Certifying Native Java Card API by Formal Refinement

Quang-Huy Nguyen and Boutheina Chetali

Axalto, Smart Cards Research,
34-36 rue de la Princesse, 78431 Louveciennes Cedex, France
e-mail: {qnguyen,bchetali}@axalto.com

Abstract. This paper describes a refinement-based approach to show that a native Java Card API function fulfills its specification. We refine a native function from its informal specification (by Sun) through several intermediate models into a low-level model which is very close to its C implementations. We formally prove the correctness of the refinement steps between two adjacent levels. The low-level model is sufficiently detailed such that its correspondence to the C implementation can be informally checked. This work provides a framework to enforce the security of the native code by formal analysis and can be generalized to verify a complete implementation of the Java Card platform.

1 Introduction

Native API methods are usually written in C and are considered as part of the Java Card platform. On the contrary, non-native methods are written in Java Card and can be seen as applications running on the Java Card platform. Formal analysis of Java Card API methods has been done in several previous works (*i.e.,* [1–3]) using languages and tools dedicated to Java such as JML [4] (and its associated tools) and JACK [5]. On the contrary, in our knowledge, native methods have never been addressed. The main obstacle is related to their C implementation which is yet to be well handled by formal analysis.

Refinement is one of the cornerstones of formal approaches for software engineering: the process of developing a more detailed design or implementation from an abstract specification through a sequence of mathematically-based steps that maintain correctness *w.r.t.* the original specification. In the formal tools like B-Method or Esterel, the informal specification can be modelled, refined and then automatically translated into C code. However, in both of these systems, the generated code is not sufficiently efficient (in terms of performance and resource consuming) to fit into smart cards. Some attempts (*e.g.,* [6]) have been done to optimize the generated code but these optimizations are usually complex and may jeopardize the rigour provided by formal tools. Furthermore, in the industry, we often need to directly deal with an already developed product rather than starting from its informal specification.

In this work, we aim at certifying an existing implementation of the native methods that are already embedded on smart cards. To this end, we build a

low-level model of the JCVM (Java Card Virtual Machine) which is sufficiently close to its C implementation such that the correspondence between them can be informally checked. We also build two intermediate models in order to refine the informal specification of the native methods to the low-level model. Both of the models are built using the Coq proof assistant [7] which allows us to formally prove the correctness of each refinement step.

The rest of this paper is organized as follows. Section 2 describes several refining models of the native API methods. In Section 3, we provide their low-level model basing on a concrete JCVM implementation. Section 4 presents the correctness of refinement steps and its proof. Section 5 shows the relation between the low-level model and the concrete implementation. We discuss the related work in Section 6 and give some concluding remarks in Section 7.

2 Refining Informal Specification

The model of a native method must be built upon a model of the whole JCVM. In this paper, the JCVM is always modelled as a state machine. A *state* is a snapshot of all components of the JCVM: installed CAP files, heap, frame stack, static fields image, JCRE elements, etc. A *primitive operation* is a basic access service to one of these component (*e.g.,* popping and pushing a frame onto the frame stack, getting and setting an object in the heap). A primitive operation takes a state and its parameters and yields a new state and (possibly) a value. Any JCVM function (*e.g.,* a bytecode or a native function) can be seen as a sequence of primitive operations. The execution of a JCVM function transforms an (initial) state into a (final) state and (possibly) returns a value. If an exception is raised during this execution, then the returned value is the address of this exception which allows the JCVM to lookup for the exception handler.

Model	State Machine	Data structures	Primitive operations	Implementation dependency ?	Specification of native methods
FSP	FIVM	abstract	Coq relations	no	expected input and output
HLD	FIVM	abstract	Coq functions	no	abstract algorithm
LLD	CVM	refined	Coq functions	yes	refined algorithm

Fig. 1. Resume of intermediate models

The informal specification of a native method is refined by the following intermediate models (see a resume in Figure 1):

FSP is the Functional SPecification of the native function and is built upon the FIVM (Formal Internal Virtual Machine) state machine. In this model, a native function is specified by its expected input and output which are respectively defined by a pre-condition and a post-condition following Hoare

logic [8]. These input and output are described in the informal specification and hence, the FSP model is completely independent of any concrete implementation.

HLD is the High-Level Description of the native function which is also built upon the FIVM state machine. However, in this model, the native function is specified by its algorithm *i.e.,* a function taking its input and returning its output. This function is written by a sequence of primitive operations. Because the data structures and the primitive operations are kept abstract in FIVM, the HLD model is also independent of any concrete implementation.

LLD is the Low-Level Description of the native function built upon the CVM (Concrete Virtual Machine) state machine. Like the HLD model, the LLD model of a native function specifies its algorithm as a sequence of primitive operations. However, all CVM data structures and primitive operations are fully defined basing on a concrete JCVM implementation (by Axalto). Therefore, the LLD model is also strongly related to this concrete implementation.

2.1 Functional Specification Model

FIVM states. In FIVM, the card memory is seen as a set of memory cells. Each cell is associated to an address which will be used to access to this cell. The `addr_null` address represents to the null pointer. A FIVM state (`fivm_state`) is a snapshot of the card memory and is composed of the following components:

1. *Installed packages* stores the list of already installed packages (CAP files).
2. *Heap* stores the heap elements which are either an object or an array. An object is represented by a header structure as follows:
   Record *fivm_object_header* : *Set* := {
   fivm_object_status : *object_context*;
   fivm_object_transient_mode : *transience*;
   fivm_object_remote_mode : *bool*;
   fivm_object_class : *address*
   }.
   This structure contains the security context of the applet that owns the object, a flag indicating its memory mode (persistent, `CLEAR_ON_RESET` or `CLEAR_ON_DESELECT` transient), a boolean flag indicating its remote mode, and the address of its `class_info` structure (which defines its class) in the installed packages. Similarly, an array is represented by its header structure which contains the type of its elements, its length, its security context and its memory mode.
3. *Frame stack* stores the stack of frames and is the core data structure needed for method interpretation [9]. In FIVM, the execution of a method is done inside a frame which is defined as follows:
   Record *fivm_frame_info* : *Set* := {
   ifrm_pc : *address*;
   ifrm_context : *frame_context*;
   ifrm_max_locals : *nat*;

ifrm_max_stack : *nat* }.
where `ifrm_pc` is the program counter and points to the next bytecode to be executed; `ifrm_context` is the currently active context in which the method is being executed; `ifrm_max_locals` is the number of local variables of the method including its parameters; `ifrm_max_stack` is the number of FIVM words allocated to the operand stack where the intermediate results are pushed in and popped out during the execution of the method.
4. *Static fields image* stores the static fields of the installed packages.
5. *JCRE* stores the information used by the JCRE (Java Card Runtime Environment).

Primitive operations. The FSP primitive operations are defined as Coq predicates *i.e.,* relations between the input and the output of the operations in order to ease the modelling of the pre- and post-conditions. The FSP primitive operations are abstract *i.e.,* only their signature is given as Coq parameters. We briefly draw the primitive operations on different FIVM components in the following:

1. *Installed packages*: FIVM provides primitive operations to check if a given package has been correctly installed on the card, and to access to all components of the installed packages.
2. *Heap*: FIVM provides primitive operations to access to all heap elements *i.e.,* object and array headers, object instance fields and array elements. For example, the access to an object header pointed by an address is done via the predicate `heap_object_header`:
   Parameter *heap_object_header*: *fivm_state* → *address* → *fivm_object_header* → *Prop*.
3. *Frame stack elements*: FIVM provides primitive operations to pop the top frame, and to push a new frame onto the frame stack.
4. *Static fields*: FIVM provides read and write services for static fields.
5. *JCRE*: FIVM provides primitive operations to access to all JCRE information. For example, the currently active applet is accessed by `fivm_selected_applet`:
   Parameter *fivm_selected_applet*: *fivm_state* → *applet_ident* → *Prop*.

Firewall control. The firewall mechanism (Chapter 6 of [10]) ensures that the access to a JCVM element (*e.g.,* objects, arrays, static fields) is allowed if and only if the currently active context (*i.e.,* the context of the currently active applet) is the security context of the element. Exceptionally, the JCRE has a global privileged context and can access to all JCVM elements. All firewall conditions can be modelled using the primitive operations described above.

Native methods. The pre-condition defines the constraints on the input which is composed of the initial FIVM state and the list of parameters encoded as FIVM words (`iword`). The post-condition defines the constraints on the output which is composed of the final FIVM state and a (possibly) returned value encoded as a

FIVM word. This optional returned value is encoded in Coq by the type `(option iword)` which covers two cases: `(Some v)` means that a value `v` of type `iword` is returned and `(None iword)` means that no value is returned (void return).

Example 1. This example describes the model of the native method `export` of the class `javacard.framework.service.CardRemoteObject`. This method allows an on-card (remote) object to be (remotely) accessed by the card reader. The method `export` has only one parameter which is the address of the object to be exported. This constraint is modelled by the following pre-condition:

Definition *export_pre* (*ctxt*:*frame_context*)(*args* : *list iword*)
: *Prop* := ∃ *theObj*:*address*, *args* = (*address2iword theObj*)::*nil*.

where `address2iword` transforms the parameter `theObj` (which is an address) into a FIVM word. The output of `export` depends on its parameter, on the initial FIVM state (`fin`) and on the firewall condition[1]:

- **if** the parameter points to an allocated object in the heap[2] and the firewall condition is satisfied, **then** the remote mode of the object is set to `true` and `export` returns void. By changing the remote mode of the object, a new machine state (`fout`) is created from `fin`.
- **else**, `export` throws a security exception and the FIVM state is not modified.

```
Definition export_post (args: list iword)
 (fin fout: fivm_state) (result: option iword): Prop :=
 ∃ theObj: address, args = (address2iword theObj)::nil ∧
 ∃ hdr: fivm_object_header, heap_object_header fin theObj hdr ∧
 ∃ selapp: applet_ident, fivm_selected_applet fin selapp ∧
   IF (obj_jcre_or_same_owner (selected_applet_context selapp)
                                          (fivm_object_status hdr))
   THEN
       let newhdr := Fivm_Object_Header obj_status
                           (fivm_object_transient_mode hdr) true
                                          (fivm_object_class hdr)
       in (heap_object_header fout theObj newhdr) ∧ result=(None iword)
   ELSE fin=fout ∧ result=(Some (address2iword SecurityException)).
```

where the predicate (*obj_jcre_or_same_owner* ...) checks if the security context of the currently active applet (`selapp`) is either the (global) JCRE context or the security context of the object (to be exported) whose the header structure is `hdr`.

2.2 High-Level Model

The HLD model is also built upon the FIVM state machine. The JCVM functions are specified in the HLD model by their algorithm *i.e.,* by a function taking their

[1] For `export`, the currently active context must be either the JCRE context or the security context of the object to be exported (*cf.* Section 6.1.4 of [10]).

[2] Actually, this condition is ensured by the Java compiler and if it does not hold, then there is an inconsistency in the card memory.

input and returning their output. In this context, the HLD primitive operations must also be specified by Coq (abstract) functions instead of Coq predicates as in the FSP model.

Primitive operations. In the FSP model, the primitive operations are expressed as *partial* functions and defined as *relations*. A function f having parameters of type $A_1, \ldots, A_n$ and yielding a value of type B is generally represented as a relation R_f on $S \times A_1 \times \ldots \times A_n \times B$, where S represents some FIVM state. For functions that modify the content of the memory, their return also includes a new FIVM state. On the other hand, the HLD functions must be defined as *total* computable functions in Coq to ensure the termination of its computations.

In order to transform partial FSP functions into HLD total functions, a new constant (`FivmaFatalError`) is introduced to lift their co-domain. That is, a partial function is set to return `FivmaFatalError` when its output is not defined:

```
Inductive fivma_fatal_error : Set := FivmaFatalError : fivma_fatal_error.
Inductive exc (V E : Set) : Set :=
 | Value : V → exc V E
 | Error : E → exc V E.
Definition fivma_val (A : Set) := exc A fivma_fatal_error.
```

Notice that `FivmaFatalError` represents a model-level error and is not related to any Java Card runtime error or exception. All HLD functions will now return a value of type `(fivma_val A)` where `A` is its return type in the normal case. For example, the function `fivma_set_remote _object_header` updates the (boolean) remote mode flag of an object pointed by an address is specified as follows:

```
Parameter fivma_set_remote_object_header: fivm_state → address → bool →
(fivma_val fivm_state).
```

This function returns a new FIVM state (because the memory content has been modified) and is abstract (like all other HLD primitive operations), that is, its specification consists only of its signature.

Error handling. The error case makes the usage of functions more complex because there is now one more case to consider in each function call. For smoothly handling this case, a new construct is defined:

```
Definition try_with (C : Set) (e : exc V1 E1) (f : V1 → C) (g : E1 → C):
          C := match e with
          | Value x ⇒ f x
          | Error y ⇒ g y
          end.
```

Actually, `try_with` allows one to handle both cases of a total function. In the error case, the error (`y`) is handled by the function `g`. In the normal case, the returned value of the function (`x`) is used in the rest of the model (`f`). A new syntactic sugar `try1` is also defined such that `(try1 w=(F e) in H with err`

=> G) compiles to (G err) if (F e) returns the error err, and to (H val) if it returns the value val. In particular, if err and G are omitted, then any error will be handled by a default procedure which consists in transferring the error to the higher level (*e.g.,* the invoking function).

Native methods. The algorithm of a native method is defined as a sequence of the HLD primitive operations. The input of a native method is composed of the initial FIVM state and the list of parameters encoded as FIVM words. A native method may return a value or throws an exception by returning its address. In any case, the output of the method is composed of the final FIVM state and the (possibly) returned value encoded by the type (option iword).

Example 2. The algorithm of export (see Example 1) is described as follows:

1. **if** the list of arguments is empty, **then** a fatal error is raised, **else**,
2. convert the first argument into an object address using iword2address;
3. extract the object header pointed by this address using fivma_get_object _header;
4. check if the currently active context is either the global JCRE context (using fivma_test_jcre_context), or the security context of the object (using fivma_test_obj_same_owner) (the firewall condition);
5. **if** the firewall condition is satisfied, **then** return void and the final state (which has been updated by fivma_set_remote_object_header), **else** return the address of the security exception and the initial state.

```
Definition fivma_export (args: list iword) (fin : fivm_state)
  : (option iword) × fivm_state :=
    match args with
      | fst :: _ ⇒ let obj := iword2address fst in
        try1 hdr := fivma_get_object_header fin obj in
        try1 selapp := fivma_selected_applet fin in
        IF (fivma_test_jcre_context (selected_applet_context selapp))||
           (fivma_test_obj_same_owner (selected_applet_context selapp)
                                       (fivm_object_status hdr))
        THEN
             try1 fout := fivma_set_remote_object_header fin obj true in
                ((None iword), fout)
        ELSE ((Some (address2iword SecurityException)), fin)
      | _ ⇒ raise FivmaFatalError
    end.
```

3 Low-Level Model of a JCVM implementation

The LLD model specifies a real JCVM implementation on a new state machine called CVM (Concrete Virtual Machine). All the components of this state machine are defined as concrete data structures. Therefore, all CVM primitive operations can now be defined as concrete algorithms. The algorithm of a native

method are then refined to be close to its C implementation. In this section, for space reason, we only concentrate on the frame stack as well as on the invoking and the returning process of a (Java Card or native) method.

3.1 Frame stack

A CVM frame is composed of the following elements:

- an operand stack is a stack of 16-bits words (`cvm_word`).
- a table of local variables, each of them being a 16-bits word.
- a security information representing the currently active context.
- a program counter pointing to the next bytecode to be executed.

As in FIVM state machine, the frame stack is part of the CVM state and is defined as follows:

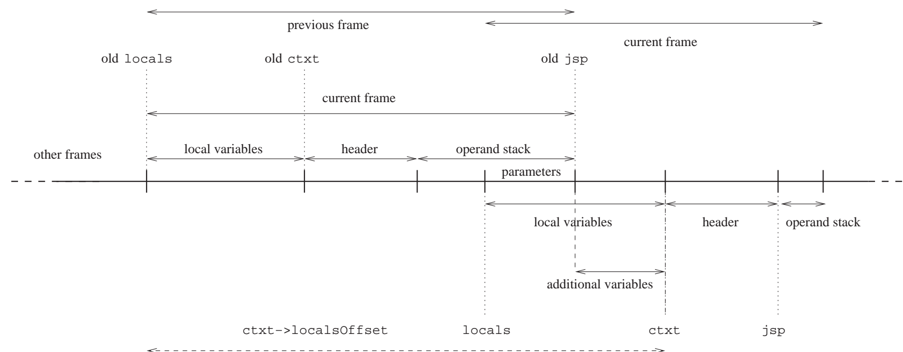

Fig. 2. The storage of the CVM frame stack

Record *cvm_state* : *Set* := {
cvm_frame_stack : *c_memory_segment*;
jsp : *c_address*;
locals : *c_address*;
ctxt : *c_address*; ... }.

- The contiguous memory segment `cvm_frame_stack` stores successively the frame stack itself. For each frame, firstly appears the local variable table, then its header, and finally its operand stack (see Figure 2).
- `jsp` is a pointer to the top of the operand stack of the current frame (*i.e.,* the top frame).
- `locals` points to the beginning of the local variable table of the current frame.
- `ctxt` points to the header of the current frame. This header is composed of:

- `localsOffset`: a byte representing the offset from the header of the current frame (current `ctxt`) to the first item of the local variable table of the previous frame. This information is needed to recover the previous frame upon return from the current method *i.e.,* to recover the old value of `locals`.
- `contextInfo`: a byte containing the currently active context.
- `nextpc`: a program counter pointing to the location where the virtual machine resumes upon return from the current method.
- `prev`: a pointer to the header of the previous frame (old `ctxt`).

3.2 Java Card methods

Invocation. When a Java Card method is invoked, a new frame is pushed onto the frame stack. The local variable table (`locals`) of the new frame is set to the first parameter of the invoked method which have been pushed onto the operand stack of the previous frame by the invoking method (see Figure 2). This is an optimization in the JCVM implementation to avoid copying these parameters and to reduce memory consumption. The header of the invoked method is stored after the new local variable table whose the length is determined by its `method_info` structure stored in its CAP file. Then the global variables are updates according to the new current frame:

- `jsp` points to the operand stack of the new frame *i.e.,* just after its header.
- `locals` points to the first item of the new local variable table.
- `ctxt` points to the header of the new frame.

Return. The returning process consists in popping the top frame by restoring the values of the global variables as follows:

1. `jsp` is assigned to the value of `locals`, that is all parameters must have been popped from the operand stack during executing the invoked method.
2. `locals` is assigned to the current value of `ctxt` minus the value of the `localsOffset` field of the header of the current frame. This indeed points to the local variable table of the previous frame.
3. `ctxt` is assigned to the value of the `prev` field of the header of the current frame.

3.3 Native methods

When a native method is invoked, its parameters are also pushed into the operand stack (of the current frame) as it is done when invoking a Java Card method. However, the native function is executed in the same frame of the invoking method and no new frame is created on the frame stack. After the execution of the native function, a returned type, which is of type short, is pushed on the top of the operand stack. If this type is 1 or 2, then there is a returned value which has been pushed onto the operand stack just under the returned type.

Otherwise, the method returns void. The CVM retrieves the returned value if there is any, pops out the parameters and moves the program counter to the next bytecode to be executed.

In the LLD model, a native method is defined as a total function using the CVM primitive operations. These primitive operations, which are abstract in the HLD model, are fully defined as Coq functions in the LLD model. The input of a native method is composed of the initial CVM state and the list of parameters encoded as CVM words (`cvm_word`). The output of the method is only composed of the final CVM state because the (possibly) returned value and its type are already pushed onto the operand stack of the current frame.

Example 3. The following LLD model of `export` is very similar to the HLD model presented in Example 2 except for the returning process: in the LLD model, the (possibly) returned value and its type are explicitly pushed onto the operand stack (by `cvm_frame_push`).

```
Definition cvm_export (args: list cvm_word) (cin: cvm_state): cvm_state :=
  match args with
    | fst :: _ ⇒ let obj := cvm_word2address fst in
        try1 hdr := cvm_get_object_header cin obj in
        try1 selapp := cvm_selected_applet cin in
        IF (fivma_test_jcre_context (selected_applet_context selapp))||
            (fivma_test_obj_same_owner (selected_applet_context selapp)
                                             (cvm_object_status hdr))
        THEN
          cvm_frame_push (cvm_set_remote_object_header cin obj true) szero
        ELSE
          cvm_frame_push
          (cvm_frame_push cin (address2cvm_word SecurityException)) stwo
    | _ ⇒ raise CvmFatalError
  end.
```

4 Correctness of Refinement

Informally, the refinement from a model to another model is correct if there is a correspondence between the executions of a native method in these two model.

Theorem 1 (Correctness of refinement). *Let $\mathcal{M}_1$ be a model of a native function and $\mathcal{M}_2$ be a refined model of $\mathcal{M}_1$. Let $\mathcal{R}_1$ be a relation between the states of $\mathcal{M}_1, \mathcal{M}_2$ and $\mathcal{R}_2$ be a relation between the data of $\mathcal{M}_1, \mathcal{M}_2$. Suppose that the two initial states of the native method are related by $\mathcal{R}_1$, and their corresponding parameters are related by $\mathcal{R}_2$. The refinement from $\mathcal{M}_1$ to $\mathcal{M}_2$ is said to be correct if:*

1. *the two final machine states are related by $\mathcal{R}_1$, and*
2. *the two returned values of the method, if there is any, are related by $\mathcal{R}_2$.*

4.1 FSP to HLD refinement

This refinement step is correct if the algorithm defined in the HLD model fulfills its specification defined in the FSP model. In Hoare logic, a function f fulfills its pre-condition Pre_f and post-condition $Post_f$ if:

$$\forall xy : y = f(x) \to Pre_f(x) \to Post_f(y)$$

where x, y respectively represent the input and the output of f. This statement is translated in Coq for the method `export` as follows:

Theorem *fivma_export_proof*:
∀ (*args*: *list iword*)(*fin fout*: *fivm_state*)(*result*: *option iword*),
(*fivma_export args fin*) = (*result*, *fout*) →
(*export_pre args fin*) → (*export_post args fin fout result*).

It is not difficult to see that this theorem is a special case of Theorem 1 where both $\mathcal{R}_1$ and $\mathcal{R}_2$ are the identity relation because both FSP and HLD models are built upon the state machine FIVM.

4.2 HLD to LLD refinement

For any native method, this refinement step is correct if the returning process from the method produce a similar effect in the FIVM and CVM state machines. In the HLD model, because the frame stack is abstract, the (possibly) returned value is pushed onto the operand stack and then, be popped by the invoking method in an opaque way. In the LLD model (see Section 3.3), the frame stack is detailed and all pop and push operations are explicitly performed on the operand stack. We need to chow that the two returning process produce the corresponding final states and returned values, providing that the initial states and the method parameters are respectively related by `cvm_fivm_link_state` (which abstractly relates CVM states to FIVM states) and `cvm_word2iword` (which abstractly converts CVM words into FIVM words).

Therefore, the correctness of the refinement must be stated for all two execution scenarios of a native method: (1) it returns a value or the address of an exception, and (2) it returns void. For example, the two theorems to be proved for `export` are described as follows:

1. `export` returns a value or the address of an exception:

   Theorem *cvm_export_value_proof*: ∀ *cst1 cst2 fst1 args cst' cst" typ val*,
   (*cvm_fivm_link_state cst1 fst1*) →
   (*cvm_export args cst1*) = *cst'* →
   (*cvm_frame_pop cst'*) = (*typ*,*cst"*) →
   (*andb* (*Zle_bool typ stwo*) (*Zge_bool typ sone*)) = *true* →
   (*cvm_frame_pop cst"*) = (*val*,*cst2*) →
   ∃ *fst2*: *fivm_state*, (*cvm_fivm_link_state cst2 fst2*) ∧

(*fivma_export* (*map cvm_word2iword args*) *fst1*) = ((*cvm_word2iword val*), *fst2*).
where the primitive operation `cvm_frame_pop` pops a short value from the operand stack of the current frame and returns a new machine state; `Zle_bool` represents the less-or-equal operator on short values; `andb` represents the conjunctive operator on boolean values.
This theorem states that in the LLD model, after executing `export` on the initial state `cst1` and on the list of parameters `args`, if we pop a short value (`typ`) from the top of the operand stack and this value is 1 or 2, then popping the next short value from the stack yields the returned value (`val`) of `export` and the final CVM state `cst2`. Now if we execute `export` in the HLD model on the corresponding FIVM state `fst1` (because `(cvm_fivm_link_state cst1 fst1)` holds), and on the corresponding parameters `(map cvm_word2iword args)`, then we obtain the final FIVM state `fst2` which corresponds to `cst2`. Moreover the LLD-returned value `val` also corresponds to the HLD-returned value (`cvm_word2iword val`).

2. `export` returns void:

   Theorem *cvm_export_void_proof*: ∀ *cst1 cst2 fst1 args cst' typ*,
   (*cvm_fivm_link_state cst1 fst1*) →
   (*cvm_export args cst1*) = *cst'* →
   (*cvm_frame_pop cst'*) = (*typ*,*cst2*) →
   (*andb* (*Zle_bool typ stwo*) (*Zge_bool typ sone*)) = *false* →
   ∃ *fst2*: *fivm_state*, (*cvm_fivm_link_state cst2 fst2*) ∧
   (*fivma_export* (*map cvm_word2iword args*) *fst1*) = ((*None iword*), *fst2*).

   In the LLD model, the short value at the top of stack is neither 1 nor 2 and there is no returned value. In this case, the HLD model of `export` must return void. Furthermore, the two final states (`fst2` and `cst2`) must also be related by `cvm_fivm_link_state`.

These two theorems are a special case of Theorem 1 where $\mathcal{R}_1$ is the relation `cvm_fivm_link_state` and $\mathcal{R}_2$ is the function `cvm_word2iword`.

4.3 General proof scheme

The general structure of a native function can be seen as a tree whose leaves are primitive operations. The internal nodes of this tree are Coq constructs used for defining the native function. The general proof scheme for the refinement on the native function between two adjacent models is described as follows:

1. Decompose the native function into more simple operations in both models until the primitive operations are reached.
2. Prove the correctness for each decomposition step: because the definitions of the native function in both models follow the same structure, this proof is feasible.
3. Apply the appropriate refinement hypotheses (see Section 4.4) to conclude the correctness for the primitive operations.

This proof scheme is closely related to the structure of the native function. For example, if it is a recursive function, then for proving the correctness of the decomposition steps over it, an proof by induction is needed. Furthermore, because a native function needs to cover all possible error cases, the proof must be done on all of its execution paths. In many cases, this leads to huge and unreadable proof. In order to ease the proof readability and maintenance, we have modularized and factorized the proofs by defining numerous common tactics and lemmas.

4.4 Refinement hypotheses

Because the HLD primitive operations are abstract, the correctness of their refinement from the FSP model must be supposed as hypotheses of the FSP-to-HLD refinement proof. Actually, those hypotheses express the internal consistency of the FIVM state machine.

On the other hand, the LLD primitive operations are fully defined and the correctness of their refinement from the HLD model must also be supposed as hypotheses of the HLD-to-LLD refinement proof. Actually, those hypotheses are part of the abstract relation between the FIVM and CVM state machines. This relation is also expressed by the abstract relations between FIVM states and CVM states (`cvm_fivma_link_state`), and between FIVM data and CVM data (*e.g.,* `cvm_word2iword`).

Example 4. Let us consider the primitive operation that yields the header structure of an object. In the FSP model, this operation is modelled by the predicate `head_object_header` and in the HLD and LLD models by the functions `fivma_get_object_header` and `cvm_get_object_header`. The refinements from the FSP model to HLD model and from the HLD model to LLD model are respectively supposed in Coq by the hypotheses `fivma_get_object_header_proof` and `fivma_get_object_header_refinement`:

Hypothesis *fivma_get_object_header_refinement*:
∀ (*fst*: *fivm_state*) (*addr*: *address*) (*hdr*: *fivm_object_header*),
(*fivma_get_object_header fst addr*)=*hdr* → (*heap_object_header fst addr hdr*).

Hypothesis *fivma_get_object_header_refinement*:
∀ (*cst*: *cvm_state*) (*fst*: *fivm_state*) (*addr*: *address*),
(*cvm_fivma_link_state cst fst*) →
(*cvm_get_object_header cst addr*) = (*fivma_get_object_header fst addr*).

5 C Implementation vs. Coq Low-Level Model

The conformance of the Axalto implementation *w.r.t.* the LLD model is informally checked by a hypertext document which relates the C code to the Coq model. This is the only informal step in the refinement chain from the informal

specification to the implementation of a native function. However, the fact that the LLD model has been refined basing on the C implementation makes the their conformance much more evident.

Example 5. The C implementation of `export` is quoted as follows.

```
void CARDREMOTEOBJECT_export()
{
    PEOBJECTHANDLE pHandle;
    u1   isExport;
    pHandle = soft_check_ref(pass_byteword_0());
    if(!isHandleRemote(pHandle)) {
      _VM_WriteU2((GEN_ADDRESS)(&pHandle->datalength),
      (u2)(pHandle->datalength | HANDLE_REMOTE)); }
}
```

In the heap, an object header is represented by a bit vector that contains the remote mode flag. Accessing to different fields of the object header is done via macros like `isHandleRemote`, which check the value of the corresponding bits. In the C code of a native function, the macro `pass_byteword_`n is used to pop its nth parameter from the operand stack of the current frame. For `export`, `pass_byteword_0` pops the address of the remote object. The `soft_check_ref` function checks the firewall condition on this object and raises a security exception if it is violated. Otherwise, the function checks if the object has been already exported before setting the remote flag of the object header (`pHandle`) using the `HANDLE_REMOTE` mask. This check is an optimization of the implementation because writing on E^{2}PROM is costly. In the LLD model, the flag is updated without this check (by `cvm_set_remote_object_header`) because the model is not executable and hence, we are not really concerned by the performance.

6 Related Work

Numerous researchers have worked on the formal analysis of the Java Card platform. However, most of them concentrate on ensuring some high-level security properties of the Java Card applets such as well-typedness [11, 12], confidentiality, noninterference, information-flow security [13–15].

While Java Card API can be formally analyzed as for Java Card applets [1–3, 16], verifying native functions requires us to work on the C code. Currently, the application of formal methods to the verification of the C code is still at its very early stage. Indeed, the semantics of C is not strictly defined and varies between different compilers[3]. C is however largely used in the embedded software industry thanks to its efficiency. There are currently two approaches for formally handling C code: in the bottom-up approach, the formal model is built using the

[3] Actually, part of the C memory management is not built in the language but is intentionally left to programmers for efficiency reason.

C code while in the top-down approach, the informal specification is formalized and refined to an C implementation.

The top-down approach is used in several works [6] using B-Method to automatically generate C code from a formal model. The bottom-up approach is used, for example, in [17] to generate Coq model of C code using tools like Caduceus and Why. The method presented in this paper can be seen as a mixed approach because the low-level model is designed by refining the higher-level models and by abstracting the C code to be certified.

While formal verification of C code is still not straightforward, many researchers have focused on the static analysis of information flow [18] (and/or abstract interpretation) as a feasible means to improve the security of C code. In this direction, the research has given rise to several industrial tools such as CAVEAT [19] or PolySpace.

7 Concluding Remarks

We described a refinement-based approach to verify the conformance of a Java Card native function *w.r.t.* their specification. The main idea is to use three intermediate models: the FSP model describes the expected input and output of the function (basing on the informal specification), the HLD model defines the algorithm of the function on an abstract JCVM, and the LLD model refines this algorithm on a concrete JCVM implementation. The refinement steps between two adjacent models are formally proved in Coq. This approach can be applied as well to the bytecode interpretation because a native function is actually a programmer-customized extension to the Java Card instruction set.

The two state machines (FIVM and CVM) used in this work were built during the French-funded FORMAVIE research project to fulfill the Common Criteria requirements [20] on the JCVM development. Using these models, we showed the conformance of the Java Card interpreter and linker developed in Axalto *w.r.t.* the JCVM specification (by Sun). The verification of the native API methods is an extension of this project and is an ongoing work. Actually, the set of native API methods varies between different implementations (this set is not precisely defined in the API specification) but for many methods, only a native implementation can be satisfactory in terms of performance and/or security (*e.g.,* the update operation on arrays or the PIN operations). We based on the Axalto implementation to built the LLD model. On the contrary, the higher-level FSP and HLD models are abstract and can be used for checking other implementations. Furthermore, both of these models can be used to reason on the high-level security properties of the native functions and of the JCVM platform.

References

1. J. van der Berg, B. Jacobs, and E. Poll. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Proc. of CARDIS'00*, pages 135–154. Kluwer Academic Publishers, 2000.

2. 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 *Lecture Notes in Computer Science*, pages 165–178. Springer-Verlag, September 2001.
3. L. Burdy, J-L. Lanet, and A. Requet. Java Applet Correctness: A Developer-Oriented Approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proc. of FME'03*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, September 2003.
4. The Java Modeling Language (JML) homepage. `http://www.cs.iastate.edu/~leavens/JML/`.
5. L. Burdy and A. Requet. Jack : Java Applet Correctness Kit, 2002. Available at `http://www.gemplus.com/smart/rd/publications/pdf/BR02jack.pdf`.
6. D. Bert, S. Boulm, M-L. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proc. of FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 94–113. Springer-Verlag, 2003.
7. The Coq Development Team. *The Coq Proof Assistant.* `http://coq.inria.fr/`.
8. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
9. B. Venners. *Inside the Java Virtual Machine, 1st edition.* McGraw-Hill Professional, 1999.
10. Sun Microsystems. *Java Card 2.2 Runtime Environment Specification*, 2002. `http://www.javasoft.com/products/javacard`.
11. G. Barthe, P. Courtieu, G. Dufay, and S. M. de Sousa. Tool-Assisted Specification and Verification of the JavaCard Platform. In H. Kirchner and C. Ringeissen, editors, *Proc. of AMAST'2002*, volume 2422 of *Lecture Notes in Computer Science*, pages 41–59. Springer-Verlag, 2002.
12. G. Barthe and G. Dufay. A Tool-Assisted Framework for Certified Bytecode Verification. In M. Wermelinger and T. Margaria-Steffen, editors, *Proceedings of FASE'04*, volume 2984 of *Lecture Notes in Computer Science*, pages 99–113. Springer-Verlag, 2004.
13. J. Andronick, B. Chetali, and O. Ly. Using Coq to Verify Java Card Applet Isolation Properties. In David A. Basin and Burkhart Wolff, editors, *Proc. of TPHOLs'03*, volume 2758 of *Lecture Notes in Computer Science*, pages 335–351. Springer-Verlag, September 2003.
14. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In Manuel Fähndrich, editor, *Proceedings of TLDI'05*, pages 103–112. ACM Press, 2005.
15. M. Eluard and T. Jensen. Secure object flow analysis for java card. In P. Honeyman, editor, *Proc. of CARDIS'02*. IFIP/USENIX, 2002.
16. M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *Proceedings of CARDIS'04*. Kluwer Academic Publishers, August 2004.
17. J. Andronick, B. Chetali, and C. Paulin-Mohring. Formal verification of security properties of smart card embedded source code. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *Proc. of FM'05*, volume 3582 of *Lecture Notes in Computer Science*, pages 302–317. Springer-Verlag, 2005.
18. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
19. The Caveat Project. `http://www-drt.cea.fr/Pages/List/lse/LSL/Caveat/index.html/`.
20. Common Criteria. `http://www.commoncriteria.org/`.