New Security Issues Raised by Open Cards

[Published in *Elsevier Information Security Technical Report* **4**(2):19–27, 1999.]

Pierre Girard and Jean-Louis Lanet

GEMPLUS R&D
Parc d'Activités de Gémenos - B.P. 100
13881 Gémenos CEDEX - France
{Pierre.Girard, Jean-Louis.Lanet}@gemplus.com

1 New Open Smart Card Systems

The methods, languages and tools for developing a smart card system share some characteristics with those of the embedded domain. Until recently, smart card programs were written in hand coded native assembly language. All programs (drivers, operating system, libraries, applications) were developed as a monolithic piece of code burned in the smart card ROM. Therefore, not only are traditional card systems difficult to develop (low-level programming language, very reduced-feature microcontroller, specific code for every microprocessor) but they also cannot support any evolution of their applications since all the application code is burned forever with the runtime engine in the ROM.

Moreover, the production of such a monolithic program dedicated to a specific hardware and with *ad hoc* functions for the application domain consumes most of the card development cycle. In order to issue a card application, it is required (i) to write precise specifications, (ii) to write or re-write the basic software (akin to an operating system) for possibly multiple platforms, (iii) to develop specific functions for the application, and (iv) to verify this software prior to deploying it on thousands or millions of cards. This process is time-consuming and costly. Since defining specifications for products that will be available long after is risky, this process has been inadequate for the creation of new markets. As it requires a long time it also severely limits the ability of a card issuer to quickly deploy new applications in accordance with the market needs.

In order to adapt to these markets (*i.e.*, to reduce the "time-to-markets" and to increase the flexibility for card applications) new generations of smart cards (called *open* smart cards) have emerged during the last two years. Most notable efforts towards such smart card systems are Java Card, MultOS and Smart Card for Windows which provide to application developers an opportunity to create applications on a common base of code. They contain a platform for dynamic (*i.e.*, on demand) storage and execution of downloaded executable content, which is based on a virtual machine for portability across multiple smart card microcontrollers and for security reasons.

While these new smart cards bring solutions regarding the market needs, they also introduce new problems for smart card manufacturers. They provide solu-

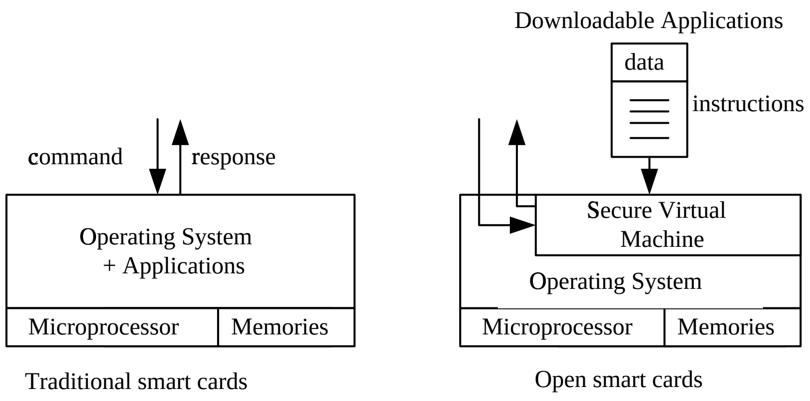

Fig. 1. From traditional to open smart cards.

tions for card application developers by enabling them to program in high-level languages, on a common base of software (an abstract machine and application programming interfaces) which isolates their code from specific hardware and operating system libraries. In that sense, they can reduce drastically the time to get new applications to market. They also tend to support both the flexibility and the evolution of applications by enabling the downloading of executable content in already deployed smart cards. This later characteristic has required more sophistication in terms of security techniques since any program (even malicious or erroneous ones) could potentially damage or misappropriate the whole card system.

Ensuring that a program cannot damage a system consists in denying it access to other memory areas than those reserved for its execution and for its code and data (containment or “sandboxing”). Containment relies on access controls to memory areas. It would be better to associate containment with a protection mechanism insuring that every instruction accesses data with respect to their types. In fact, a more fine-grained protection consists in verifying that every instruction that accesses to a memory area is compliant with the definition of the data stored in this area (*i.e.*, its type, or its class in an object-oriented system). This later technique has been popularized in the Java language with the verification step performed by the virtual machine each time it loads a new class from an unsecure source. In the field of open smart cards, this security issue is generally addressed by an off-card verifier-converter which performs a static verification of type-safety.

In this paper, we aim to discuss various threats raised by Java Cards at various levels of the system. First, we present the security offered by the card environment, *i.e.*, how to securely download code on cards. Then we address the Java Card platform security itself, from the chip security features to the Java Card virtual machine. Next, we expose how to deal with application security which is a standard problem for smart card manufacturers but a quite new one for third party Java developers beginning to code Java Card applets. Finally, we highlight the security issues coming from objects sharing inside the card and we present a few hints to prevent faults in smart card software.

2 Secure Code Downloading

Currently, the Java Card framework does not support Java-like dynamic class loading. The main problem is that a smart card environment is too small for running the Java class file verifier. Thus the verification process must be done off-card. Open card architectures hence propose a downloading framework with reduced flexibility, in which the downloaded unit is the application (or package in Java) rather than a single class file.

Off-card byte code verification is performed by a third-party (the card issuer in the figure below). This entity signs the successfully verified applet to ensure that any attempt to tamper with the byte code after its verification will be detected. The applet and the certificate are carried out on a server in a network. A client (the end user) can download the applet and check the digital signature before execution.

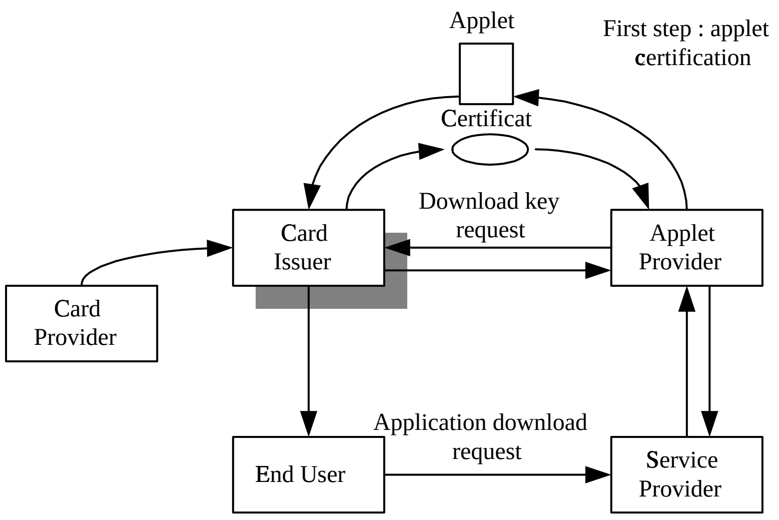

Fig. 2. Certification scheme.

2.1 VISA Open Platform

The benefits of Java Card in term of applets portability are undoubtedly of great interest to deploy an application on a wide range of smart card. However, until now, the standardization of applets loading is not in the scope of the Java

Card standard. This turns out to be a disappointment for the customers as it prevents them from transparently distributing an applet on Java Cards from different manufacturers.

To fill this last gap, Visa has written a specification, called Visa Open Platform (see figure 3), for its own needs (credit/debit cards and electronic purses) but, as this proposal is the only one, it is likely to become a *de facto* standard for all the smart card industry.

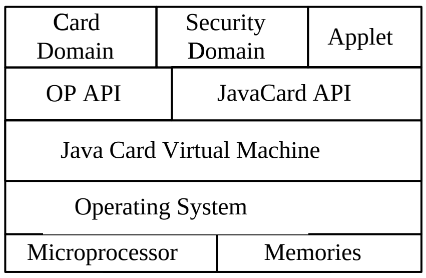

Fig. 3. Visa Open Platform Card Architecture.

Visa Open Platform defines extensions to the Java Card components to achieve card interoperability for applets downloading and management. The main additional components are the following:

– The Open Platform API: In addition to the Java Card API, the Open Platform API provides to the downloaded applets the interface to access the Open Platform services offered by the following components.
– The Security Domain: Every applet on a card belongs to a Security Domain (specified by the terminal when the applet is downloaded), which basically represents the authority of the applet's provider. The security domains are implemented through special applets, called Security Domain applets, one applet for each domain. They are used to authenticate and verify applets during the downloading process. They also offer common services for all applets of a given provider.
– The Card Domain: This component represents the Issuer's interest on the card and keeps the entire card under its control. The Card Domain can be viewed as a security domain with additional features. It has the exclusive right to download code to the card and to authenticate and verify them through the appropriate security domain. Finally, this component keeps track of the card state and the applets state and various others parameters for auditing purpose.
– The Card Executive: This part of the Open Platform receives all incoming commands from the external world and dispatches them to the proper part of the card. The Card Executive monitors applets selection commands to keep

track of the current applet. All commands are routed to the current applet until another select command is issued. It also has a significant contribution to the data sharing mechanism.

2.2 Card Management System

An essential range of back-office software are also currently under development: applets servers and card management software. Applets servers will be used as secure applet repository whereas card management software will be able to deal with the complexity of huge number of emitted cards with cards having content different from one another.

Applet servers need to meet a lot a security criteria. They must of course guarantee the applet integrity (but also allow an authorized entity to update the reference version of an applet), but also confidentiality, because an applet can contain proprietary algorithms and/or sensible data such as commercial figures. Finally, the availability should be carefully assessed as applet distribution is an asynchronous mechanism. When the issuer decides to download an applet on a range of cards disseminated world-wide, they are not updated immediately. The terminals have to watch for the cards to be updated and require the needed applet when the event occurs.

The card management systems can be viewed as the control panel of Java Card batches. They contain detail information about each card: card state, applets state, applets loaded, free memory available, etc. They can communicate with the terminals to instruct them to update or modify cards, to get feed-back information on instructed operations and to obtain up-to-date information from the cards (audit data, security alerts, etc.).

Card management systems will probably be strategic tools to manage a set of Java Cards and to offer the best cooperation to third parties such as applets providers.

3 The Java Card

3.1 Physical and Operating System security

The operating system security, as for classical smart cards, relies on the physical security of the microprocessors. The chip provides a wide range of security features and embedded counter-measures against potential attacks. Many of them are still classified, but some are well-known such as sensors monitoring the chip power supply, clock frequency, temperature, etc. The persistent memory is also entirely designed from a security point of view. The memory layout does not match with the logical addresses and integrates witness cells, which are not addressable but checked by the memory logic in order to detect a tampering attempt, as well as dummy cells which are useless but can confuse an attacker. The memory busses can be ciphered to avoid internal data probing and be scrambled within dummy logic.

The security of the Java Card virtual machine is built on the operating system security which must provide reliable services. This is of special concern for the cryptographic primitives which must be safe regarding all the state-of-the-art attacks such as timing attacks, power analysis, etc. The memory management is also a critical issue and must be safe and robust against card power-loss or memory failure.

3.2 The Java Card system

A Java Card system includes several components *i.e.,* the Java Card virtual machine, the Java Card converter and installation tools on both terminal and card side. The aim of the converter is to transform a compiled Java program *i.e.,* a class file, into a format downloadable into the smart card *i.e.*, the CAP file format. The converter uses the export file which contains naming and linking information about other packages imported by the classes being converted. In the same way, once the class file is converted it generates the export files for the converted package.

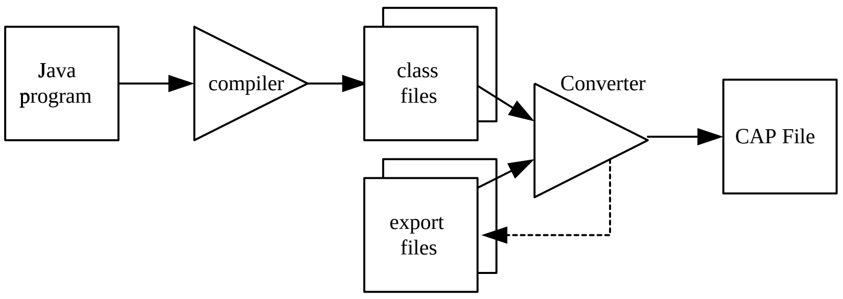

Fig. 4. From Java program to the CAP file.

After conversion, the applet is transferred into the card using the loader program (Installer) and the last part of the linking process is achieved. Then, the virtual machine is able to run the applet. The applets are selected and deselected by the Java Card Runtime Environment (JCRE). It keeps track of applets that are selected and the current active applet. The JCRE is made up of the virtual machine and the Java Card API classes. The Java Card virtual machine is specially designed for smart card, several features have been removed. The Java Card API is a set of tools or services to help programmers in designing Java Card applets. The API services propose access to the basic storage, transaction, atomicity, communication and security features of the Java Card. Instead of having to write sensitive code, the programmer can use these services to rapidly and securely produce applications.

While the virtual machine insures Java language-level security, the JCRE performs additional runtime checks. Thus it modifies the original virtual machine by adding specific features. The additional runtime checks are done by the applet firewall. This program is in charge of controlling the segregation between applets and the correct access to objects. It prevents unauthorized access to the fields and methods of class instances. Due to the fact that the applet must share objects in some situations, the applet firewall controls the strict access to the shareable interface of these objects.

3.3 The virtual machine

The virtual machine interprets Java byte codes and translates them into instructions that are understandable by the hardware. Each hardware platform has its own Java Card virtual machine but this provides a support for hardware independence. The other benefit is the improvement of the security. The interpreter controls that each Java byte code precondition is met prior to its execution. By this means, any attempt of an applet to illegally access part of another applet space is detected by the interpreter and a runtime exception is propagated.

Most of the controls of the interpreter can be done statically. Performing off-card verification has significant advantages for long life applets. Using a powerful machine, it is possible to execute checks that use time consuming heuristics avoiding to reject correct applets. The drawback of this approach lies on the fact that it requires a trusted service and the various participants have to trust a same authority. Moreover such an approach creates a single point of failure due to the use of a unique secret.

Alternative solutions are based on a defensive virtual machine or on verification using the Proof Carrying Code technique. Within the first solution all safety checks are performed at runtime. For instance, before a write operation, the virtual machine checks the type and access conditions associated with the data located in the target area. Defensive virtual machines have two major drawbacks. First, the number of runtime checks to be performed severely hinders their performances, moreover additional data (such as typing information) need to be stored which increases their memory requirements. Another approach is to adapt the Proof Carrying Code technique as a means to perform on-card verification of the type safety of Java Card programs. In this approach a first verification process is done off-card and a typing proof is produced. This proof represents the type information for labels. Then the proof is sent along with the byte code without any cryptographic means. On card, a type checker verifies the code using the proof. The embedded algorithm performs type inference between two labels and checks the coherence between the on-card inference and the proof. Such technique seems very promising but the complexity of the on-card processing for a full type checking makes its implementation still difficult on current smart cards.

A full implementation of the Java virtual machine cannot fit on the resources-constrained devices available today. Several elements of Java are not supported:

- dynamic class loading: program executing on the card must refer to masked classes or already downloaded packages;

– security manager: the Security Manager class is not implemented but the virtual machine shares its functionality with the applet firewall;
– threads;
– garbage collection: the garbage collector is not mandatory. Without a system for freeing allocated memory, the card may become unusable due to denial of service attacks;
– multi dimensional arrays;
– floating operations.

Java Card allows the use of native methods. In such a case the execution is neither under the control of the virtual machine nor the firewall. The Java Card does not provide any security model and means for native methods. The native functions are not restricted by the JCRE access control mechanism. A misuse of native methods can break the security of the Java Card.

3.4 Object sharing

As explained previously the applet firewall is in charge of controlling the separation between the different object spaces called contexts. A context is associated with each package in such a way that all applets of a package share the same context. If the virtual machine executes an invoke byte code and if certain conditions are met, the virtual machine performs a context switch by pushing the current context on the stack. The caller context is restored (popped from the stack) at the end of the method. The JCRE has its own context with specific privileges.

Each object is owned by either an applet or the JCRE context. The ownership of a created object is related to the applet of the current context. An object can be accessed only by the owning context if the latter is the active one. Such a mechanism prevents unauthorized access to an object.

To enable interaction between applets, the standard defines the *shareable interface* concept. It is a means to invoke from one context methods an object it does not own. Note that the fields and other methods of the object are not accessible, only the methods defined in a shareable interface are available through the firewall. When a method in a shareable interface is invoked, a context switch occurs to the context of the object's owner under the control of the JCRE. Such a mechanism allows secure inter-applet communication.

4 Application Security

Until now, we have only dealt with the platform level security but the application level security should also be addressed.

The first one (platform level security) concerns applications segregation as well as the quality of security services offered by the platform e.g., correctness of the Java virtual machine including the verifier, tamper resistance, cryptographic algorithms and post-issuance loading mechanism. This part is under the issuer's responsibility. One of the means for fault prevention consists in using formal methods as described in §6.

The second one (application security), is under the provider's responsibility, but relies necessarily on the platform security. Moreover, applications should assume that the OS will not be aggressive and will act as supposed. Conversely, the OS does not make any assumption about the application and should still work and protect the other applications even if an aggressive or unsecure piece of code is loaded. However, one should note that even if this is technically perfectly acceptable, and if an unsecure application cannot threaten the platform or other applications, the end users or potential customers could get confused by a break-in of an application and mix up the platform security and the application security. To avoid this potential damage to its brand image, an issuer could enforce a minimum security level for applications loaded on its card by reviewing them or operating a scheme including some certification authorities.

Another trend is to give a manufacturer security libraries with a high level of security assurance containing the manufacturer cutting edge security know-how. With some precise security guidelines, even a naïve Java developer would be able to produce secure applets.

5 Secure Data Sharing

Apart from platform and application security aspects, a third one must be addressed. All the difficulties arise from data sharing inside a card. Actually, most of multi-application smart cards, in order to build cooperative schemes and optimize memory usage, allow data and service sharing (*i.e.*, objects sharing) between applications. Beyond this point there is a need for a card-wide security policy concerning all applications. A small example should clarify this point. When an application provider A decides to share (or more probably to sell) some data with an application provider B, it asks for guarantees that B is not able to resell those data or make them available world-wide. The threat could be a commercial concern or a privacy concern.

A mandatory security policy is necessary to solve the problem of re-sharing shared objects as mentioned above. The security policy should model the information flows between the applications which, themselves, reflect the trust relationships between the participants of the applicative scheme. The best candidate for such a mandatory policy appears to be a multilevel policy.

Enforcing the security policy could be done dynamically by a reference monitor (part of the card OS), which is called each time an object reference is used by the virtual machine, or statically by checking the correctness of the information flows in an applet. The first solution would be too costly in memory and execution time which are both critical in a smart card as each object should be tagged with its level and check the validity of each read/write operation.

The second solution has been studied for a long time and could be integrated to existing static verifier of Java byte code. A current Gemplus project, called

PACAP[1] is focused on this approach and aims at checking the data flows between objects on the card by static analysis prior to applets downloading, for a given configuration.

Practically, this means that an applet provider delivers its code to the issuer along with the security level of all the objects contained in it. The issuer verifies that the code and the declared levels of the objects comply with the other applets and their objects security levels.

6 Fault Prevention

All Java Card mechanisms prevent hostile applets from breaking the security of the smart card. However the smart card security is based on two assumptions: the JCRE is correctly implemented and applets loaded onto the card have been previously checked. The correctness of the JCRE is very important because it is the mean to avoid an applet to reference illegally another applet's objects. In fact not only the JCRE but also the virtual machine must be correctly implemented. At this level, it is possible to use formal methods in order to prove such a correctness. Formal methods insure that the implantation is a valid interpretation of the specification using a mathematical proof.

6.1 The B Method

The B Method is a formal method for software engineering developed by J.R. Abrial [1]. It is a *model oriented* approach to software construction. This method is based on set theory and first order logic. The basic concept is the *abstract machine* which is used to encapsulate data describing the state of the system. Invariants can be expressed on the state of the machine which can only be accessed by the specified operations.

The abstract machine is refined by adding specification details. Several steps can be used before reaching the implementation level where the specification is detailed enough to generate code. The refinements and the implementation have other invariants which express relations between the states of the different refinements.

The proof process allows to check the consistency among the mathematical model, the abstract machine and the refinements. This way, it is possible to prove that the implementation is correct according to its specification. The tool *AtelierB* generates the proof obligations of the specification according to the mathematical model. A theorem prover is provided to automatically discharge some of the proof obligations and an interactive theorem prover allows the user to interact in the proof process. Currently we model several parts of the Java Card using the B method. We provide hereafter an example of a virtual machine model.

[1] The PACAP project is partially founded by MENRT contract n°98B0252.

6.2 A formal model of the virtual machine

One of the most difficult problem in Java security is to preserve the type safety. The security model can be broken if you can confuse the virtual machine about the types you manipulate. The use of formal methods for the design of the virtual machine aims at:

– providind a formal description of the virtual machine;
– extracting the static checks;
– formally demonstrating that the interpreter fulfills the static constraints;
– providing a reference implementation of both the verifier and the interpreter.

With this approach it is possible to reduce runtime checks. It is based on a Defensive Java Virtual Machine (DJVM) that we split to obtain in the one hand the byte code verifier and in the other hand the interpreter. At the abstract level, we formalize the DJVM as described in the sun Java Card description. By successive refinements we add new details to refine the specification. Then, we extract the runtime checks in order to de-synchronize verification and execution process. We obtain invariants representing the formal specification of the static checks. We implement those specifications with an on-the-shelf type inference algorithm.

Using the B method, it is possible to model the complete JCRE. Currently we provide a mathematical proof of the correct implementation of the interpreter, the firewall and the backup mechanism (OS functionality).

7 Conclusion

The new generation of smart cards provides solutions for card developpers to reduce the time-to-market and software evolution of their already deployed smart cards. The strong typing of Java enforces the language based security, but is not sufficient. The application security relies on a proven implementation of the OS and the associated JCRE. Ensuring the correctness of this implentation is the basis of the platform security. This can be done through a mathematical proof of the implementation.

At the application level, a Java programmer can break the security with a naïve implementation of security mechanisms. Using a correct implementation of security mechanisms through the Gemplus library allows to avoid a potential attack. Moreover the JCRE control can be bypassed by different ways: native methods, transitive flow information by data sharing... A static analysis can protect the JCRE from this kind of attack.

As a conclusion, it is clear that the Java Card is a powerful framework to develop and deploy applications. But care must be taken in order to guarantee the security of the system.

8 Acknowledgments

The authors are grateful to Jean-Jacques Vandewalle and Jean-Luc Giraud for corrections, suggestions and improvements.

References

[1] Abrial, J.R. *The B Book. Assigning Programs to meanings*, Cambridge University Press, 1996.

[2] Pierre Girard. *Which security policy for multiapplication smart Cards?*, in USENIX Workshop on Smart Card Technology, May 10–11, 1999, Chicago, Illinois, USA. To appear.

[3] Gary McGraw and Edward W. Felten. *Securing Java*, Wiley computer publishing, 1999, chapter 8.

[4] Jean-Louis Lanet and Antoine Requet. *Formal Proof of Smart Card Applets Correctness*, in Third Smart Card Research and Advanced Application Conference, September 14–16, 1998, Louvain-la-Neuve, Belgium, To appear.

[5] Visa. Visa Open Platform, Overview Document, September 3, 1998.

[6] Sun Microsystems. Java Card 2.1 Realtime Environment (JCRE) Specification, February 24, 1999.

[7] Sun Microsystems, Java Card 2.1Application Programming Interface, February 24, 1999.

[8] Sun Microsystems, Java Card 2.Virtual Machine Specification, March 1, 1999.