JCAT: An environment for attack and test on Java Card™*

Serge CHAUMETTE Iban HATCHONDO

Damien SAUVERON[†‡]

{serge.chaumette,iban.hatchondo,damien.sauveron}@labri.fr

LaBRI, Laboratoire Bordelais de Recherche en Informatique

UMR 5800 – Université Bordeaux 1

351 cours de la Libération, 33405 Talence CEDEX, FRANCE.

ABSTRACT

Till recently it was only possible to have one single application running on a smart card. Multi-applicative cards, and especially Java Cards, now makes it possible to have several applications sharing the same physical piece of plastic. This causes security problems and security assessment has to be provided. Therefore we have developed the JCAT environment for attack and test on Java Card. This environment is the topic of this paper.

KEYWORDS: *Smart Card, Java, Java Card, Emulator, Security, Test, Open Source.*

1. INTRODUCTION

The work presented in this paper is carried out at the LaBRI, *Laboratoire Bordelais de Recherche en Informatique*, in the *Distributed Systems and Objects team*. The main goal of our team is to provide the final users and the programmers with the software tools required to easily and securely develop and use applications based on distributed and mobile codes. Our focus is on Java [1, 2], or Java like technologies, because it offers mechanisms that go much further than those found in other languages especially regarding mobility and security. We also insist on the fact that a software tool must rely on solid foundations that establish that it is effectively doing what it pretends to. This is especially true in the context of one of the target architectures we are working on: smart cards and more precisely Java Cards. Smart cards are becoming more and more important in our lives. Therefore Sun microsystems proposed a new concept of multi-applicative smart cards based on the Java technology (figure 1). The main feature of this new standard is to make it possible to gather on a unique medium a set of services, called *applets*, that can cooperate with each other.

The aim of this paper is to present the implementation of the Java Card specifications [3, 4, 5, 6] that we have achieved within an emulation framework that we are setting up. This environment is being developed in the context of a cooperation with SERMA Technologies, an ITSEF (Information Technology Security Evaluation Facility) center specialized in smart cards security and in particular in Java Cards. Our joint project is called *Java Card Security*[1]. Within this project we have developed the JCAT environment so as to support software security assessment. This tool suite provides an easy way to simulate a large set of high level software attacks and to help in the validation of securized products implementations, *i.e.* products that are subject to the Common Criteria [7] or ITSEC (Information Technology Security Evaluation Criteria) evaluation. This environment includes a Java Card Virtual Machine Emulator referred to as the JCAT Emulator. This emulator is the topic of this paper.

2. RELATED WORK

The reasons why we have decided to develop our own software environment, including a JCVM emulator, are the following.

First, the available Java Card implementations are mainly commercial solutions, both when embedded – either in hardware [12] or in software [11] –, and when provided as simulators [10]. Due to the high commercial impact of the technology, these solutions are not open. There are only few non-commercial Java Card simulators [8] which are often incomplete (no transactions, no atomicity, etc.) and a bit slow.

*Java and all Java-based marks are trademarks or registered trademarks of Sun microsystems, Inc. in the United States and other countries. The authors are independent of Sun microsystems, Inc.

†LaBRI and ITSEF center of SERMA Technologies.

‡This work is partly achieved in the framework of a doctoral grant from the french ministry of research and SERMA Technologies.

[1]This work is supported and labelized *Société de l'information.* by the French Industry Department

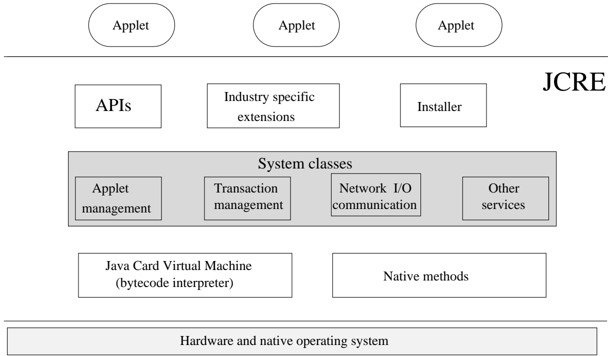

Figure 1: Overall Java Card architecture.

Second, the knowledge of the low-level details of the effective implementation of the Java Card specifications are kept confidential by the industry. We believe that gaining knowledge of these details is worth for the community.

Third, for security and certification reasons, both the virtual machine and parts of the resident applets need a third party validation. This clearly requires software tools that should be open enough to be adapted to different contexts.

These are the main reasons why we decided to develop a complete, open source, simulator tool. Additionaly, our platform is the only one that can simulate both hardware and software attacks and that remains as close as possible to an embedded implementation. Simulating hardware attacks consists for instance in modifying the values of some system objects, what is comparable to the effect that could have electromagnetic radiations [13].

3. JCAT EMULATOR

The open source Java Card Virtual Machine emulator that we have designed and implemented relies on a flexible and open architecture. This solves the problem of confidentiality that usually prevents the major actors of the domain from working together: we offer an open kernel that serves as a common platform (*cf.* figure 2) and the different actors can plug their own specific modules that are the effective added value of their commercial products. They just need to implement specific interfaces so as to allow our kernel to interact with their modules. There is no need for them to open their implementations in anyway.

In the process of implementing our JCVM, we have identified a number of points that are left obscure in the specifications. Because of that and because we want our emulator to support external plugins, we decided to design our software architecture as modular and as extensible as possible even though this may imply a penalty in term of efficiency. The rest of this paper is very technical and it is thus expected that the reader is familiar with the Java Card specifications [3, 4, 5, 6].

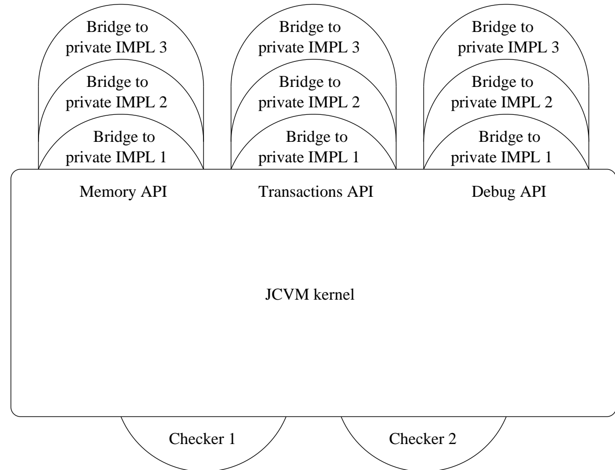

Figure 2: Overall architecture of our framework.

4. SUPPORTING EXTENSIBILITY

In this section, we present how we have been dealing with two of the fundamental parts of our JCVM implementation. We focus on these two points, because we believe that they play an important role in the extensibility that we claim is supported by our platform.

4.1. Factory design patterns

We use several factory patterns within our emulator. In this section we illustrate this point by means of the `CAP`[2] file reader. This reader has to be extensible. First, because the Java Card specifications will certainly be augmented with new components (the basic structure stored in `CAP` files) in the future, and second because a user may want to add his own custom components and be able to read them the way he wants to. Therefore we used a factory design pattern [14]: it provides an interface for creating families of component readers without specifying their concrete classes. A user can then create a new component reader or replace a pre-existent one by providing its implementation and registering it into the factory catalog. For example in Java, the following code:

```
public class FooComponentReader
             extends ComponentReader {

  static final int FOO_TAG = 1234;
  static {
```

[2]The standard binary file format of the Java Card platform.

```
    ComponentReaderFactory.addReader(
                             FOO_TAG,
                             new FooComponentReader());
  }

  public void parse( ... ) {
    ...
  }

  ...
}
```

makes so that each time the tag FOO_TAG is read in the CAP file the parse method of FooComponentReader is invoked.

We use the same mechanism for the implementation of the different kinds of arrays, objects, transactions and memories, with each time a specific interface.

4.2. Native interfaces

The Java Card specifications do not allow the programmer to call native functions at the user level. Nevertheless, implementing some of the Java Card APIs implies to use native code, *i.e.* code that cannot be written in pure Java Card. For instance the transactions are closely connected to the memory management and are then impossible to implement with the Java Card language. It requires to implement some additional features at the JCVM level. Since the Java Card specifications do not impose a specific way to deal with native interfaces, we chose to use the impdep1 private bytecode. Each time the interpreter reads this bytecode it reads the next 16 bits and invokes the dispatchInvokation method of the native interface factory with these bits serving as an index. The native interface factory is responsible for dispatching the method invocation according to that specified index. Since a handle to the interpreter is passed to the dispatching method, the programmer has complete control over the runtime. The only constraint is to pay attention to what is put in the current frame context (operand stack, local variables, etc).

5. SUPPORTING DIFFERENT SPECIFICATION INTERPRETATIONS

In this section we describe the choices that we had to make to cope with some unclear parts of the specifications regarding the persistence of objects and the heap[3] location. We present the problems and then four possible solutions that lead to completely different implementations, each one having its own advantages and drawbacks.

5.1. Objects and memory

In Java Card technology, the JCRE (Java Card Runtime Environment) and applets create objects to represent, store, and handle data. Applets are written using the Java programming language. Runnable applets on the card are instances of the Applet class. Objects in the Java Card platform obey the following Java basic rules:

- all the objects of the Java Card platform are instances of classes or array types, which have the same root class java.lang.Object;
- the fields in a new object and the components in a new array are set to their default values (0, null or false) unless they are initialized to some other value in the constructor.

Java Card technology supports both *persistent* and *transient* objects. However, the concepts of persistent and transient objects and the mechanisms to support them in the Java Card platform are not the same as in the standard Java platform. The persistent objects (objects whose fields have persistent values) are stored in persistent memory. The structure of objects is also persistent. The term "transient object" is a misnomer. It can be incorrectly interpreted to mean that the object itself is transient. However, only the values of the fields of the object are transient. There are two types of transient objects refered to as CLEAR_ON_RESET and CLEAR_ON_DESELECT objects. Each type of transient object is associated with an event that, when occuring, causes its fields to be cleared by the JCRE.

5.2. Problem of the contents of the heap

5.2.1. Persistent objects

The Java Card specifications are imprecise regarding the location of objects. It is written that the JCRE developer shall make an object persistent when the method Applet.register is called or when a reference to the object is stored in a field of any other persistent object or in the static field of a class. In addition, it is also specified that an object is created from the heap when a bytecode new, anewarray or newarray is executed or a method makeTransient*XXX*Array is invoked.

What is then the nature of objects between their creation and their assignment to the field of a persistent object? In the rest of this paper *pre-persistent space* will refer to the memory location where the objects are created after the execution of a bytecode new, anewarray or newarray. Note that the name of this space does not imply that the objects that it contains will necessarily become persistent.

The glossary of the JCRE specifications 2.1.1 and 2.2 defines the objects as being persistent by default. This definition goes against the recommendations of the specifications. What part of the specifications is it necessary

[3] Defined since Java Card specifications 2.2 as a common pool of free memory usable by a program.

to implement? The developer has to choose between two possible implementations (*cf.* figure 3): one having the concept of pre-persistent space; the other directly allocating the objects in the persistent space. What are the ad-

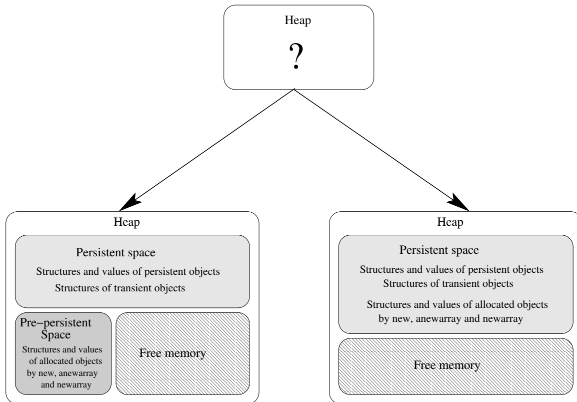

Figure 3: The problem of the contents of the heap.

vantages and the drawbacks of these two solutions? The solution with pre-persistent space offers several advantages:

- speed of the update operations since only the fields of the persistent objects are concerned by atomicity and transactions;
- an easy implementation of garbage collection in the pre-persistent space that consists in cleaning this space when the device is restarted;
- the possibility to use it to create the *Transient Environments* described in [15] that gives to the language a greater power of expression and solves some problems when objects are created within a transaction that is aborted later [16]. This feature will not be detailed here.

The other solution leads to a much simpler implementation since it just implies to allocate all the objects in persistent space.

5.2.2. Transient objects

The specifications are very precise concerning the location of transient objects. Informations on their structure are located in persistent memory whereas the values of their fields are located in transient space. The specifications forbid to locate transient space in persistent memory. It should thus be located in RAM.

Writing in the fields of a transient object does not cause any problem of efficiency because writing in RAM is much faster than writing in EEPROM. The properties of these objects (*i.e.* efficient access and security) directly result from the properties of the RAM.

The definition of persistent objects (see above, Section 5.2.1) also implies that a transient object whose reference would be stored in the field of a persistent object or of a class would become persistent. Thus, only transient objects referenced by local variables or from the operand stack could keep their transient nature. In the other cases these transient objects would become persistent objects and the values of their fields would migrate from transient space to persistent space. If we consider the solution that uses pre-persistent space, the objects of this space can reference a transient object without changing its nature since the specifications do not say anything about the pre-persistent space. We decided like the other developers that a field of a persistent object can reference a transient object still remaining transient.

5.3. Heap location

The second hole of the specifications relates to the location of the heap. Prior to release 2.2 the specifications did not give any definition of the concept of heap and still, in spite of the precise details that are now given, freedom is left to the developer of the JCRE to choose the extent and the location of the heap. Should it be located (*cf.* figure 4) only in EEPROM, or in both EEPROM and RAM? The only certainty about the heap is that it is partly in EEPROM since it must contain the persistent objects (*i.e.* structures and values) and the structures of the transient objects.

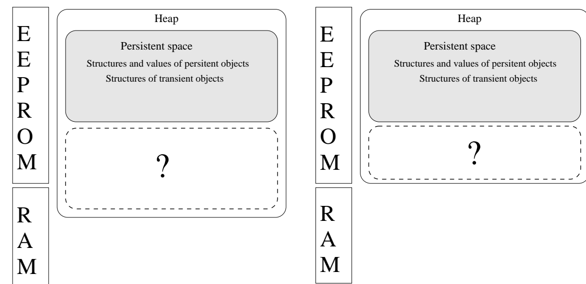

Figure 4: The heap location problem.

5.4. Global problem

These properties of the heap imply properties on the nature of the objects when they are created. Therefore it is necessary to study the global problem which results from the combination of the two problems explained above in

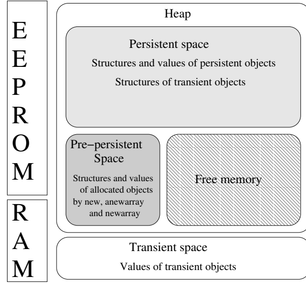

Figure 5: The global problem.

Section 5.2.1 and Section 5.3 (*cf.* figure 5).

For implementing the heap and its contents there are now 4 solutions each having its own advantages and drawbacks.

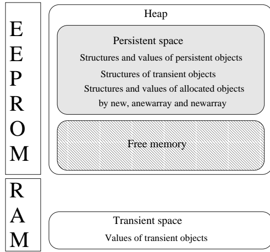

Figure 6: Solution n°1

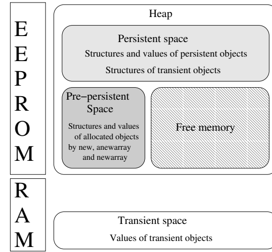

Figure 7: Solution n°2.

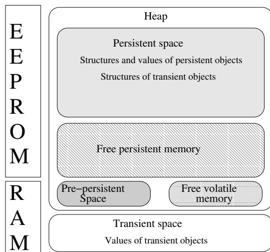

Figure 8: Solution n°3.

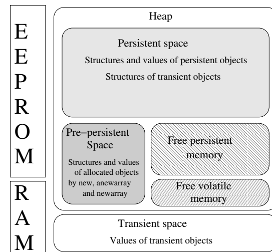

Figure 9: Solution n°4.

Solution n°1 (figure 6) is chosen by most Java Card implementations. Its main advantage is the easy management of the allocation of objects.

Solution n°2 (figure 7) is a bit more difficult to implement because it implies to handle the transfer of objects from pre-persistent space to persistent space. Its adcantage is that it is much more efficient because there is no impact of the atomicity and transaction mechanisms. The implementation of a partial garbage collector is also rather easy; each time the device is restarted it is enough to clear the pre-persistent space.

Solution n°3 (figure 8) is certainly the most efficient in terms of security and speed. The speed is increased thanks to the location of pre-persistent space in RAM. The drawback being the small size of the RAM. Security is increased by the possibility of creating objects in the RAM. With this solution transient arrays of objects realy make sens: they are in volatile memory and disappear between two sessions with the reader. This property seems in conformity with the essence of the Java Card specifications. A last advantage of this solution is that the RAM is garbage collected each time the card is powered off (*i.e.* the garbage collection mechanism simply consists in cleaning the allocation table of the RAM). The main problem is to properly move the objects from the pre-persistent space from RAM to the persistent space which is in EEPROM.

Solution n°4 (figure 9) is a good compromise between the speed and the size of the memory. The RAM part of the pre-persistent space can be used as the default location to create the objects and when there is no more free memory the objects can then be allocated in EEPROM. Its drawback is a lower execution speed and less security when pre-persistent space in RAM is full. This solution also has the same charateristics as solutions n°2 and 3.

Even though we adopted solution n°1, as most developers do, our extensible architecture makes it possible to plug modules that support other implementations. This can be rather useful within our context, *i.e.* test and validation of embedded applets.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented the way we have implement a JCVM to support both evolutions and refinements of the Java Card specifications. It also supports external plugins provided by third parties. We have now reached a point where we have an operational JCVM that includes the most sensible parts of the specifications, *i.e.* transactions, atomicity, *persistent* and *transient* objects.

In the mean term our goal is to distribute the code under an open solution (MIT, BSD, GPL or LGPL license) and to have our JCVM certified by Sun microsystems.

In the longer term we intend to promote our implementation as a common open kernel and to extend it with specific modules and tools, either from universities or from the industry. We are participating in a national effort supported by the CNRS (*Centre National de la Recherche Scientifique*) to share security experience and resources and this will be a good context to achieve the diffusion of our software suite.

7. REFERENCES

[1] James GOSLING, BillJOY and Guy STEELE.
The Java Language Specification.
Addison Wesley, 1996.

[2] Ken ARNOLD and James GOSLING.
The Java programming language.
Addison Wesley, 1996.

[3] Zhiqun CHEN.
Java Card™ Technology for Smart Cards.
Addison-Wesley – ISBN 0-201-70329-7.

[4] Sun microsystems.
Java Card™ 2.2 Application Programming Interface.
`http://java.sun.com/products/javacard/`

[5] Sun microsystems.
Java Card™ 2.2 Runtime Environment (JCRE) Specification.
`http://java.sun.com/products/javacard/`

[6] Sun microsystems.
Java Card™ 2.2 Virtual Machine Specification.
`http://java.sun.com/products/javacard/`

[7] Common Criteria.
`http://www.commoncriteria.org/`

[8] Carine COURBIS.
Simulation d'applications Java Card.
Rapport de stage du DEA d'Informatique de Lyon. 1998.
INRIA de Sophia-Antipolis.

[9] Xavier LEROY.
Bytecode verification on Java smart cards.
Software Pratice & Experience 2002; 32:319-341 (DOI: 10.1002/spe.438).

[10] IBM Zurich Research Laboratory.
JCOP Development Tools 2.2.
`http://www.zurich.ibm.com/csc/infosec/`

[11] Giesecke & Devrient.
Sm@rtCafé Java Card.
`http://www.gdm.de/`

[12] STMicroelectronics.
Instant Java for your Smartcard. 32 bits processing for Smartcards.

[13] Sergei SKOROBOGATOV and Ross ANDERSON.
Optical Fault Induction Attacks.

[14] Erich GAMMA, Richard HELM, Ralph JOHNSON, John VLISSIDES.
Design Patterns.
Addison Wesley – ISBN 0-201-63361-2.

[15] Marcus OESTREICHER.
Object lifetimes in Java Card.

[16] Marcus OESTREICHER.
Transactions in Java Card.