example. Finally, we discuss related research in Section 6 and draw conclusions from our work in 7.

We assume the reader to be familiar with the basics of model checking [EC82], some familiarity with Java's security architecture [Yel95,LY96] and Java byte code [LY96] is also helpful for the understanding of this paper.

2 Java Byte Code Verification

The Java runtime system loads classes from class files, which are often fetched over a network on demand. For each newly loaded class, its class file is checked by the class file verifier, part of which is the byte code verifier.

The checks performed can be roughly divided into mainly syntactic and mainly semantic checks. Examples of syntactic checks are verifying the correct syntactic format of the class file, the length of attributes, the correct declaration of the class, its fields and methods, etc. Since these syntactic checks are comparably simple and not of particular interest for the sequel of this paper, we will not elaborate on them. We will concentrate on the semantic checks of the verifier, in particular, we will show in Section 4 how these can be implemented using a model checker.

2.1 Semantic Checks of the Java Byte Code Verifier

The byte code verifier ensures that the byte code itself, which implements Java methods, is "legal", i.e. it follows certain rules. A detailed, though informal description of what this is supposed to mean can be found in [Yel95,LY96].[1] Essentially, the idea behind byte code verification is to ensure that the byte code has been generated by a conformant compiler, thus observing several implicit rules. The Java interpreter within the client takes conformance of the byte code

[1] From a rigorous point of view one can argue that the byte code verifier does in fact not perform a verification step: verification as an isolated notion does not make sense, instead one can only verify something *against* a specification. If such a specification is not explicitly given, the notion of "verification" is at least from a formal point of view misleading.

to these rules for granted, mainly for reasons of efficiency. Therefore, byte code verification is essential for Java security since it is not obvious that the byte code always conforms to these rules: it could have been manipulated or an attacker could even have written malicious byte code by hand for crashing the Java interpreter.

The semantic checks in the byte code verification process form the most complex part of the class file verification. In particular, a data flow analysis of each method is carried out, which is is closely related to the Java type system: Starting with the first instruction of a method, the effect of that instruction to the operand stack and to local variables is computed. This computation does not consider particular values of fields or variables but their type information, only. The state reached thereby is taken as a precondition for the following instructions. Subsequently, such a simulation is applied to all instructions of a method.

For secure execution, several conditions must hold for each instruction, depending on the type of instruction:

- For each instruction, an appropriate number of parameters must be on the operand stack, local variables must have appropriate types, and there must be enough room on the stack for storing the results of the instruction.
- If an instruction can be reached by different execution paths, the values of the operand stack and the local variables must have compatible types in all these paths.
- Each method of an object that is called must have been initialised by certain initialisation methods, and the type returned by a method must match its declaration.
- Jumps to invalid instructions are forbidden, and execution must not go beyond the last byte of the code.
- Additionally, there are certain restrictions on how the instructions for subroutines may be used. Subroutines are (among other things) used for implementing `try/finally` constructs in the Java language [LY96].

In summary, a byte code program that meets these criteria is supposed to be safe and not capable of "breaking" the virtual machine. A program which fails to meet one of these criteria is refused since its safe execution can not be guaranteed, and it might lead to gaining unauthorised access to resources on the client by crashing the Java interpreter.

2.2 Problems of the Current Approach

We see several disadvantages of the byte code verifier and the current security architecture of Java:

Lack of formality. Current implementations of byte code verification lack formality, in the sense that there is no formal model of what the byte code verifier exactly performs, and what exact properties a successfully verified applet actually has.

This lack of formality makes it difficult to avoid bugs in a verifier's implementation, as the history of Java bugs clearly demonstrates [DFW96,Pos98].

Waste of space on the client's side. The byte code verifier is a program of considerable size, which needs to be installed on the client. This might be acceptable on a PC, but other Java-enhanced devices like PDAs, Java-based phones or Java smart cards have much tighter space constraints.

Waste of client's CPU time. The process of byte code verification consumes a significant amount of CPU time, which is again a problem in PDAs, phones, or smart cards that use comparatively slow processors. Furthermore, the verifier is called each time an applet is loaded, thus resulting in multiple checks of the same byte code.

Altogether, we found that the current security architecture of the Java platform has a number of deficiencies, in particular for application areas like Java smart cards. On such a card, only a few KBytes are available for the complete Java platform including applications, and it will take at least some years until progress in chip manufacturing will allow for implementing an explicit byte code verification on the chip itself. We therefore considered an alternative architecture, where the byte code verification proceeds offline on a server in a network. This also allows to apply a more formal framework for byte code verification which can significantly strengthen Java security.

3 Offline Byte Code Verification

Offline byte code verification is carried out on a server in a network, rather than on the client itself. Successfully verified byte code programs are signed with the verifier's digital signature and passed to the client for execution. Figure 3 depicts this principle. A client loading byte code of an applet just has to check the digital signature of the code to make sure that the byte code has been checked by a verifier it trusts.

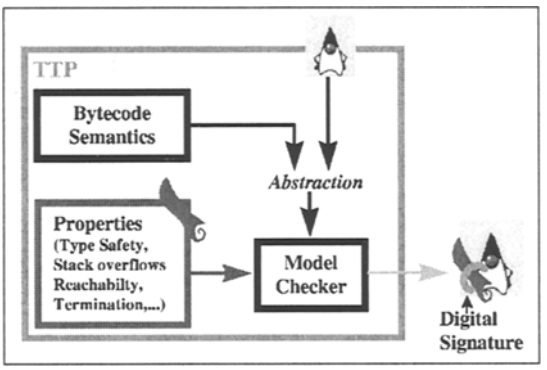

Fig. 3. Offline Byte Code Verification

Note that the verification server in the network does not need to check an applet more than once: it can behave like a proxy server and store results for a while or even check applets in advance. We will first discuss the general advantages of offline byte code verification, and turn to our particular approach using model checking in Section 4.

Offline byte code verification has a number of significant advantages:

- It allows the use of (nearly) unlimited resources in the process of byte code verification, because the resource limitations on a client platform (both in time and space) are circumvented.
  Offline verification is only limited by the amount of time a user is willing to wait when loading applets. In many application areas it is even acceptable that applets are made known to the verifier before being used, so offline byte code verification may indeed spend several minutes or even hours or days for analysing the byte code of an applet and invest as many resources as are available on a (powerful) server. In particular, this allows one to use a formal framework for byte code verification, like the one we propose in Section 4.
- Offline byte code verification can be customised to special needs without having access to the runtime platform.
  So far the byte code verification in the standard Java environments has come as a fixed "bundle" of tests applied to the byte code. These tests might or might not suit the needs of users. For instance, one could think to include approaches like proof-carrying code [NL96] or even apply semi-automatic approaches, like a functional verification of the byte code.
- It saves resources at the client's platform since a client that loads an applet just has to check the digital signature of the incoming code.
- It offers potentially more security than the current architecture: The current architecture depends on verification engines that are distributed and individually used. Several bugs in commercial products have shown that it is occasionally necessary to replace these components, which is expensive in time and money for both the manufacturer and the customer. Further, it is unlikely that all customers are aware of security issues and thus use the latest versions or bugfixes. In contrast to this, a central component can easily be replaced as errors show up. Overall security in the network is no longer defined by the weakest client, but depends only on one single well-maintained, well-implemented, well-protected component.

We see also some disadvantages of offline byte code verification; in particular:

- It requires a trusted service in the network since otherwise the result of the verification process would be arbitrary.
- There might be problems with scaleability if such a verification service is implemented naively: If plenty of different applets and clients are involved, a verification server might run into performance problems and form a bottleneck for the clients. However, for application areas where only a small number of different applets are used, this drawback is negligible, and it vanishes completely if these applets can be made known to the server in advance.

Overall, we consider the advantages of offline byte code verification to clearly outweigh its disadvantages for most application areas. In particular, such an approach allows the integration of formal methods into the process of byte code verification, as described in the sequel.

4 Model Checking of Java Byte Code

Given a description of a state transition system and a temporal logic formula, a model checker [EC82] decides whether the formula holds in the system. This works by completely examining the system's state space, which therefore needs to be finite.

A byte code program can be viewed as a description of a state transition system performing transitions on states of the JVM. The state space of the JVM is of course potentially infinite, therefore model checking is not directly applicable to such a system, and we need to derive a *finite abstraction*, first (cf. [CGL94]).

On the abstract model, it is possible to check properties which are formulated in a temporal logic like CTL. In such a logic, it is possible to express system properties involving information on states (e.g. typing of variables) and the system's dynamic behaviour. The properties which are to be checked by the process of byte code verification as defined in [LY96] can be expressed in CTL. Thus, by feeding a model checker with the model description and the respective CTL formulas, the model checker acts as a byte code verifier.

4.1 Building an Abstract Model

Byte code verification is applied to each method separately, therefore only the information relevant to a single method has to be included in the abstract model. Thus, a state of the abstract model essentially consists of a program counter, an operand stack, and the local and instance variables. Additional state variables are added for the bookkeeping of subroutine information and exceptions. References to the constant pool, which contains data used class-wide, can be pre-evaluated as these data items do not change.

Certain instructions can throw exceptions. This means that an exception object is placed on the operand stack and execution continues at a specific point which is the entry of an exception handler. If no handler is defined explicitly, a default handler in the JVM is invoked: it takes the exception object from the stack, terminates the current method, returns to the calling method and rethrows the exception there. This recursive procedure ends finally with terminating the program. In contrast, if there exists an explicitly defined exception handler, execution simply continues with its code. This behaviour is coded into the transition relation, where the default handler is modelled by a transition to the final state with the *exception flag* set to *true*.

Since the properties of the byte code related to its security deal mainly with type safety, the abstract model can be essentially restricted to type information. This allows the abstract model to be finite because

- a method compiled from a Java program uses only an operand stack of restricted size,
- the number of local variables is finite,
- there are only finitely many instructions in a method,

– the number of types used in a method is finite.

As we want the properties checked on the abstracted system to hold also on the original JVM, a homomorphic mapping between the JVM and the abstract model is established. A formal treatment requires, of course, a formal model of the original JVM (a concrete model); there are several approaches to achieve this, e.g. [Coh96,Qia97]. The mapping ensures the following property:

if φ holds in the abstract model, then it holds also in the concrete one

It can be shown that this holds for CTL formulas φ quantified over all execution paths (see e.g. [CGL94]), and we will see that the relevant byte code properties can be expressed as formulas of that class.

To make the description of the transition system complete, the set of initial states is defined by setting the state variables to their initial values: the operand stack is empty and the program counter points to the first instruction; the values of the local variables are determined by the method's signature, which specifies the types of the parameters, the initialization of the remaining state variables is straighforward.

This describes the idea behind constructing a finite state transition system from a byte code program. Since our research is motivated by Java smart cards, we will not consider certain features of the Java Virtual Machine (JVM) which are not supported by Java for smart cards (see [Sun97b] for details). This includes multi-dimensional arrays, multi-threading, and dynamic class loading.

4.2 Formal Security Properties

The security properties described informally in Section 2.1 have to be formulated in an appropriate logic (such as CTL) if a model checker is supposed to reason about them. These formulæ can then be easily validated against the informal description as they are high-level and understandable by human readers.

Another way of obtaining the formulæ is to extract them from a formal description of a defensive JVM. A defensive JVM performs all checks at runtime needed for the secure execution of a program. Such a model of the JVM is partially described in [Coh96]. The conditions to be checked by the model checker could be formally derived from such a description.

Most properties are local to a specific program point. That means: Before an instruction can be safely executed, the state of the JVM has to meet a certain condition. This is expressed by the CTL formula

$$\texttt{AG pc} = n \rightarrow \phi$$

where pc denotes the program counter, n a program point (the address of an instruction) and ϕ is a propositional condition on the JVM state. In plain words: Whenever program point n is reached during execution, the condition ϕ must hold.

To make sure that a local variable i contains a value of a specific type T, ϕ is of the form $\texttt{loc}_i = \texttt{T}$ (loc is the array of local variables). The formulæ for checking the size and typing of the operand stack are similar.

Some conditions can be checked for all program states in general, like that there are not stack overflows. This is achieved by specifying the formula

$$\texttt{AG}\ \textit{size(stack)} \leq \textit{max\_stack}$$

which imposes on all states the condition that the stack does not exceed its maximum size.

```
public class Purse {
  int cash;  // money stored
             //   in the purse
  int ta;    // transaction
  ...        //   counter
  void notifyBank() throws
    Exception {
    ...
  }
  boolean debit(int amount) {
    try {
      if (this.cash < amount) {
        notifyBank();
        return false;
      } else {
        this.cash -= amount;
        return true;
      }
    }
    finally {
      this.ta++;
    }
  }
}

  0 aload_0
  1 getfield #7 <Field int cash>
  4 iload_1
  5 if_icmpge 19
  8 aload_0
  9 invokevirtual #8 <notifyBank()>
 12 iconst_0
 13 istore_2
 14 jsr 42
 17 iload_2
 18 ireturn
 19 aload_0
 20 dup
 21 getfield #7 <Field int cash>
 24 iload_1
 25 isub
 26 putfield #7 <Field int cash>
 29 iconst_1
 30 istore_2
 31 jsr 42
 34 iload_2
 35 ireturn
 36 astore_3
 37 jsr 42
 40 aload_3
 41 athrow
 42 astore 4
 44 aload_0
 45 dup
 46 getfield #9 <Field int ta>
 49 iconst_1
 50 iadd
 51 putfield #9 <Field int ta>
 54 ret 4
Exception table:
   from   to  target type
     0    36    36   any
```

Fig. 4. Implementation of a Simple Electronic Purse: Source Code and Byte Code.

5 Example: A Simple Electronic Purse

As an example for performing byte code verification through a model checker, we consider the method `debit` of class `Purse` (see figure 4), implementing a simple electronic purse running on a Java smart card; this example was implemented for the Cyberflex Java Smart Card [Sch97].

The method `debit` is invoked on a `Purse` object whenever money is withdrawn. `debit` takes the amount of money as an argument and returns *true* if the transaction was successful and *false* in case the value stored in the purse would become negative. In the latter case, a method for notifying the bank is invoked. In any case, the transaction counter is incremented. The respective code is enclosed in a `finally` clause, so it is performed even if an exception is raised from calling `notifyBank`.

We first take a look at the state transition system built from the compiled method `debit` and how it reflects the semantics of the byte code instructions. Then we consider the formulæ describing the properties established during byte code verification.

5.1 A Transition System for `debit`

Figure 4 shows the compiled method `debit` (as created by javac from Javasoft). From this piece of byte code, a state transition system is built and given to a model checker, together with conditions to check. Figure 5 shows the corresponding input for the model checker SMV. This description was created manually, but it can be easily computed automatically with an appropriate tool.

A state of the transition system consists of the following state variables:

- a program counter `pc` ranging over the instruction addresses and an additional final state;
- an operand stack, implemented by a stack pointer `sp` and stack locations `st0` through `st2` (the maximum height of the stack is 3 as recorded in the class file);
- local variables `loc0` through `loc4`;
- a stack for active subroutines `rstack`;
- an `exception` flag which indicates whether the method ends because of an exception;
- object variables `field7` and `field9`.

Other state types are possible, and it is desirable to restrict the state space as much as possible. A candidate for a more efficient representation is the operand stack. Here, the currently unused stack locations unnecessarily expand the state space.

Some simplifications were made for building the model: The only primitive data type is integer (booleans are treated as integers, and no other primitive types are used in the program). Furthermore, the access conditions for object variables (fields) and methods which could be violated by the `getfield`, `putfield` and `invokevirtual` instructions are not handled.

The types for stack locations and local variables are similar: They range over a set of reference values (null, this, exception, and arbitrary reference), a primitive type (integer) and return addresses (used by the `ret` instruction). The additional value `UNDEF` for local variables indicates that a variable has not yet been initialised and therefore cannot be used.

The object variables are of type integer. They are strictly typed and may not hold arbitrary values like local variables do.

Note that the stack pointer ranges over values from `0` to `4`, where `4` stands for "invalid". The formulæ describing the transitions are partial in the sense that for invalid values, the value of `sp` in the next state is not determined. This leads to the existence of execution paths, where `sp` has a value of `4`. However, by the specification `SPEC AG !sp=4` we can assure that these paths are not reachable and therefore the stack never exceeds its maximum (or minimum) size.

Initial state In the initial state, the stack is empty and the program counter points to the first instruction of the program. The local variable `0` contains the reference to the object for which the method was called. `loc1` holds the method parameter which is an integer value. The other local variables are marked `UNDEF` and are therefore unusable for any reading instruction. There are no active subroutines, no exception has been raised, and the object variables are assumed to contain valid values (see the `INIT` lines in Figure 5).

State transitions The transitions are described by formulæ using the variable values of the current and the following states. The description reflects the structure of the byte code program. Generally, the actual transition formula is guarded by a formula `pc=Lnn`, indicating that this transition should be made when program point `Lnn` is reached (see the `TRANS` statements in Figure 5).

A transition manipulates the stack, changes some variables or raises an exception. This is expressed by describing which values the state variables will have in the next state, depending on their values in the current state. It must be stated explicitly if state variables do not change their values. Otherwise, the model checker chooses an arbitrary value (as for unused stack locations).

Exception handling The Java method considered here performs no explicit exception handling, i.e.: all exceptions raised while executing the method are handed over to the caller of the method. However, the `finally` clause must be executed even if an exception occurs. Therefore the compiler added an exception handler (ranging from address 36 to 41 in Figure 4) which calls the `finally` clause and then re-throws the exception, i.e. hands it upwards.

The instructions in our example which could possibly raise exceptions are `invokevirtual`, `getfield` and `putfield`. There are several `getfield` instructions in the method which raise exceptions if their respective object reference is null. As an example, look at the instruction at address 1 which is covered by the exception handler, so in case of a null reference a jump to the code of the exception handler is performed. Thus, for address 1 the model description includes a transition to address 36 guarded by the appropriate condition.

In contrast, if the instruction at address 46 raises an exception, the method is immediately interrupted and control is returned to the caller of the method. This is reflected in the abstracted model description by a transition to the final state with the exception flag set. We do not model the transfer of control here, as we consider only one method at a time.

Subroutine calls There is one subroutine in the method, implementing the code enclosed in the `finally` clause. This code can be called from 3 different locations (the `jsr` instructions at addresses 14, 31, 37). Although not obvious, the variable `rstack` holds a stack where labels for all active subroutines are pushed on. As we have only one here, there are two possible values for `rstack`: `EMPTY` if the subroutine is not active and `JSR42` in the other case. (The label `JSR42` indicates that the subroutine starts at address 42.)

Whenever a `jsr` instruction is executed, the subroutine must not already be active, as the Java byte code specification forbids recursive calls to subroutines. Therefore, the condition `rstack=EMPTY` has to be true for each program point with a `jsr` instruction.

5.2 Byte Code Properties

The properties sketched below describe the type safety of the program. They ensure that all operations have operands of suitable types, and that there are no stack underflows or overflows. They also ensure that subroutines are not recursively called.

```
SPEC AG (pc=L0 ->(sp<3 &(loc0=NULL | loc0=THIS | loc0=EXC | loc0=REF)))
SPEC AG (pc=L1 ->(sp>0 &
   (((sp=1 -> NULL=st0) & (sp=2 -> NULL=st1) & (sp=3 -> NULL=st2)) |
    ((sp=1 -> THIS=st0) & (sp=2 -> THIS=st1) & (sp=3 -> THIS=st2)))))
                                    ...
                                [omitted]
                                    ...
SPEC AG (pc=L54 -> (loc4=L17 | loc4=L34 | loc4=L40))
SPEC AG (!sp=4)
```

5.3 Results

The model description shown can be fed into the model checker SMV [CGL95]. It reports a total state space of $8.5 \cdot 10^{10}$ states, of which 8531 are reachable. All properties are checked and reported to be valid in a total amount of time of 0.86 seconds (on a Sun SparcStation 20).

Several points that could be verified by the model checker were already pre-evaluated while building the model. For example, the model checker could verify that jumps go only to valid instructions. The model shown here incorporates this condition by having only valid instruction addresses in the range of the program counter.

6 Related Work

The principle of offline byte code verification has, to our knowlege, first been described within the context of the Kimera project [SGB98]; the separation was proposed mainly for reasons of efficiency. The project also addresses the security of Java runtime environments, by providing a more carefully tested and "cleaner" implementation of the byte code verifier. Kimera does not attempt to put the verification step on rigorous approaches like formal methods.

Closely related work towards a formal treatment of Java is currently being undertaken at several places: Attempts to define a formal semantics for subsets of the Java language are pursued by [NvO98,DE97,Sym97]; this work chiefly aims at proving that the Java type system is sound and uses type-theoretic approaches. These approaches do not consider Java byte code and are therefore not directly applicable to our approach.

An approach to formally define the semantics of the complete Java language using Abstract State Machines is described in [BS98]. Their formulation defines the mapping from the Java language to Java byte code in several layers of abstraction. This work is related to our approach since ASM can be seen as state transition systems. Although [BS98] do not consider proving properties of concrete byte code programs, the results could eventually contribute to mapping byte code programs into the finite abstractions used in our approach.

A definition of the semantics of the Java Virtual Machine is described in [Qia97]. The approach considers a subset of the JVM instructions and aims at proving the run-time type correctness of byte code programs from their static typing. It yields an operational semantics of Java byte code and a static type inference system.

A similar approach is taken in [SA98]; the paper concentrates on the `try/finally` construct in the Java language and the Java byte code instructions relevant to this. The difference to our approach is that [SA98] uses a language more expressive than the finite state systems we consider; therefore, a more detailed description of the semantics of byte code verification is possible, at the price of a more complex modelling. The approach goes into a very similar direction to our work, but it concentrates on describing the byte code verification, whilst we focus on carrying it out with declarative means.

Our approach differs –on the meta level– from proof carrying [NL96,FL97] code in that we replace the process of checking the proof by verifying a digital signature. Rather than requiring a proof checker on the client side, we assume safe distribution of keys, trust in a network node and signature verification on the client side. However, it is foreseeable that a public key infrastructure will be widely available in the future, so very little *additional* overhead is brought into the client by our procedure. Technically, we use a less expressive (because decidable) language for expressing properties than proof carrying code. This allows us to perform proofs fully automatically, while with proof carrying code, the question of where the proofs actually come from, still needs to be sorted out in detail.

7 Conclusion

We described an approach to Java byte code verification, where the verifier is not integrated into the Java run time environment, but proceeds offline. The process is implemented by using a model checker on an explicit, finite model of a byte code program. The checker verifies that the required security conditions, which are also given explicitly, hold for that model and are therefore fulfilled by the original byte code program.

The proposed method of formal byte code verification helps achieving the highest level of security in execution of Java applets. This is done by replacing a crucial part of the Java security architecture, the byte code verifier, by a tool based on formal methods that avoids the pitfalls of conventional implementations.

To summarise, the advantages of our approach are:

More flexibility and extensibility. In the sketched framework, we deal with high-level descriptions of system models and safety properties. The described abstract model of the JVM is not the only one possible, and the model and the properties to be checked can be adapted to individual needs.

Achieving formal correctness. The highest achievable level of correctness is a formal proof. The correctness of an implementation of the byte code verifier is almost impossible to prove, and the benefit is questionable, as a little change invalidates the whole proof.
In our approach, we shift the proof condition from the implementation level to the level of validation, of both the model description and the formulæ. But as these descriptions are high-level, the task becomes relatively easy. Furthermore, as errors show up, they can be corrected quickly and safely.
Of course, there is still no guarantee that an actual implementation of a JVM will execute a verified byte code program correctly. To achieve this, we need a formally verified implementation of the JVM and the whole environment. For smart cards, even this task seems tractable.

We believe that through model checking, a formal treatment of byte code can be achieved in a simple and effective way. Understanding the formalisms needed for this kind of byte code verification is not more complicated than implementing a byte code verifier, and the gain is a formal, highly trustable verifier.

References

[BS98] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[CGL94] E. Clarke, D. Grumberg, and D. Long. Model Checking and Abstraction. *ACM Trans. on Prog. Languages and Systems*, 16(5):1512–1542, 1994.

[CGL95] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency - Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[Coh96] Richard M. Cohen. The Defensive Java Virtual Machine Specification Version, Alpha 1 Release. Technical report, Computational Logic, Inc; http://www.cli.com/software/djvm/html-0.5/djvm-report.html, 1996.

[DE97] Sophia Drossopoulou and Susan Eisenbach. Proving the Soundness of the Java Type System. Working Paper, Imperial College, Dept. of Computing, London, UK, Feb. 1997.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From HotJava to Netscape. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1996. IEEE. http://www.cs.princeton.edu/sip/pub/secure96.html.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2(3):241–266, December 1982.

[FL97] J. Feigenbaum and P. Lee. Trust Management and Proof-Carrying Code for Mobile Code Security. In *DARPA Workshop on Foundations of Mobile Code Security*, Monterey, CA, 26-28 March 1997.

[GS96] J. Gosling and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[KP98] Matthias Kaiserswerth and Joachim Posegga. Java Chipkarten. *Informatik-Spektrum*, 21(1):27–28, 1998.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[NL96] G. Necula and P. Lee. Proof-Carrying Code. Technical Report CMU-CS-96-165, Carnegie Mellon University, School of Computer Science, Pittsburg, PA, September 1996.

[NvO98] Tobias Nipkow and David von Oheimb. Java$_{tight}$ is Type-Safe — Definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, 1998.

[Pos98] Joachim Posegga. Die Sicherheitsaspekte von Java. *Informatik-Spektrum*, 21(1):16–22, 1998.

[Qia97] Z. Qian. A Formal Specification of Java Virtual Machine Instructions. (unpublished manuscript), 1997.

[SA98] Raymie Stata and Martín Abadi. A Type System for Java Bytecode Subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, 1998.

[Sch97] Schlumberger, Inc. Cyberflex 2.0 Multi 8K Programmer's Guide. http://www.cyberflex.austin.et.slb.com/cyberflex/docs/docs-page3.htm, 1997.

[SGB98] Emin Gün Sirer, Arthur J. Gregory, and Brian N. Bershad. Kimera: A Java System Architecture. http://kimera.cs.washington.edu/, 1998.

[Sun97a] Sun Microsystems, Inc. Java Card 2.0 Application Programming Interfaces, Revision 1.0 Final. http://java.sun.com:80/products/javacard/, October 13 1997.

[Sun97b] Sun Microsystems, Inc. Java Card 2.0 Language Subset and Virtual Machine Specification, Revision 1.0 Final. http://java.sun.com:80/products/javacard/, October 13 1997.

[Sym97] Don Syme. Proving Java Type Soundness. Technical report, University of Cambridge Computer Laboratory, 1997.

[Yel95] Frank Yellin. Low Level Security in Java. In *WWW4*, 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html.

```
MODULE main
VAR pc : {L0,L1,L4, ... ,L51,L54,FINI};
    sp : {0,1,2,3,4};
    st0,st1,st2 : {NULL,THIS,EXC,REF,INT,L17,L34,L40};
    loc0..4 : {UNDEF,NULL,THIS,EXC,REF,INT,L17,L34,L40};
    rstack : {EMPTY, JSR42};  exception : {0,1};
    field7 : {UNDEF,INT};  field9 : {UNDEF,INT};
INIT pc=L0 & sp=0 & loc0=THIS & loc1=INT & loc2=UNDEF & loc3=UNDEF
INIT loc4=UNDEF & rstack=EMPTY & exception=0 & field7=INT & field9=INT
-- aload_0
TRANS pc=L0 ->((sp=0 ->(next(st0)=loc0 & next(sp)=1)) &
  (sp=1 ->(next(st0)=st0 & next(st1)=loc0 & next(sp)=2)) &
  (sp=2 ->(next(st0)=st0 & next(st1)=st1 & next(st2)=loc0 & next(sp)=3))) &
  (next(loc0)=loc0 & next(loc1)=loc1 & next(loc2)=loc2 & next(loc3)=loc3 &
   next(loc4)=loc4 & next(rstack)=rstack & next(field7)=field7 &
   next(field9)=field9 & next(exception)=exception) & next(pc)=L1
-- getfield #7
TRANS (pc=L1 & !((sp=1 ->NULL=st0) & (sp=2 ->NULL=st1) & (sp=3 ->NULL=st2)))
 ->((sp=1 ->next(sp)=1 & next(st0)=field7) &
    (sp=2 ->next(sp)=2 & next(st0)=st0 & next(st1)=field7) &
    (sp=3 ->next(sp)=3 & next(st0)=st0 & next(st1)=st1 & next(st2)=field7)) &
  (next(loc0)=loc0 & next(loc1)=loc1 & next(loc2)=loc2 & next(loc3)=loc3 &
   next(loc4)=loc4 & next(rstack)=rstack & next(field7)=field7 &
   next(field9)=field9 & next(exception)=exception) & next(pc)=L4
TRANS (pc=L1 & ((sp=1 ->NULL=st0) & (sp=2 ->NULL=st1) & (sp=3 ->NULL=st2)))
->next(pc)=L36 &
  (next(loc0)=loc0 & next(loc1)=loc1 & next(loc2)=loc2 & next(loc3)=loc3 &
   next(loc4)=loc4 & next(rstack)=rstack & next(field7)=field7 &
   next(field9)=field9 & next(exception)=exception) & next(sp)=1 & next(st0)=EXC
                                    ...
                                [omitted]
-- ret 4                            ...
TRANS pc=L54 ->((sp=0 ->(next(sp)=sp)) & (sp=1 ->(next(sp)=sp & next(st0)=st0))
 & (sp=2 ->(next(sp)=sp & next(st0)=st0 & next(st1)=st1)) &
    (sp=3 ->(next(sp)=sp & next(st0)=st0 & next(st1)=st1 & next(st2)=st2))) &
  (next(loc0)=loc0 & next(loc1)=loc1 & next(loc2)=loc2 & next(loc3)=loc3 &
   next(loc4)=UNDEF & next(rstack)=EMPTY & next(field7)=field7 &
   next(field9)=field9 & next(exception)=exception) & next(pc)=loc4
-- final state
TRANS pc=FINI ->((sp=0 ->(next(sp)=sp)) & (sp=1 ->(next(sp)=sp & next(st0)=st0))
 & (sp=2 ->(next(sp)=sp & next(st0)=st0 & next(st1)=st1)) &
    (sp=3 ->(next(sp)=sp & next(st0)=st0 & next(st1)=st1 & next(st2)=st2))) &
  (next(loc0)=loc0 & next(loc1)=loc1 & next(loc2)=loc2 & next(loc3)=loc3 &
   next(loc4)=loc4 & next(rstack)=rstack & next(field7)=field7 &
   next(field9)=field9 & next(exception)=exception) & next(pc)=FINI
```

Fig. 5. Sketch of the `debit` Method as a State Transition System.