Using Coq to verify Java Card™ applet isolation properties

June Andronick, Boutheina Chetali, and Olivier Ly

Schlumberger Systems - Advanced Research on Smart Cards

Abstract This paper reports on the use of the Coq proof assistant for the formal verification of applet isolation properties in Java Card technology. We focus on the confidentiality property. We show how this property is verified by the card manager and the APIs, extending our former proof addressing the Java Card virtual machine. We also show how our verification method allows to complete specifications and to enhance the secure design of the platform. For instance, we describe how the proof of the integrity puts the light on a known bug. Finally, we present the benefits of the use of high order modelling to handle the complexity of the system, to prove security properties and eventually to construct generic re-usable proof architectures.

Key words: Theorem Proving, Smart Card, Security.

Introduction

A multi-application smart card can hold several applications coming from different vendors of different sectors and possibly loaded after issuance. This implies a new security model that has to assure the card issuer that the embedded applications will not corrupt its system, and the application provider that its applications are protected against the other ones.

In order to face these new security needs Java Card technology strengthens the inherent security of Java technology with a complex mechanism to control the sharing of information and services between on-card applications. This mechanism is known as the *applet isolation principle*, a central security issue in Java Card technology (see [6]). This principle relies on the classical sandbox model of Java security (see [13,10,19]) which consists in partitioning on-card applications into contexts, and verifying accesses across these contexts. This verification is supposed to face two main concepts: the *integrity* and the *confidentiality* of data of applets. Confidentiality (integrity respectively) means that data are protected from any unauthorized disclosure (modification respectively).

The work described here fits in the global objective of proving the correctness of the security architecture of the Java Card platform. It deals with the formal verification of the applet isolation principle. We actually focus on the formal verification of the confidentiality property. Our formalization of this property relies on the classical concept of *non-interference* (see [8,9]). We define the confidentiality as a non-interference property between on-card applets, assuming that they do not provide any *shareable interface*.

In a former work (see [1]), we proved that the virtual machine ensures the confidentiality property. This means that at the level of the bytecode interpretation, the property holds for applets that do not provide shareable interface. At this level, confidentiality mainly relies on the firewall mechanism used by the virtual machine which enforces a runtime checking during bytecode interpretation.

Although the virtual machine is central in Java Card technology, it fits however in a complex architecture relying on several other components. The execution process of an applet involves those other components of the Java Card platform like the communication module. Given that the whole security depends on the weakest link, proving the confidentiality at the virtual machine level is not sufficient. For instance, some global objects like the APDU buffer, used by the applet to communicate with the external world, are managed outside the control of the virtual machine. Therefore, the next step consists in verifying that the confidentiality property is respected during the whole process of applet's execution. For that, we have to consider the card management specific operations, for instance the management of the ADPU buffer, and the API which is extensively involved in executions of Java Card applications. The formal proof presented here consists in proving that the confidentiality holds at the card manager level, generalizing the result of [1] to the whole architecture of the Java Card platform.

The formal verification relies on the formal modelling of the Java Card architecture which has been developed in FORMAVIE project[1] (see [5]). This modelling has been developed within the language of the *Calculus of (Co)Inductive Constructions* and mechanically checked using the Coq proof-assistant (see [16]).

The paper is organized as follows: Section 1 gives an overview of Java Card technology from the security point of view. Section 2 gives a brief account about the formal modelling of the Java Card architecture which has been developed in FORMAVIE project. In Section 3, we describe the proof architecture taking into account the card management on one hand, and the API on the other hand; in particular, we show how we extend the formal statement of the confidentiality to the levels of the card manager and of the API. In Section 4, we point out some results about the integrity property.

Acknowledgement. The authors acknowledge Pr. C. Paulin-Mohring for her advises for this work.

1 Security in Java Card Technology

1.1 Java Card Platform

A Java Card based smart card is made of (see [11]):
- The *operating system* and the *native methods*[2].

[1] FORMAVIE project is an R&D project in the domain of information systems, partially funded by the French government. The partners of the project are Schlumberger Systems and the French research institute INRIA.

[2] i.e. written in a low level language.

- The *Java Card virtual machine* whose task is to interpret bytecode programs (the code obtained after Java compilation) and to enforce secure data sharing between Java Card applications at runtime, in particular data confidentiality.
- The *Application Programming Interface* (API for short) which handles Java Card specific features and also provides system services to embedded applications as class or library packages.
- The *card manager* which handles the life cycle of the card and its embedded applications. It is in charge of loading Java Card applications and managing inputs and outputs.
- The *applets* which are compiled Java Card applications.

A smart card communicates with the outside world via its eight contact points when it is inserted into a *Card Acceptance Device* (CAD for short), i.e. a reader or a terminal. The CAD supplies the card with power and establishes a data-carrying connection. The communication between the card and the CAD is done by exchanging *Application Protocol Data Units* (APDU for short). An APDU is a data packet containing either a *command* from the host to the card or a *response* from the card to the host: the card waits for an APDU command from the terminal; when it receives one, it executes the specified action; and then it replies to the terminal with an APDU response.

The card manager is the component which is in charge of the storage of Java Card packages, applet installation and initialization, card resource management, and communications. When the card is inserted into the CAD, first a reset occurs, then the card manager enters into a loop, waiting for APDU commands from the CAD. When an APDU command arrives, the card manager either selects an applet to run as instructed in the command or forwards the command to the running applet (the currently selected applet). By forwarding the command to the currently selected applet, we actually mean requesting from the virtual machine the execution of the `process` method of this applet. Once the processing of the command is finished, the card manager takes back the control and forwards the response to the CAD. The whole process repeats for each incoming command.

1.2 The sandbox Model

The *applet isolation principle*, i.e. the *isolation* regarding data access between applets embedded on the same card, is central for the security of multi-applicative smart cards. For instance, a smart card may contain a purse application together with a loyalty application both coming from different providers. In this case, the loyalty application should be able neither to get nor to modify any private information belonging to the purse. This example highlights the two concepts of isolation: the *integrity* and the *confidentiality* of data.

To enforce the isolation principle, Java Card security implements a sandbox-like policy. Each applet is confined to a particular space called a *context*, which is associated to its package; and the verification of the isolation between contexts is enforced at runtime by a *firewall* mechanism. More precisely, each package

defines a *context* and each applet it contains is associated to this context. In addition, there is a privileged context – the *JCRE context* – devoted to system operations.

Isolation of contexts relies on the concepts of *object ownership* and *active owner* which are defined as follows:

- An *owner* is either an applet instance, or the JCRE.
- An object *belongs* to the owner who created it, i.e. the owner which was *active* when it was created. The owner is then unique, determined at the creation of the object and never changed.
- During the execution of a non-static method, the *active owner* is the owner of the object that contains the method[3].
- During the execution of a static method, the *active owner* is the one which was active during the execution of the calling method. There is an exception to this rule: to install a new applet instance, the static method `install` of the class of the applet being installed is called by the JCRE; during this execution, the *active owner* is the applet instance to be created, instead of the JCRE. Therefore, any applet instance belongs to it-self, as all the objects it creates.

During the execution of a method, the *currently active context* is then defined as the context of the active owner. For instance, when the card manager forwards an APDU command to the selected applet, the `process` method of the applet is invoked and its context becomes the currently active context.

An object can only be accessed by a subject within the same context, i.e. when the object's context is the currently active context. Object accesses *across contexts* are allowed only in the four following cases:

1. Services and resources provided by the JCRE (the *entry point objects*) belong to the JCRE context, but can be accessed by any object.
2. Applets and the JCRE can share data through *global arrays* like the byte array parameter of the method `install` or the APDU buffer. These global arrays can be accessed from any context. However references to them cannot be stored; this avoids their re-use in non specified situations.
3. When the currently active context is the JCRE context, methods and fields of any object can be accessed.
4. Interaction between applets of different contexts is possible via *shareable interfaces*: when a given applet wants to make some methods available for other applets of different contexts, it provides an object whose class implements a shareable interface, i.e. an interface which extends the interface `javacard.framework.Shareable`. Such an interface defines a set of methods, that are the services that the applet in question makes accessible to other applets. These methods can be invoked *from any context.*

1.3 The Firewall

To enforce the isolation between the contexts, Java Card technology provides a dynamic check of these rules by the *firewall* (see [11]). This means that for each

[3] Let us note that Java Card technology does not provide any multi-thread mechanism.

bytecode instruction (`getfield`, `putfield` ...), a specific set of rules is defined. When the virtual machine interprets a bytecode instruction, it checks that the access conditions specified by the firewall rules are fulfilled. If they are not, a `SecurityException` is thrown.

The execution of a method m of a given applet consists of the interpretation of the successive bytecode instructions of m, including the execution of the methods of the API which are called in m. But the firewall mechanism enforced by the virtual machine only occurs during the bytecode interpretation, i.e. when the virtual machine interprets the methods written in Java. This dismiss the execution of API methods not written in Java.

In fact, the API methods are mostly implemented as *native*[4] methods. For instance, the method `javacard.framework.Util.arrayCopy(...)` is native for obvious performance reasons; while `javacard.framework.JCSystem.beginTransaction()` is intrinsically native because it is a direct request to the system. This implies that firewall rules cannot be directly applied in this context since there is no bytecode instruction interpretation. Moreover no security mechanism is specified for the execution of API methods; decisions are left to the developer.

However, such a mechanism is crucial for Java Card security to make sense. For example, let us consider the method `static short arrayCopy(byte[] src, short srcOff, byte[] dest, short destOff, short length)` of the class `javacard.framework.Util`. This method intends to copy an array of length `length` from the specified source `src` to the specified destination array `dest` according to offsets `srcOff` and `destOff`. No security constraints are imposed by the specifications about the owners of `src` and `dest`. Therefore, without any additional constraints on the use of this API method, any applet could steel the contents of any array of any other context, which definitely is contrary to Java Card security goals. Therefore, some security constraints must be added to the specification. In this case, the situation is clear: by extension of the firewall rules, we have to impose that `src` and `dest` both are accessible from the context of execution, i.e. the context of the caller of the method since this last one is static; if not, a `SecurityException` is thrown.

The purpose here is then to prove that under suitable conditions, the principle of isolation of applets holds for all the life cycle of a Java Card based smart card, from the card manager specific operations to the interpretation of each bytecode instruction of a method of an applet, including calls to API methods.

2 The Modelling of Java Card Technology

Here we present the formal modelling of the Java Card platform developed in FORMAVIE project (see [5]) on which the proof is based. The JCVM and the card manager have been formally modelled in the Coq system ([16]). This modelling is exhaustive regarding Sun specification (see [11]). Moreover a part of the API has also been modelled.

[4] i.e. written in a low level language.

2.1 The Virtual Machine

The Java Card virtual machine has been modelled as a *state machine*, the JCVM. The state encloses all the data needed by the virtual machine to interpret bytecode instructions and transitions describe this interpretation.

The definition of this state machine, fully described in [1], is given in Appendix A.

2.2 The Card Manager

The card manager is in charge of managing the whole life cycle of the card. The card life is divided into sessions. A session is the period from the time a card is inserted into the CAD and is powered up until the time the card is removed from the CAD. During a session, the card manager is in charge of dispatching the commands it receives and returning the responses to the CAD. The modelling of the card manager thus involves several concepts, depending on the level of abstraction.

At the highest level, the card manager can be seen as a relation between an infinite sequence of commands and an infinite sequence of responses, according to an initial state of the JCVM. In accordance with the partitioning of the card life into sessions, these sequences are represented by streams respectively defined by:

```
Definition card_in := (Stream input_card_session).
Definition card_out := (Stream output_card_session).
```

where `input_card_session` represents a list of *commands* and `output_card_session` a list of *responses*. A response is defined by:

```
Inductive typ_response : Set :=  Success : typ_response
                               | OutData : (list byte) -> typ_response
                               | Atr     : typ_response
                               | Fail    : status_word -> typ_response.
Inductive response : Set :=  NoResponse : response
                           | Response   : typ_response -> response.
```

The definition of a command is the generalization of the notion of APDU command (describing only the selection of an applet and a plain command), in order to take into account all the features of the card manager, such as the installation of applets or the loading of packages:

```
Inductive command : Set :=
    Select       : apdu_comm -> command
  | Command      : apdu_comm -> command
  | Load_File    : cap_format -> command
  | Install      : aid -> package_info -> install_params -> command
  | Load_Install : cap_format -> aid -> package_info ->
                   install_params -> command
  | Reset        : command.
```

The card manager is then represented by the co-inductive type:

```
card_life : card_in -> jcre_state -> card_out -> Prop :=
```

where the state records the packages installed on the card, together with the state of the JCVM:

```
Record jcre_state : Set :=
  JCRE_State  { installed_packages : package_table;
                execution_state :> jcvm_state        }.
```

The `card_life` is defined as an infinite sequence of sessions. Each session is in turn modelled inductively using the modelling of the dispatching of a single command. This last one is modelled by the predicate:

```
dispatcher: package_table -> jcvm_state -> command ->
            package_table -> jcvm_state -> response -> Prop.
```

that associates to an initial state of the JCVM and a received command, a final state resulting from the execution of this command and an output response to be sent back to the CAD. This predicate takes into account a set of loaded packages and defines a new set of packages if a loading of new packages has occurred.

The execution of an APDU command involves the interpretation of the `process` method by the virtual machine, including possible calls to API methods. For instance, the receiving of the command `Select apdu` causes an applet to become the currently selected applet. This applet is the one associated to the AID mentioned in the argument `apdu` of the command. Prior to the selection, the card manager shall deselect the previously selected applet, by invoking its `deselect` method. Then it informs the applet of selection by invoking its `select` method. The applet may refuse to be selected by returning `false` or by throwing an exception. If it returns `true`, the actual `Select apdu` command is supplied to the applet in a subsequent call to its `process` method. The `process` method is then interpreted by the virtual machine. The interpretation ends in a final state, returning the result of the method. The card manager analyses this value and sends a response to the CAD.

2.3 Modelling the API.

The formal specification of each method of the API is defined by pre and post conditions expressed on the JCVM states. These conditions are defined in a relational way as inductive types.

Precondition. For each method, the precondition specifies the necessary conditions on the JCVM state for the method to be invoked. It mainly defines the form of the operand stack, which contains the arguments of the method. In particular, it specifies the types of these arguments.

Postcondition. The postcondition of a method specifies the returned value of the method, together with the JCVM state resulting from the execution of the method. Let us note that the result may be the throwing of an exception. Postcondition of API methods have to specify the resulting JCVM state because some methods of the API are used as system entry points, i.e. they are used to send

requests to the system. This is the case of methods of the class `JCSystem`. For instance, methods like `beginTransaction()` or `commitTransaction()` allow applets to use the transaction mechanism, which is managed by the system; therefore, the specification of these methods describe their effects on the state of the system it-self, and especially the part dealing with transaction. Therefore the postconditions have to deal with the state of the system, and specify the effect the method.

Example. Let us consider for instance the method `boolean equals(byte[] bArray, short offset, byte length)` of the class `AID` which checks if the specified AID bytes in `bArray` (the ones starting at `offset` in `bArray` and of length `length`) are the same as those encapsulated in `this` AID object instance.

The precondition of this method specifies that at the invocation, the operand stack must have the form:

byte	short	reference	reference	...

where the first item represents `length`, the second one `offset`, the third one `bArray` and the fourth one the reference, let us say `ref`, to the object instance of class AID on which the method is called. On top of the required types, the precondition specifies that `ref` must be different from the `null` reference. The modelling of this precondition in Coq is straightforward and is not described here.

Let us note that the precondition does not specifies that `bArray` must be different from `null`. Indeed in this case, the specification of the method is that a `NullPointerException` must be thrown by the method; therefore, this is a part of the behaviour of the method which is specified by the postcondition.

The postcondition is defined as an inductive type of signature
`equals_post_cond : jcvm_state -> equals_info -> method_state_result -> Prop`.
This relation specifies the returned value of the method together with the resulting JCVM state from the initial state at method invocation and the arguments. The returned value and the resulting state are both enclosed in a term of type `method_state_result`. The arguments are enclosed in a term of type `equals_info` defined by the precondition from the initial state. Each case (normal case and exception throwing) gives rise to constructor (see Appendix B for full details).

All the postconditions of API methods follow the same scheme; in particular, the signature of each postcondition is as follows:
`jcvm_state -> pre_info -> method_state_result -> Prop`
where the type `pre_info` depends on the method being specified.

3 A Formal Validation Method based on Coq

3.1 Formalization of the Confidentiality

Non-interference. Our modelling of the confidentiality is based on the classical concept of *non-interference* (see [8,9,17]) which states that the confidentiality

of data is ensured if the values of these confidential data have no effect on the behaviour of external entities.

In our framework, we consider a context C, a selected applet α not belonging to C and a received command c which does not request the selection of an applet of C (see Remark 2 below). We want to verify that there is no disclosure of data of the context C during the processing of c by α. Let $\mathcal{F}_C(s)$ denote the contents of the fields of all objects belonging to C, i.e. the data to be protected, and $\mathcal{F}_{\overline{C}}(s)$ the contents of the fields of all objects *not* belonging to C.

Let us consider two states s_1 and s'_1 of the JCVM that may differ only on data of the context C. This means that we have $\mathcal{F}_{\overline{C}}(s_1) = \mathcal{F}_{\overline{C}}(s'_1)$ and that nothing in assumed on the values of $\mathcal{F}_C(s_1)$ neither on the ones of $\mathcal{F}_C(s'_1)$.

Then, the confidentiality is ensured if the two processings of the same command c from s_1 and s'_1 respectively leads to two final states s_2 and s'_2 that may differ only on data of the context C, and the two responses that are equal[5].

Equivalence of JCVM states. The simple equality of the states up to the confidential data is too restrictive and a notion of equivalent states is in fact needed. Indeed, when the virtual machine interprets the bytecode `new`, it needs a fresh reference. This fresh reference is retrieved from the operating system of the card, whose mechanism is not specified. This operation is non-deterministic. But the execution of the bytecode `new` on two states that are equivalent should lead to equivalent states, even if the fresh references which have been used are different. So, we must consider the equality of terms concerned *up to a one-to-one mapping of references.* But such a mapping can only be defined on references appearing on the heap. Therefore the JCVM state equivalence is only defined for *consistent* states, where a JCVM state is said to be consistent regarding references (consistent for short) if it contains in its data structures only null references or references appearing as index in the heap.

We can now define the equivalence of JCVM states up to one context.

Definition 1 (JCVM State Equivalence up to a Context).
Two consistent JCVM states s and s' are said to be equivalent up to the context C, which is denoted by $s \sim_C s'$, if there exists a one-to-one mapping φ such that all the components of s' except the heap, as well as the objects of the heap not belonging to C, are obtained from their counter-parts in s by replacing any reference ρ by $\varphi(\rho)$. No assumption is done about objects belonging to C.

Hypotheses. Let us now look at the hypotheses we have to assume to state a coherent definition of the confidentiality. We assume that C does not provide any shareable interface, since it corresponds to an authorized access across context (see page 4). Moreover, α is the selected applet in the initial state s_1 (and thus in s'_1), and therefore we suppose that this applet does not belong to C, otherwise the objects belonging to C could be accessed by any applet and in particular by α. For the same reasons, we assume that the received command does not request the selection of an applet belonging to C.

[5] The detailed steps to transform this informal definition of the confidentiality in a formal statement are presented in [1]; we explain here only the main needed notions.

Statement. We are now able to state a formal definition of the confidentiality:

Definition 2 (Confidentiality Property).
Let C be a context which does not provide any shareable interface. Let s_1 and s'_1 be two consistent states such that $s_1 \sim_C s'_1$. Let us assume that the selected applet in s_1 does not belong to C. Let c be a command received from the CAD which is not a command of selection of an applet of the context C. Let s_2 (s'_2 respectively) and r (r' respectively) be the state and the response resulting from the processing of c from s_1 (s'_1 respectively). Then $s_2 \sim_C s'_2$ and $r = r'$.

Remark 1. This definition specifies the confidentiality at the level of the processing of a single command, i.e. concerning the `dispatcher` predicate. This processing of the command is specified from a JCVM state, according to loaded packages. But the generalization to the card life is easily defined by stating that the executions of the same stream of inputs from two equivalent JCRE states respectively lead to the same stream of output. At this upper level, JCRE states are considered, enclosing the JCVM state and the loaded packages used by the `dispatcher` predicate.

Remark 2. The confidentiality property does not address commands which request the selection of an applet of the context C. The processing of such a command puts into action the applet to be selected, say β, in the context C. In particular, β becomes the active owner and C the currently active context[6]. As mentioned page 7, the method β.`process(apdu)` is then invoked. During the execution of this method, β can read or modify the data belonging to its context C. Actually β is responsible for protection of its context's data. In particular, it must ensure that no information disclosure may occur through the execution of its method `process` or within the response sent back to the CAD. Since this execution depends on the argument `apdu` for which β is not responsible, β can deny its selection according to its proper security policy defined at the level of the application, and not of the system.

3.2 Formal Verification

Architecture.

The verification of the isolation of applets has to be done at each step of the execution of a command received from a CAD. These steps are the following ones.

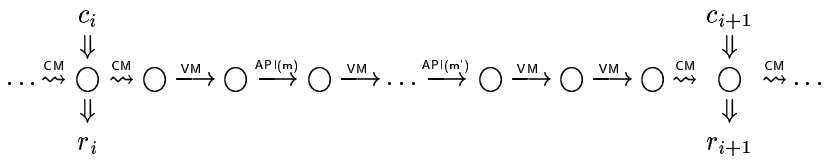
1. First, a pre-treatment is performed by the card manager in order to "prepare" the initial state of the JCVM. For instance, if the command has to be processed by the selected applet, the method `public void process (APDU apdu)` of this last one is called with the *APDU object* in argument. This APDU object is an instance of the `APDU` class, created by the card manager from the data contained in the received APDU command. Let us denote by $\overset{CM}{\rightsquigarrow}$ such a treatment of the card manager.

[6] See page 4 for the definition of active owner and currently active context.

2. Then the method (either `process`, `install`, `select` or `deselect`) is interpreted by the virtual machine. This interpretation consists of a finite sequence of transitions. Each transition can be either a JCVM transition (denoted by $\xrightarrow{\text{VM}}$), i.e. the execution of a single bytecode instruction, or a call to a method m of the API (denoted by $\xrightarrow{\text{API(m)}}$), i.e. a relation between the state at the call of m and the resulting state obtained after the full execution of m.
3. Eventually, the card manager performs final operations (also denoted by $\overset{\text{CM}}{\rightsquigarrow}$) in order to build the response to send to the CAD according to the returned value of the method, and also to prepare the state to receive the following command, including for instance the zeroing of the APDU buffer.

Let us note however that the processing of some command may not involve the JCVM such as the loading of a new package.

The verification of the isolation at the level of the card manager implies a proof at each step of this process:



Concerning the specific operations of the card manager, the verification consists in proving that the processing of commands and the building of the response do not introduce any leak regarding isolation.

Concerning the interpretation of the method, we have to prove that neither the interpretation of bytecode instructions nor the execution of methods of the API violate the confidentiality of data of applets.

Proofs.

At the level of the card manager, we have proved the confidentiality property as defined in Definition 2. The proof is done by case analysis on the kind of the received command. It depends on two main theorems, corresponding to the different kinds of steps needed to process the command.

The first theorem concerns the confidentiality of the execution of one bytecode instruction. It states that, assuming suitable hypotheses (in particular that there is no shareable interface in C), if there are two JCVM transitions $s_1 \xrightarrow{\text{VM}} s_2$ and $s_1' \xrightarrow{\text{VM}} s_2'$ such that $s_1 \sim_C s_1'$, then $s_2 \sim_C s_2'$. This proof has already been presented in [1].

Similarly, the second main theorem states that for each method m of the API, if $s_1 \xrightarrow{\text{API(m)}} s_2$ and $s_1' \xrightarrow{\text{API(m)}} s_2'$ such that $s_1 \sim_C s_1'$, then $s_2 \sim_C s_2'$ and the results of the executions are the same. This proof is presented in detail in the next section.

Eventually, the proof also relies on several lemmas stating that the confidentiality is preserved during the specific operations of the card manager such as the loading of new packages.

Confidentiality Property for the API.

Here we focus on the execution of API methods. We saw that API methods are specified with pre and postconditions. In particular, for each method, the postcondition specifies the JCVM state resulting from the execution of the method. Therefore, the relation that we denoted by $\xrightarrow{\text{API(m)}}$ is defined by the postcondition of the method m.

We define a generic confidentiality property for the API as follows:

```
Section ApiConfidentialityProperty.

Variable pre_info : Set.
Variable method_post_condition:
  jcvm_state -> pre_info -> method_state_result -> Prop.
Variable confidentiality_condition: jcvm_state -> pre_info -> Prop.
Variable pre_info_isom: pre_info->(reference->reference)->pre_info.

Definition confidentiality_api :=
 (* equivalent jcvm states at method invocation *)
   (invkst,invkst':jcvm_state)
   (own:owner)(phi:(reference -> reference))
   (jcvm_state_equivalence_up_to_one_context invkst invkst' own phi) ->
 (* equivalent parameters for the method *)
   (inf,inf':pre_info) inf'=(pre_info_isom inf phi) ->
 (* the caller context is not the JCRE *)
  ~(jcvm_caller_context invkst JCRE_Context) ->
 (* the context which must not be accessible *)
   (own:owner) (jcvm_caller_context invkst (Applet_Context own)) ->
   (hiddenown:owner) ~(same_owner own hiddenown) ->
 (* the hypothesis for confidentiality *)
   (confidentiality_condition invkst inf) ->
 (* the execution *)
   (poststate,poststate':jcvm_state)
   (res,res':method_result)
   (method_post_condition invkst  inf  (Meth_Res poststate  res )) ->
   (method_post_condition invkst' inf' (Meth_Res poststate' res')) ->
   (jcvm_state_equivalence_up_to_one_context poststate poststate' own phi)
   /\ (res'=(method_result_isom res phi)).

End ApiConfidentialityProperty.
```

Remark 3. Since the arguments and the result of a method may contain references, we must consider two executions from equivalents states, but also from equivalent arguments; and we prove that the results are equivalent.

Variables are used to instantiate the property to each method of the API:

- `pre_info` is a type depending on the method; it is designed to enclose the parameters of the method (see page 8).
- `method_post_condition` is the specification of the postcondition of the method.

- `confidentiality_condition` encloses the additional hypotheses, if any, to ensure the confidentiality. We saw page 14 that the specification of the security mechanism is not stated concerning the API. Therefore, it may happen that some additional conditions have to be fulfilled in order to ensure confidentiality; they are enclosed in this variable.
- `pre_info_isom` defines a specific equivalence for the type `pre_info` of the parameters of the method.

Remark 4. Here no hypothesis is done concerning shareable interface, since the API does neither provide nor call any.

To prove the confidentiality for the API, we instantiate this definition for each method, and prove it. Let us look at the particular example of the method `boolean equals(byte[] bArray, short offset, byte length)`. The confidentiality property is obtained by instantiating `pre_info` with `equals_info` and `method_post_condition` with `equals_post_condition` as defined in B. To achieve the proof of the property, we had to assume that `bArray` is accessible from the caller context, which is the instantiation of `confidentiality_condition`. This gave rise to the following result:

```
Theorem equals211_confidentiality:
let confidentiality_condition =
 [invkst:jcvm_state; inf:equals_info]
 (caller_array_access_security_constraints invkst (equalsinfo_arr inf))
in
(confidentiality_api equals_info equals211_post_cond confidentiality_condition).
```

We do not give the details of the definition of
`caller_array_access_security_constraints`
which expresses that `bArray` (here `(equalsinfo_arr inf)`) is accessible from the caller context according the security policy to regulate access across contexts (see [11]).

Remark 5. For each method, the instantiation of the variable `confidentiality_condition` actually completes the specification of the method from the security point of view. Therefore, the collection of these conditions for all the methods of the API specifies a sufficient security mechanism for the API to enforce the confidentiality property. This is an important application of this work.

Remark 6. Independently of our work, this rule has been added in Java Card 2.2 specification. However, we also did the proof of the same property on the models of Java Card 2.2; it is very similar to the one for Java Card 2.1.1; the only difference is that the security hypothesis is no longer needed.

4 About the Integrity

Concerning the API, a similar method can be used to prove the integrity property; it has been done in the case of the `AID` class.

The notion of integrity is simpler that the one of confidentiality since its violation is "observable". Indeed, the integrity is ensured during the execution of a method if the values of the confidential data at the end of the execution are the same that the ones at the beginning. Thus, with the notation of Section 3.1, the integrity for the API can by formally stated as follows.

Definition 3 (Integrity Property for the API).
Let m be an API method, Cxt a context and s a JCVM state. If the calling context in s, i.e. the one of the caller of the method m, is different from C and from the JCRE, and if $s \xrightarrow{\mathsf{API}(m)} s'$ then $\mathcal{F}_C(s) = \mathcal{F}_C(s')$.

This gives rise to the definition of a generic integrity property to be instantiated and proved for each method of the API (see Appendix C for details).

Example. Let us look here at the following example, well known from Java Card technology developer community. Let us consider the method `byte getBytes(byte[] dest, short offset)` of the class `javacard.framework.AID` which is supposed to put in the array `dest` all the AID bytes encapsulated within the `AID` instance from which the method is called (`offset` specifies where the AID bytes begin within `dest`). Here the security problem concerns the access to the array `dest`. The basic extension of the firewall rules is not sufficient to ensure applet isolation. Indeed, this natural extension would be to allow access to `dest` only if the context of execution of the method is the JCRE context, or is the same as the context of `dest`. Following such a policy would introduce a security hole. Indeed, `AID` instances are provided by the JCRE and belong to it. Any applet α can get such an instance, say `a`, via the method `javacard.framework.JCSystem.lookupAID(...)`. Now, if α invokes `a.getBytes(dest,offset)`, there is a context switch into the JCRE context since `a` belongs to the JCRE; this gives to the method all the rights to access any object in any context. In particular, the method can put the bytes enclosed in `a` into `dest`, whatever the context of this last one is. It particular, even if the context of `dest` is different from the context of α, the content of `dest` is erased. In conclusion, α would be able to modify any byte array in an other context in an indirect way by using the JCRE. This is again contrary to Java Card security goals. So, a stronger security rule must be enforced to control the use this method: `dest` must be accessible from the context of the caller of the method.

This information leak has also been revealed during the development of the proof of integrity. Indeed, the introduction of an additional hypothesis has been necessary to complete the proof:

```
Theorem getBytes211_integrity:
let integrity_condition =
 [invkst:jcvm_state; getbinf:getB_info]
 (caller_array_access_security_constraints invkst (getbinfo_arr getbinf))
in
(integrity_api getB_info getBytes211_post_cond integrity_condition).
```

Similarly to the case of the method `equals`, this rule has been added in Java Card 2.2 specification.

Conclusion

A Formal Method to Complete Specifications. The security mechanism of the API is not specified in Sun specification (see [11]); it is actually left to the design and implementation stages. Therefore, our work turned out to be no longer the verification of a specification, as our former work [1], but mainly the completion of the functional specification of the API aiming at assuring confidentiality. Our method of validation actually provides a way to complete the specification with sufficient conditions to ensure applet isolation. Indeed, for each method of the API, the development of the proof of the confidentiality property introduces a set of hypotheses. This set is actually used to complete the specification of the method.

Let us emphasize that the use of a proof assistant handling higher order logic has been crucial in this work. The definition of infinite traces describing the executions of a machine as well as that of propositions over those traces are directly expressed in terms of the mechanisms for supporting co-inductive definitions. Moreover the formalization of the confidentiality property uses quantification on bijections between Java references.

Related Work. In the framework of Java Card technology, the problem of confidentiality has been investigated in [7]. This work concerns applet interactions and focuses on the qualification of admissible information flows. Verification of applet isolation has also been investigated in [12] by setting up a type system at applet source code level to check applet isolation. More generally, formal verification techniques has been investigated for Java source code, and in particular for the Java Card API in [15,14,18]. These works deal with applets and not with the underlying system, i.e. the virtual machine, the API, etc. Our approach is complementary: we focus on the verification of applet isolation at the level of the Java Card platform itself.

Java Card platform security has been investigated in [2,4,3]. In particular, these studies established a correspondence between offensive and defensive Java Card virtual machine. Here we focus on applet isolation and especially on confidentiality, adding another building block to prove Java Card security.

Future Work. A first direction to investigate is to generalize our result about applet isolation, in order for instance to take into account the case of shareable interfaces. A second direction consists in checking the property for actual implementation of the Java Card platform.

References

1. J. Andronick, B. Chetali, and O. Ly. Formal Verification of the Confidentiality Property in Java Card™ Technology. Submitted at Journal of Logic and Algebraic Programming.

2. G. Barthe, G. Dufay, M. Huisman, and S Melo de Sousa. Jakarta: a Toolset to Reason about the JavaCard Platform. In I. Attali and T. Jensen, editors, *Proceedings of E-SMART'01*, volume LNCS 2140, pages 2–18. Springer-Verlag, 2001.
3. G. Barthe, G. Dufay, L. Jakubiec, and S. Melo de Sousa. A Formal Correspondence between Offencive and Defensive JavaCard Virtual Machine. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume LNCS 2294, pages 32–45, 2002.
4. G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume LNCS 2028, pages 302–319. Springer-Verlag, 2001.
5. G. Betarte, B. Chetali, E. Gimenez, and C. Loiseaux. Formavie: Formal Modelling and Verification of the JavaCard 2.1.1 Security Architecture. In *E-SMART 2002*, pages 213–231, 2002.
6. Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison Wesley, 2000.
7. Mads Dam and Pablo Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *13th IEEE Computer Security Foundations Workshop*, pages 233–244. IEEE Computer Society Press, July 2000.
8. J.A. Goguen and J. Meseguer. Security Policy and Security Models. In *Proc. of the 1982 Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.
9. J.A. Goguen and J. Meseguer. Unwinding and interference control. In *Proc. of the 1982 Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society Press, 1984.
10. Gary McGrow and Ed Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, 1999.
11. Sun Microsystems. *Java Card 2.1.1 Specification*. `http://java.sun.com/products/javacard/`, 2000.
12. P. Müller and A. Poetzsch-Heffter. A Type System for Checking Applet Isolation in Java Card. In S. Drossopoulou et al, editor, *Proceedings of FTfJP'01*, 2001.
13. Scott Oaks. *Java Security*. O'Reilly, 1998.
14. E. Poll, P. Hartel, and E. de Jong. A java reference model of transacted memory for smart cards. In *Fifth Smart Card Research and Advanced Application Conf. (CARDIS'2002)*, 2002. To Appear. See `http://www.cs.kun.nl/VerifiCard/files/publications.html`
15. E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the java card api in jml: the apdu class. *Computer Networks*, 36(4):407–421, 2001.
16. The Coq Development Team LogiCal Project. *The Coq Proof Assistant Reference Manual*. `http://pauillac.inria.fr/coq/doc/main.html`
17. John Rushby. Noninterference, transitivity, and channel-control security policies, dec 1992.
18. J. van den Berg, B. Jacobs, and E. Poll. Formal specification and verification of javacard's application identifier class. In *Proceedings of the Java Card 2000 Workshop*, 2000. `http://www.irisa.fr/lande/jensen/jcw-program.html`
19. Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 1997.

A Virtual machine modelling

A.1 JCVM States

A *JCVM state* is made of:

- the *execution status*, which can be:
  - either a *frame stack*, if the machine is interpreting a method; each frame of the stack corresponds to a method call; it records: the *current context* in which the method is executed, the *operand stack*, the *local variables* and the *program counter* which points the bytecode to be executed.
  - or a *return value*, if the processing of the method is finished,
  - or an *exception* which has been raised and not caught,
  - or eventually a *fatal error*, if the machine goes into an unrecoverable error,
- the *heap*, which is an association table mapping references to object instances.
- the *static field image* , which records, for each loaded package, all the static fields of all the classes of the package.
- and finally, the *log status*, which records data needed to manage transactions; a transaction is a piece of code that must be executed *atomically* (see [11]); if a failure occurs, for instance if the card is removed from the CAD, all the operations executed from the beginning of the transaction must be canceled and the state of the card restored; thus, in order to allow the roll-back, all these operations have to be stored in the log status.

The set of JCVM states has been encoded in Coq as an inductive type:

```
Record jcvm_state : Set :=
  JCVM_State
    { state_execution_status : jcvm_execution_status;
      state_heap : jcvm_heap;
      ...
    }.
```

This type uses several inductive types like `jcvm_execution_status` which encodes the four kinds of execution status, and so on.

Remark 1. Some components have not been modelled but left as parameters of the specification since they are not explicitly described in Sun specification (see [11]). For instance, this is the case of *Java references* which are functionally defined by their properties.

A.2 Transitions

A *JCVM transition* between two JCVM states represents the execution a single bytecode instruction. More precisely, a transition from a state s to a state s' (denoted by $s \xrightarrow{\text{VM}} s'$) corresponds to the execution on s, of the current bytecode of s, i.e. the one pointed by the program counter of its top frame.

Let us look for instance at the bytecode instruction `getfield_s idx` which aims at fetching the value of an object field (see [11]). The interpretation of this bytecode roughly consists of:

1. popping the top word[7] of the operand stack of the top frame of the frame stack of the state. This word must be a valid reference to an object of the heap. If the reference is `null`, then a `NullPointerException` is thrown. Regarding the firewall rules, if the current active context[8] is not authorized to access the object in question, then a `SecurityException` is thrown.
2. retrieving the field encoded by the token `idx` in the object in question, and pushing it onto the operand stack.

Transitions have been encoded in Coq in a *relational* way by an inductive type of the following signature:

```
jcvm_transition : package_table->applet_ident->jcvm_state->jcvm_state->Prop.
```

which associates two states connected by a transition of the JCVM according to the table of loaded packages and the currently selected applet. The choice of a *relational* modelling allows the specification to follow exactly the informal specification [11], without any addition of information. However let us note that some low level unambiguous operations have been modelled in a *functional* way like for instance the jump operation of the program counter.

[7] The word is the atomic data unit used in Java Card technology.
[8] See page 4 for the definition of the current active context.

B Postcondition of the method `equals`

The postcondition of the method `equals` of the class `javacard.framework.AID`[9] is defined in Coq by the following predicate:

```
Inductive equals_post_cond
  [ invkst:jcvm_state;
    equalsinf:equals_info;
    methdres:method_state_result] : Prop :=

  Normalequals:
     let arrf  = (equalsinfo_arr   equalsinf) in  (* bArray *)
     let offs  = (equalsinfo_offs  equalsinf) in  (* offset *)
     let lngth = (equalsinfo_lngth equalsinf) in  (* length *)
     let rf    = (equalsinfo_this  equalsinf) in  (* this *)
     let hp    = (state_heap invkst) in

   (* rf is a reference to an AID instance of length aidlth
      with value aiddata *)
     (aid_instance_ref invkst rf) ->
     (aiddata:(list byte))
     (get_aid_instance_value hp rf (AID_Inst_Value aiddata)) ->
     let aidlth = (length aiddata) in

   (* offset is not negative and offset + aidlth is less than
      or equal than the length of the array parameter *)
     ...

   (* The returned value *)
     IF   ~arrf=null
       /\(get_byte_array_segment hp arrf offs (byte2short lngth) aiddata)
     then  (res_app methdres) = (Meth_Val (Some (Data_w (Short sone))))
     else  (res_app methdres) = (Meth_Val (Some (Data_w (Short szero))))) ->

   (* The returned state *)
     (res_st methdres) = invkst ->

     (equals_post_cond invkst equalsinf methdres)

| OutOfBounds_equals:
     ...
   (* The returned value is a reference to the IndexOutOfBoundsException
      object *)
     (res_app methdres) =
           (Meth_Exc (exception_reference IndexOutOfBoundsException)) ->
     ...
     (equals_post_cond invkst equalsinf methdres).
```

[9] Java Card Version 2.1.1 (see [11]).

This relation specifies the result of the method together with the resulting JCVM state, from the JCVM state `invkst` at method invocation and the arguments. The result and the resulting state both are enclosed in a term of type `method_state_result`. The arguments are enclosed in a term of type `equals_info` defined by the precondition from `invkst`. The first constructor of the relation specifies the normal behaviour of `equals(byte[] dest, short offset, byte length)` where each control is successful and no exception is thrown. The second constructor specifies the error case where the exception `IndexOutOfBoundsException` must be thrown.

C Integrity property

Here is the definition of a generic integrity property for the API:

```
Section ApiIntegrityProperty.

Variable pre_info : Set.
Variable method_post_condition:
    jcvm_state -> pre_info -> method_state_result -> Prop.
Variable integrity_condition:
    jcvm_state -> pre_info -> Prop.

Definition integrity_api :=
 (* the jcvm state at method invocation *)
   (invkst:jcvm_state)
 (* the caller context is not the JCRE *)
  ~(jcvm_caller_context invkst JCRE_Context) ->
 (* the context which must not be accessible *)
   (own:owner) (jcvm_caller_context invkst (Applet_Context own)) ->
   (hiddenown:owner) ~(same_owner own hiddenown) ->
 (* the hypothesis for integrity *)
   (inf:pre_info)
   (integrity_condition invkst inf) ->
 (* the execution *)
   (poststate:jcvm_state)
   (res:method_result)
   (method_post_condition invkst inf (Meth_Res poststate res)) ->
   (context_equal hiddenown invkst poststate).

End ApiIntegrityProperty.
```

AUTHORS' CONTACT :

June Andronick, Boutheina Chetali and Olivier Ly

36-38, rue de la Princesse, BP45, 78431 Louveciennes Cedex, France

Phone number: (+33) (0)1 30 08 45 00

{june.andronick, boutheina.chetali, olivier.ly}@
louveciennes.sema.slb.com