An Efficient and Simple Way to Test the Security of Java Cards$^{\text{TM}\star}$

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.

Abstract. Till recently it was impossible to have more than one single application running on a smart card. Multiapplication cards, and especially Java Cards, now make it possible to have several applications sharing the same physical piece of plastic. Today, these cards accept to load code only after an authentication. But in the future, the cards will be open an everybody should be authorized to upload an application. This raises new security problems by creating additional ways to attack Java Cards. These problems and the method to test them are the topic of this paper. The attacks will be illustrated with code samples. The method presented here can be applied right now by authorised people (e.g. Information Technology Evaluation Facility – ITSEF) to test the security of Java Cards since they have the authentication keys and tomorrow a hacker may also be able to use this method to attack cards without needing the keys.

1 Introduction

Java Cards [1,2] are multiapplication smart cards proposed by Sun microsystems based on the Java technology. 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.

Even if it is not currently possible to freely load an applet on the actual Java Cards without being authenticated, the next generation should allow it. Although these cards will embed a verifier to protect them against malicious code loading, they raise new security problems.

The aim of this paper is to describe a method to test the security of the actual Java Cards and to present the new threats that arise when dealing with open Java Cards. First we will present the common way to explore and attack classical Java Cards and then we will precisely explain what is an open Java Card. In section 4 we will show the well-known attacks against these new cards. Then in the context of the open Java Cards we

$\star$ 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.

$\star\star$ LaBRI (Talence, FRANCE) and LMSI (Limoges, FRANCE).

$\star\star\star$ This work was partly supported by a doctoral grant from the french ministry of research and SERMA Technologies.

will expose in section 5 a characterization method using glitches on I/Os and we will investigate in section 6 how it is possible to use it in order to quickly and efficiently achieve physical attacks on these cards. Finally we will present our experiments and we will conclude.

2 Exploring and attacking closed Java Cards

A Java Card is able to embed several applets that do not run simultaneously because the OS has a single thread. Its architecture (*cf.* Fig. 1) consists of:

- an embedded virtual and homogeneous Operating System that supports the loading and the execution of several applets. The OS is a runtime environment (Java Card Runtime Environment – JCRE) with a virtual machine (VM) that provides security features (e.g. a firewall between applets and the system).
- the applets that are interpreted by the VM.

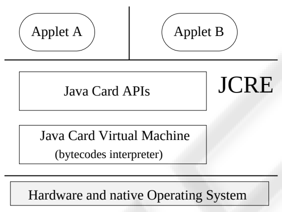
Applet A
Applet B
Java Card APIs
JCRE
Java Card Virtual Machine
(bytecodes interpreter)
Hardware and native Operating System

Fig. 1. Architecture of a Java Card.

Even though the ways to explore a closed Jard Card (*i.e.* a Java Card that allows applets to be uploaded after it has been issued only if the authentification was successful) before trying to attack it are reduced, there are at least two possibilities that remain:

- Software approach. Since an end user cannot load code on a closed Java Card he can only access its external interfaces. More precisely, the only external interface visible on a card is the communication layer. One of the possibilities to set up an attack at the level of the communication layer is to send all the possible APDU[1] commands [3,4] to the Java 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 the 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.
  Smart cards are also prone to many other invasive (microprobing, etc.) and non-invasive (glitch on the different pads of the card, etc.) attacks [11,12,13,14] which are out of the scope of this paper.

[1] The APDU (Application Protocol Data Unit) is the communication unit between a reader and a card at the application level.

But, if the user owns the authentication keys, he will be able to load his code and he can apply the characterization method described in section 5 to set up attacks as shown in section 6. He can also use the internal attacks presented in section 4. We consider that this method to test the security is mainly intended to be used by the ITSEF or by card manufacturers.

3 The open Java Cards

Till now the Java Card technology has not completely been proven secure by formal methods and consequently it is not safe to run uncertified applets that may be provided by a hacker to attack the assets of the platform and those of the other applets. To prevent these problems, the Java Card issuers began to support standards such as GlobalPlatform [15] which specifies how to securely load, install and manage applets on a card. Indeed this standard is used to easily set up and use cryptographic mechanisms to authorize or prevent the loading of an applet on the card. For example the applet 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 applet. But the major drawback of this solution is its centralized model because of the trusted third party required to sign the applet; it thus decreases flexibility. Therefore, on-card verifiers have been proposed [16,17,18,19,20,21,22] although Java Card designers first thought it was impossible to embed a verifier because of the resource constraints of smart cards. Among the different solutions that have been proposed, three of them (*i.e.* the defensive VM, the verifier based on code transformation and the stand-alone verifier) allow to withdraw the signature step of the applet without jeopardizing the card security and thus allow everybody to freely load his/her own applet – *i.e.* anyone can still load a rogue applet but it will eventually be rejected by the verifier or blocked by the defensive VM. Note that the verifier based on code transformation [18,19] requires an off-card program to normalize the applet whereas the two others (*i.e.* the defensive VM [20] and the stand-alone verifier [22]) are stand-alone. We only consider as ***open Java Cards*** the Java Cards based on one of these two stand-alone solutions. Both are equivalent [21] but the defensive VM dynamically checks each executed bytecode whereas the stand-alone verifier statically checks the applet once at load time and it is associated with an offensive VM that does not check the bytecode at execution time. Even if these open Java Cards are not yet available on the market they most probably are the future of Java Cards.

4 Internal attacks

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

– identify services present on the card and then try to deduce the possible behavior of an official[2] applet (since it can only use services available on the card!). For

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

example an applet can try to use all the cryptographic services defined in the Java Card specifications [2] in order to work out if they are present on the card or not.
- collect information on the card to elaborate hardware and software attacks. For example, an applet can be loaded to gather information about the applets already installed on the card.
- attack the VM and the firewall. 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 [23] and attacks against the VM based on type confusion are shown in [24]. There also exist attacks coming from problems in the specifications such as those presented in [25].

5 Characterization method

Once the available services have been identified, an interesting possibility consists in observing their signature. 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, 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.

In this section we present an efficient and simple way to to determine the signature of services on open Java Cards using glitches on the I/O channel of the card. This method consists in surrounding the pattern to observe with events that are visible for an observer outside the smart card. Fig. 2 shows the trace of a normal execution of an applet that includes a pattern to observe. Obviously it is difficult to isolate this pattern from all those composing the trace.

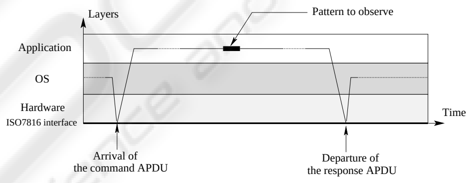

Fig. 2. Trace of a normal execution in the layers.

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

Another advantage of this approach is that the trace to save is shorter (because the area to observe is smaller) and thus it is possible to get a better sampling of the signal, still producing the same amount of data.

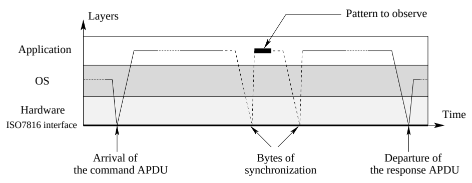

Fig. 3. Trace of a glitched execution in the layers.

5.1 Using the time extension request

The ISO 7816-3 standard [3] 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 1.1 shows an example using the `waitExtension()` glitches to surround an encryption.

Listing 1.1. Encryption surrounded by `waitExtension()` glitches

```
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.

5.2 Using the standard card communication method

The second solution uses the standard card 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 1.2 shows an example using response based glitches to surround an encryption.

This method was successfully tested on many Java 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 APDUs (*i.e.* ISO7816-4 [4]) are achieved by a sequence of TPDU[3]

[3] The TPDU (Transmission Protocol Data Unit) is the communication unit between a reader and a card at the transmission level

Listing 1.2. 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);
...
}
```

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).

5.3 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 1.3 corresponds to the result of the compilation and conversion of the three last lines of listing 1.2.

Listing 1.3. 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)
```

It makes it clear that it is possible by working at the bytecode level to improve the precision of the observation. The sequence of bytecodes presented in listing 1.3 can be modified as shown listing 1.4 and still have the same global behavior but with a better accuracy regarding the surrounding of the encryption by the glitches (*cf.* the Appendix).

This improvement is also possible when using the `waitExtension()` 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[4] file implies to change many other parts. For instance changes in the

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

Listing 1.4. 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)
```

array of bytecodes 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 furthermore imply modifications of the Directory component. To simplify these operations, we have developed a tool provided with the JCatools suite [26,27] 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.

Note that if the manufacturers find countermeasures against this method, it will still be possible to locate the pattern by surrounding it with something that induces high power consumption (a cryptographic operation, etc.) to still produce an event visible from the outside.

6 Quickly evaluating the feasibility of an attack

Based on the characterization method presented above it is possible to quickly evaluate the feasibility of an attack against an official applet since a hacker can possibly set up physical attacks [12,13,14] on the pattern identified in her own applet. Indeed she is in the best experimental position to easily and quickly attack a card implementation (e.g. to 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 a way to attack 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 security. It is also possible to add a glitch just before accessing data of another applet so as to synchronize a physical attack to bypass the checks of the firewall thanks to the disturbance of the hardware component.

7 Experiments

The method presented in this paper and some improved versions are used by the ITSEF of SERMA Technologies to test Java Card platforms, applets and their service for instance for a Common Criteria evaluation [28]. Attacks described in the previous section were already successfully used during real evaluations. The technical details of their implementation and the results are confidential but the main principles are published in this paper. Most of the time they are used with the theoretical assumption that the attacker could load her applet (the Java Cards that are evaluated are not yet open). Obviously this assumption is taken into account in the quotation of the attacks to know if they exceed the desired evaluation assurance level.

8 Conclusion and future work

This paper describes a way to quickly test the security of the next generation of Java Cards. 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 Java Cards. Even if the problems raised may seem obvious, we have already used them successfully during a real product evaluation to quickly set up attacks.

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 1.5.

Listing 1.5. 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 1.6.

Listing 1.6. 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 position of the glitches or to replace the dummy sequence by the proper bytecodes. Listing 1.7 shows how we update the listing 1.6 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 1.7. Pattern to observe surrounded by glitches

```
...
nop         // The nop bytecodes 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 1.7 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. Chen, Z.: Java Card™ Technology for Smart Cards: Architecture and Programmer's Guide. Addison-Wesley (2000)
2. Sun microsystems: Java Card™ 2.2.1 Specifications. Sun microsystems (2003)
3. International Organization for Standardization: Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part 3: Electronic signals and transmission protocols. (ISO)

4. International Organization for Standardization: Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part 4: Interindustry comands for interchange. (ISO)
5. Muir, J.A.: Techniques of Side Channel Cryptanalysis. Master's thesis, University of Waterloo, Ontario, Canada (2001) Master of Mathematics in Combinatorics and Optimization.
6. Kocher, P.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology, Springer-Verlag (1996) 104–113
7. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology, Springer-Verlag (1999) 388–397
8. Coron, J.S., Kocher, P., Naccache, D.: Statistics and Secret Leakage. In: Proceedings of Financial Cryptography (FC2000). Volume 1962 of Lecture Notes in Computer Science., Springer-Verlag (2001) 157–173
9. Quisquater, J.J., Samyde, D.: ElectroMagnetic Analysis (EMA): Measures and Countermeasures for Smart Cards. In: Proceedings of E-smart 2001. Volume 2140 of Lecture Notes in Computer Science., Springer-Verlag (2001) 200–210
10. Gandol, K., Mourtel, C., Olivier, F.: ElectroMagnetic Analysis: Concrete Results. In: Proceedings of CHES'2001. Volume 2162 of Lecture Notes in Computer Science., Springer-Verlag (2001) 251–261
11. Kömmerling, O., Kuhn, M.G.: Design Principles for Tamper-Resistant Smartcard Processors. In: Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99), Chicago, Illinois, USA (1999) 9–20
12. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. In: Proceedings of Workshop on Fault Detection and Tolerance in Cryptography, Italy (2004)
13. Giraud, C., Thiebeauld, H.: A survey on fault attacks. In: Proceedings of CARDIS'04, Smart Card Research and Advanced Applications VI, Toulouse, France, Kluwer academic publisher (2004) 159–176
14. Skorobogatov, S., Anderson, R.: Optical Fault Induction Attacks. In: Proceedings of Workshop on Cryptographic Hardware and Embedded Systmes (CHES 2002), San Francisco Bay (Redwood City), USA (2002)
15. GlobalPlatform: GlobalPlatform. (http://www.globalplatform.org/)
16. Rose, E., Rose, K.: Lightweight bytecode verification. In: In Workshop on Fundamental Underpinnings of Java, OOPSLA '98 Workshop., Vancouver, Canada (1998)
17. Casset, L., Burdy, L., Requet, A.: Formal Development of an embedded verifier for Java Card Byte Code. In: Proceedings of the IEEE International Conference on Dependable Systems & Networks, Washington, D.C., USA (2002)
18. Leroy, X.: On-Card Bytecode Verification for Java Card. In: Proceedings of the International Conference on Research in Smart Cards, E-Smart 2001, Springer-Verlag (2001) 150–164
19. Leroy, X.: Bytecode verification on Java smart cards. Software-Practice & Experience **32** (2002) 319–340
20. Cohen, R.M.: Defensive Java Virtual Machine Version 0.5 alpha. (1997)
21. Barthe, G., Dufay, G., Jakubiec, L., Melo de Sousa, S.: A Formal Correspondence between Offensive and Defensive JavaCard Virtual Machines. In: Proceedings of VMCAI'02. Volume 2294 of Lecture Notes in Computer Science., Venice, Italy, Springer-Verlag (2002) 32–45
22. Deville, D., Grimaud, G.: Building an "impossible" verifier on a Java Card. In: 2nd USENIX Workshop on Industrial Experiences with Systems Software, Boston, USA (2002)
23. Montgomery, M., Krishna, K.: Secure Object Sharing in Java Card. In: Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99), Chicago, Illinois, USA (1999)

24. Witteman, M.: Java card security. Information Security Bulletin **8** (2003) 291–298
25. Betarte, G., Giménez, E., Chetali, B., Loiseaux, C.: FORMAVIE: Formal Modelling and Verification of Java Card 2.1.1 Security Architecture. In: Proceedings of E-Smart 2002, Nice, France (2002) 215–229
26. Chaumette, S., Hatchondo, I., Sauveron, D.: JCAT: An environment for attack and test on Java Card. In: Proceedings of CCCT'03 and 9th ISAS'03. Volume 1., Orlando, FL, USA (2003) 270–275
27. Hatchondo, I., Sauveron, D.: The JCatools website. (http://sourceforge.net/projects/jcatools/)
28. CCIMB: International Common Criteria home page. (http://www.commoncriteriaportal.org/)