Certifying an Embedded Remote Method Invocation Protocol

June Andronick
Gemalto, Security Lab
6 rue de la Verrerie
92190 Meudon, France
june.andronick@gemalto.com

Quang-Huy Nguyen
Gemalto, Security Lab
6 rue de la Verrerie
92190 Meudon, France
quang-huy.nguyen@gemalto.com

ABSTRACT

This paper describes an approach to formally prove that an implementation of the Java Card Remote Method Invocation protocol on smart cards fulfills its functional and security specification. For that, we refine the specification in two intermediate formal models: the functional specification and the high level design. These two models are both defined upon an existing complete formal model of the Java Card virtual machine, allowing to formalize all the security requirements. We focus on certifying the Java code portion since the native portion has been handled in a previous work. The correctness is showed to be preserved while composing the native and Java codes. Our refinement scheme has been designed to fulfill the requirements of a high-level Common Criteria security evaluation.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Software/Program Verification—*Formal methods, Correctness proofs*

Keywords

Formal verification, Security and functional certification, Embedded software, Common Criteria, JCRMI

1. INTRODUCTION

Java Card Remote Method Invocation (JCRMI) is a communication protocol introduced in the Java Card 2.2 platform (see [12]). This protocol allows the terminal to directly call remote methods embedded in a smart card. For example, in a m-commerce application, a handset may call a user-authentication function embedded in the SIM card to check the authenticity of the buyer. In a banking application, the terminal may call the debit/credit methods of an e-purse embedded on the payment card. JCRMI simplifies the interface between the terminal and the card by translating the remote method invocation into APDU[1] commands in a transparent way for the terminal. In the Java Card platform, the server side of JCRMI is implemented as part of the API package `javacard.framework.service`. This package may contain both Java and native code (usually written in C).

In this work, we prove that a given embedded implementation[2] of the JCRMI protocol verifies the functional and security specification defined by Sun (see respectively [11] and [13]). We follow a classical refinement approach: the informal specification is refined, possibly through several steps, into a model whose link with the code is as straightforward as possible. Since we target a formal certification (*i.e.,* EAL5-7 levels of the Common Criteria's ladder), all the models and refinement proofs are formal and developed using the Coq proof assistant (see [14]).

More precisely, the refinement of the informal specification into its implementation is done through two intermediate formal models (see Figure 1):

- the functional specification (FSP) model, where an API method is defined by a precondition on the input and a postcondition on the output of the method. In this stage, we handle the native and Java methods in the same manner.

- the high level description (HLD) model, where an API method is specified by a function that computes the output of the method from its input. In this level, the Java methods are specified using a Java-like language[3] embedded in Coq and following their implementation, whereas the native methods are directly specified as Coq functions.

Both models are based on an underlying formal model of the Java Card Virtual Machine (JCVM).

A formal refinement proof is realized to ensure that the HLD model respects the FSP model. A strong point of this approach is that the correspondence between the HLD model and the implementation code is straightforward because both of them are based on the same code. Actually, the correspondence between the two levels is equivalent to

[1]Application Protocol Data Units: the standardized format of the exchanged packets of data between cards and terminals (ISO7816-4).
[2]The implementation that has been verified is the Axalto's Java Card generic platform.
[3]This language has been developed within the French OPPIDUM funded project FORMAVIE2.

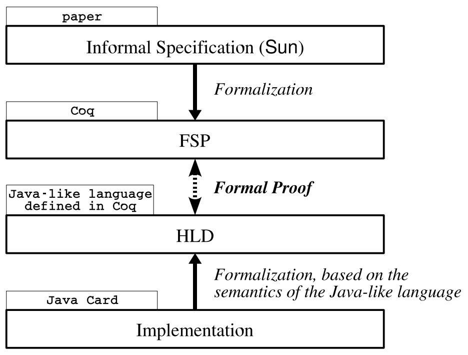
paper
Informal Specification (Sun)
Formalization
Coq
FSP
Formal Proof
Java-like language defined in Coq
HLD
Formalization, based on the semantics of the Java-like language
Java Card
Implementation

Figure 1: Refinement scheme for Java methods

the correctness of the semantics of the Java-like language. This strengthened link is a great benefit compared to a manual analysis linking the most refined model and the implementation.

In addition, our approach also contains the following advantages:

- this approach benefits from a complete semantics of Java Card using the state machine modeling the JCVM: the access control and the information flow control policies of the JCRMI protocol can be fully formalized as required in the security specification (see [13, 12]). For example, it is required that an array parameter of a remote method shall be stored in a global array in order to avoid the reuse of this parameter in other sessions. We cannot fully express this information flow control without formalizing the notion of *global arrays* in Java Card. Furthermore, for an invoked remote method to be executed in the card, its calling context shall fulfill the same firewall condition as that of the bytecode `invokevirtual`. This policy requires the full model of the Java Card firewall (described in section 3.1).

- this approach handles smoothly both the Java code and the native code. The correctness of native and Java methods can be soundly combined.

- the models issued from this approach can be naturally used to fulfill the requirements of a high-level Common Criteria evaluation (EAL5-7). Actually, the FSP and HLD models are designed following these requirements.

This paper is organized as follows. First, the JCRMI protocol and its security requirements are briefly described in Section 2. Then the formal models involved in the refinement chain, *i.e.,* the FSP model and the HLD model, are presented in Section 3. The refinement proof between these models is explained in Section 4. Finally, the related work and some concluding remarks are respectively discussed in Sections 5 and 6.

2. JCRMI PROTOCOL

We provide in this section a brief specification of the JCRMI protocol. A more detailed description can be found in Sun specifications (see [12, 11, 13]).

When an applet wants to make some of its methods remotely accessible, it first defines a *remote* class C that implements the standard `javacard.rmi.Remote` interface. The methods declared in C are remotely accessible and must be declared as throwing a `RemoteException` (in order to encapsulate any communication problem) in addition to possible user exception. An initial remote object is then created as an instance of C. Besides, the applet creates a new `javacard.framework.service.RMIService` object and binds it to the initial remote object. The `RMIService` object plays the role of the JCRMI server stub, *i.e.,* all received JCRMI APDU commands are forwarded to the `RMIService` object, which translates them into calls to methods of the initial object or later of other remote objects.

As described in Figure 2, a typical JCRMI working session contains two phases. In the first phase, the terminal selects the target applet using its AID (Applet IDentifier). This selection is translated into a SELECT FILE command sent to the selected applet. This command is then forwarded to the `RMIService` object defined in this applet. This object has normally been bound to an initial remote object. The SELECT FILE command is then handled by the method `processCommand()` of the `RMIService` object. This method returns the reference of the initial remote object ($InitRef$). Having this reference, the terminal may start the next phase, where the remote methods are invoked.

In the second phase, the terminal sends an INVOKE command which contains the reference of the remote object ($InitRef$), the identifier of the remote method (m) and the parameters of this method ($p1, \ldots, pn$). The INVOKE command is also handled by `processCommand()`. Concretely, `processCommand()` unpacks the parameters and then simulates the execution of the bytecode `invokevirtual` on m. The result of this execution is then popped out of the operand stack, packed and sent back to the terminal. If this result is a reference Ref to an *exported* [4] remote object in the card, then the terminal may invoke a method m' of this object by a new INVOKE command. In other words, any object whose reference has been returned to the terminal may be used for remote invocation, as soon as it has been exported.

The second phase is finished by a new SELECT FILE command for selecting a new applet (or by a card reset).

Security requirements. The following security features are required on the server side of the JCRMI protocol (see [12, 13]):

1. Residual information protection: if the parameter of a remote method is an array, then it shall be stored in the card by a JCRE [5]-owned global array. This requirement ensures that a parameter is not reused in another session.

2. Access control: the firewall condition needed for executing a remote method is that of the `invokevirtual` bytecode on this method.

3. Information flow control: a reference is sent back to the terminal as the result of a remote method only

[4] `Remote` objects are automatically exported at creation, but may be manually unexported or exported by invoking the `unexport` and `export` methods.

[5] Java Card Runtime Environment

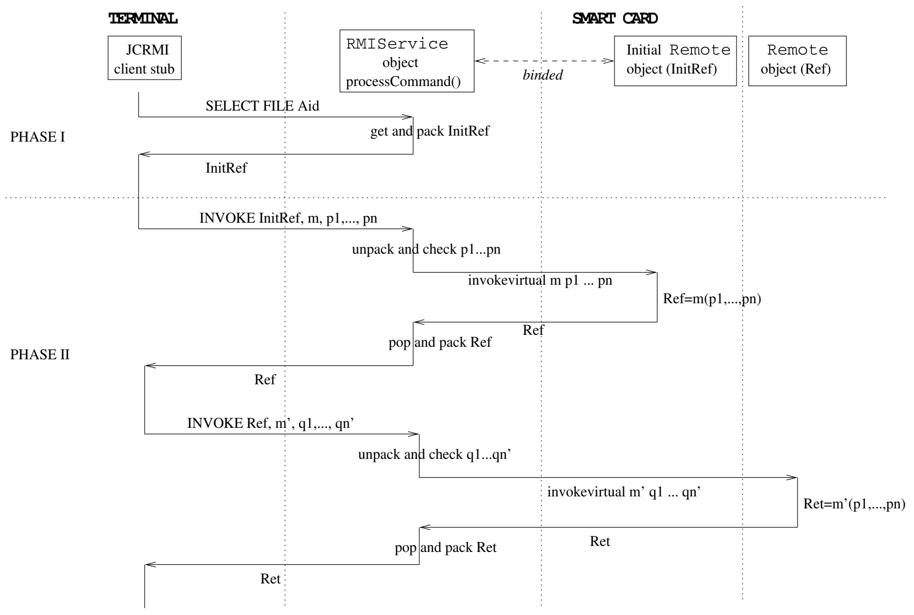

Figure 2: A typical JCRMI session

if it points to an exported object. This requirement avoid a non-accessible reference to be leaked out.

4. Lifetime of a remote reference: a returned remote reference is valid as long as the applet that owns it is active in the Java Card platform.

3. FORMAL MODELS

3.1 FIVM State Machine

The Java Card virtual machine has been fully formalized by a state machine called FIVM (for Formal Internal Virtual Machine). A FIVM state is a snapshot of the card memory and is composed of the following components: the installed applications (Java Card packages), the frame stack, the heap, the static fields image and the transaction state. The FIVM transitions represent the bytecode execution by the virtual machine in terms of modification of these components. Both the FSP and HLD models of the JCRMI protocol are built upon this state machine.

Security contexts and firewall control. Java Card 2.2 is a multi-applicative platform *i.e.,* several applets may co-exist on the platform. In order to protect the data of an applet from illegal access by other applets, the concept of *security context* (or simply *context*) is introduced. Each Java Card package is associated to a security context. That is to say all applets contained in this package share the same security context. The context of the currently selected applet (there is always such an applet during the lifetime of the JCVM) is the *currently active context.* The JCVM elements (*e.g.,* objects, arrays, static fields) created during the life-cycle of an applet are said to be owned by this applet. In addition, there are some special JCVM elements that are owned by the JCRE such as the (temporary and permanent) entry points and the global arrays. The security context of a JCVM element is that of its owner.

The Java Card firewall mechanism ensures that a given applet may only access to JCVM elements belonging to its security context. More precisely, the access to a JCVM element is allowed if and only if the currently active context is the owning security context of the element. There exists some exceptions to the firewall rules (see [12]) but they are not detailed here since there are not needed for the comprehension of the presented work.

Java Card-specific features. FIVM takes into account all Java Card-specific features, in particular:

- the firewall controls (see above);
- the different types of JCVM elements: persistent objects stored in the non-volatile memory (such as EEPROM or Flash), transient objects and arrays stored in the volatile memory (such as RAM). There are two categories of transient elements: a clear-on-deselect (COD) element is reset if the applet owning this element is deselected, while a clear-on-reset (COR) element is reset only if the card is reset;
- the transaction mechanism;
- the support for native codes in the API methods.

The RMIService internal state. A FIVM state contains all the information needed by the JCVM to interpret the bytecodes and therefore execute methods. However, some additional information is needed during the process of the JCRMI protocol. Indeed a `RMIService` object must contain the reference of the remote object which it is bound to. Besides, during the phase II of a JCRMI session, if a reference to an exported remote object is returned, then methods of this object may be invoked. Therefore such references must also be stored by the `RMIService` object. In the FSP model, these two pieces of information are stored in a record *service_state* associated to a given `RMIService` object:

$$\begin{array}{l} \mathit{service\_state} \triangleq \\ \quad \mathsf{ServiceState}\ \{\ \mathit{srvst\_initial\_ref} :\ \mathit{address}; \\ \qquad\qquad\qquad \mathit{srvst\_returned\_refs\_array} :\ \mathit{address}\ \}. \end{array}$$

whose value, at a given state of the virtual machine, may be read or modified by two predicates:

$$\begin{array}{l} (\mathit{get\_service\_state}\ s_{in}\ \mathit{ref}\ \mathit{srvst}) \\ (\mathit{set\_service\_state}\ s_{in}\ \mathit{ref}\ \mathit{srvst}\ s_{out}) \end{array}$$

where *ref* is the reference of the `RMIService` object. This structure is called the *internal state* of a `RMIService` object and is composed of:

- the address of the initial remote object;
- a pointer to the list of the remote objects references that have been returned to the terminal during the current JCRMI session:
  - the list is initialized to the empty list when the `RMIService` object is created;
  - at each new JCRMI session, the list is reset in the phase I to a list containing only the initial exported remote reference;
  - in the phase II, all the returned exported references, if any, are added to the list.

  This list is stored in the volatile memory (through a COD array) in order to fulfill the security requirement 4 described in Section 2.

3.2 Functional Specification

The FSP model of the API is a formal description of the functional specification. Each method m is described by a precondition Pre_m and a postcondition $Post_m$.

The precondition specifies the required conditions on the input so that the method can be called. The input is made of the initial memory state s_{in} of the virtual machine (*i.e.,* a FIVM state), the context c_{in} that is active when the method is called (for the verification of the firewall conditions) and the parameters $\overrightarrow{a}$ given to the method:

$$(Pre_m\ c_{in}\ \overrightarrow{a}\ s_{in}) \tag{1}$$

The postcondition describes the output of the execution of the method on the initial state with the given parameters. The output is made of the resulting state s_{out} of the virtual machine, the resulting context c_{out} (a context switch may occur during the execution) and a possible returned value r:

$$(Post_m\ \overrightarrow{a}\ s_{in}\ s_{out}\ c_{out}\ r) \tag{2}$$

Let us note that the returned value may be an exception raised during the execution of the method. More precisely, the returned value may have one of the three following form:

$$r\ \triangleq\ \mathit{raise}\ \mathit{exc} \mid \mathit{return}\ \mathsf{None} \mid \mathit{return}\ (\mathsf{Some}\ \mathit{val})$$

where *exc* is an exception raised by the method and *val* is a value.

Example 1 As mentioned in Sun's API specification [11] (see Figure 3), the `RMIService` constructor initializes a new `RMIService` object *i.e.,* initializes the internal state associated to this object.

In the FSP model, the precondition $Pre_{\mathtt{RMIService}}$ describes the constraints for invoking the constructor:

$$\begin{array}{l} (Pre_{\mathtt{RMIService}}\ c_{in}\ \overrightarrow{a}\ s_{in}) \triangleq \\ \quad \exists \mathit{this}\ .\ \exists \mathit{initialObject}\ . \\ \qquad \overrightarrow{a} = \mathit{initialObject} :: \mathit{this} :: \mathit{nil}. \end{array} \tag{3}$$

It expresses that this method has two parameters which must have been put on the operand stack:

1. the address of the newly created `RMIService` object to be initialized (*this*);
2. the address of the `Remote` object (*initialObject*) which will be bound to *this*.

The postcondition $Post_{\mathtt{RMIService}}$ describes the memory state after the execution of the constructor and the returned value:

$$\begin{array}{l} (Post_{\mathtt{RMIService}}\ \overrightarrow{a}\ s_{in}\ s_{out}\ c_{out}\ r) \triangleq \\ \quad \exists \mathit{this}\ .\ \exists \mathit{initialObject}\ . \\ \qquad \overrightarrow{a} = \mathit{initialObject} :: \mathit{this} :: \mathit{nil}\ \wedge \\ \qquad \mathsf{IF}\ \mathit{initialObject} = \mathit{addr\_null} \\ \qquad \mathsf{THEN}\ s_{in} = s_{out}\ \wedge \\ \qquad\qquad r = (\mathit{raise}\ \mathtt{NullPointerException}) \\ \qquad \mathsf{ELSE} \\ \qquad\quad \mathsf{IF}\ \exists (s_{aux}, \mathit{arr}). \\ \qquad\qquad (\mathit{create\_ext\_ref\_array}\ c_{in}\ s_{in}\ s_{aux}\ \mathit{arr}) \\ \qquad\quad \mathsf{THEN} \\ \qquad\qquad \mathit{let}\ \mathit{srvst} := \\ \qquad\qquad\quad (\mathsf{ServiceState}\ \mathit{initialObject}\ \mathit{arr})\ \mathit{in} \\ \qquad\qquad (\mathit{set\_service\_state}\ s_{aux}\ \mathit{this}\ \mathit{srvst}\ s_{out}) \wedge \\ \qquad\qquad r = (\mathit{return}\ \mathsf{None}) \\ \qquad\quad \mathsf{ELSE}\ s_{in} = s_{out}\ \wedge \\ \qquad\qquad r = (\mathit{raise}\ (\mathtt{SystemException} \\ \qquad\qquad\qquad \mathtt{NO\_TRANSIENT\_SPACE})). \end{array} \tag{4}$$

This postcondition first states that if *initialObject* is null, then the method raises a `NullPointerException`. Otherwise, the method allocates and initializes a new transient array (using an intermediate state s_{aux}) which is used to store the list of returned remote object references. If there is not enough memory space, the method should raise a `SystemException` with the reason code `NO_TRANSIENT_SPACE`. If the allocation succeeds, the created array should be referenced in the internal state of the `RMIService` object in the resulting state s_{out}. Moreover, this internal state should contain the reference *initialObject* of the bound remote object. Finally, the method returns no value (and raises no exception) and its result is (*return* None).

Of important note is the straightforward link between the FSP model and Sun's informal specification. In our goal to prove that a given implementation is correct with respect to its specification, the FSP model represents the formalization of the specification. The next step is to formally describe the algorithm implemented by each method. This is done in the HLD model, described in the next section.

3.3 High Level Design

The HLD model aims at representing the algorithms of the methods. On one hand, the Java methods are specified as functions in a Java-like language which is defined

```
public RMIService(java.rmi.Remote initialObject)
            throws NullPointerException
```

Creates a new `RMIService` and sets the specified remote object as the initial reference for the applet. The initial reference will be published to the client in response to the `SELECT APDU` command processed by this object.

The `RMIService` instance may create session data to manage exported remote objects for the current applet session in `CLEAR_ON_DESELECT` transient space.

Parameters:
`initialObject` - the remotely accessible initial object.

Throws:
`NullPointerException` - if the `initialObject` parameter is null.

Figure 3: Sun's specification of the `RMIService` constructor

upon the FIVM state machine. In other words, the methods algorithms are expressed directly using this language and following their Java Card implementation. The translation of the language constructors into FIVM concepts in Coq is transparent in the HLD model.

On the other hand, native methods are directly represented by Coq functions. Since the correctness of native methods has already been addressed in a previous work (see [9]), we only present here the HLD models of the Java methods.

The Java-like language. In order to describe the algorithms of the Java methods of the API, a subset of Java has been *embedded* in Coq. This means that each constructor of this language is mapped to an action defined on the FIVM state machine. The language syntax, defined as *sugars* in Coq (*i.e.,* notations), has been chosen to be close to the syntax of Java.

For example, the access to an object field (`o.fld` in Java) is represented in the HLD model by the notation o `@_a` fld, whose semantics describes the effect of this instruction on the states of the virtual machine:

$$\texttt{Notation "}o\ \texttt{'@a\_'}\ fld\texttt{"} \triangleq (inst\_getfield\ o\ fld)$$

where $(inst\_getfield\ o\ fld)$ is of type *instruction*, as any other instruction of the Java-like language. The *instruction* type is defined as a function between *statements*:

$$instruction \triangleq statement \rightarrow statement$$

where *statement* is a record containing the components needed for the execution of an instruction. A statement *stmt* of type *statement* is made of the currently active context c, a possible value *val* returned by the instruction, the global state s of the virtual machine (a FIVM state), and a *control* variable *ctr*:

$$stmt \triangleq (\mathsf{Stmt}\ c\ \mathsf{None}\ s\ ctr) \mid (\mathsf{Stmt}\ c\ (\mathsf{Some}\ val)\ s\ ctr)$$

The variable *ctr* of type *control* represents either the fact that the execution has terminated normally, or that an exception has been raised, or that a fatal error has occurred:

$$control \triangleq \mathsf{Normal} \mid \mathsf{Exception} \mid \mathsf{Fatal}$$

In other words, this language enables to describe a Java code by a Java-like code with a very close syntax, but which is actually a notation for a Coq definition of its effects on a FIVM state.

The HLD model of the Java methods. This Java-like language enables to describe the algorithms of each method without having to explicitly describe how the global state of the virtual machine is modified. Indeed, the HLD model of a method m is given by a function f_m that may be described by a sequence of instructions:

$$(f_m\ \overrightarrow{a}) \triangleq \{i_1;\ ...i_n;\} \quad : instruction \tag{5}$$

which is in fact a notation for:

$$(f_m\ \overrightarrow{a}\ (\mathsf{Stmt}\ c_{in}\ \mathsf{None}\ s_{in}\ \mathsf{Normal}))) \triangleq \\ (\mathsf{Stmt}\ c_{out}\ res\ s_{out}\ ctr) \tag{6}$$

Invoking native functions. A Java method can invoke native functions written in C. In the HLD level, a native method is modeled directly as a Coq function (see [9]), not using the Java-like syntax. In other words, an interface is needed to combine the model of the native code and the one of the Java methods. This interface is done by the Coq function *api_call_native_fct* taking in argument the native function and its parameters:

$$(api\_call\_native\_fct\ f_{native}\ \overrightarrow{a}) : instruction \tag{7}$$

where f_{native} is a native function, therefore defined as a Coq function:

$$\begin{aligned}(f_{native}\ \overrightarrow{a}\ c_{in}\ s_{in}) \triangleq\ & (\mathsf{NormalRes}\ (\mathsf{Some}\ val)\ s_{out}) \\ & |(\mathsf{NormalRes}\ \mathsf{None}\ s_{out}) \\ & |(\mathsf{ExcRes}\ exc\ s_{out}) \\ & |(\mathsf{FatalErrorRes})\end{aligned} \tag{8}$$

Once again, the semantics of the Java-like language in terms of FIVM concepts is transparent in the HLD model. In practice, since an *instruction* is a function computing a resulting *statement* from an initial *statement*, a call to a native method is defined as follow:

$$
\begin{array}{l}
(api\_call\_native\_fct \\
\quad f_{native}\ \overrightarrow{a}\ (\mathsf{Stmt}\ c_{in}\ \mathsf{None}\ s_{in}\ \mathsf{Normal}))) \triangleq \\
\quad match\ (f_{native}\ \overrightarrow{a}\ c_{in}\ s_{in})\ with \\
\qquad |\ (\mathsf{NormalRes}\ res\ s_{out}) \Rightarrow \\
\qquad\qquad (\mathsf{Stmt}\ c_{in}\ res\ s_{out}\ \mathsf{Normal}) \\
\qquad |\ (\mathsf{ExcRes}\ exc\ s_{out}) \quad \Rightarrow \\
\qquad\qquad (\mathsf{Stmt}\ c_{in}\ (\mathsf{Some}\ exc)\ s_{out}\ \mathsf{Exception}) \\
\qquad |\ (\mathsf{FatalErrorRes}) \quad \Rightarrow \\
\qquad\qquad (\mathsf{Stmt}\ c_{in}\ \mathsf{None}\ s_{out}\ \mathsf{Fatal})
\end{array}
$$

This definition shows that the combination of the two modeling methods (using the Java-like language and using directly Coq) really relies on this function. Indeed the fact that the semantics is defined in Coq enables to call directly the native method written in Coq, while the notations enable to call the function in the Java-like language.

Example 2 As described in the FSP model of the `RMIService` constructor (see Example 1), this method stores the reference of the initial remote object and allocates a new array to record the registered references. In the HLD model, these two elements are stored as instance fields of the `RMIService` class. The constructor is modeled by the following function:

$$
\begin{array}{l}
(f_{RMIService}\ this\ initialObject) \triangleq \\
\quad \{ \\
\quad\quad \_if\ (initialObject == \_null) \\
\quad\quad\quad \_throw\ exp\_NullPointerException; \\
\quad\quad this\ \texttt{@!\_}\ sessionReturnedReference\ \texttt{<-} \\
\quad\quad\quad (api\_call\_native\_fct \\
\quad\quad\quad\quad f_{makeTransientObjectArray} \\
\quad\quad\quad\quad (\texttt{MAX\_REFERENCE\_NUMBER\_PER\_INSTANCE} \\
\quad\quad\quad\quad\ \texttt{::CLEAR\_ON\_DESELECT::nil})\ ); \\
\quad\quad this\ \texttt{@!\_}\ initialReference\ \texttt{<-}\ initialObject; \\
\quad \}.
\end{array}
$$

This function has two parameters: the `RMIService` object to be initialized (*this*), and the initial remote object (*initialObject*) to be bound to *this*. The algorithm modeled is the following:

1. If the initial remote object is null then a `NullPointerException` must be thrown (see the specification in Figure 3).

2. The *sessionReturnedReference* field of the current `RMIService` object points to a new transient array of references, that is allocated by calling $f_{makeTransientObjectArray}$ which is the HLD model of the native API method `makeTransientObjectArray` (of the `javacard.framework.Util` class).

3. The *initialReference* field of the current `RMIService` object is assigned to the reference of the remote object given in argument.

Of important note is the similarity between the HLD modeling and the implementation in the Java Card language. For instance, the Java Card source code of the `RMIService` constructor is the following:

```
public RMIService(Remote initialObject)
  throws NullPointerException
{
  if (initialObject==null)
    throw new NullPointerException();
  this.sessionReturnedReference =
    JCSystem.makeTransientObjectArray(
      (short)MAX_REFERENCE_NUMBER_PER_INSTANCE,
      JCSystem.CLEAR_ON_DESELECT);
  this.initialReference = initialObject;
}
```

Therefore the link between the HLD model and the implementation is straightforward. Actually this link is equivalent to the correctness of the Java-like language semantics. Now, to ensure that our implementation verifies its specification, we prove that the HLD model is a refinement of the FSP model.

4. CORRECTNESS OF REFINEMENT

The proof that the HLD model refines the FSP model ensures that the algorithms described in the HLD model verify their specification described in the FSP model. This ensures that the actual implementation on card of the JCRMI protocol is correct with respect to Sun informal specification (see Figure 1). We present here the formal proof of the correctness of the Java methods (the correctness of the native methods has been described in [9]). Then, we show that the overall correctness can be preserved when combining the Java code portions and the native portions.

General proof scheme. The refinement between the HLD model and the FSP model is correct if the algorithms defined in the HLD model fulfill their specifications defined in the FSP model. In Hoare logic, a function f fulfills its precondition Pre_f and postcondition $Post_f$ if:

$$\forall\ x.\ \forall\ y.\ y{=}f(x) \wedge Pre_f(x) \Rightarrow Post_f(y)$$

where x, y respectively represent the input and the output of f.

Here, the preconditions and postconditions are defined in the FSP model as described in (1) and (2), whereas the algorithm of a method is given by the Java-like code as described in (6). Therefore the refinement proof is stated as follow:

$$
\begin{array}{l}
\forall\ \overrightarrow{a}.\ \forall\ c_{in}.\ \forall\ s_{in}.\ \forall\ s_{out}.\ \forall res.\ \forall\ c_{out}.\ \forall\ ctr. \\
\quad (f_m\ \overrightarrow{a}\ (\mathsf{Stmt}\ c_{in}\ \mathsf{None}\ s_{in}\ \mathsf{Normal})\ ){=} \\
\qquad\qquad (\mathsf{Stmt}\ c_{out}\ res\ s_{out}\ ctr) \wedge \\
\quad (Pre_m\ c_{in}\ \overrightarrow{a}\ s_{in})\ \wedge \\
\quad ctr \neq \mathsf{Fatal} \Rightarrow \\
\quad (Post_m\ \overrightarrow{a}\ s_{in}\ s_{out}\ c_{out}\ res)
\end{array}
\tag{9}
$$

This statement means that if the method m is executed on a state that verifies the precondition and if no fatal error occurs, then the resulting state verifies the postcondition.

In practice, this lemma will be proved by computing the postcondition after each instruction of the method. More precisely, the following predicate is defined:

$$
\begin{array}{l}
(correctness\ Pre\ i\ \overrightarrow{a}\ Post) \triangleq \\
\quad \forall\ c_{in}.\ \forall\ s_{in}.\ \forall\ s_{out}.\ \forall res.\ \forall\ c_{out}.\ \forall\ ctr. \\
\qquad (i\ (\mathsf{Stmt}\ c_{in}\ \mathsf{None}\ s_{in}\ \mathsf{Normal})\ ){=} \\
\qquad\qquad (\mathsf{Stmt}\ c_{out}\ res\ s_{out}\ ctr) \wedge \\
\qquad (Pre\ c_{in}\ \overrightarrow{a}\ s_{in})\ \wedge \\
\qquad ctr \neq \mathsf{Fatal} \Rightarrow \\
\qquad (Post\ \overrightarrow{a}\ s_{in}\ s_{out}\ c_{out}\ res)
\end{array}
$$

This predicate represents the notion that the instruction i is correct with respect to the specification given by the precondition Pre and the postcondition $Post$. In other words, the predicate is true if the execution of i on a state verifying Pre results in a state verifying $Post$, assuming that no fatal error has occurred.

Using this predicate, one can notice that the refinement proof amounts to proving:

$$\forall\ \overrightarrow{a}\ .\ (correctness\ Pre_m\ (f_m\ \overrightarrow{a})\ \overrightarrow{a}\ \ Post_m)$$

From then, the refinement is proved by using a set of lemmas dealing with each kind of instruction of the Java-like language. The lemma that is most used is the one decomposing a sequence of instructions:

$$\begin{array}{l} Lemma\ correctness\_sequence : \\ \quad \forall\ Pre\ .\ \forall\ Post\ .\ \forall\ i_1\ .\ \forall\ i_2\ .\ \forall\ P\ . \\ \qquad \begin{array}{lllll} (correctness & Pre & i_1 & P &)\ \wedge \\ (correctness & P & i_2 & Post &)\ \Rightarrow \\ (correctness & Pre & \{i_1; i_2\} & Post &) \end{array} \end{array}$$

We can notice that to use this lemma, we have to explicitly give an intermediate proposition P that is the postcondition of the first instruction. In other words, the proof is done by computing by hand a "strongest postcondition" for each instruction, in order to obtain the global postcondition for the sequence of all the instructions. Then each instruction correctness is proved using some specific predicate for this instruction.

Combining the correctness of Java and native codes. Let us consider a method m, that calls a native method nat. In the HLD model, the method m is represented by a function f_m whose body is written in the Java-like language (see (5)). The native method is represented by a Coq function f_{nat} (see (8)) and the method call is done using the function $api\_call\_native\_fct$ (see (7)). The HLD model of the method m has the following form:

$$\begin{array}{lll} (f_m\ \overrightarrow{a}) & \triangleq & \{\ i_1; \\ & & \quad ... \\ & & \quad (api\_call\_native\_fct\ f_{nat}\ \overrightarrow{a_{nat}}) \\ & & \quad ... \\ & & \quad i_n;\ \ \} \end{array}$$

During the refinement proof, *i.e.,* the proof that f_m verifies Pre_m and $Post_m$ defined in the FSP model, we will reach a point where we will have to prove:

$$(correctness\ P_1\ (api\_call\_native\_fct\ f_{nat}\ \overrightarrow{a_{nat}})\ \overrightarrow{a_{nat}}\ P_2)$$

for a P_1 and a P_2 computed during the proof development. This proof step amounts to proving that for all current context c_{in} and initial state s_{in}, if no fatal error occurs, we have:

$$(P_1\ c_{in}\ \overrightarrow{a_{nat}}\ s_{in})\ \Rightarrow\ (P_2\ \overrightarrow{a_{nat}}\ s_{in}\ s_{out}\ c_{in}\ res) \qquad (10)$$

where s_{out} and res are respectively the resulting state and the returned value of the execution of the native method.

At this point we have to take into consideration the fact that native methods have already been proved correct with respect to their specification (see [9]). In the FSP model, native methods are modeled in the same way as the Java methods, *i.e.,* by a precondition Pre_{nat} and a postcondition $Post_{nat}$. The native methods refinement proof thus ensures that for all current context c_{in} and initial state s_{in}, if no fatal error occurs, then we have:

$$\begin{array}{l} (Pre_{nat}\ c_{in}\ \overrightarrow{a_{nat}}\ s_{in})\ \Rightarrow \\ \qquad (Post_{nat}\ \overrightarrow{a_{nat}}\ s_{in}\ s_{out}\ c_{in}\ res) \end{array} \qquad (11)$$

where s_{out} and res are respectively the resulting state and value of the execution of the native method.

Therefore, the correctness step (10) will be proved using this refinement proof (11) together with two intermediate lemmas (see Figure 4):

- one lemma stating that the property P_1 assumed in the proof development is stronger than the precondition Pre_{nat} of the native function:

$$(P_1\ c_{in}\ \overrightarrow{a_{nat}}\ s_{in}) \Rightarrow (Pre_{nat}\ c_{in}\ \overrightarrow{a_{nat}}\ s_{in}) \qquad (12)$$

- one lemma stating that the property P_2 needed by the proof development is weaker than the postcondition $Post_{nat}$ of the native function:

$$\begin{array}{l} (Post_{nat}\ \overrightarrow{a_{nat}}\ s_{in}\ s_{out}\ c_{in}\ res) \Rightarrow \\ \qquad (P_2\ \overrightarrow{a_{nat}}\ s_{in}\ s_{out}\ c_{in}\ res) \end{array} \qquad (13)$$

Benefits and errors found. In this paper, the whole certification process is illustrated by a single method, the constructor `RMIService`, chosen for a better understanding. The verification has actually been done for each method of the JCRMI protocol, in particular the method `ProcessCommand`.

A major benefit of the approach is the direct link between the implementation and its formalization. This enables to prove that the source code is correct with respect to its functional and security specification, or to find and fix errors if any.

Indeed, our work highlighted some implementation errors, such as potential buffer overflow or disclosure of unexported object reference, and enabled to fix them. For confidentiality reasons, the errors may not be explained here. What can be said is that they could be found thanks to the use of the Java-like language that enables to copy exactly the embedded source code in the HLD model.

5. RELATED WORK

In previous attempts [7], part of the Sun reference implementation of Java Card API has been specified (in JML) and verified by automatic and interactive provers (Simplify [4], PVS [10]) using LOOP [3]. However, that work was done in a previous version of Java Card which does not include the JCRMI protocol.

Java Card applets can also be specified and verified using Krakatoa [5] (and a theorem prover like Coq [14]) as described in [6]. However, Krakatoa does not have a complete Java Card semantics. In particular, the different object lifetimes (transient objects, JCRE-owned entry points and global arrays) as well as the firewall control are not included. In other words, some JCRMI security requirements described in Section 2 may not be correctly specified. Furthermore, the native code is not considered in Krakatoa. Similar work has also been done to specify Java Card applet in JML and then, generate proof obligations using JACK [1].

In KeY framework [2], the JCVM is modeled in a domain-specific logic (Dynamic Logic). One can then specify the

$$(P_1\ c_{in}\ \overrightarrow{a_{nat}}\ s_{in}) \overset{(12)}{\Rightarrow} (Pre_{nat}\ c_{in}\ \overrightarrow{a_{nat}}\ s_{in}) \overset{(11)}{\Rightarrow} (P_2\ \overrightarrow{a_{nat}}\ s_{in}\ s_{out}\ c_{in}\ res) \overset{(13)}{\Rightarrow} (Post_{nat}\ \overrightarrow{a_{nat}}\ s_{in}\ s_{out}\ c_{in}\ res)$$

Figure 4: Refinement proof scheme for API native calls (proof of (10))

(functional) properties to be ensured on a Java Card code in the same logic following the JML style. However, the KeY model has not covered the firewall control yet and it is not well defined how the security requirements on the code are handled in this framework. In a recent work [8], the author has verified a part of the Java Card 2.2 API in KeY. Still, the JCRMI protocol is not considered in that work.

Comparing to these approaches, our certification requires more workload because we refine the JCRMI protocol from the informal specification to the implementation by different models and then, prove the correctness of this refinement. However, thanks to a complete model of the JCVM, this approach allows us to target all the functional and security requirements of the protocol. Furthermore, the direct link with the implementation, thanks to the Java-like language, enables to verify the source code itself. Besides, the Java and native codes are smoothly combined in this work while they need usually to be handled by two separated tools in the literature (except for KeY which provides an interface between Java and native codes). Finally, the refinement scheme described in this paper can be used for a high-level Common Criteria evaluation (EAL5-7) that requires the use of formal models and proofs.

6. CONCLUDING REMARKS

In this paper, we have described a refinement method to formally certify a smart card embedded software, the JCRMI protocol. Although the size of the source code is not huge (300 lines of Java and 800 lines of C), the software is actually more complex because these codes use several specific features of the Java Card platform. The aim of this work is twofold: (1) ensure that the specified requirements are fulfilled and (2) detect the implementation errors.

On the first point, the two intermediate models FSP and HLD have been built following respectively the informal specification and the implementation of the software. The correspondence between these two levels has been then formally proved in Coq. Thanks to the use of a complete model of the JCVM, we managed to model and prove not only the functional but also the security requirements on the software.

On the second point, this work reveals some errors in the implementation, that have been fixed since then, as mentioned at the end of Section 4.

The presented work is part of a high-level Common Criteria evaluation of the Java Card platform where securing RMI mechanism is one of the security objectives to be fulfilled. The refinement framework can be however used to deal with the other API classes of the platform. We are also investigating the application of this framework to the verification of Java Card applets. Expressiveness is an advantage of this framework because any Java Card-related security requirement can be expressed on the model and checked against any Java/C mixed implementation. However, the correctness proof is time-consuming because we need to proceed instruction by instruction. A more automatic proof tool is needed to handle the commercial-size Java Card applets. Another direction is to focus on the security kernel of the applet because for the rest we are only interested in its functional properties that can be handled by the tools like KeY, JACK or Krakatoa.

7. REFERENCES

[1] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: a tool for validation of security and behaviour of Java applications. In *FMCO'06*, 2006.

[2] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[3] Loop. `http://www.sos.cs.ru.nl/research/loop`.

[4] The Simplify decision procedure (part of ESC/Java2). `http://secure.ucd.ie/products/opensource/ESCJava2/`.

[5] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa Tool for Java Program Verification, 2002. `http://krakatoa.lri.fr/`.

[6] C. Marché and N. Rousset. Verification of Java Card Applets Behavior with respect to Transactions and Card Tears. In Dang Van Hung and Paritosh Pandya, editors, *SEFM'06*. IEEE Comp. Soc. Press, 2006.

[7] H. Meijer and E. Poll. Towards a Full Specification of the Java Card API. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *LNCS*, pages 165–178. Springer-Verlag, 2001.

[8] W. Mostowski. Fully verified Java Card API reference implementation. In Bernhard Beckert, editor, *Verify'07*, volume 259 of *CEUR WS*, 2007.

[9] Q-H. Nguyen and B. Chetali. Certifying Native Java Card API by Formal Refinement. In J. Domingo-Ferrer, J. Posegga, and D. Schreckling, editors, *CARDIS'06*, volume 3928 of *LNCS*, pages 313–328. Springer-Verlag, 2006.

[10] The PVS system. `http://pvs.csl.sri.com/`.

[11] Sun Microsystems. *Java Card 2.2 Application Programming Interface*, 2002. `http://www.javasoft.com/products/javacard`.

[12] Sun Microsystems. *Java Card 2.2 Runtime Environment Specification*, 2002. `http://www.javasoft.com/products/javacard`.

[13] Sun Microsystems. *Java Card System Protection Profile Collection - Version 1.1*, 2003. `http://java.sun.com/products/javacard/pp.html`.

[14] The Coq Development Team. *The Coq Proof Assistant*. `http://coq.inria.fr/`.