A Formal Specification of the Java Card Firewall*

Igor Siveroni[1], Thomas Jensen[2] and Marc Eluard[2]

[1] Department of Computing
Imperial College of Science, Technology and Medicine
London, England
[2] IRISA, Campus de Beaulieu
35042 Rennes Cedex, France

SecSafe Document	
Date:	**October 12, 2001**
Number:	**SECSAFE-ICSTM-003**
Version:	**1.0**
Classification	**Public**
Status:	**Accepted**

1 Introduction

Secure processing is fundamental for Java Card applications. The Java Card platform supports a subset of the Java programming language features including the security mechanisms associated to the supported language subset. Due to the nature of the Java Card system and its applications, Java language-level security mechanisms are insufficient to provide a satisfactory security level. Additional runtime security mechanisms are needed.

The aim of this paper is to provide a formal specification of the Java Card firewall, a mechanism designed to protect the security and integrity of the Java Card Runtime Environment (JCRE) and of each applet. In the Java Card firewall scheme, applets can only access objects that were created by applets defined in the same package or through special object sharing mechanisms such as shareable interfaces.

A formal definition of the semantics of the Java Card Virtual Machine and firewall will provide the basis for the construction of program simulators and the derivation of formal verification tools. We are particularly interested in the application of static analysis techniques to the validation of the security and safety aspects of Java Card applications.

We present a precise specification of the Java Card Firewall for a small subset of the Java Card virtual machine language. We define the operational semantics of the language subset and include the firewall security checks as part of the evaluation rules. We start by defining the language subset and program structures to be used in the paper. Then we model the runtime structures created during the execution of a Java Card program and define a small step operational semantics for the language subset. We finish by presenting the complete definition of the firewall security checks and by outlining future work.

*The authors are partially supported by the European Commission project Secsafe IST-1999-29075.

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 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 enforced by the firewall is based on the Java Card's package structure (which is the same as Java's one) and the notion of *contexts* (in Java Card, this notion is called *group context*).

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 is identified by the package name). In addition to the contexts defined by the applets executed 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 *Context* of contexts is defined by:

$$ctxt \in Context = \{\texttt{JCRE}\} \cup PackageAID$$

Every object is assigned a unique *owner context viz.*, the context of the applet which created the object. A method of an object is said to execute in the context of its owner[1]. It is with this context that the JCRE determines 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 specifically designated 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 entry points can be the elements of the table containing the AIDs of the applets installed on the card. Entry points can be marked as *temporary*. 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. Java Card offers a sharing mechanism, called *shareable objects*, that gives limited access to objects across contexts. An applet can allow another applet to access an object's methods from outside its context. The mechanism is restricted to methods and cannot be applied to fields. It uses a shareable interface, that is, an interface which extends `javacard.framework.Shareable`. In this interface, the applet lists the methods that it is prepared to share with other applets. The class of the object to share must implement this interface. The "server" applet must define a method, `getShareableInterfaceObject`, 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 to share different objects with different client applets.

2.1 An example of sharing mechanisms

Figure 1 contains a example illustrating the sharing mechanisms of the firewall. We have 3 applets: Alice, Bob and Charlie. Alice implements a shareable interface `MSI` (we assume an interface `MSI` that extends `Shareable` in which the method `foo` is listed) and is prepared to share an object `MSIO` (an instance of the class that implements the interface `MSI`) with Bob. When Alice receives a request for sharing (via a call to her method `getSIO`[2]) by the JCRE, she verifies that the caller is Bob. If it is Bob, she returns `MSIO` else she returns null.

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

[2]In reality, this method is called `getShareableInterfaceObject` and is invoked by the Java Card run time system that mediates all requests for shared objects

```
public class Alice extends Applet implements MSI {
   private Secret ObjectSecret;
   public Shareable getSIO (AID Client) {
      if (client.equals (BobAID))
         return (this);
      return null; }
   public Secret foo () {
      AID Client;
      Secret Response;
      Client = getPrevCxt();
      if (client.equals (BobAID))
         Response = ObjectSecret;
      return Response; } }
```

```
public class Bob extends Applet {
   public static MSI AliceObj;
   public void bar () {
      AliceObj = (MSI) getASIO
         (AliceAID); } }
```

```
public class Charlie extends Applet {
   private static MSI AliceObj;
   private static Secret AliceSecret;
   public void bar () {
      AliceObj = Bob.AliceObj;
      AliceSecret = AliceObj.foo (); } }
```

Figure 1: Example of shareable objects

Bob can ask for a shareable object from Alice using the JCRE method `getASIO`[3]. Assume now that Bob (inadvertantly) leaks a reference to `MSIO` to the third applet Charlie. With this reference, Charlie can invoke the same methods of the `MSIO` object as Bob.

Alice has some doubts about Bob so she decides to verify, in each access to one of her shared methods, the identity of the caller. Java Card offers a method for obtaining the AID of the context in operation before the last context switch, here called `getPrevCxt`[4]. Using this information Alice can discover when applets from contexts other than Bob's attempt to access the `MSIO` object.

2.2 Limitations of the firewall

A potential difficulty with the Java Card firewall is that shareable objects can be accessed by any applet and not only by the applet to which the reference was given. Since references can be passed from one applet to another, this opens up the possibility for methods to share objects can be invoked by applets other than those for which they were intended. As illustrated by the example, Java Card has a limited form of the stack inspection mechanism that underlies the Java 2 security architecture for preventing such unwanted access. The Java Card `getPrevCxt` can be called to get access to the last context switch that took place. When a method is called from another applet, this context switch indicates the identity of the caller. This information can then be used to decide what value the method should return to the caller. However, it is up to the programmer to implement this correctly. If the security mechanisms provided by the language are not used properly, unwanted information flow can arise as a result of objects flowing from one applet to another. More generaly, the security mechanisms provided by Java Card are sufficient to implement the wanted security policy. But there is nothing to prove that the implementation respects the wanted policy.

[3] In reality, this is a call to the system method `JCSystem.getAppletShareableInterfaceObject`

[4] In reality, this method is called `JCSystem.getPreviousContextAID`

3 The Java Card Virtual Machine Language

3.1 The Instruction Set

We use a modified subset of the Java Card virtual machine language (JCVMLe) defined in [6]. We have kept most of the instructions that interact with the JCVM firewall and reduced the language syntax in order to simplify the presentation of the operational semantics and concentrate on the presentation of the JCVM applet firewall. Mainly we have removed operand types from instructions - an operand type indicates the type of the arguments - and reduced the number of the instructions. For example, several stack operations have been omitted and subroutines are not considered. The changes are orthogonal to the firewall specification.

The instruction set used for this presentation is:

$$
\begin{array}{rcll}
I \in \mathit{Instruction} & ::= & \texttt{nop} \mid \texttt{push}\ c \mid \texttt{pop} \mid \texttt{numop}\ op & \text{Core Language} \\
& \mid & \texttt{load}\ i \mid \texttt{store}\ i \mid \texttt{goto}\ addr & \\
& \mid & \texttt{if}\ cmp\ \texttt{goto}\ addr & \\
& \mid & \texttt{new}\ \tau_r \mid \texttt{checkcast}\ \tau_r \mid \texttt{instanceof}\ \tau_r & \text{Object Language} \\
& \mid & \texttt{getstatic}\ f \mid \texttt{putstatic}\ f & \\
& \mid & \texttt{getfield}\ f \mid \texttt{putfield}\ f & \\
& \mid & \texttt{invokedefinite}\ m \mid \texttt{invokevirtual}\ m & \text{Method Language} \\
& \mid & \texttt{invokeinterface}\ m \mid \texttt{return} & \\
& \mid & \texttt{arraylength} \mid \texttt{arrayload} \mid \texttt{arraystore} & \text{Array \& Exception Language} \\
& \mid & \texttt{throw} &
\end{array}
$$

where

$$
\begin{array}{rcl}
i \in \mathit{Index} & = & \mathit{integer} \\
c \in \mathit{Constant} & = & \mathit{integer} \cup \{\texttt{null}\} \\
op \in \mathit{NumericOperator} & = & \{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}, \texttt{rem}, \texttt{cmp}\} \\
cmp \in \mathit{ComparisonOperator} & = & \{\texttt{eq}, \texttt{ne}, \texttt{ge}, \texttt{gt}, \texttt{le}, \texttt{lt}, \texttt{and}, \texttt{or}, \texttt{xor}, \texttt{shl}, \texttt{shr}, \texttt{ushr}\}
\end{array}
$$

We also consider a simplified version of the JCVM types:

$$
\begin{array}{rcl}
\tau \in \mathit{Type} & ::= & \mathit{ReferenceType} \mid \mathit{PrimitiveType} \\
\mathit{PrimitiveType} & ::= & \texttt{short} \\
\tau_r \in \mathit{ReferenceType} & ::= & \mathit{ArrayType} \mid \mathit{ClassType} \mid \mathit{InterfaceType} \\
\mathit{ClassType} & ::= & \mathit{Class} \\
\mathit{InterfaceType} & ::= & \mathit{Interface} \\
\mathit{ComponentType} & ::= & \mathit{PrimitiveType} \mid \mathit{ClassType} \mid \mathit{InterfaceType} \\
\mathit{ArrayType} & ::= & (\texttt{array}\ \ \mathit{ComponentType})
\end{array}
$$

The return address type (used for subroutines) is not considered and numerical values are restricted to be of type `short`. In addition, we assign the special type `Null` to the `null` constant. A special type *MethodType* is defined:

$$
\begin{array}{rcl}
\tau_m \in \mathit{MethodType} & = & (\mathit{parameterTypes} : [\mathit{Type}]) \ \times\ (\mathit{resultType} : \mathit{ResultType}) \\
\tau_r \in \mathit{ResultType} & = & \mathit{Type} \cup \mathit{VoidType}
\end{array}
$$

and will write $(\tau_i)_1^n \to \tau_r \in \mathit{MethodType}$, where an empty array of parameters can be represented as $(\tau_i)_1^n$ and $n = 0$.

3.2 Program Structures

Java Card programs, unlike Java, are not represented as `class` files. Instead, Java Card CAP - converted applet - files are used. A Java Card CAP file is the binary representation of a package of classes that can be installed on a device and used to execute the package's classes on a Java Card virtual machine[10]. In our framework, the information contained in the loaded CAP files is represented by a *program structure*. A program P contains a set of packages and each package a set of classes and interfaces. A package is uniquely identified by an AID (application identifier). We define the following domains:

$$P \in \mathit{Program} \quad p \in \mathit{Package} \quad \mathit{aid} \in \mathit{PackageAID} \quad \mathit{cl} \in \mathit{Class} \quad \mathit{iface} \in \mathit{Interface}$$

In JCVML*e*, a class structure *cl* contains information about the *package* where it was declared, its direct *superClass* - $\perp$ for `Object` class - and *subClasses*, the interfaces it directly implements, the set of direct *fields* and *methods*, and the access flags. Similarly, an interface structure *iface* contains information about its direct *subInterfaces* and *superInterfaces*, and the set of implementing classes *implementingClasses*.

Information about inherited methods and fields have to be looked up in the superclasses and superinterfaces. A set of auxiliary functions is defined to get information from the class hierarchy. The functions $\mathit{superClass}^* : \mathit{Class} \rightarrow \mathcal{P}(\mathit{Class})$ and $\mathit{superInterfaces}^* : \mathit{Interface} \rightarrow \mathcal{P}(\mathit{Interface})$ return *all* the superclasses and superinterfaces of a class or interface, respectively. The function $\mathit{implements} : \mathit{Class} \rightarrow \mathcal{P}(\mathit{Interface})$ returns *all* the interfaces implemented by a class.

A complete description of the method and field structures can be found in [6]. A method structure

$$m \in \mathit{Method} = (\mathit{id} : \mathit{MethodID}) \times (\mathit{type} : \mathit{MethodType}) \times (\mathit{classI} : \mathit{Class} \cup \mathit{Interface})$$

contains information about a method's ID (used for virtual method invocation), its type and the class or interface where it was declared. A non-abstract method also carries information about its body. The sequence of instructions that make up a method's body are accessed via addresses (*Address*). The following interface:

$$\begin{array}{lcl} \mathit{addr} \in \mathit{Address} & & \\ \mathit{firstAddress} & : & \mathit{Method} \rightarrow \mathit{Address} \\ \mathit{nextAddress} & : & \mathit{Method} \times \mathit{Address} \rightarrow \mathit{Address}_{\perp} \\ \mathit{instructionAt} & : & \mathit{Method} \times \mathit{Address} \rightarrow \mathit{Instruction} \end{array}$$

is used to access the first instruction of a method, the address following a given address, and the instruction at a given address. The instruction set is given in section 3.1.

A field is uniquely identified by a field id. A field structure

$$f \in \mathit{Field} = (\mathit{id} : \mathit{FieldID}) \times (\mathit{type} : \mathit{Type})$$

contains the field's id and type.

4 Runtime Structures

4.1 The Heap, Objects and Object Ownership

Objects are stored in the heap. The heap is defined as a mapping from locations to objects, which can be class instances or arrays.

$$\begin{array}{lcl} H \in \mathit{Heap} & = & \mathit{Location} \rightarrow \mathit{Object} \\ \mathit{Object} & = & \mathit{ClassInst} \cup \mathit{ArrayObject} \end{array}$$

All objects carry ownership information, which is kept by a special structure:

$$\mathit{Owner} = (\mathit{context} : \mathit{Context}) \times (\mathit{aid} : \mathit{AID})$$

where *aid* keeps track of the applet id.

A *ClassInst* is defined by:

$$\begin{array}{rcl} o \in \mathit{ClassInst} & = & (\mathit{class} : \mathit{ClassType}) \\ & \times & (\mathit{owner} : \mathit{Owner}) \ \times \ (\mathit{entryPoint} : \mathit{EntryPoint}) \\ & \times & (\mathit{fieldValue} : \mathit{FieldID} \ \rightarrow \ \mathit{Value}) \\ & \times & (\mathit{isInit} : \mathit{boolean}) \end{array}$$

The *class* element stores the object's class. *fieldValue* is a map from field ids to runtime values and the *isInit* flag is used to know if the object has been initialised or not. *owner* contains information about the owning context and applet, while *entryPoint* indicates if the object is a permanent or temporary `JCRE` entry point:

$$\mathit{EntryPoint} = \{\texttt{perm}, \texttt{temp}, \texttt{no}\}$$

We will explain the use of `JCRE` entry points in the section dedicated to the applet firewall (section 6). New objects are created and allocated by the function *newObject*:

$$\begin{array}{l} \mathit{newObject} : \mathit{Owner} \times \mathit{ClassType} \times \mathit{EntryPoint} \times \mathit{Heap} \ \rightarrow \ \mathit{Heap} \times \mathit{Location} \\ \mathit{newObject}(\mathit{own}, \tau, \mathit{ep}, H) = < H', \mathit{loc} > \text{ where} \\ \quad \forall \mathit{loc}' \in \mathit{dom}(H).\ H(\mathit{loc}') = H'(\mathit{loc}') \ \wedge \\ \quad H'(\mathit{loc}) = o \ \wedge \ o \in \mathit{ClassInst} \ \wedge \ o.\mathit{class} = \tau \\ \quad o.\mathit{owner} = \mathit{own} \ \wedge \ o.\mathit{entryPoint} = \mathit{ep} \\ \quad \mathit{dom}(o.\mathit{fieldValue}) = \{f.\mathit{id} \mid f \in \mathit{instanceFields}(\tau.\mathit{class})\} \ \wedge \ o.\mathit{isInit} = \texttt{false} \\ \quad \forall f \in \mathit{instanceFields}(\tau.\mathit{class}), \\ \qquad o.\mathit{fieldValue}(f.\mathit{id}) = \mathit{def}(f.\mathit{type}) \end{array}$$

where *instanceFields* is the set of non-static fields of a class, including the fields defined in its superclasses, and $\mathit{def}(\tau)$ returns the default value for a particular type τ (numeric types map to 0 and reference types to `null`). For this presentation we assume that fields and array values are initialised to the default values indicated by their types.

For security reasons, sometimes it is not desirable to preserve data stored in objects across sessions. An object is transient if the data stored in its fields is cleared whenever the card is reset or the applet is deselected. Currently only arrays with primitive components or references to `Object` can be designated as transient objects. There are two types of transient objects, namely, `CLEAR_ON_RESET` and `CLEAR_ON_DESELECT`. The interaction between transient objects and the applet firewall is explained in section 6. We define:

$$\mathit{Transient} = \{\texttt{CLEAR\_ON\_RESET}, \texttt{CLEAR\_ON\_DESELECT}, \texttt{NOT\_TRANSIENT}\}$$

An *ArrayObject* is defined by:

$$\begin{array}{rcl} a \in \mathit{ArrayObject} = & & (\mathit{elementType} : \mathit{Type}) \ \times \ (\mathit{length} : \mathit{integer}) \\ & \times & (\mathit{owner} : \mathit{Owner}) \ \times \ (\mathit{global} : \mathit{boolean}) \ \times \ (\mathit{transient} : \mathit{Transient}) \\ & \times & (\mathit{values} : \mathit{integer} \ \rightarrow \ \mathit{Value}) \end{array}$$

New arrays are created and allocated with the *newArray* function:

$$\begin{array}{l} \mathit{newArray} : \mathit{Owner} \times \mathit{Type} \times \mathit{integer} \times \mathit{boolean} \times \mathit{Transient} \times \mathit{Heap} \ \rightarrow \ \mathit{Heap} \times \mathit{Location} \\ \mathit{newArray}(\mathit{own}, \tau, n, g, t, H) = < H', \mathit{loc} > \text{ where} \\ \quad \forall \mathit{loc}' \in \mathit{dom}(H).\ H(\mathit{loc}') = H'(\mathit{loc}') \ \wedge \\ \quad H'(\mathit{loc}) = a \ \wedge \ a \in \mathit{ArrayObject} \ \wedge \ a.\mathit{elementType} = \tau \\ \quad a.\mathit{owner} = \mathit{own} \ \wedge \ a.\mathit{global} = g \ \wedge \ a.\mathit{transient} = t \\ \quad \mathit{dom}(a.\mathit{values}) = \{0, \ldots, n-1\} \\ \quad a.\mathit{length} = n \ \wedge \ \forall i.\ 0 \leq i < n, a.\mathit{values}(i) = \mathit{def}(\tau) \end{array}$$

$$\frac{\tau \in \mathit{Type} - \mathit{ArrayType}}{\tau \preceq \tau} \qquad \frac{cl' \in \mathit{superClass}^*(cl)}{cl \preceq cl'} \qquad \frac{\mathit{iface} \in \mathit{implements}(cl)}{cl \preceq \mathit{iface}}$$

$$\frac{\tau \in \mathit{ReferenceType} \cup \{\mathtt{Null}\}}{\mathtt{Null} \preceq \tau} \qquad \frac{}{\mathit{iface} \preceq \mathtt{Object}} \qquad \frac{\mathit{iface}' \in \mathit{superInterfaces}^*(\mathit{iface})}{\mathit{iface} \preceq \mathit{iface}'}$$

$$\frac{}{(\mathtt{array}\ \tau) \preceq \mathtt{Object}} \qquad \frac{\tau \preceq \tau'}{(\mathtt{array}\ \tau) \preceq (\mathtt{array}\ \tau')} \qquad \frac{\mathit{iface} \in \mathit{implements}(\mathtt{Array})}{(\mathtt{array}\ \tau) \preceq \mathit{iface}}$$

Figure 2: Subtype Relation Rules

4.2 Runtime Values

The subset of JCVMLe used in this paper supports two kind of values: numeric values and reference values (return addresses are not considered). Numeric values correspond to the Java type `short`. Reference values consist of references to class instances and arrays, and the special constant `null`.

$$\begin{aligned} v \in \mathit{Value} &= \mathit{NumericValue} \cup \mathit{ReferenceValue} \\ \mathit{NumericValue} &= \mathit{integer} \\ \mathit{ReferenceValue} &= \mathit{Location} \cup \{\mathtt{null}\} \end{aligned}$$

By choosing a single numeric value type we avoid all the complications regarding different number representations (they are considered in the full version of the semantics).

The type relation $H \vdash v\ :\ \tau$ assigns type τ to runtime value v under heap H. The relation is defined by:

$$\frac{v \in \mathit{integer}}{H \vdash v\ :\ \mathtt{short}} \qquad \frac{H(loc) \in \mathit{ClassInst} \quad cl = H(loc).\mathit{class}}{H \vdash loc\ :\ cl} \qquad \frac{H(loc) \in \mathit{ArrayObject} \quad \tau = H(loc).\mathit{elementType}}{H \vdash loc\ :\ (\mathtt{array}\ \tau)}$$

The subtype relation $\tau \preceq \tau'$ is defined following the "can be assigned to" criteria, as described by the rules for `checkcast` and `instanceof` in [10, ch 7: pp 139,182]. The subtype relation $\tau \preceq \tau'$ is defined by the rules in Figure 2. The judgement $H \vdash v\ :\preceq\ \tau$ - as in [5] - indicates that runtime value v can by assigned to a Java type τ. The following is true:

$$H \vdash v\ :\preceq\ \tau \quad \Leftrightarrow \quad (H \vdash v\ :\ \tau'\ \wedge\ \tau' \preceq \tau)$$

4.3 Frames

A frame contains information about the state of execution of a method. It is defined as follows:

$$\begin{aligned} &F \in \mathit{Frame} = \mathit{Owner} \times \mathit{Method} \times \mathit{Address} \times \mathit{LocalVar} \times \mathit{OperandStack} \\ &F ::= <\mathit{own}, m, pc, V, S> \end{aligned}$$

A frame contains the following components:

- The owning context and applet, *own.context* and *own.aid*, respectively.
- The current method structure $m \in \mathit{Method}$.
- The program counter register $pc \in \mathit{Address}$, which contains the address of the instruction currently being executed in method m.

- The array of local variables[5] $V \in [Value_\perp]$. A local variable is not referenced by name but by the position it occupies in the array of local variables. Uninitialised global variables are denoted by $\perp$.
- The operand stack S, defined as a sequence of values:

  $$S = v_1 :: \ldots :: v_n \in OperandStack = Value^*$$

  where values on top of the stack appear on the right-hand side of the sequence i.e. v_n is on top of the stack above. The empty stack is represented with the symbol ϵ.

We write $V = v_0 :: \ldots :: v_n$ when local variable array V is initialized with the sequence of values (usually from the operand stack) $v_0 :: \ldots :: v_n$.

4.4 Configurations

A runtime configuration describes the state of execution of the JCVM. We define three kinds of configurations:

$$C \in Config \ = \ RConfig \cup EConfig \cup HConfig$$

A running configuration - *RConfig* - keeps track of the chain of invoked methods by using a stack of call frames. An exception configuration - *EConfig*- represents the state of uncaught exceptions or machine errors. The halt configuration is the result of returning from the method that started the execution of the current applet[6].

A **running configuration** $C = <K,H,SF>$ contains the following components:

- The heap H, a mapping from locations to runtime objects (class and array instances).
- Static fields memory K, a mapping $FieldID \to Value$.
- The call stack SF, where $SF \in Frame^*$.

The frame of the method currently being executed - the current method - sits on top of the call stack. Given $SF = SF' :: <own, m, pc, V, S>$, the current instruction I can be extracted with $I = m.instructionAt(pc)$. The next instruction can be found at address $m.nextAddress(pc)$.

Exception and halt configurations represent terminal configurations. They are defined by:

$$\begin{array}{rcl} <H,K,loc> \in EConfig & = & Heap \times StaticMem \times Location \\ <H,K,\mathtt{halt}> \in HConfig & = & Heap \times StaticMem \times \{\mathtt{halt}\} \end{array}$$

where the location *loc* in the exception configuration is a reference to the uncaught exception object.

5 The Operational Semantics

5.1 The Evaluation Relation and the Firewall Conditions

The small step semantics of JCVMLe is defined as a relation between configurations: $\Rightarrow \subseteq Config \times Config$. The evaluation relation is defined as the least relation that satisfies the rules in section 5.2.

All conditions and security checks related to the JCVM applet firewall are presented inside boxes. We present a detailed explanation of each condition in section 6. In the case that a firewall security condition fails, a `SecurityException` is thrown. We are not including explicit rules for the cases when runtime exception are thrown by the JCVM though they are handled in a similar way as shown in the rule for instruction `throw`.

[5] Arrays of elements of domain t are denoted by $[t]$.
[6] We can also use the halt configuration to model uncaught exceptions

$$\frac{I = \texttt{nop}}{<K, H, SF::<own, m, pc, V, S>> \;\Rightarrow\; <K, H, SF::<own, m, pc+1, V, S>>}$$

$$\frac{I = \texttt{push}\;\; c}{<K, H, SF::<own, m, pc, V, S>> \;\Rightarrow\; <K, H, SF::<own, m, pc+1, V, S::c>>}$$

$$\frac{I = \texttt{pop}}{<K, H, SF::<own, m, pc, V, S:v>> \;\Rightarrow\; <K, H, SF::<own, m, pc+1, V, S>>}$$

$$\frac{I = \texttt{numop}\;\; op \;\wedge\; (op \in \{\texttt{div}, \texttt{rem}\} \;\Rightarrow\; v_2 \neq 0) \;\wedge\; r = applyBinary(op, v_1, v_2)}{<K, H, SF::<own, m, pc, V, S::v_1::v_2>> \;\Rightarrow\; <K, H, SF::<own, m, pc+1, V, S::r>>}$$

$$\frac{I = \texttt{load}\;\; i}{<K, H, SF::<own, m, pc, V, S>> \;\Rightarrow\; <K, H, SF::<own, m, pc+1, V, S::V[\mathrm{i}]>>}$$

$$\frac{I = \texttt{store}\;\; i}{<K, H, SF::<own, m, pc, V, S::v>> \;\Rightarrow\; <K, H, SF::<own, m, pc+1, V[i \;\mapsto\; v], S>>}$$

$$\frac{I = \texttt{goto}\;\; addr}{<K, H, SF::<own, m, pc, V, S>> \;\Rightarrow\; <K, H, SF::<own, m, addr, V, S>>}$$

$$\frac{\begin{array}{l} I = \texttt{if}\;\; cmp\;\; \texttt{goto}\;\; addr \\ pc' = \begin{cases} addr & \text{if } applyComparison(cmp, v_1, v_2) = \texttt{true} \\ m.nextAddress(pc) & \text{otherwise} \end{cases} \end{array}}{<K, H, SF::<own, m, pc, V, S::v_1::v_2>> \;\Rightarrow\; <K, H, SF::<own, m, pc', V, S>>}$$

Figure 3: The Core Language Semantics

5.2 The Evaluation Rules

We first show the process of reading and executing an instruction by presenting the evaluation rule for `nop`:

$$\frac{I = m.instructionAt(pc) \;\wedge\; I = \texttt{nop}}{<K, H, SF::<own, m, pc, V, S>> \;\Rightarrow\; <K, H, SF::<own, m, m.nextAddress(pc), V, S>>}$$

For convenience, from now on we will consider $I = m.instructionAt(pc)$ as the default and write 0 and $pc + 1$ instead of $firstAddress(m)$ and $m.nextAddress(pc)$, respectively, whenever m and pc are clear from the context. Then we have:

$$\frac{I = \texttt{nop}}{<K, H, SF::<own, m, pc, V, S>> \;\Rightarrow\; <K, H, SF::<own, m, pc+1, V, S>>}$$

The rest of evaluation rules for the instructions of the core language are shown in figure 3. The `push` instruction places the constant c on top of the stack and `pop` removes the top element from the stack. The `numop` instruction uses the top two elements of the stack - after removing them - as arguments to the numeric operation op and places the result back on the stack. The `load` instruction places the value of local variable i on top of the stack and `store` updates local variable i with the value found on top of the stack. The `goto` instruction jumps to the instruction found in address $addr$. The `if` instruction jumps to $addr$ only if

$$\frac{\begin{array}{c} I = \texttt{new}\ (\texttt{array}\ \tau)\ \wedge\ n \geq 0 \\ < H', loc > = newArray(own, \tau, n, \texttt{false}, \texttt{NOT\_TRANSIENT}, H) \end{array}}{<K, H, SF::<own,\ m,\ pc,\ V,\ S::n>> \ \Rightarrow\ <K, H', SF::<own,\ m,\ pc+1,\ V,\ S::loc>>}$$

$$\frac{\begin{array}{c} I = \texttt{new}\ \ cl\ \wedge\ cl \in Class \\ < H', loc > = newObject(own, cl, \texttt{no}, H) \end{array}}{<K, H, SF::<own,\ m,\ pc,\ V,\ S>> \ \Rightarrow\ <K, H', SF::<own,\ m,\ pc+1,\ V,\ S::loc>>}$$

$$\frac{I = \texttt{getstatic}\ \ f\ \wedge\ v = K(f.id)}{<K, H, SF::<own,\ m,\ pc,\ V,\ S>> \ \Rightarrow\ <K, H, SF::<own,\ m,\ pc+1,\ V,\ S::v>>}$$

$$\frac{I = \texttt{putstatic}\ \ f\ \wedge\ \boxed{checkPutStatic(own.context, H, v)}}{<K, H, SF::<own,\ m,\ pc,\ V,\ S::v>> \ \Rightarrow\ <K[f.id \mapsto\ v], H, SF::<own,\ m,\ pc+1,\ V,\ S>>}$$

$$\frac{\begin{array}{c} I = \texttt{getfield}\ \ f\ \wedge\ loc \neq \texttt{null}\ \wedge\ o = H(loc) \\ v = o.fieldValue(f.id)\ \wedge\ \boxed{checkGetField(own.context, o)} \end{array}}{<K, H, SF::<own,\ m,\ pc,\ V,\ S'::loc>> \ \Rightarrow\ <K, H, SF::<own,\ m,\ pc+1,\ V,\ S'::v>>}$$

$$\frac{\begin{array}{c} I = \texttt{putfield}\ \ f\ \wedge\ loc \neq \texttt{null}\ \wedge\ o = H(loc)\ \wedge\ fv = o.fieldValue[f.id \mapsto v'] \\ o' = o[fieldValue \mapsto fv]\ \wedge\ \boxed{checkPutField(own.context, o, H, v)} \end{array}}{<K, H, SF::<own,\ m,\ pc,\ V,\ S'::loc::v>> \ \Rightarrow\ <K, H[loc \mapsto o'], SF::<own,\ m,\ pc+1,\ V,\ S'>>}$$

$$\frac{\begin{array}{c} I = \texttt{checkcast}\ \ \tau_r\ \wedge\ H \vdash loc\ :\preceq\ \tau \\ \boxed{loc \neq \texttt{null}\ \Rightarrow\ checkCast(own.context, \tau_r, H(loc))} \end{array}}{<K, H, SF::<own,\ m,\ pc,\ V,\ S::loc>> \ \Rightarrow\ <K, H, SF::<own,\ m,\ pc+1,\ V,\ S::loc>>}$$

$$\frac{\begin{array}{c} I = \texttt{instanceof}\ \ \tau_r\ \wedge\ v = \begin{cases} 1 & loc \neq \texttt{null}\ \wedge\ H \vdash loc\ :\preceq\ \tau_r \\ 0 & \text{otherwise} \end{cases} \\ \boxed{loc \neq \texttt{null}\ \Rightarrow\ checkCast(own.context, \tau_r, H(loc))} \end{array}}{<K, H, SF::<own,\ m,\ pc,\ V,\ S::loc>> \ \Rightarrow\ <K, H, SF::<own,\ m,\ pc+1,\ V,\ S::v>>}$$

Figure 4: The Object Language Semantics

$$\frac{\begin{array}{l} I = \texttt{invokedefinite}\ m' \ \wedge\ m'.\mathit{isStatic} \ \wedge\ (\tau_i)_1^n \to \tau_r = m'.\mathit{type} \\ V' = v_1 :: \ldots :: v_n \ \wedge\ F' = <own, m', 0, V', \epsilon> \end{array}}{<K,H,SF::<own, m, pc, V, S::v_1::\ldots::v_n>> \ \Rightarrow\ <K,H,SF::<own, m, pc, V, S>::F'>}$$

$$\frac{\begin{array}{l} I = \texttt{invokedefinite}\ m' \ \wedge\ \neg m'.\mathit{isStatic} \ \wedge\ (\tau_i)_1^n \to \tau_r = m'.\mathit{type} \\ loc \neq \texttt{null} \ \wedge\ o = H(loc) \ \wedge\ \boxed{\mathit{checkInvokeDefinite}(own.context, o)} \\ V' = loc :: v_1 :: \ldots :: v_n \ \wedge\ F' = <o.owner, m', 0, V', \epsilon> \end{array}}{<K,H,SF::<own, m, pc, V, S::loc::v_1::\ldots::v_n>> \ \Rightarrow\ <K,H,SF::<own, m, pc, V, S>::F'>}$$

$$\frac{\begin{array}{l} I = \texttt{invokevirtual}\ m' \ \wedge\ (\tau_i)_1^n \to \tau_r = m'.\mathit{type} \\ loc \neq \texttt{null} \ \wedge\ o = H(loc) \\ m_v = \begin{cases} \mathit{methodLookup}(m'.id, \texttt{Object}) & o \in \mathit{ArrayObject} \\ \mathit{methodLookup}(m'.id, o.class) & \text{otherwise} \end{cases} \\ \boxed{\begin{array}{l} \mathit{appCtxt} = \mathit{getAppletContext}(SF::<own, m, pc, V, S::loc::v_1::\ldots::v_n>) \\ \mathit{checkInvokeVirtual}(own.context, \mathit{appCtxt}, o) \end{array}} \\ V' = loc :: v_1 :: \ldots :: v_n \ \wedge\ F' = <o.owner, m_v, 0, V', \epsilon> \end{array}}{<K,H,SF::<own, m, pc, V, S::loc::v_1::\ldots::v_n>> \ \Rightarrow\ <K,H,SF::<own, m, pc, V, S>::F'>}$$

$$\frac{\begin{array}{l} I = \texttt{invokeinterface}\ m' \ \wedge\ (\tau_i)_1^n \to \tau_r = m'.\mathit{type} \\ loc \neq \texttt{null} \ \wedge\ o = H(loc) \\ m_v = \begin{cases} \mathit{methodLookup}(m'.id, \texttt{Object}) & o \in \mathit{ArrayObject} \\ \mathit{methodLookup}(m'.id, o.class) & \text{otherwise} \end{cases} \\ \boxed{\begin{array}{l} \mathit{appCtxt} = \mathit{getAppletContext}(SF::<own, m, pc, V, S::loc::v_1::\ldots::v_n>) \\ \mathit{checkInvokeIface}(own.context, \mathit{appCtxt}, m'.classI, o) \end{array}} \\ V' = loc :: v_1 :: \ldots :: v_n \ \wedge\ F' = <o.owner, m_v, 0, V', \epsilon> \end{array}}{<K,H,SF::<own, m, pc, V, S::loc::v_1::\ldots::v_n>> \ \Rightarrow\ <K,H,SF::<own, m, pc, V, S>::F'>}$$

$$\frac{\begin{array}{l} I = \texttt{return} \\ F' = <own', m', pc', V', S'> \end{array}}{<K,H,SF::F'::<own, m, pc, V, S::v>> \ \Rightarrow\ <K,H,SF::<own', m', pc' + 1, V', S'::v>>}$$

$$\frac{I = \texttt{return}}{<K,H,<own, m, pc, V, S>> \ \Rightarrow\ <K,H,\texttt{halt}>}$$

Figure 5: The Method Language Semantics

$$\frac{I = \texttt{arraylength} \wedge loc \neq \texttt{null} \wedge v = H(loc).length \quad \boxed{\begin{array}{l} appCtxt = getAppletContext(SF::<own, m, pc, V, S::loc>) \\ checkArrayLoad(own.context, appCtxt, H(loc)) \end{array}}}{<K,H,SF::<own, m, pc, V, S::loc>> \Rightarrow <K,H,SF::<own, m, pc+1, V, S::v>>}$$

$$\frac{\begin{array}{l} I = \texttt{arrayload}\ t \wedge loc \neq \texttt{null} \wedge (0 \leq i < H(loc).length) \\ v = H(loc).values(i) \wedge \boxed{\begin{array}{l} appCtxt = getAppletContext(SF::<own, m, pc, V, S::loc::i>) \\ checkArrayLoad(own.context, appCtxt, H(loc)) \end{array}} \end{array}}{<K,H,SF::<own, m, pc, V, S::loc::i>> \Rightarrow <K,H,SF::<own, m, pc+1, V, S::v>>}$$

$$\frac{\begin{array}{l} I = \texttt{arraystore}\ t \wedge loc \neq \texttt{null} \wedge (0 \leq i < H(loc).length) \\ values' = (H(loc).values)[i \mapsto v] \wedge o' = H(loc)[values \mapsto values'] \\ H' = H[loc \mapsto o'] \wedge \boxed{\begin{array}{l} appCtxt = getAppletContext(SF::<own, m, pc, V, S::loc::i::v>) \\ checkArrayStore(own.context, appCtxt, H(loc), H.v) \end{array}} \end{array}}{<K,H,SF::<own, m, pc, V, S::loc::i::v>> \Rightarrow <K,H',SF::<own, m, pc+1, V, S>>}$$

$$\frac{\begin{array}{l} I = \texttt{throw} \wedge H \vdash loc : \tau \wedge \tau \preceq \texttt{Throwable} \\ loc \neq \texttt{null} \wedge \boxed{checkThrow(own.context, H(loc))} \\ SF' = catchException(SF :: <own, m, pc, V, S>, H, loc) \end{array}}{<K,H,SF::<own, m, pc, V, S::loc>> \Rightarrow <K,H,SF'>}$$

Figure 6: The Array and Exception language Semantics

the result of performing the comparison operation *cmp* between the top two elements of the stack is true, otherwise it continues to the next instruction.

The evaluation rules for the instructions of the object language are shown in figure 4. New arrays are initialised as non-global and not transient. Instance objects created by the `new` instruction are not `JCRE` entry point objects. The owner of the current frame is designated as the owner of the object created by a `new` instruction. In `getstatic` and `putstatic`, the field id $f.id$ is used to access the static field f in K. In the other instructions, the location of the object to be read, modified or inspected is taken from the stack. The difference between `checkcast` and `instanceof` is that the former throws a `ClassCastException` when the operation is not valid. Firewall security checks take as arguments the current context, the object being accessed and, in the case of `putstatic` and `putfield`, the value being stored.

The evaluation rules for the instructions of the method language are shown in figure 5. The `invokedefinite` instruction is used for static methods and for non-static methods that have been resolved statically. The *methodLookup* function is used by `invokevirtual` and `invokeinterface` to find the first method structure with method id $m'.id$ in the superclass hierarchy of the class of the current object. The `return` instruction has a special case to handle a return from the last frame. Since `invokevirtual` and `invokeinterface` can work on arrays, the *getAppletContext* function - which returns the context of the currently **selected** applet - is used to handle the firewall security check for transient arrays (see section 6).

The evaluation rules for the instructions of the array and exception language are shown in figure 6. The `throw` instruction uses the $catchException(SF :: <own, m, pc, V, S>, H, loc)$ function to get back a stack of frames where the top frame contains an appropriate exception handler for (H, loc).

6 The JCVM Firewall

Arrays and class instances are owned by the applet that created them. The context (or package) where the applet is defined determines the owning context. The JCVM firewall provides a security mechanism that prevents an object from being accessed by code running in a different context (different from the owning context).

Prior to performing an operation on an object, the JCVM performs an access check. The access checks depend on the type and owner of the referenced object, the instruction, and the currently active context. We have followed the specifications found in [9] and have defined the following access check predicates:

$$\begin{array}{rcl}
\mathit{checkPutStatic} & : & \mathit{Context} \times \mathit{Heap} \times \mathit{Value} \to \mathit{boolean} \\
\mathit{checkGetField} & : & \mathit{Context} \times \mathit{Object} \to \mathit{boolean} \\
\mathit{checkPutField} & : & \mathit{Context} \times \mathit{Object} \times \mathit{Heap} \times \mathit{Value} \to \mathit{boolean} \\
\mathit{checkThrow} & : & \mathit{Context} \times \mathit{Object} \to \mathit{boolean} \\
\mathit{checkInvokeDefinite} & : & \mathit{Context} \times \mathit{Object} \to \mathit{boolean} \\
\mathit{checkInvokeVirtual} & : & \mathit{Context} \times \mathit{Context} \times \mathit{Object} \to \mathit{boolean} \\
\mathit{checkInvokeIface} & : & \mathit{Context} \times \mathit{Context} \times \mathit{Interface} \times \mathit{Object} \to \mathit{boolean} \\
\mathit{checkCast} & : & \mathit{Context} \times \mathit{Context} \times \mathit{ReferenceType} \times \mathit{Object} \to \mathit{boolean} \\
\mathit{checkArrayLoad} & : & \mathit{Context} \times \mathit{Context} \times \mathit{ArrayObject} \to \mathit{boolean} \\
\mathit{checkArrayStore} & : & \mathit{Context} \times \mathit{Context} \times \mathit{ArrayObject} \times \mathit{Heap} \times \mathit{Value} \to \mathit{boolean}
\end{array}$$

Access to objects is restricted by the JCVM firewall. In general, a security exception is thrown if the current context - the context of the current frame - is different from the owning context of the object that is being referenced, unless it is the `JCRE` context, which has special privileges. Given current context $ctxt$ and object o, access is allowed if:

$$(ctxt = \texttt{JCRE}) \ \vee \ (ctxt = o.owner.context)$$

The `JCRE` provides several mechanisms to allow object access across contexts. `JCRE` entry point objects - temporary and permanent - and global arrays are objects owned by the `JCRE` context that can be accessed from any context. A context switch occurs when a method of an object owned by a different context is invoked i.e. two consecutive frames have different contexts. The *anyContext* predicate checks if an object is a global array or if it has been designated as a `JCRE` entry point:

$$\mathit{anyContext}(o) = (o \in \mathit{ClassInst} \ \wedge \ o.\mathit{entryPoint} \neq \texttt{no}) \ \vee \ (o \in \mathit{ArrayObject} \ \wedge \ o.\mathit{global})$$

Objects designated as temporary `JCRE` entry points and global arrays cannot be stored in class variables, instance variables or array components. Let $\tau_r \in \mathit{ReferenceType}$. The *isStorable* predicate is defined as:

$$\mathit{isStorable}(H, v) = [H \vdash v \ : \ \tau_r \ \wedge \ o = H(v)] \ \Rightarrow \ \begin{cases} (o \in \mathit{ClassInst} \ \wedge \ o.\mathit{entryPoint} \neq \texttt{temp}) \ \vee \\ (o \in \mathit{ArrayObject} \ \wedge \ \neg o.\mathit{global}) \end{cases}$$

Access to static and instance fields is checked by the following predicates:

$$\begin{array}{rcl}
\mathit{checkPutStatic}(ctxt, H, v) & \equiv & ctxt = \texttt{JCRE} \ \vee \ \mathit{isStorable}(H, v) \\
\mathit{checkGetField}(ctxt, o) & \equiv & ctxt = \texttt{JCRE} \ \vee \ ctxt = o.owner.context \\
\mathit{checkThrow}(ctxt, o) & \equiv & \mathit{checkGetField}(ctxt, o) \\
\mathit{checkPutField}(ctxt, o, H, v) & \equiv & ctxt = \texttt{JCRE} \ \vee \ (ctxt = o.owner.context \ \wedge \ \mathit{isStorable}(H, v))
\end{array}$$

Arrays with primitive type components or references to `Object` can be transient objects. Transient objects of `CLEAR_ON_DESELECT` type can only be accessed when the currently active context is the context of the currently selected apple. The *checkTransient* predicate checks for `CLEAR_ON_DESELECT` transient objects:

$$\mathit{checkTransient}(ctxt, appCtxt, o) \equiv \begin{cases} (o.transient = \texttt{CLEAR\_ON\_DESELECT}) \ \Rightarrow \ (ctxt = appCtxt) & o \in \mathit{ArrayObject} \\ \texttt{true} & o \in \mathit{ClassInst} \end{cases}$$

Access to array elements is checked by the following predicates:

$$checkArrayLoad(ctxt, appCtxt, a) \quad \equiv \quad ctxt = \texttt{JCRE} \lor \begin{pmatrix} (ctxt = a.owner.context \ \lor \ a.global) \land \\ checkTransient(ctxt, appCtxt, a) \end{pmatrix}$$

$$checkArrayStore(ctxt, appCtxt, a, H, v) \quad \equiv \quad ctxt = \texttt{JCRE} \lor \begin{pmatrix} (ctxt = a.owner.context \ \lor \ a.global) \land \\ checkTransient(ctxt, appCtxt, a) \land \\ isStorable(H, v) \end{pmatrix}$$

checkInvokeDefinite follows the same restrictions for contexts but takes into consideration the invocation of methods on objects designated as entry points - *anyContext* is used. Putting everything together we have:

$$checkInvokeDefinite(ctxt, o) \quad \equiv \quad (ctxt = \texttt{JCRE}) \ \lor \ (ctxt = o.owner.context) \ \lor \ anyContext(o)$$

It is possible to invoke methods of the `Object` class on an array. In particular, methods can be invoked on a transient array. The *checkTransient* check must be added to the security checks for `invokevirtual`:

$$checkInvokeVirtual(ctxt, appCtxt, o) \quad \equiv \quad ctxt = \texttt{JCRE} \lor \begin{pmatrix} [(ctxt = o.owner.context) \ \lor \ anyContext(o)] \\ \land \\ checkTransient(ctxt, appCtxt, o)] \end{pmatrix}$$

Another mechanism that allows object access among contexts is the use of shareable interfaces. Methods of shareable interfaces can be invoked from one context even if the object implementing them is owned by an applet in another context. The following must be true:

$$isShareable(o, \tau) \equiv \texttt{Shareable} \in (implements(o.class) \cap superInterfaces^*(iface)$$

The security check for the invoke interface instruction is defined by:

$$\begin{array}{l} checkInvokeIface(ctxt, appCtxt, iface, o) \equiv \\ \quad (ctxt = \texttt{JCRE}) \ \lor \ \begin{pmatrix} [(ctxt = o.owner.context) \ \lor \ anyContext(o) \ \lor \ isShareable(o, \tau)] \\ \land \\ checkTransient(ctxt, appCtxt, o) \end{pmatrix} \end{array}$$

Security checks for `checkcast` and `instanceof` are performed by:

$$\begin{array}{l} checkCast(ctxt, appCtxt, \tau, o) \equiv \\ \left\{ \begin{array}{l} (ctxt = \texttt{JCRE}) \lor \\ \left(\left\{ \begin{array}{l} (ctxt = o.owner.context) \ \lor \ anyContext(o) \lor \\ (\tau \in InterfaceType \ \land \ isShareable(o, \tau.interface)) \end{array} \right\} \land \ checkTransient(ctxt, appCtxt, o) \right) \end{array} \right. \end{array}$$

7 Related and Future Work

There are several works that deal with the formal semantics of Java[1] and the JVM[2, 5]. Current work of Java Card semantics include recent efforts to formalise different aspects of the Java Card API [7] and firewall[3, 4]. Caromel, Henrio and Serpette [3] propose a static analysis for verifying whether object accesses in a Java Card application violate the firewall rules for shared objects. The analysis produces a set of constraints whose solution allows to identify those instructions that can lead to a security exception being raised. Eluard et al. [4] provides a formalisation of the Java Card firewall checks. This formalisation is the same as the one used in this paper; however, the language treated in *op. cit.* is much smaller than the set of bytecodes considered here.

The work presented in this paper has been developed as part of the SECSAFE[8] project: Secure and safe Systems based on Static Analysis. The objective of the project is to assess the scalability of static analysis technology to the validation of security and safety aspects of realistic languages and applications. We intend to develop methods that apply to the domain of smart cards by focusing a substantial part of our efforts on the Java programming language and its dialect Java Card , treating source-level as well as bytecode-level applications.

In order to reason about security properties of Java Card programs, we first have to formally define the operational semantics of the language and the applet firewall. This paper focuses mainly on aspects related to the firewall. A complete report containing the whole language definition, including aspects of the JCRE is in progress.

Other immediate goals are the specification of a model for control flow analyses for Java Card programs and the specification of other analysis based on security properties, to be validated using the current operational semantics (correctness proofs).

References

[1] J. Alves-Foss and Lam. F. Dynamic denotational semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 201–240. Springer-Verlag, 1999.

[2] Peter Bertelsen. Semantics of Java byte code. Technical report, Department of Information Technology, Technical University of Denmark, http://www.dina.kvl.dk/˜pmb/, March 1997.

[3] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Context inference for static analysis of Java Card object sharing. In *I. Attali and T. Jensen (Eds): Conference on Research in Smart Cards, E-smart 2001*, volume 2140 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, September 19–21 2001.

[4] Marc Éluard, Thomas Jensen, and Ewen Denney. An operational semantics of the Java Card firewall. In *I. Attali and T. Jensen (Eds): Conference on Research in Smart Cards, E-smart 2001*, volume 2140 of *Lecture Notes in Computer Science*, pages 95–110. Springer-Verlag, September 19–21 2001.

[5] Stephen N. Freund and John C. Mitchell. A formal framework for the Java Bytecode Language and Verifier. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA '99*, volume 34(10) of *SIGPLAN Notices*, pages 147–166, Denver, Colorado, USA, November 1999. ACM Press.

[6] Renaud Marlet. Syntax of the JCVM language to be studied in the SecSafe project. Technical Report SECSAFE-TL-005, v1.7, Trusted Logic, May 2001. SecSafe public.

[7] Erik Poll, Joachim van den Berg, and Bart Jacobs. Formal specification of the JavaCard API in JML. In *J. Domingo-Ferrer, D. Chan and A. Watson (Eds): 4th Smart Card Research and Advanced Application Conf. (CARDIS)*, pages 135–154. Kluwer Acad. Publ., 2000.

[8] The SECSAFE project. http://www.doc.ic.ac.uk/˜siveroni/secsafe.

[9] Sun Microsystems, Inc., Palo Alto, CA 94303 USA. *The Java Card^{TM} 2.1.1 Runtime Environemnt (JCRE) Specification*, revision 1.0 edition, May 18 200.

[10] Sun Microsystems, Inc., Palo Alto, CA 94303 USA. *The Java Card^{TM} 2.1.1 Virtual Machine Specification*, revision 1.0 edition, May 18 2000.