New security problems raised by open multiapplication smart cards
RR-1332-04

Serge Chaumette and Damien Sauveron*,**

LaBRI, Laboratoire Bordelais de Recherche en Informatique
UMR 5800 – Universite Bordeaux 1
351 cours de la Liberation, 33405 Talence CEDEX, FRANCE.
{serge.chaumette,damien.sauveron}@labri.fr, http://www.labri.fr/

Abstract. Till recently it was impossible to have more than one single application running on a smart card. Multiapplication cards, and especially Java Cards™ [1], now make it possible to have several applications sharing the same physical piece of plastic. This raises new security problems by creating additional ways to attack a card. These problems are the topic of this paper. The attacks will be described for multiapplication cards in general and illustrated by means of code samples for Java Cards.

KEYWORDS: *Smart Card, Java Card, Security, Attack.*

1 Introduction

The work presented in this paper is carried out at the LaBRI, *Laboratoire Bordelais de Recherche en Informatique*, LaBRI 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 claims that it is doing. This is especially true in the context of one of the target architectures we are working on: smart cards and more precisely Java Cards. Java Cards are a concept of multiapplication smart cards proposed by Sun microsystems based on the Java technology (*cf.* Fig. 2). The main feature of this standard makes 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 describe new threats that arise when dealing with open multiapplication smart cards. First we will present the common way to explore and attack classical smart cards and then we will describe what is an open multiapplication smart card. In section 4 we will show the general attacks against these new cards. Then in the context of the open multiapplication smart cards we will expose in section 5 some physical identification helper methods and we will investigate in section 6 how it is possible to use them in order to achieve some attacks on these cards. Finally we will present our experiments, the related work and we will conclude.

2 Exploring and attacking closed smart cards

Even though the ways to explore a closed card (*i.e.* a card that does not allow codes to be uploaded after it has been issued) before trying to attack it are reduced, there are at least two possibilities that remain:

* LaBRI and ITSEF of SERMA Technologies.

** This work is partly supported by a doctoral grant from the french ministry of research and SERMA Technologies.

[1] 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.

– Software approach. Since you cannot load code on a closed card you can only access its external interfaces. More precisely, the only external interface visible on a card is the communication layer. The communication architecture between a smart card and a reader connected to a host can be seen as a protocol stack, from the physical layer to the application layer (*cf.* Fig. 1). It is described in the ISO7816-3 [3] and ISO7816-4 [4] standards.

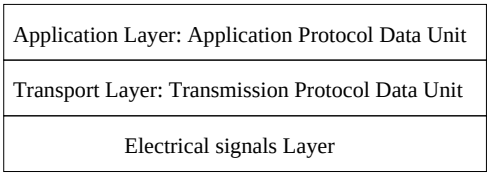

Fig. 1. Stack of communication protocols.

One of the possibilities to set up an attack at the level of the communication layer is to send all the possible APDU commands to the card in order to identify all the services that it makes available to the outside.
– Hardware approach. Based on the results obtained by Kocher, the main path of attack to explore a card without damaging it, is to observe side channels [5] such as the physical emanations from the chip or the time consumptions [6]. The methods that use power analysis (PA) [7,8] or electromagnetic analysis (EMA) [9,10] make it possible to identify patterns corresponding to operations achieved by the card. These attacks are carried out as a blind man, or a semi-blind man if the specifications of the card operations are known.
The non-invasive attacks cited above can be used to build the dictionary that we propose in section 6.1 but the smart cards are prone to many other invasive (e.g. microprobing, etc.) and non-invasive (e.g. glitch on the different pads of the card, etc.) attacks [11,12,13,14].

3 The open multiapplication smart cards

A multiapplication card is able to embed several applications. These applications do not run simultaneously because the smart card OS has a single thread. The two main standards of multiapplication cards are Java Card [15,16] and MULTOS [17]. But recently two new technologies have just joined them with the birth of Smartcard.NET [18,19] by Hive-Minded and of the MultiApplication BasicCard ZC6.5 [20] by ZeitControl. Windows for Smart Cards by Microsoft can also be cited even if it seems to be dead. Java Card was the first to appear but they share many concepts anyway. The architecture of a smart card implementing these technologies (*cf.* Fig. 2) consists of:

– an embedded Operating System that supports the loading and the execution of different applications. The OS is a runtime environment with a virtual machine (VM) that provides security features (e.g. a firewall between applications).
– the applications that are interpreted by the VM.

Until now no multiapplication technology was completely proven secure by formal methods and consequently it is not safe to run uncertified applications that may be provided by a hacker to attack the assets of the platform and those of the other applications. To prevent these problems, the smart card issuers began supporting standards such as GlobalPlatform [21] which specifies how to securely load, install and manage applications on a card. Indeed this standard is used to easily set up and use cryptographic mechanisms to authorize or not the loading of an application on the card.

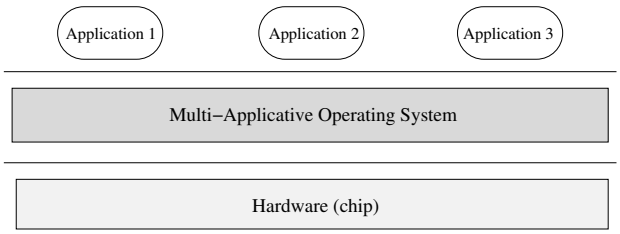

Fig. 2. Architecture of a multiapplication smart card.

For example the application can be digitally signed off-card by a trusted party (e.g. the card issuer). Once loaded the card can check the signature and accept or reject the application. But the major drawback of this solution is that it is based on a centralized model because of the trusted third party required to sign the application; it thus decreases flexibility. Therefore, on-card verifiers have been proposed [22,23,24,25,26,27,28] although Java Card designers first thought it was impossible to embed a verifier due to the resource constraints of the smart cards. Among the different solutions proposed, three (*i.e.* the defensive VM, the verifier based on code transformation and the stand-alone verifier) allow to withdraw the signature step of the application without jeopardizing the card security and thus allow everybody to freely load his/her own application – *i.e.* anyone can still load a rogue application but it will eventually be rejected by the verifier or blocked by the defensive VM. Note that the verifier based on code transformation [24,25] requires an off-card program to normalize the application whereas the two others (*i.e.* the defensive VM [26] and the stand-alone verifier [28]) are stand-alone. We only consider as ***open multiapplication cards*** the multiapplication cards based on one of these two stand-alone solutions. Both are equivalent [27] but the defensive VM dynamically checks each executed bytecode whereas the stand-alone verifier statically checks once at load time and it is associated with an offensive VM that does not check the bytecode at execution time.

4 Internal attacks

Obviously one of the main problems of the open multiapplication cards is the possibility to create internal attacks since it is possible to load a malicious application. Such an application may:

- identify services present on the card and then try to deduce the possible behavior of an official[2] applet (since it cannot use service unavailable on the card);
- collect information on the card to elaborate hardware and software attacks;
- attack the VM and the firewall.

4.1 Identification of services

For example the listing 1.1 shows an application that uses all the cryptographic services defined in the Java Card specifications [16] in order to work out if they are present on the card or not.

4.2 Collection of information

Listing 1.2 shows how to detect all the applications available on a Java Card and if they propose some services how to get the `Shareable` interface that is necessary to use them. This code assumes that the targeted applets perform an authentication based only on the `parameter` value of the `getAppletShareableInterfaceObject(AID serverAID, byte parameter)`. Note that this

Listing 1.1. Discovering cryptographic services

```
public class TestCryptoAlgo extends Applet {
 ...
 public void process(APDU apdu) throws ISOException {
     byte[] buffer = apdu.getBuffer();
     // Try to get an instance of all the algorithms available in the specifications.
     switch ( buffer[ISO7816.OFFSET_INS] ) {
         case 0x02: isValidAlgorithm(Cipher.ALG_DES_CBC_NOPAD);
         case 0x04: isValidAlgorithm(Cipher.ALG_DES_ECB_NOPAD);
         case 0x06: isValidAlgorithm(Cipher.ALG_RSA_PKCS1);
         case ...
     }
 }

 public void isValidAlgorithm(byte bAlgo) throws ISOException {
     try {
         // Try to create an instance of algorithm of type bAlgo.
         Cipher cipherInst = Cipher.getInstance(bAlgo, true);
     } catch (CryptoException e) {
         // If the type is not implemented, Status Word = 0x6FFF is sent on the I/O
         // else the successful SW = 0x9000 is sent by the card itself at the end.
         if (e.getReason() == CryptoException.NO_SUCH_ALGORITHM)
             ISOException.throwIt((short) 0x6FFF);
     }
 }
 ...
}
```

strong assumption is not realistic. But if the targeted applets require a more realistic authentication based on the AID of the client applet (*i.e.* `ScannerApplet` here) it is still possible on some Java Card implementations to get an authorized AID. This attacks known as AID Impersonation [29] is made possible by some naming contradictions between the Java Card and the GlobalPlatform standards. Indeed the GlobalPlatform allows to install an applet with a chosen AID whereas Java Card restricts. Thus depending of the implementation choices it may be possible to perform this attack or not.

We may also add a glitch (e.g. see the methods described in the section 5) just before accessing the data of the targeted applets so as to synchronize a physical attack (e.g. fault injection [12,13,14]) to bypass the checks of the firewall thank to the disturbance of the hardware component.

4.3 Attacks against the VM and the firewall

Obviously there exist many attacks directed against the VM and the firewall but we will not detail them here because other works have already explained them. For example two attacks against the firewall mechanism (AID impersonation and illegal reference casting that provides access to all interface methods of a class) are described in [29]. There also exist attacks coming from problems in the specifications such as those presented in [30]. The attacks against the VM also exist and may be due to a bad specification or a bad implementation. For example the program of listing 1.3 that forges a pointer would be rejected by an open multiapplication card: at loading time for those based on the stand-alone verifier; at execution time for those based on the defensive VM. But an attacker should always try to load such malicious codes to test if there are implementation problems or not.

The piece of code shown listing 1.3 (provided in an intermediate language) traverses all the allocation table and tries to read one byte of the object by considering it as an array. It is based on the strong assumption that the stack is not typed (e.g. it is the case of an offensive VM or a

[2] an applet installed by some trustworthy organization, e.g. a banking applet.

Listing 1.2. Scan all the applications and try to get a shareable interface

```
public class ScannerApplet extends Applet {
 ...
 public void process(APDU apdu) throws ISOException {
     byte[] tab = new byte[16];
     ...
     // Put here the code to test all the possible values for tab
     ...
     for(byte i=5; i <= 16; i++) {
         AID aid = JCSystem.lookup(tab, (short) 0, i);
         if (aid != null) tryToGetServices(aid);
     }
 }

 void tryToGetServices(AID aid) {
     for(byte i=0; i <= 255; i++) {
         Shareable sio = JCSystem.getAppletShareableInterfaceObject(aid, i);
         if (sio != null) callMethodOfSIO(sio);
     }
 }

 void callMethodOfSIO(Shareable sio) {
     // Put here some code to call methods of this interface
 }
 ...
}
```

bad implemented defensive VM) in order to do arithmetic operations on the references. Note that if the reference is represented as a memory address (*i.e.* the equivalent of a pointer) in the VM implementation this code allows to traverse all the memory.

Listing 1.3. Scan all the references

```
  sconst_0 // From reference 0
:loop
  dup
  sconst_0
  baload    // Try to get one byte and push it on the stack (but we can imagine other operations)
  invoke do_something // Method that removes the byte read and returns nothing
  sconst_1
  sadd     // Increment the reference by steps of 1
  goto loop

  void do_something(byte value@address) {
      // Take a byte operand on the stack (i.e. the byte read) and save it somewhere
  }
```

5 Physical identification helper methods

Once the services have been identified, an interesting possibility consists in observing their signatures. The main physical signals that can be observed are issued from the side channels such as the power consumption, the electromagnetic emissions or the execution time consumption. For instance we can use a cryptographic service and determine the characteristics of the physical signals emitted during the execution of this service (e.g. duration in time, location of the best electromagnetic emissions, power consumption, etc.). This characterization can be targeted to the use of a whole service or to an elementary operation e.g. the interpretation of a single bytecode or a sequence of bytecodes.

There are two ways to determine the signatures. The first which is also the simplest is achieved by using glitches on the I/O channel of the card. The second consists in causing the pattern to observe to be repeated. In the following sections we present these two approaches and discuss their advantages and drawbacks. We eventually propose a hybrid method that overcomes the problems and take the best of these two methods.

5.1 The glitches based method

This method consists in surrounding the pattern to observe with events that are visible for an observer from outside the smart card. Fig. 3 shows the trace of a normal execution of an applet that includes the pattern to observe. Obviously it is difficult to isolate this pattern from all those composing the trace.

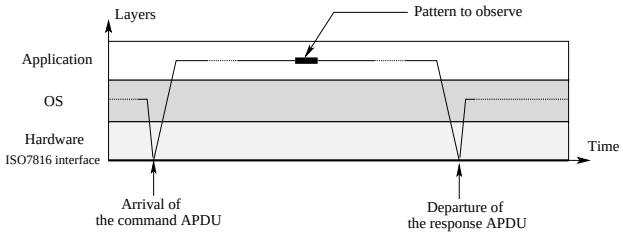

Fig. 3. Trace of a normal execution in the layers.

The only event visible to an external observer that the card can produce is the emission of bytes on the communication channel. If the execution is glitched using this event, it is easier to find the pattern in the trace (*cf.* Fig. 4).

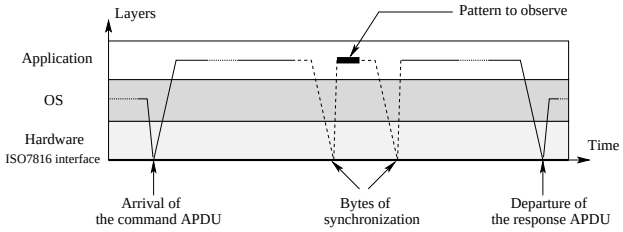

Fig. 4. Trace of a glitched execution in the layers.

The code inserted to trigger the glitches causes an overhead and thus the pattern to observe does not appear at the same instant in the first trace and in the second trace.

Moreover to easily locate the pattern, another advantage of this approach is that the trace to save is shorter and thus it is possible to get a better sampling of the signal, still producing the same amount of data.

Using the time extension request

The ISO 7816-3 standard defines a special mechanism that allows a smart card to request a time extension to the reader *i.e.* let it know that it should wait a bit more before receiving a result (to prevent a timeout). This mechanism depends on the transport protocol (e.g. T=0 or T=1) and allows the reader to know that the smart card is not mute and still works. For instance for the

T=0 protocol this mechanism consists in sending the NULL procedure byte (*i.e.* the byte 0x60) to the reader. In Java Card the method invoked to use this mechanism is `apdu.waitExtension()`. Listing 1.4 shows an example using the `waitExtension()` glitches to surround an encryption.

Listing 1.4. Encryption surrounded by `waitExtension()` glitches

```
public void process(APDU apdu) {
...
    // Request a time extension that puts data on the I/O (glitch 1).
    apdu.waitExtension();
    cipherLength = cipher.doFinal(clearData, (short) 0,
                                  clearData.length,
                                  cipherData, (short) 0);
    // Request again a time extension (glitch 2).
    apdu.waitExtension();
...
}
```

In recent implementations of many manufacturers the `waitExtension()` method is deactivated at user level. The necessary time extension requests are automatically managed by the platform.

Using the classic communication method

The second solution uses the classic communication method. It is possible to use it to generate an event used as a glitch by sending the card response in several pieces. Normally the communication model from the smart card to the host consists in sending all the data of the response at once. We propose to send a pseudo response in two times, the first piece being sent before the pattern to observe and the second piece being sent after. Listing 1.5 shows an example using response based glitches to surround an encryption.

Listing 1.5. Encryption surrounded by data glitches

```
public void process(APDU apdu) {
   byte[] buffer = apdu.getBuffer();
...
   buffer[0] = (byte)0xFF;
   apdu.setOutgoing();
   apdu.setOutgoingLength((short) 2);
   // Send the synchro ... means the beginning of the process (glitch 1)
   apdu.sendBytes((short) 0, (short) 1);
   cipherLength = cipher.doFinal(clearData, (short) 0,
                                 (short) clearData.length,
                                  cipherData, (short) 0);
   // Send the synchro ... means the end of the process (glitch 2)
    apdu.sendBytes((short) 0, (short) 1);
...
}
```

This method was successfully tested on many smart cards and it should work on any card supporting ISO7816-3–4. People often think that it does not work because they reason at APDU level and they think that the emissions (e.g. `sendBytes(...)`) are batched. But as we have shown in Fig. 1 the APDUs (*i.e.* ISO7816-4) are achieved by a sequence of TPDU exchanges (*i.e.* ISO7816-3). Besides this can be confused due to `APDU` class in Java Card is not really an APDU representation and is only an APDU emulation (e.g. with T=0 cards the mandatory call to `getBuffer()` sends a TPDU response to the reader in order to inform it can send the TPDU with the data field of the APDU to the card).

Possible improvements
Using the above method, it is possible to target a big pattern but sometimes we wish to enhance the precision of the target. Let us consider an example. The sequence of bytecodes of listing 1.6 corresponds to the result of the compilation and conversion of the three last lines of listing 1.5. It makes clear that it is possible by working at the bytecode level to improve the precision of the

Listing 1.6. Generated bytecodes for the encryption invocation surrounded by glitches

```
aload_1
sconst_0
sconst_1
invokevirtual   0x0 0x6 // glitch 1 (real glitch)
getfield_a_this 0x0
getfield_a_this 0x1
sconst_0
getfield_a_this 0x1
arraylength
getfield_a_this 0x2
sconst_0
invokevirtual   0x0 0xa // pattern to observe (real encryption)
sstore_2
aload_1
sconst_0
sconst_1
invokevirtual   0x0 0x6 // glitch 2 (real glitch)
```

observation. The sequence of bytecodes presented in listing 1.6 can be modified as shown listing 1.7 and still have the same global behavior but with a better accuracy regarding the surrounding of the encryption by the glitches. This improvement is also possible when using the `waitExtension()`

Listing 1.7. Improved pattern surrounding

```
aload_1
sconst_0
sconst_1
getfield_a_this 0x0
getfield_a_this 0x1
sconst_0
getfield_a_this 0x1
arraylength
getfield_a_this 0x2
sconst_0
aload_1
sconst_0
sconst_1
invokevirtual   0x0 0x6 // glitch 1 (real glitch)
invokevirtual   0x0 0xa // pattern to observe (real encryption)
sstore_2
invokevirtual   0x0 0x6 // glitch 2 (real glitch)
```

method when the platform makes it available.

One of the problems when working at the bytecode level is that a minor modification in the CAP[3] file implies to change many other parts. For instance changes in the bytecodes array of the Method component of the CAP file often implies to modify the information related to the maximum size of the stack and the maximum number of local variables of the frame of the modified method. The ReferenceLocation component and its size should also often be modified. These modifications

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

furthermore imply modifications of the Directory component. To simplify these operation, we have developed a tool provided with the JCatools suite [31,32] to modify the Method component of a CAP file. The JCatools suite was developed during the Java Card Security project between the LaBRI and the ITSEF of SERMA Technologies. This suite of tools mainly consists in a Java Card Emulator and additional facilities. More information is available in the Appendix.

Preventing these attacks
Some solutions to prevent theses attacks can be considered. For instance it is possible:

- to introduce a random delay on the I/Os to block these attacks. But it is a bad solution because with many tries it should still be possible to cope with this randomness.
- to store all the data in a buffer until the end of the execution before sending the buffer on the I/O. This would most likely require a second buffer much bigger than what remains acceptable on a card. Thus it is not a good solution.
- to store the data in a buffer until their size reaches a fixed threshold before to send them on the I/O. In this case the hacker may instrument her applet to generate the exact amount of data needed to obtain a response on the I/O and to work out the beginning of the pattern to observe.
- to store the data as previously in a buffer but with a random threshold. Despite of this, a hacker could still bypass this security using a probabilistic technique.

For all these reasons we believe that mixing a random delay and a random threshold storage of the data in a buffer seems to be a solution that may be worth considering.

Note that if the manufacturers find a better countermeasure, it will still be possible to locate the pattern by surrounding it with something that induces high power consumption (e.g. a cryptographic mechanism, etc.) to produce an event visible from the outside.

5.2 Repeated pattern based method

Since the manufacturers can add the countermeasures proposed above, we consider another method that still makes it possible to capture the signature of the operations achieved on the card. This solution consists in repeating many times an identical sequence so that it is easier to locate the area to study and then to identify the elementary pattern.

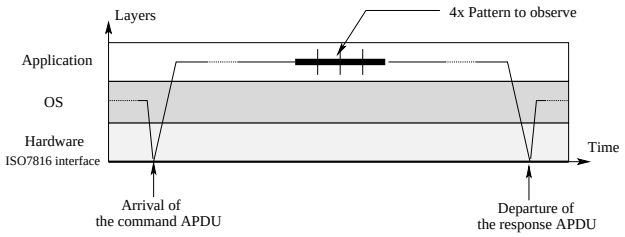

Fig. 5. Trace of a multiple patterns execution in the layers.

For instance inserting the code shown listing 1.8 in a Java Card applet makes it easier to find the `dup` elementary pattern. The physical emanations corresponding to the execution of the above program, will contain four times the pattern corresponding to the `dup` operation.

Listing 1.8. Patterns method

```
...
// assume that there is at least one element on the stack
dup      // pattern to observe
dup      // pattern to observe
dup      // pattern to observe
dup      // pattern to observe
...
```

Solutions against this attack

The first solution is to put constraints on the on-card verifier to check for instance that the loaded code does not contain a sequence of two `dup` bytecodes, *i.e.* `dup dup`. Indeed the converter will produce a `dup2` and not this sequence. But it is not possible to do such verifications for all the possible bytecodes (e.g. it is not possible on `sxor`) due to the fact that it would require that the verifier is able to understand the semantic of the bytecodes sequence at application level (*i.e.* what the application wants to do) and clearly this is impossible. The verifier can only understand the semantic of the bytecodes sequence at VM level (*i.e.* if the bytecodes chain is valid). A defensive VM can also do the same checks but the problem is same.

A better solution would be to use hardware countermeasures (e.g. a random internal clock or asynchronous processors [33,34], etc.) to disturb the physical emanations.

5.3 The hybrid method using both glitches and patterns

This method avoids the overhead caused by the transition between the application, the OS and the chip due to the insertion of glitches. It allows to easily locate the sequence of identical patterns in the trace and it is then easy to find the elementary patterns in the located sequence.

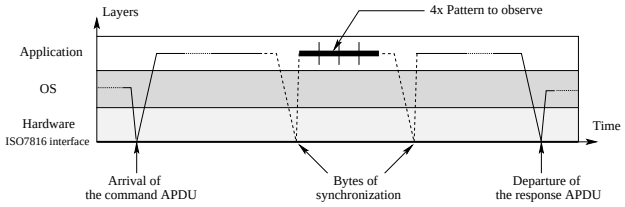

Fig. 6. Trace of a glitched multiple patterns execution in the layers.

The listing 1.9 shows an example of this hybrid method to observe `sxor`.

6 Applications of the physical identification helper methods

Based on the physical identification helper methods explained above we describes in this section two possible applications:

- the matching attacks allowing to reverse-engineer an official applet;
- the physical attacks allowing to determine quickly the feasibility of an attack against an official applet.

Listing 1.9. Sample code used by the hybrid method

```
...
aload_1
sconst_0
sconst_m1
sconst_m1
sconst_m1
sconst_m1
sconst_1
aload_1
sconst_0
sconst_1
invokevirtual   0x0 0xc // glitch 1
sxor    // pattern to observe
sxor    // pattern to observe
sxor    // pattern to observe
sxor    // pattern to observe
invokevirtual   0x0 0xc // glitch 2
...
```

6.1 Matching attacks

This attack consists in matching the signal identified for the pattern in the malicious applet with the signal of an official applet to reverse-engineer it. The first step required to set up this attack is to build a dictionary matching pattern signals with bytecode instructions. The second step is to identify patterns of the dictionary in the trace of the official applet to know the operations effectively performed. It is a difficult task but some research projects are in progress to automatically recognize patterns in signal [35]. This attack jeopardizes the confidentiality of the code of the official applet (*i.e.* the code is known) but it does not indicate that an attack will be possible against the execution of this official applet. In the Common Criteria[4] [36] evaluation method, the code of an official applet is often an asset to protect and a such attack is considered a problem. Moreover the knowledge of the code allows to use analysis tools to find breaches of security. Finally it is also possible to couple the knowledge of the code with the attacks described below.

6.2 Physical attacks

Thanks to the presented identification methods, a hacker can possibly set up physical attacks [12,13,14] on the pattern identified in her own applet. Indeed she can put in the best states to easily and quickly attack a card implementation (e.g. try to bypass the access conditions or perturb a cryptographic algorithm) avoiding to perform many useless tests and saving a lot of time. If her attacks succeed then she can attack the official applet or the platform. For instance if she knows an attack against cryptographic algorithm that the official applet uses, she can try to set up the attack against this algorithm by calling it from her own applet and surrounding it by glitches to quickly test its implementation.

7 Experiments

A few years ago, Sebastien Garcia[5] carried out experiments based on identical techniques to achieve matching attacks for the ITSEF of SERMA Technologies. He worked at the assembler level and

[4] The Common Criteria for Information Technology Security Evaluation, abbreviated "CC", defines a language for defining and evaluating information technology security systems and products. The framework provided by the CC allows a manufacturer to define set of security functional and assurance requirements for his product. The CC also provides evaluation laboratories with procedures for evaluating the products or systems against the specified requirements.

[5] IXL laboratory of the University of Bordeaux

he obtained interesting results but with the product that he used it was very hard to get a good dictionary because for some instructions it was even difficult to differentiate the signals. We are in a different situation. A bytecode is in fact a sequence of assembler instructions that represents a bigger pattern than a single assembler instruction, we believe that it may be possible to effectively build such a dictionary of Java Card bytecodes. Furthermore, since each bytecode corresponds to a really different sequence of assembler instructions, it is most likely that their signature will also be significantly different.

We have carried out some experiments on Java Cards at the ITSEF of SERMA Technologies using the physical identification helper methods described in this paper in order to develop a methodology to reverse engineer official applications. These were achieved on Java Cards of the GemXpresso Pro family already certified at level EAL5+ of the Common Criteria. It was certainly a bad idea to begin with these products because we were only given two full days to make our experiments and no funding. Moreover we wanted to work on the most interesting – but also the most difficult – tests: the physical identification of bytecodes (we had already successfully tested the attacks described in the section 6.2 during real evaluations). We have failed to establish a simple dictionary using the electromagnetic emanations. This is mainly due to the many expensive techniques required by the recent chips to really perform a systematic and complete research (removing the shield against the electromagnetic emanations, required time to get good and reproducible setups, etc.).

But obviously some attacks presented in this paper or some improved versions are used by the ITSEF of SERMA Technologies to test and attack Java Card platforms, applets and their services for instance throughout a Common Criteria evaluation.

8 Related work

Some similar projects on multiapplication smart card [37,38,39] were achieved in 1999 by the Gemplus teams. They addressed many issues related to the loading, the virtual machine, the object sharing, the information flow between the applets, but they did not identify the attacks that we have described in section 6 thanks to the physical identification methods. More recently other work [40,41] related to the multiapplication smart cards did not either address the problems we raised in this paper. The first enables a smart card issuer to verify that a new applet securely interacts with already downloaded applets [40] and the second presents a generic security model for operating systems of multiapplication smart cards that formalizes the main security aspects of secrecy, integrity, secure communication between applications and secure downloading of new applications [41].

We have not found any related work based on our physical identification methods to characterize a platform.

9 Conclusion and future work

This paper describes what open multiapplication smart cards are and it surveys the problems that they raise. The majority of the problems and attacks presented in this paper (*i.e.* sections 5 and 6) were unpublished till now and we hope that sharing our experience with others interested in the area will help to secure the future open smart cards. Even if the problems raised and the countermeasures proposed may sometimes seem obvious, we have already use them successfully during a real product evaluation to quickly set up attacks. This paper opens some new aspect of research. Particularly, in the near future, we intend to go on with our experiments so as to get a bytecode dictionary.

Appendix: Methodology to modify a CAP file

In this appendix we show in details the method used to set up trace based identification of operations by working at the bytecode level.

- First, write a code that uses the pattern to observe or a dummy Java code that will be replaced by the pattern to observe. In this last case the dummy sequence should generate a large enough sequence of bytecodes so that it can be replaced by the pattern to observe. For instance to observe the bytecode `sxor`, write the Java code of listing 1.10.

Listing 1.10. Dummy code

```
...
   apdu.setOutgoing();
   apdu.setOutgoingLength((short) 2);

   apdu.sendBytes((short) 0, (short) 1); // glitch 1
   // Here introduce the interesting pattern to observe
   short s = (short) 0;    // Dummy
   s ^= (short) 1;         // Dummy
   apdu.sendBytes((short) 0, (short) 1); // glitch 2
...
```

- Compile and convert it to produce a CAP formated file. Then use the *parseComponent* tool of the JCatools suite to get the methods of the CAP file in a human readable format. The interesting part including the dummy code of the file obtained by *parseComponent* is detailed listing 1.11.

Listing 1.11. Bytecodes generated from the dummy code

```
...
aload_1
sconst_0
sconst_1
invokevirtual   0x0 0xc // glitch 1
sconst_0                //
sstore_3                // The
sload_3                 // generated
sconst_1                // dummy
sxor                    // bytecodes
sstore_3                //
aload_1
sconst_0
sconst_1
invokevirtual   0x0 0xc // glitch 2
...
```

- Modify the generated bytecode sequence so as to improve the glitches position or to replace the dummy sequence by the proper bytecodes. Listing 1.12 shows how we update the listing 1.11 to set the `sxor` pattern surrounded by glitches with the best accuracy.
- Modify the maximum size of the stack and the maximum number of local variables.
- Use the *methodRewriter* tool available in the JCatools suite to rewrite the Method component of the CAP file. It also modifies the ReferenceLocation component and if needed the Directory component.

Listing 1.12. Pattern to observe surrounded by glitches

```
...
nop             // The nop bytecode do nothing
nop             // and we use them to reach the same
nop             // size for the bytecodes array as
nop             // previously to simplify the modification process.
aload_1
sconst_0
sconst_0        // TIP: the result of the sxor
sconst_1        // will be the third argument
aload_1
sconst_0
sconst_1
invokevirtual   0x0 0xc // glitch 1
sxor            // pattern to observe
invokevirtual   0x0 0xc // glitch 2
...
```

The resulting CAP file is still valid for the verifier and the defensive VM and can be uploaded to a Java Card.

Listing 1.12 shows that in some cases it is possible to isolate a single bytecode. It uses a tip to avoid the addition of a bytecode (e.g. `pop`, `sstore`, etc.) just after `sxor` that would remove the `short` value resulting of the `sxor` operation from the stack.

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. ISO/CEI.
   Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part 3: Electronic signals and transmission protocols.
4. ISO/CEI.
   Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part 4: Interindustry comands for interchange.
5. James Alexander Muir.
   Techniques of Side Channel Cryptanalysis.
   Master of Mathematics in Combinatorics and Optimization.
   University of Waterloo, Ontario, Canada, 2001.
6. P. Kocher.
   Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems.
   CRYPTO'96, vol. 1109 of Lecture Notes in Computer Science, pp. 104-113.
   Springer-Verlag 1996.
7. P. Kocher, J. Jaffe and B. Jun.
   Differential Power Analysis.
   CRYPTO'99, vol. 1666 of Lecture Notes in Computer Science, pp. 388-397.
   Springer-Verlag 1999.
8. Jean-Sébastien Coron, Paul Kocher, and David Naccache.
   Statistics and Secret Leakage.
   Proceedings of Financial Cryptography (FC2000), vol. 1962 of Lecture Notes in Computer Science, pp. 157-173.
   Springer-Verlag, 2001.
9. Jean-Jacques Quisquater and David Samyde.
   ElectroMagnetic Analysis (EMA): Measures and Countermeasures for Smart Cards.
   E-smart 2001, vol. 2140 of Lecture Notes in Computer Science, pp. 200-210.
   Springer-Verlag 2001.

10. Karine Gandol, Christophe Mourtel, and Francis Olivier.
Electromagnetic Analysis: Concrete Results.
Proceedings of CHES'2001, LNCS 2162, pp. 251-261, 2001.
11. Oliver Kömmerling and Markus G. Kuhn.
Design Principles for Tamper-Resistant Smartcard Processors.
Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99), pp. 9-20.
Chicago, Illinois, USA, May 10-11, 1999.
12. Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall and Claire Whelan.
The Sorcerer's Apprentice Guide to Fault Attacks.
Proceedings of Workshop on Fault Detection and Tolerance in Cryptography.
Italy, 2004.
13. Christophe Giraud and Hugues Thiebeauld.
A survey on fault attacks.
Proceedings of CARDIS'04, Smart Card Research and Advanced Applications VI ,Kluwer academic publisher, pp. 159-176.
Toulouse, France, August 24-26, 2004.
14. Sergei Skorobogatov and Ross Anderson.
Optical Fault Induction Attacks.
Proceedings of Workshop on Cryptographic Hardware and Embedded Systmes (CHES 2002).
San Francisco Bay (Redwood City), USA, August 13-15, 2002.
15. Zhiqun Chen.
Java Card™ Technology for Smart Cards.
Addison-Wesley – ISBN 0-201-70329-7.
16. Sun microsystems.
Java Card™ 2.2 Specifications.
`http://java.sun.com/products/javacard/`
17. Multos.
The MULTOS™ Specification.
`http://www.multos.com/`
18. Smartcard.NET
`http://www.hiveminded.com/`
19. *.NET Brings Web Services to Smart cards.*
SmartCards Trends, vol. 1, p. 12, April-May 2004, ISSN 1763-3494.
`http://www.smartcardstrends.com/det_atc.php?idu=779`
20. BasicCard.
`http://www.basiccard.com/`
21. Global Platform.
Global Platform Card Specification.
`http://www.globalplatform.org/`
22. E. Rose and K. Rose.
Lightweight bytecode verification.
In Workshop on Fundamental Underpinnings of Java, OOPSLA '98 Workshop.
Vancouver, Canada, October 1998.
23. L. Casset, L. Burdy, and A. Requet.
Formal Development of an embedded verifier for Java Card Byte Code.
In the IEEE International Conference on Dependable Systems & Networks.
Washington, D.C., USA, June 2002.
24. Xavier Leroy.
On-Card Bytecode Verification for Java Card.
Proceedings of the International Conference on Research in Smart Cards, E-Smart 2001, pp. 150-164.
Springer-Verlag – ISBN 3-540-42610-8.
25. Xavier Leroy.
Bytecode verification on Java smart cards.
Software-Practice & Experience volume 32, pp. 319-340, 2002.
26. R. M. Cohen.
Defensive Java Virtual Machine Version 0.5 alpha., 1997
27. G. Barthe, G. Dufay, L. Jakubiec and S. Melo de Sousa.
A Formal Correspondence between Offensive and Defensive JavaCard Virtual Machines.
Proceedings of VMCAI'02, LNCS 2294, pp. 32-45.
Venice, Italy, 2002.

28. Damien Deville and Gilles Grimaud.
Building an "impossible" verifier on a Java Card.
Proceedings of the 2nd USENIX Workshop on Industrial Experiences with Systems Software.
Boston, USA, 2002.
29. Michael Montgomery and Ksheerabdhi Krishna.
Secure Object Sharing in Java Card.
Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99).
Chicago, Illinois, USA, May 10-11, 1999.
30. Gustavo Betarte, Eduardo Giménez, Boutheina Chetali and Claire Loiseaux.
FORMAVIE: Formal Modelling and Verification of Java Card 2.1.1 Security Architecture.
Proceedings of E-Smart 2002, pp. 215-229.
Nice, France, September 19-20, 2002.
31. Serge Chaumette, Iban Hatchondo and Damien Sauveron.
JCAT: An environment for attack and test on Java Card.
Proceedings of CCCT'03 and 9th ISAS'03, vol.1 pp. 270-275.
Orlando, FL, USA.
32. The JCatools website.
`http://sourceforge.net/projects/jcatools/`
33. Simon Moore, Ross Anderson, Paul Cunningham, Robert Mullins, George Taylor.
Improving Smart Card Security using Self-timed Circuits.
Proceedings of ASYNC'02, pp. 211-218, 2002.
34. Jacques J.A. Fournier, Simon Moore, Huiyun Li, Robert Mullins and George Taylor.
Security Evaluation of Asynchronous Circuits.
Proceedings of CHES'2003, LNCS 2779, pp. 137-151, 2003.
35. Jean-Jacques Quisquater and David Samyde.
Automatic Code Recognition for smart cards using a Kohonen neural network.
Proceedings of the 5th Smart Card Research and Advanced Application Conference (CARDIS'02)
36. Common Criteria.
`http://www.commoncriteria.org/`
37. Pierre Girard and Jean-Louis Lanet.
Java Card or How to Cope with the New Security Issues Raised by Open Cards?.
Proceedings of Gemplus Developer Conference.
Paris, France, June 21-22, 1999.
38. Pierre Girard and Jean-Louis Lanet.
New Security Issues raised by Open Cards.
In Information Security Technical Report, Vol4, N2, pp. 19-27, 1999.
39. Pierre Girard.
Which security policy for multiapplication smart cards?
Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99), pp. 21-28.
Chicago, Illinois, USA, May 10-11, 1999.
40. Pierre Bieber, J. Cazin, Pierre Girard, Jean Louis Lanet, Virginie Wiels and Guy Zanon.
Checking Secure Interactions of Smart Card: Applets Extended Version.
Journal of Computer Security, vol. 10 , Issue 4 (December 2002).
Special issue on ESORICS 2000, pp. 369-398, 2002.
41. Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul Karger, Vernon Austel, David Toll.
Verification of a Formal Security Model for Multiapplicative Smart Cards.
Proceedings of ESORICS 2000, LNCS 1895, pp. 17-36, 2000.