An Operational Semantics of the Java Card Firewall*†

Marc Éluard[1], Thomas Jensen[1] and Ewen Denney[2]

[1] IRISA
Campus de Beaulieu
35042 Rennes Cedex, France
{eluard,jensen}@irisa.fr

[2] Division of Informatics
University of Edinburgh
Edinburgh, Scotland
ewd@dai.ed.ac.uk

Abstract This paper presents an operational semantics for a subset of Java Card bytecode, focussing on aspects of the Java Card firewall, method invocation, field access, variable access, shareable objects and contexts. The goal is to provide a precise description of the Java Card firewall using standard tools from operational semantics. Such a description is necessary for formally arguing the correctness of tools for validating the security of Java Card applications.

1 Introduction

Java Card is being promoted as a high-level language for programming of multi-application smart cards. The high-level nature of the language should ease the programming and the reasoning about such applications. Java Card keeps the essence of Java, like inheritance, virtual methods, overloading, etc, but leaves out features such as large primitive data types (`long`, `double` and `float`), characters and strings, multidimensional arrays, garbage collection, object cloning, the security manager, etc. (see the specification [1] and also [8]). Furthermore, given the security-critical application areas of Java Card, the language has been endowed with an elaborate security architecture.

Central to this architecture is the Java Card firewall. Applets installed on the card are separated by a firewall that prevents one applet from accessing objects owned by another applet. Shareable objects and interfaces are used to provide communication between otherwise separated applets. A limited form of stack inspection allows a server applet to know the identity of the client that requested a particular service. These mechanisms (that will be further detailed in section 2) facilitate the design of secure applications but do not in themselves guarantee security. They do, however, offer the possibility of formal verification of the security of an application using tools from semantics and static program

* Work partially funded by the GIE Bull-Inria "Dyade" and by European FET/Open Project IST–1999–29075 "Secsafe"

† Published in the Proceeding of Smart Card Programming and Security (ESMART), volume 2140 of *Lecture Notes in Computer Science*, pages 95–110. © Springer-Verlag, September 2001.

analysis. The purpose of this paper is to give a formal semantic description of the Java Card firewall. The interest of such a description lies in its use as a foundation for designing and proving static analysis methods for verifying the security of a multi-application Java Card, but for lack of space we do not detail this here.

The paper is organised as follows. In section 2 we give a description of the central security features of Java Card 2.1.1. This is followed by the definition of semantics domains in section 3 and the operational semantic of selected bytecode in section 4. In section 5, we discuss related work.

2 The Java Card firewall

The Java Card platform is a multi-application environment in which an applet's sensitive data must be protected against malicious access. In Java, this protection is achieved by using class loaders and security managers to create private name spaces for applets. In Java Card, class loaders and security managers have been replaced with the Java Card firewall. The separation that is enforced by the firewall is based on the package structure of Java Card (which is the same as that of Java) and the notion of *contexts*.

When an applet is created, the JCRE gives it a unique applet identifier (AID) from which it is possible to retrieve the name of the package in which it is defined. If two applets are instances of classes coming from the same Java Card package, they are said to belong to the same context (which we identify by the package name). In addition to the contexts defined by the applets executing on the card, there is a special "system" context, called the JCRE context. Applets belonging to this context can access objects from any other context on the card. Thus, the set *Contexts* of contexts can be defined by:

$$Contexts = \{JCRE\} \cup \{pckg : pckg \textit{ is a legal package name}\}$$

Every object is assigned a unique *owner context viz.*, the context of the applet that created the object. A method of an object is said to execute in the owner context of the object[1]. It is this context that decides whether an access to another object will succeed. The firewall isolates the contexts in the sense that a method executing in one context cannot access any fields or methods of objects belonging to another context.

There are two ways in which the firewall can be circumvented: via JCRE entry points and via shareable objects. JCRE entry points are objects owned by the JCRE that have been designated specifically as objects that can be accessed from any context. The most prominent example is the APDU buffer in which commands sent to the card are stored. This object is managed by the JCRE and in order to allow applets to access this object, it is designated as an entry point. Other examples include the elements of the table containing the AIDs of the applets installed on the card. Entry points can be marked as *temporary*.

[1] In the case of static call, the execution is in the caller's context.

References to temporary entry points cannot be stored in objects (this is enforced by the firewall).

Two applets in different contexts may want to share some information. For that, Java Card offers a sharing mechanism, called *shareable objects*, that gives limited access to objects across contexts. An applet can allow access to an object's methods from outside its context (it is impossible to share fields) by using a shareable interface that is, an interface which extends `javacard.framework.Shareable`. In this interface, the applet gives the list of the method's signatures that it wants to share. The class of the object to be shared must implement this interface. The "server" applet must define a method, `getShareableInterfaceObject`. This method is called when an applet is asked to provide a shared object. It is passed as parameter the AID of the "client" applet which requested the shared object. This allows different objects to be shared with client applets.

In the following section, we give a small example to illustrate these sharing mechanisms.

2.1 A simple scenario

We have 3 applets: Alice, Bob and Charlie, each belonging to a different context. Alice implements a shareable interface MSI and she is prepared to share an object MSIO with Bob (MSIO is an instance of a class that implements MSI). When Alice receives a request for sharing (using the method `getShareableInterfaceObject`, she verifies that the caller is Bob. If it is Bob, she returns the MSIO, otherwise she returns null (see also section 4).

```
public class Alice extends Applet implements MSI {
  public Shareable getShareableInterfaceObject (AID client, byte param){
     if (client.equals (BobAID, (short)0, (byte)BobAID.length) == false)
        return null;
     return (this); } }
```

Using the method `JCSystem.getAppletShareableInterfaceObject`, Bob asks for a shareable object from Alice. Assume now that Bob (inadvertantly) leaks a reference to MSIO to the third applet Charlie[2]. With it, Charlie can access the same methods as Bob.

```
public class Bob extends Applet {
  public static MSI AliceObject;
  AliceObject =
     (MSI)JCSystem.getAppletShareableInterfaceObject(AliceAID,(byte)0);}

public class Charlie extends Applet {
  private static MSI AliceObject;
  AliceObject = Bob.AliceObject;
  // The method void foo () exists in MSI
  AliceObject.foo (); }
```

----

[2] for example, by storing the reference into a public static field (there are other more subtle ways in which this can happen)

Alice has some doubts about Bob so she decides to verify, at each access to one of her shared methods, the identity of the caller. In this case Charlie can't access MSIO anymore.

```
public class Alice extends Applet implements MSI {
  public void foo () {
     // The caller is Bob?
     AID client = JCSystem.getPreviousContextAID ();
     if (client.equals (BobAID, (short)0, (byte)BobAID.length) == false)
        ISOException.ThrowIt (SW_UNAUTHORIZED_CLIENT);
     ... // OK, the caller was Bob } }
```

2.2 Limitations of the firewall

As illustrated by this example, Java Card has a limited form of the stack inspection mechanism that underlies the Java 2 security architecture. The Java 2 `checkPermission` instruction verifies whether all callers on the call stack have a specific permission (e.g. to write a file in a given directory). Java Card contains a mechanism for knowing from which context a method was called but there is no mechasnism for obtaining the identity of all the callers. More precisely, an applet can get a description of the last *context switch* that took place, by calling the method `getPreviousContextAID`. (Notice that this context switch could have happened several levels down in the call stack.) In the example, Alice does not know whether the call made by Bob is in turn a result of Bob being called by some other applet. Neither can she know what Bob will do with the result of the call. This is problematic since an object is only marked as shared, not with whom it is supposed to be shared. Thus, while the firewall can serve to prevent direct information flow, further program analysis is required in order to verify that all information flow of the application respects a given security policy.

3 Semantic description of the Java Card firewall

In the following we describe the semantic domains of a modified version of Java Card bytecode. Rather than a stack-oriented bytecode we shall be working with a "three-address" bytecode where arguments and results of a bytecode instruction are fetched and stored in local variables instead of being popped and pushed from a stack. This format is similar to the intermediate language used in the Java tool Jimple [17]. Furthermore, we assume that the constant pool has been expanded *i.e.* that indices into the constant pool have been replaced by the corresponding constant. For example, the bytecode instruction `invokevirtual` takes as parameter the signature of the method called, rather than an index into the constant pool. The transformation of code into this format is standard and straightforward.

3.1 Notation

The term $\mathcal{P}(X)$ denotes the power set of X: $\mathcal{P}(X) \equiv \{S \mid S \subseteq X\}$. A product type $X = A \times B \times C$ is sometimes treated as a labelled record: with an element x of type X we can access its field with the names of its constituent types ($x.A$, $x.B$ or $x.C$). A list can be given by enumeration of its elements: $x_1 :: \cdots :: x_n$. Given a list v we can access one of its element by its position in the list ($v(i)$ for the ith element). And finally, we can concatenate two lists: $(x_1 :: \cdots :: x_n) ::: (x_m :: \cdots :: x_p) = x_1 :: \cdots :: x_n :: x_m :: \cdots :: x_p$. We denote, by X^*, the type of finite lists, whose elements are of type X. We use the symbol $\rightarrow$ to form the type of partial functions: $X \rightarrow Y$. We can update a function f with a new value v for an argument x: $g := f[x \mapsto v]$. We (ab)use the same notation for objects: the object obtained from object o by modifying field f to have value v is written $o[f \mapsto v]$.

3.2 Semantic types

Our semantic domains follow the same structure as the domains defined by Bertelsen [5,6].

Before introducing the representation of the different elements, we define some basic types. Id_p, Id_c, Id_i, Id_f and Id_m are the types of qualified name of a package, a class, an interface, a field and a method, respectively[3]. When we want to talk about a class or an interface name, we can use the set Id_{ci} ($Id_{ci} = Id_c \cup Id_i$). Id_v is the type of (unqualified) names of variables. We assume furthermore a set Pc of program counters. A program counter identifies an instruction within the whole class hierarchy (i.e. it is relative to a class hierarchy and not just a method). We assume a set *Label* which represents the different labels used with a jump instructions.

General types We use types to stand for abstract primitive values. For example, `byte` instead of *12*. A type can be an array type (type between square brackets), or a simple type where a simple type is a *Primitive* or the name of a class or of an interface (Id_{ci}).

$$\begin{aligned} \mathit{Primitive} &= \{ \texttt{boolean}, \texttt{short}, \texttt{byte}, \texttt{int} \} \\ \mathit{Type} &= \text{'['}\mathit{SType}\text{']'} \cup \mathit{Stype} \\ \mathit{SType} &= \mathit{Primitive} \cup Id_{ci} \end{aligned}$$

Classes A class or an interface descriptor consists of the set of the associated access modifiers ($\mathcal{P}(Mod_{ci})$[4]), the name of the class or interface (Id_{ci}), the name

[3] The qualified name of an entity is the complete name. For a class, it is $p.c$ where p is the name of the package and c the (unqualified) name of the class. For a method ($c.m$) or a field ($c.f$), it is the qualified name of the class and the (unqualified) name of the method or field.

[4] The access modifier `Interface` is used to specify that the declaration is for an interface.

of the direct superclass or the names of direct superinterfaces (*Ext*), the name of the interfaces that the class implements (*Imp*), the name of its package (Id_p), field declarations (*Fld*), method declarations and implementations (*Mtd*). A class must have a superclass, the default being `java.lang.Object`, but an interface can have zero, one or more superinterfaces. Only a class can implement an interface, so for an interface this field is the empty set.

The fields of a class are described by a map from field names (Id_f) to a set of access modifiers ($\mathcal{P}(Mod_f)$) together with a type descriptor (*Type*). The type descriptor defines what type of values can be stored in the field.

$$
\begin{aligned}
Desc_{ci} &= \mathcal{P}(Mod_{ci}) \times Id_{ci} \times Ext \times Imp \times Id_p \times Fld \times Mtd \\
Mod_{ci} &= \{ \texttt{Public}, \texttt{Package}, \texttt{Final}, \texttt{Abstract}, \texttt{Shareable}, \texttt{Interface} \} \\
Ext &= Id_c \cup \mathcal{P}(Id_i) \\
Imp &= \mathcal{P}(Id_i) \\
Fld &= Id_f \rightarrow Desc_f \\
Desc_f &= \mathcal{P}(Mod_f) \times Type \\
Mod_f &= \{ \texttt{Public}, \texttt{Package}, \texttt{Private}, \texttt{Protected}, \texttt{Final}, \texttt{Static} \}
\end{aligned}
$$

Methods The methods are described by a map that to a method signature (*Sig*) associates a method descriptor ($Desc_m$). This structure consists of the set of the associated access modifiers ($\mathcal{P}(Mod_m)$), the code of the method (*Code*), a description of the formal parameters (*Param*) and the local variables of the method (*Varl*). A signature is the name of the method (Id_m) and the list of type descriptors for its parameters ($Type^*$). Code is a list whose elements consist of a program counter value (*Pc*) and the instruction at this address (*Bytecode*). The set of local variables is the list of all variable names (Id_v) with their type descriptor (*Type*).

$$
\begin{aligned}
Mtd &= Sig \rightarrow Desc_m \\
Sig &= Id_m \times Type^* \\
Desc_m &= \mathcal{P}(Mod_m) \times Code \times Param \times Varl \\
Mod_m &= \{ \texttt{Public}, \texttt{Package}, \texttt{Private}, \texttt{Protected}, \texttt{Static} \} \\
Code &= Inst^* \\
Inst &= Pc \times Bytecode \mid Pc \times Bytecode \times Label \\
Varl &= (Id_v \times Type)^* \\
Param &= (Id_v \times Type)^*
\end{aligned}
$$

For this paper we consider a small set of bytecodes that is sufficient for illustrating the different features of the semantics. In the following, NT and T range over local variables. The `invokevirtual` instruction takes as argument a fully qualified method name, indicating the point of declaration of the method. The explainations of these intructions are given in section 4.2.

$$
\begin{aligned}
\mathit{Bytecode} = & \ NT := \texttt{getstatic}\ C.f \\
& \mid \texttt{putfield}\ C.f\ T_1\ T_2 \\
& \mid NT := \texttt{invokevirtual}\ C.m\ T_0\ T_1\ \cdots\ T_n\ S_1{::}\cdots{::}S_n \\
& \mid \texttt{goto}\ \mathit{label} \\
& \mid NT := \texttt{new}\ C \\
& \mid NT := \texttt{invokestatic}\ \mathit{getAID} \\
& \mid NT := \texttt{invokestatic}\ \mathit{getPrevCtx} \\
& \mid NT := \texttt{invokestatic}\ \mathit{getASIO}\ T_1\ T_2
\end{aligned}
$$

3.3 The run-time state

This section defines the run-time values used in the semantics. We are primarily interested in modelling the object structure and ownership so we abstract primitive values such as booleans and integers to their type. In addition to the values already introduced we have a set, *Ref*, of references for modelling the heap of objects. A particular element $\mathit{Null} \in \mathit{Ref}$ denotes the undefined reference. We introduce two kinds of reference, Ref_i (respectively Ref_a) can point to class instances Obj_i (respectively array instances Obj_a) and the union of them with *Null* is *Ref*.

Ownership The notion of ownership in Java Card is very clear, an object is owned by the active applet at the moment of its creation. We extend the definition of an owner with the context (package) of its creation. We model this notion with a pair *(Package, Applet)*.

$$\mathit{Owner} = \mathit{Id}_p \times \mathit{Ref}_i$$

The owner of an applet is the applet itself.

Values

$$
\begin{aligned}
\mathit{Value} &= \mathit{Ref} \cup \mathit{Primitive} \\
\mathit{RValue} &= \mathit{Object} \cup \mathit{Primitive} \\
\mathit{Object} &= \mathit{Obj}_i \cup \mathit{Obj}_a \\
\mathit{Obj}_i &= \mathit{Id}_{ci} \times \mathit{Owner} \times \mathit{JCREep} \times \mathit{tJCREep} \times \mathit{Fldv} \\
\mathit{Fldv} &= \mathit{Id}_f \rightarrow \mathit{Value} \\
\mathit{Obj}_a &= \text{'['}\ \mathit{SType}\ \text{']'} \times \mathit{Owner} \times \mathit{JCREep} \times \mathit{tJCREep} \times \mathit{global} \times \mathit{Elt} \\
\mathit{Elt} &= \mathcal{P}(\mathit{Value})
\end{aligned}
$$

We have three kinds of values: class instances, arrays and primitive values (such as bytes and booleans). A class instance contains the name of the class, the owner of this instance, boolean flags indicating whether or not it is a JCRE entry point and a temporary JCRE entry point (*cf.* section 2) and the set of fields. The set of fields maps a field name to a value.

An array instance is described by the type of its elements, the owner, the information about being an entry point or not, a flag indicating whether the array is global or not and finally the set of its elements.

The state With these types, we can define the state used in the semantics. A state consists of a call stack of frames, the memory and the class hierarchy. The latter is part of the run-time state because it is used to store the values of static fields. We write the call stack as a sequence of frames such that the currently active frame appears as the first element of the sequence.

$$
\begin{aligned}
\mathit{State} &= \mathit{Frame}^\star \times \mathit{Mem} \times E_{ci} \\
\mathit{Frame} &= \mathit{Inst} \times \mathit{Ref} \times \mathit{Locals} \\
\mathit{Locals} &= \mathit{Id}_v \rightarrow \mathit{Value} \\
\mathit{Mem} &= \mathit{Value} \rightarrow \mathit{RValue} \\
E_{ci} &= \mathit{Id}_{ci} \rightarrow \mathit{Desc}_{ci} \times \mathit{FldS} \\
\mathit{FldS} &= \mathit{Id}_f \rightarrow \mathit{Value}
\end{aligned}
$$

The current frame contains the current instruction and a reference to the object on which the method currently executing has been invoked. This object represents the currently active context. An element of the set *Locals* maps a local variable to its value. The memory (also called the heap of objects) is modelled by a map from references to values. The class hierarchy is represented as a map that to a class or an interface name associates its descriptor and its static fields.

AIDs and the table of installed applets The JCRE keeps a table recording the applets installed on the card. These applets are identified by an applet identifier, an *AID*. An AID is an object of class `AID` containing a byte array with a number that identifies the applet together with a method for testing whether two AIDs represent the same number. With a reference to an AID, we can find a reference to the corresponding applet instance through the table *Applet_tbl* of installed applets:

$$\mathit{Applet\_tbl} : \mathit{Ref} \rightarrow \mathit{Ref}.$$

Were we to model the dynamic installation of applets this table would have to be made part of the state but we do not consider this here.

3.4 Auxiliary Functions

We follow Bertelsen [5,6] and define a number of functions that abstract the syntactic structure of a list of bytecodes.

$$
\begin{aligned}
&\mathit{Succ} : \mathit{Inst} \rightarrow \mathcal{P}(\mathit{Inst}) \\
&\mathit{Find} : \mathit{Label} \rightarrow \mathit{Inst} \\
&\mathit{First} : \mathit{Desc}_m \rightarrow \mathit{Inst}
\end{aligned}
$$

The flow of control inside a method is modelled by the function *Succ* that for each instruction yields the set of instructions that can follow in the execution. The need for returning a set of instructions is due to the fact that we have abstracted away all primitive values; in particular, the semantics will not be able to evaluate the value of the condition in a branching statement. The function *Find* permits us to find an instruction with a given label. Finally, the function *First* takes a method descriptor and returns the first instruction of this method.

The function *Lookup* models the dynamic resolution of virtual method calls. It takes as arguments the signature of a method, the dynamic class of the object on which the method is invoked, the class in which the method is declared and the class hierarchy. It returns the method descriptor of the implementation of the method designated by the signature.[5]

$$\mathit{Lookup} : \mathit{Sig} \times \mathit{Id}_{ci} \times \mathit{Id}_{ci} \times E_{ci} \rightarrow \mathit{Desc}_m.$$

The function *Imp_Shareable?* determines if the class of an object implements a *Shareable* interface. This function is recursive on the field *Imp* of the class.

$$\mathit{Imp\_Shareable?} : \mathcal{P}(Id_i) \rightarrow \mathit{boolean}.$$

The function *Ext_Shareable?* determines if an interface extends the interface `Shareable`. This function is recursive on the field *Ext* of the interface.

$$\mathit{Ext\_Shareable?} : Id_i \rightarrow \mathit{boolean}.$$

Initialisation The function *Init_Var* constructs a function that maps the local variables of a method to their initial values.

$$\mathit{Init\_Var} : \mathit{Param} \times \mathit{Value}^* \times \mathit{Varl} \rightarrow \mathit{Locals}.$$

It takes the name and the descriptor of the formal parameters, the value of the actual parameters and the name of local variables of the frame to be constructed. The result is a function that maps the formal parameters to the value of the corresponding arguments and is default on local variables. The default value for a object and an array is *Null*. For a primitive, the default value is type of the primitive value.

Similarly, when creating a new object instance, we use a function *Init_Fields* to prepare the set of instance fields for a specified class and all of its superclasses:

$$\mathit{Init\_Fields} : \mathit{Id}_{ci} \rightarrow \mathit{Fldv}.$$

It takes the name of the class and returns a function in which a field name maps the default value for its type (the default value is the same as defined for *Init_Var*).

4 Operational semantics for instructions with the firewall

In this section we give an operational semantics for a small subset of Java Card instructions. The main feature that distinguishes this semantics from a Java bytecode semantics is the modelling of the Java Card firewall. The rules for instructions that can violate the firewall include an extra hypothesis that formalises when the instruction can be executed without raising a security exception.

[5] In this paper we do not describe any further the details of dynamic method lookup in Java(see [11, section 15.12] and [9]).

4.1 Firewall checks

The checks made by the firewall are formalised through a collection of predicates. Covering all bytecode instructions requires eight different predicates. We give the exact formula only for the two predicates used in this paper.

CheckVirtual? This check is performed during a call to a virtual method.

$$CheckVirtual?: Object \times Object \rightarrow boolean.$$

For $(o_1,o_2) \in Object \times Object$, the access is authorized if and only if the context represented by o_1 is the context of the JCRE ($o_1.Owner.Id_p = JCRE$) or if the contexts of o_1 and o_2 are the same ($o_1.Owner.Id_p = o_2.Owner.Id_p$) or if the object o_2 is global ($o_2.global$) or if the object o_2 is a JCRE entry point ($o_2.JCREep$).

$CheckVirtual?\ (o_1,o_2) =$
$(o_1.Owner.Id_p = JCRE) \vee (o_1.Owner.Id_p = o_2.Owner.Id_p) \vee (o_2.global) \vee (o_2.JCREep)$

CheckPutfield? This check is performed when storing a value in a field.

$$CheckPutfield?: Object \times Obj_i \times RValue \rightarrow boolean.$$

For $(o_1,o_2,v) \in Object \times Obj_i \times RValue$, the access is authorized if and only if the context represented by o_1 is the context of the JCRE ($o_1.Owner.Id_p = JCRE$) or if the contexts of o_1 and o_2 are the same ($o_1.Owner.Id_p = o_2.Owner.Id_p$) and if the value is not a global object ($\neg\ v.global$) and is not a temporary JCRE entry point ($\neg\ v.tJCREep$).

$CheckPutfield?\ (o_1,o_2,v) =$
$(o_1.Owner.Id_p = JCRE) \vee ((o_1.Owner.Id_p = o_2.Owner.Id_p) \wedge (\neg\ v.global) \wedge (\neg\ v.tJCREep))$

CheckALoad? This check is performed when read access is made to an array.

$$CheckALoad?: Object \times Obj_a \rightarrow boolean.$$

For $(o,a) \in Object \times Obj_a$, the access is authorized if and only if the context represented by o is the context of the JCRE ($o.Owner.Id_p = JCRE$) or if the contexts of o and a are the same ($o.Owner.Id_p = a.Owner.Id_p$) or if the array represented by a is a global array ($a.global$).

The instruction `arraylength` performes exactly the same check, so for this instruction we use the *CheckALoad?* predicate.

CheckAStore? This check is performed when storing an element in an array.

$$CheckAStore?: Object \times Obj_a \times RValue \rightarrow boolean.$$

For $(o,a,v) \in Object \times Obj_a \times RValue$, the access is authorized if and only if the context represented by o is the context of the JCRE ($o.Owner.Id_p = JCRE$) or if the contexts of o and a are the same ($o.Owner.Id_p = a.Owner.Id_p$) or if the array represented by a is a global array ($a.global$) and if the value not represents a global array ($\neg\ v.global$) or a temporary JCRE entry point ($\neg\ v.tJCREep$).

CheckClass? This check is performed during a cast or an `instance_of` check.

$$CheckClass?: Object \times Object \times Id_i \rightarrow boolean.$$

For $(o_1,o_2,Id) \in Object \times Object \times Id_i$, the access is authorized if and only if the context represented by o_1 is the context of the JCRE ($o_1.Owner.Id_p = JCRE$) or if the contexts of o_1 and o_2 are the same ($o_1.Owner.Id_p = o_2.Owner.Id_p$) or if the object o_2 is global ($o_2.global$) or if the object o_2 is a JCRE entry point ($o_2.JCREep$) or if the object's class implements a *Shareable* interface (*Imp_Shareable? (o_2.Imp)*) and if the object being cast into or an instance of an interface that extends `Shareable` (*Ext_Shareable? (Id)*).

CheckGetfield? This check is performed when reading access on a field is made.

$$CheckGetfield?: Object \times Obj_i \rightarrow boolean.$$

For $(o_1,o_2) \in Object \times Obj_i$, the access is authorized if and only if the context represented by o_1 is the context of the JCRE ($o_1.Owner.Id_p = JCRE$) or if the contexts of o_1 and o_2 are the same ($o_1.Owner.Id_p = o_2.Owner.Id_p$).

CheckInterface? This check is performed during a call to an interface method.

$$CheckInterface?: Object \times Obj_i \times Id_i \rightarrow boolean.$$

For $(o_1,o_2,Id) \in Object \times Obj_i \times Id_i$, the access is authorized if and only if the context represented by o_1 is the context of the JCRE ($o_1.Owner.Id_p = JCRE$) or if the contexts of o_1 and o_2 are the same ($o_1.Owner.Id_p = o_2.Owner.Id_p$) or if the object o_2 is a JCRE entry point ($o_2.JCREep$) or if the object's class implements a *Shareable* interface (*Imp_Shareable? (o_2.Imp)*) and if the interface being invoked extends `Shareable` (*Ext_Shareable? (Id)*).

CheckPutstatic? This check is performed when storing a value in a static field.

$$CheckPutstatic?: Object \times RValue \rightarrow boolean.$$

For $(o,v) \in Object \times RValue$, the access is authorized if and only if the context represented by o is the context of the JCRE ($o.Owner.Id_p = JCRE$) or if the value is not a global array ($\neg$ *v.global*) and is not a temporary JCRE entry point ($\neg$ *v.tJCREep*).

4.2 The semantics

The present semantics does not take visibility into account. Although the model has enough information to deal with visibility modifiers, we omit this for brevity.

Concerning the Java Card API, we only consider methods directly related to the firewall. These are `getAppletShareableInterfaceObject`, `getShareableInterfaceObject`, `getAID` and `getPreviousContextAID`. The static method `getAppletShareableInterfaceObject` that belongs to the `JCSystem` package is called by a client when it wants to obtain a shareable object from a server applet (cf. Section 2.1). The JCRE in turn invokes the method `getShareableInterfaceObject` that returns a shareable object based on the identity of the client. Thus, the modelling of a call to `getAppletShareableInterfaceObject` is a combination of a static and a virtual method call. The call to `getShareableInterfaceObject` is made directly by the rule of `getAppletShareableInterfaceObject`, the `invoke-static` is transformed into a `invokevirtual` if the call to `getShareableInterface-Object` is possible. The semantics contains a rule that treats the invocation of this method separately. Similarly, there is a separate semantic rule for the invocation of the two static methods of the `JCSystem` package `getAID` and `getPreviousContext-AID` for accessing the AID of the applet that owns the currently executing object and the AID of the context in action before the switch to the current context, respectively. In addition, we give semantics to five bytecodes: `getstatic`, `putfield`, `invokevirtual`, `goto` and `new`.

getstatic The `getstatic` instruction loads a value stored in a static class or interface field and stores it in a local variable.

$$\frac{\begin{array}{l} I = (pc, NT := \texttt{getstatic}\, C.f) \\ V' = V[NT \mapsto E_{ci}(C).FldS(C.f)] \\ I' \in Succ(I) \end{array}}{\langle\langle I, r, V\rangle :: A, mem, E_{ci}\rangle \Rightarrow \langle\langle I', r, V'\rangle :: A, mem, E_{ci}\rangle}$$

The class or the interface C must have a descriptor in E_{ci}. The field $C.f$ must exist in the set of static field of C. Then the value of the field $(E_{ci}(C).FldS(C.f))$ is loaded and stored into variable NT.

putfield The `putfield` instruction loads a value from a local variable and stores it into an instance field.

$$\frac{\begin{array}{l} I = (pc, \texttt{putfield}\, C.f\ T_1\ T_2) \\ o = mem(V(T_1)) \\ g = o.Fldv[C.f \mapsto V(T_2)] \\ o' = o[Fldv \mapsto g] \\ mem' = mem[V(T_1) \mapsto o'] \\ I' \in Succ(I) \\ [\ \mathit{CheckPutfield?}(mem(r), mem(V(T_1)), mem(V(T_2)))\ ] \end{array}}{\langle\langle I, r, V\rangle :: A, mem, E_{ci}\rangle \Rightarrow \langle\langle I', r, V\rangle :: A, mem', E_{ci}\rangle}$$

The value stored in T_1 must be a reference to a class object. This object must have a field $C.f$. Then the value stored in T_2 is stored in the field.

invokevirtual The `invokevirtual` instruction makes a call to an instance method.

$$\frac{\begin{array}{l} I = (pc, NT := \texttt{invokevirtual}\, C.m\ T_0\ T_1\ \cdots\ T_n\ S_1 :: \cdots :: S_n) \\ desc = Lookup((m, S_1 :: \cdots :: S_n), mem(V(T_0)).Id_{ci}, C, E_{ci}) \\ V' = Init\_Var(desc.Param, V(T_0) :: \cdots :: V(T_n), desc.Varl) \\ I' = First(desc) \\ r' = V(T_0) \\ [\ \mathit{CheckVirtual?}(mem(r), mem(V(T_0)))\ ] \end{array}}{\langle\langle I, r, V\rangle :: A, mem, E_{ci}\rangle \Rightarrow \langle\langle I', r', V'\rangle :: \langle I, r, V\rangle :: A, mem, E_{ci}\rangle}$$

The value stored in T_0 must be a reference to a class instance. We search for the implementation of the method called using the function *Lookup*. We construct the new list of local variables with the variables set to the actual parameters.

goto The `goto` instruction makes a jump to an instruction labelled *label*.

$$\frac{\begin{array}{l} I = (pc, \texttt{goto}\ label) \\ I' = Find(label) \end{array}}{\langle\langle I, r, V\rangle :: A, mem, E_{ci}\rangle \Rightarrow \langle\langle I', r, V\rangle :: A, mem, E_{ci}\rangle}$$

new The `new` instruction creates a new object in memory.

$$\frac{\begin{array}{l} I = (pc, NT := \texttt{ new } C) \\ O = (C, mem(r).Owner, false, false, Init\_Fields(C)) \\ R \in Ref \setminus dom(mem) \\ V' = V[NT \mapsto R] \\ mem' = mem[R \mapsto O] \\ I' \in Succ(I) \end{array}}{\langle\langle I, r, V\rangle :: A, mem, E_{ci}\rangle \Rightarrow \langle\langle I', r, V'\rangle :: A, mem', E_{ci}\rangle}$$

A new object of class C is created in the memory with the flags for entry point and temporary entry point set to *false*. A reference to this object is stored in the variable NT.

getAID

$$\frac{\begin{array}{l} I = (pc, NT := \texttt{ invokestatic } JCSystem.getAID) \\ V' = V[NT \mapsto mem(r).Owner.AID] \\ I' \in Succ(I) \end{array}}{\langle\langle I, r, V\rangle :: A, mem, E_{ci}\rangle \Rightarrow \langle\langle I', r, V'\rangle :: A, mem, E_{ci}\rangle}$$

The AID of the currently active applet is the AID of the owner of the current object. A reference to the current object can be retrieved from the frame as r.

getPreviousContextAID

$$\frac{\begin{array}{l} I_1 = (pc, NT := \texttt{ invokestatic } JCSystem.getPreviousContextAID) \\ \forall i \in \{2, \cdots, n-1\}, Mem(r_i).Owner.Id_p = mem(r_1).Owner.Id_p \\ mem(r_n).Owner.Id_p \neq mem(r_1).Owner.Id_p \\ V' = V[NT \mapsto mem(r_n).Owner.AID] \\ I' \in Succ(I_1) \end{array}}{\begin{array}{l} \langle\langle I_1, r_1, V_1\rangle :: \cdots :: \langle I_n, r_n, V_n\rangle :: A, mem, E_{ci}\rangle \Rightarrow \\ \quad \langle\langle I', r_1, V'\rangle :: \langle I_2, r_2, V_2\rangle :: \cdots :: \langle I_n, r_n, V_n\rangle :: A, mem, E_{ci}\rangle \end{array}}$$

The previous context is found by searching down the call stack for the most recent frame whose current object has an owner context that differs from the owner context of the current object on top of the call stack. If none such is found, *Null* is returned.

getAppletShareableInterfaceObject

$$\frac{\begin{array}{l} I = (pc, NT := \texttt{invokestatic}\ \mathit{JCSystem.getAppletShareableInterfaceObject} \\ \qquad\quad T_1\ T_2) \\ \mathit{server} = V(T_1) \\ \mathit{server} \in \mathit{Dom}(\mathit{Applet\_tbl}) \\ \mathit{class} = \mathit{Mem}(\mathit{Applet\_tbl}(\mathit{Server})).\mathit{Id}_{ci} \\ \mathit{desc} = \mathit{Lookup}((\mathit{getShareableInterfaceObject}, \texttt{AID} :: \texttt{byte}), \mathit{class}, \texttt{Applet}, E_{ci}) \\ \mathit{client} = \mathit{mem}(r).\mathit{Owner}.\mathit{AID} \\ I' = \mathit{First}(\mathit{desc}) \\ r' = \mathit{Applet\_tbl}(\mathit{server}) \\ V' = \mathit{Init\_Var}(\mathit{desc}.\mathit{Param}, r' :: \mathit{client} :: \mathit{mem}(V(T_2)), \mathit{desc}.\mathit{Varl}) \end{array}}{\langle\langle I, r, V\rangle :: A, \mathit{mem}, E_{ci}\rangle \Rightarrow \langle\langle I', r', V'\rangle :: \langle I, r, V\rangle :: A, \mathit{mem}, E_{ci}\rangle}$$

This method is called by the *client* to get the *server* applet's shareable object. Although a static method call, it functions as a virtual call of the method `getShareableInterfaceObject` of the *server*. If the firewall conditions are not respected, the result is *Null*.

5 Related works

There are several works on a formal semantics for Java [4,10]. The Bali project [2] provides an axiomatic semantics for a substantial subset of Java and Java Card but does not give an axiomation of the firewall. To formalize the language, they use a Hoare-style calculus [18,19]. All definitions and proofs are done formally with the theorem prover Isabelle/HOL. The resulting proof system can be proved sound, is easy to use, and complete. A similar goal is pursued in the Loop project [3], where they develop an interface specification in JML of the Java Card API [14] and provide proofs that the current Java Card API classes satisfy these interface specifications [15]. The more comprehensive semantics was proposed by Bertelsen [5] and was taken as the starting point for the present work. This semantics models the stack-based Java bytecode. Our choice of passing to a variable-oriented language means that we no longer have an operand stack in the frames. Moreover, we had to add certain attributes to the run-time structures to keep track of the owner of objects, whether they are entry points *etc.* Pusch has formalised the JVM in HOL [16]. Like us, she considers the class file to be well-formed so that the hypotheses of rules are just assignments. The operational semantics is presented directly as a formalisation in HOL, whereas we have chosen (equivalently) to use inference rules. Several works have focussed on formalising aspects of the Java Card firewall. Motré has formalised the firewall in the B language [13]. She transforms the informal specification into an abstract machine which can then be refined into an actual implementation. Each operation of this machine corresponds to a specific object access. This description of the firewall provides a formal description of the security policy as defined in JCRE specification [12] and provides a reference implementation of the firewall.

She formally demonstrates that the firewall verifications of bytecodes are sufficient to fulfil the security policy and to ensure the memory integrity (that only an authorised operation can access the memory). Bieber *et al.* [7] propose a verification technique based on model checking for detecting illegal information flow between Java Card applets. They associate a level with each applet and the legal flow between applets are given as a lattice of levels. Each applet is abstracted into a set of call graphs. All call graphs that do not include an interface methods are discarded (the sharing mechanism uses interface method). All values of variables are abstracted by computed levels, a variable having the level of the applets which use it. They give an invariant that is a sufficient condition for the security property, and verify it by model checking. It would be worth examining how the semantics defined in this paper can be used to provide a formal proof of correctness of their analysis.

6 Summary

We have described a small-step operational semantics of a representative subset of byte codes pertaining to the Java Card firewall. In doing so, we have deliberately abstracted away certain aspects of the language; for example, numeric calculations are not modelled. The continuation of this work is to demonstrate that the level of abstraction chosen is suitable for constructing and arguing the correctness of verification techniques for the firewall.

References

1. Java Card 2.1.1. http://java.sun.com/products/javacard/javacard21.html.
2. The BALI project, Last visited 2001. http://www4.informatik.tu-muenchen.de/ isabelle/bali/.
3. The LOOP project, Last visited 2001. http://www.cs.kun.nl/ bart/LOOP/.
4. Jim Alves-Foss, editor. *Formal syntax and semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999. 404 pages.
5. Peter Bertelsen. Semantics of Java Byte Code. Technical report, Dep. of Information Technology, Technical University of Denmark, March 1997. Home page http://www.dina.kvl.dk/ pmb/.
6. Peter Bertelsen. Dynamic semantics of Java bytecode. In *Workshop on Principles on Abstract Machines*, September 1998. Home page http://www.dina.kvl.dk/ pmb/.
7. Pierre Bieber, Jacques Cazin, Abdellah El Marouani, Pierre Girard, Jean-Louis Lanet, Virginie Wiels, and Guy Zanon. The PACAP prototype : a tool for detecting Java Card illegal flow. In Isabelle Attali and Thomas Jensen, editors, *Java Card Workshop (JCW)*, volume 2041 of *Lecture Notes in Computer Science*, September 2000.
8. Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison Wesley, 2000.
9. Ewen Denney and Thomas Jensen. Correctness of Java Card method lookup via logical relations. In *9th European Symp. on Programming (ESOP)*, pages 104–118. Springer-Verlag, March 2000.

10. Stephen N. Freund and John C. Mitchell. A formal framework for the Java bytecode language and verifier. *Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 34(10):147–166, November 1999.
11. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000. 896 pages, http://java.sun.com/docs/books/jls/index.html.
12. Sun Microsystems. Java Card 2.1.1 runtime environment (JCRE) specification, May 2000. Revision 1.0, 61 pages.
13. Stéphanie Motré. Modélisation et implémentation formelle de la politique de sécurité dynamique de la Java Card. In *Approches Formelles dans l'Assistance au Développement de Logiciel (AFADL)*, pages 158–172. LSR/IMAG, January 2000.
14. Erik Poll, Joachim van den Berg, and Bart Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *4th Smart Card Research and Advanced Application Conf. (CARDIS)*, pages 135–154. Kluwer Acad. Publ., 2000.
15. Erik Poll, Joachim van den Berg, and Bart Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Computer Networks Magazine*, 2001.
16. Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für informatik, Technische Universität München, 1998.
17. Soot: a Java optimization framework. http://www.sable.mcgill.ca/soot/.
18. David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency: Practice and Experience*, 2001.
19. David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 119–156. Springer-Verlag, 1999.