Embedded Eavesdropping on Java Card

Guillaume Barbu, Christophe Giraud, Vincent Guerin

Embedded Eavesdropping on Java Card

Guillaume Barbu[1,2], Christophe Giraud[1], and Vincent Guerin[1]

[1] Oberthur Technologies
Technologies and Innovation,
4 allée du Doyen George Brus, 33600 Pessac, France.

[2] Institut Télécom / Télécom ParisTech, CNRS LTCI,
Département COMELEC,
46 rue Barrault, 75634 Paris Cedex 13, France.

{g.barbu,c.giraud,v.guerin}@oberthur.com

Abstract. In this article we present the first Combined Attack on a Java Card targeting the APDU buffer itself, thus threatening both the security of the platform and of the hosted applications as well as the privacy of the cardholder. We show that such an attack, which combines malicious application and fault injection, is achievable in practice on the latest release of the Java Card specifications by presenting several case studies taking advantage for instance of the well-known GlobalPlatform and (U)SIM Application ToolKit.

Key words: Java Card, APDU Buffer, Fault Attack, Logical Attack, Combined Attack.

1 Introduction

When introduced in the mid-nineties, Java Cards revolutionized the development process for smart cards applications. Indeed before then, applications were always developed in a native way, i.e. by taking into account the specificities of the corresponding hardware on which the application is going to be executed. This meant in particular that if a developer wanted to execute the same application on several different devices, he had to develop as many implementations as devices. Java Cards on the other hand allow the developer to implement an application independently from the device on which it is going to be executed. Such an abstraction layer is provided by the Virtual Machine which interprets the Java code, called *bytecode*, and executes the corresponding instructions for a specific device. Therefore, executing a brand new Java Card application on each and every Java Card on the market costs only one development, leading to the very fast deployment of such an application which cannot be achieved when using native products. Moreover, Java Cards allow one to easily load new applications when the card is in the field whereas such a functionality extension is very difficult to achieve when using a native card.

Originally used in the mobile environment, Java Cards are now widely used in banking and identity environments where the constraints in terms of security

are very strong. Indeed, Java Cards are generally considered as intrinsically safer than native ones due to the security brought by the Java Card Runtime Environment which for instance constantly checks that objects of an application do not access objects of another application. However, as it is often the case when a new system appears, many attackers try to circumvent the inherent security of Java Card by using so-called *ill-formed applications*. To do so, the attacker modifies the binary representation of a Java Card application in order to allow it to access unauthorised objects [1–5]. Fortunately, such logical attacks can be counteracted by the use of a *bytecode verifier* [6, §4.9.2] whose aim is to ensure that an application is conform to the Java Card specifications [7–10]. In addition, implementors may operate certain verifications at run time in a so-called *defending virtual machine*, by opposition to the *offensive* ones which merely interpret the bytecode and totally rely on the security brought by the bytecode verification. However, such a verification was not mandatory up to the very recent Java Card 3.0 *Connected* Edition specifications which make the use of an on-card bytecode verifier compulsory. Therefore, each and every Java Card prior to the Java Card 3.0 *Connected* Edition is likely to be vulnerable to software attacks if the bytecode verification is not performed or if the embedded virtual machine does not implement additional security checks at run time.

At the same time as Java Cards were introduced, two new kinds of attacks specific to the embedded environment were published. These attacks take advantage of the *physical* properties of the embedded device on which the application is being executed. The first kind of these physical attacks, called *Side Channel Analysis*, takes into account the physical interactions between a device and its environment to obtain information about the secrets manipulated by the device. Examples of such interactions are the power consumption [11] or the electromagnetic radiation [12] of the device. The second kind of attacks, called *Fault Attacks*, aim to disturb the execution of an application. Such a disturbance could lead to a faulty output or to executing the application with granted privileges [13, 14]. Nowadays, the main mean of disturbing an embedded device is to use light beams [15] or electromagnetic pulses [16]. Physical attacks were mainly studied in the literature to break cryptographic implementations but they can also target any function implemented on embedded devices.

The idea to combine software and physical attacks appears recently in [17]. Such attacks, called *Combined Attacks*, aim at allowing a malicious application to bypass the security of Java Cards even using a bytecode verifier. Since then, many Combined Attacks have been published to attack several vital points of a Java Card such as the operand stack or the garbage collector [18–22].

In this paper, we use a Combined Attack to compromise another vital point of a Java Card which has not yet been targeted: the *APDU buffer*. This buffer is used to exchange all data that passes between the smart card and the terminal. It is therefore a central element of any smart card. In the following parts, we will show how an attacker can spy on the content of the APDU buffer or modify it through concrete examples on Java Cards. This study exhibits the fact that the developer must always take into account that the APDU buffer can

be compromised at any time and not only during communication with the card reader.

The rest of this paper is organised as follows. In Section 2, we detail the usage of the APDU buffer on smart cards before exposing the main specificities of this buffer in a Java Card. We also briefly present the characteristics of the two Java Card platforms which are described in the latest specifications, namely the *Classic* and *Connected* Editions. In Section 3, we show how a Combined Attack can allow an attacker to access the content of the APDU buffer during the execution of an application running on either Java Card *Classic* or *Connected* Edition. In Section 4 we present several cases studies, in particular to break Secure Channels. Finally Section 5 concludes the paper.

2 Forewords on the APDU Buffer and Targeted Platforms

The attack we propose is about the attacker's ability to illegally access the Application Protocol Data Unit buffer within an applet of her own. In this section, we start by discussing the possible usage of the APDU buffer and the potential security issues. Then we outline a couple of statements from the JCRE specification relative to the security of the APDU buffer in a Java Card platform. Finally, we present the two platforms we consider in the scope of this article.

2.1 The APDU Buffer Usage

At first glance, one could say that having hand over the APDU buffer is only a Man-In-The-Middle attack between the card and the card reader. However when looking more carefully, one can observe that the APDU buffer contains not only received or emitted data but it is also a temporary buffer that the application uses during its execution. We expose hereafter two examples which illustrate how powerful an adversary is compared to a Man-In-The-Middle attack when she can spy on and/or modify the APDU buffer.

The first example is based on *Secure Channel* which is the most common countermeasure to counteract Man-In-The-Middle. The principle of a Secure Channel is to share a key between the card and the reader and then to ensure confidentiality and integrity of the communication by using this key with cryptographic functions such as encryption or MAC verification for instance. For simplicity reasons and memory consumption optimisation, such operations are often performed in the APDU buffer. In such a case a Man-In-The-Middle cannot alter the exchanged data without being detected nor recover the sensitive information which are exchanged. However, an attacker having hand over the APDU buffer can not only spy on the communication, bypassing the confidentiality insurance, but she can also modify the command after the MAC verification leading to very powerful potential attacks.

A second example concerns the management of the APDU buffer during commands requiring a very large amount of memory space, such as asymmetric cryptographic computations for instance. In such a case, the developer can

use the APDU buffer as a temporary buffer during the application execution to extend the memory space capability of the device. However, an attacker spying on the APDU buffer during the cryptographic computation can compromise the security of the system if sensitive values such as cryptographic keys are manipulated in this buffer. Moreover, if the attacker can modify the values manipulated in the APDU buffer, a logical fault can be injected on a temporary value leading to an erroneous cryptographic output. Such a faulty output can then be used to recover the corresponding cryptographic secret key by using Differential Fault Analysis [13].

The examples described above emphasize the strength of an attacker if she succeeds in spying on and modifying the APDU buffer during the execution of an application. In the following, we will present in more detail the specificities of the APDU buffer in the context of a Java Card.

2.2 Specificities of the APDU Buffer in a Java Card

On a Java Card platform, a global array is a particular type of array object that is owned by the JCRE but accessible by different applications. The APDU buffer object contains such an array which is accessible through the `javacard.framework.APDU.getBuffer()` virtual method. Indeed this array is the buffer containing the incoming APDU command and the outgoing APDU response.

The sensitive nature of such arrays is quite obvious since they are potentially shared amongst all applications. Therefore the JCRE specifications mandate several restrictions and verifications concerning its use.

Firstly, to prevent an application from accessing a global array when it should not, the following restriction applies:

> *"Accessing Class Instance Object Fields (...). Otherwise, if the bytecode is* `putfield` *and the field being stored is a reference type and the reference being stored is a reference to a temporary JCRE entry point object or a global array, access is denied."* [10, §2.4.2.8]

Secondly, to avoid any data leakage from one application to another, the following statement is specified:

> *"Because of the global status of the APDU buffer, it MUST be cleared to zeroes whenever an applet is selected, before the applet container accepts a new APDU command. "* [10, §2.4.2.2]

In the following, let us present the two different Java Card platforms we consider in this paper.

2.3 Targeted Platforms

In its latest version, the Java Card standard has been divided into two different Editions: the *Classic* and the *Connected*. Let us briefly present below these two editions.

Java Card 3.0 Classic Edition The Java Card 3.0 *Classic* Edition appears as a regular update of the Java Card 2.2.2 standard. It is as of today the most widespread type of Java Card platform. Therefore several frameworks have been defined with a strong link with such platforms. This is the case for instance of the GlobalPlatform environment and the Card/(U)SIM Application Toolkits.

Java Card 3.0 Connected Edition. The Java Card 3.0 *Connected* Edition stands as the major evolution of the latest release of the Java Card specifications. It offers several new capabilities, such as an embedded web server, coming with standard network protocols, a widely upgraded API, or on-card bytecode verification. Besides the on-card bytecode verification, making an ill-formed-applet-based attack hardly possible, the main features used in the following of this article are the multithreading support and the notion of *Restartable Tasks*.

The multithreading consists in allowing the concurrent execution of several processes (*threads* of execution) on a given system. On a single-core system such as a smart card, multithreading is then typically implemented by alternatively giving the system resources to the different threads of execution.

In the scope of this article, the multithreading will be used through the definition of a restartable task, also introduced in the latest Java Card specifications. The notion of *Restartable Tasks* is based on a task registry in which an application can register/unregister tasks it wishes to launch automatically whenever the system is powered on. In particular, an application can register an object instance of a class implementing the interface `Runnable` (by extending the class `Thread` typically) into the registry by a call to the static method `TaskRegistry.register(Runnable t)`. Subsequently, the `run()` method of this object instance will be automatically executed in a new thread of execution every time the system starts up.

This section has introduced the APDU buffer and its specificities. As just seen, the specifications are aware of its sensitiveness, hence the quoted restrictions. In Section 3 we present different ways to overcome these restrictions depending on the targeted platform.

3 The APDU Buffer Storage Attacks

In this section we firstly detail a fault attack allowing an attacker to store the APDU buffer despite the JCRE restrictions. Subsequently, we show how to exploit such a capability to mount a full attack path on platforms implementing either the *Classic* or the *Connected* Editions of the Java Card 3.0 specifications.

3.1 A Fault Attack to Store the APDU Buffer

Attacks against Java Card platforms often take advantage of ill-formed applications loaded on-card without going through the bytecode verification process. We will describe below how the combination of a malicious, but yet well-formed,

application with a single physical fault injection can allow an attacker to gain a permanent access to the APDU buffer array whatever Edition the Java Card implements.

As stated in Section 2.2, the JCRE is responsible for preventing an application from storing references to global arrays, and in particular to the APDU buffer array. Therefore, the JCRE must perform runtime checks to enforce this rule. Without loss of generality, we assume that the JCRE operates the check described in Listing 1.1 when executing a `putfield` instruction.

Listing 1.1. Detection of APDU buffer storage attempt in `putfield`

```
// ref points to the object to store
if (isGlobalArray(ref)) {
  // Handle storage attempt
  throw SecurityException
}
```

Obviously, the aim of the attacker is to force the jump in the *else-branch* in our case. Since such a disturbance can be achieved thanks to a fault injection [13], the attacker can run within her own applet a method trying to store the reference of the APDU buffer array into a global array and disturb the conditional branching execution to avoid the `SecurityException`.

Let us assume that the attacker has been successful with the fault injection. As a result, she has been able to store the reference of the APDU buffer into a non-volatile field. Using this field, she is then likely to access the APDU buffer at any time. The following shows how such a capability can be exploited by an attacker on the Java Card 3.0 *Classic* and *Connected* Editions.

3.2 Attacking a Java Card 3.0 Classic Edition

By using the attack presented in Section 3.1, we assume that the attacker is able to access the APDU buffer at any time. However, without any interaction with other entities on-card, she cannot take advantage of this privilege in other ways than accessing the command and response of her own application.

In order to be able to exploit her new facility, the attacker's application must be given a chance to run when the APDU buffer is meant for another application. It is therefore necessary that it exposes one or several method(s) that might be called by another entity on the platform through shareable interfaces. The point is that when called, these shared method would allow the attacker to read or corrupt the APDU buffer "belonging" to the entity calling the method.

One can think that such a situation where an entity on-card calls the shared method of another applet would only appear in an attack proof of concept. However, we demonstrate in Section 4 that different contexts can lead to this situation in practice.

3.3 Attacking a Java Card 3.0 Connected Edition

In the following, we show how an attacker can take advantage of the Java Card 3.0 *Connected* Edition multithreading facility to exploit the privilege of accessing the APDU buffer whenever she wants.

Considering the attacker has been able to store the APDU buffer, we can then imagine a restartable task whose `run()` method infinitely loops and spies upon the APDU buffer, such as detailed in Listing 1.2.

Listing 1.2. The eavesdropping restartable task

```
/**
 * - fieldBuf is the stored APDU buffer reference.
 * - BUF_LEN is the assumed APDU buffer length.
 * - tmpBuf is a byte array initialized with a size of BUF_LEN.
 * - os is an OutputStream used by the attacker
 */
public void run() {
  while (true) {
    // APDU buffer is different, copy its content.
    if (arrayCompare(fieldBuf, 0, tmpBuf, 0, BUF_LEN) != 0) {
      System.arraycopy(fieldBuf, 0, tmpBuf, 0, BUF_LEN);
      os.write(tmpBuf);
    }
  }
}
```

The attacker is then potentially able to dump every byte written in the APDU buffer inside her `run` method. Moreover, she can also modify the content of the APDU buffer by writing into instead of copying it.

We have seen how a Combined Attack on a Java Card can allow an attacker to spy the content of the APDU buffer during the execution of an application. In the following, we expose different case studies based on this capability.

4 Case Study

In this section, we exhibit two case studies from two important specifications of the Java Card ecosystem, namely the GlobalPlatform (GP) environment [23, 24] and the Card and (U)SIM Application ToolKit (CAT/(U)SAT) [25, 26]. In addition, we detail a possible exploitation of the restartable task described in Section 3.3.

4.1 Attacking through the GP Environment: OPEN

GP is an entity developing and publishing specifications relative to the deployment and management of embedded applications on secure chip technologies. As part of the GP specifications [23], we find the description of the GP environment: the OPEN.

The GP Environment: OPEN As per [23], the OPEN is the on-card entity responsible for command dispatch, card content management operations, security management operations and secure inter-application communication. According to this last responsibility, the OPEN including its contactless extension [24], is the GP entity that sends notifications to other on-card entities when certain events occur. In order to keep track of the different on-card entities, the OPEN owns and uses an internal GP registry as an information resource. This registry contains information for managing the card, executable load files, applications, *Security Domain* associations, and privileges.

Shareable Interface Method Call from the OPEN. The event notifications are operated through calls to a shareable interface method. The only limitation is then for the attacking application to register for such notifications. In the scope of GP's contactless services, applications that implement the `CLApplet` interface shall be notified of changes occurring to their GP registry entry. These changes can have various origins, depending on the Contactless Registry Service (CRS). For instance, the application is notified of its installation on the platform, of the modification of the contactless communication protocol, etc... (the complete list of events can be found in [27]). These notifications are implemented by calls to the `notifyCLEvent` method of this interface.

Therefore, the applet detailed in Listing 1.3, which has gained a permanent access to the APDU buffer array as described in Section 3, is likely to analyse its content each time the registry entry of the applet is updated. As such updates occur quite often, the attacker is then able to access frequently to the APDU buffer. One could note that these updates of the GP registry are typically privileged operations performed by the `CRSApplication`. Therefore, the data potentially contained in the APDU buffer is likely to be particularly sensitive. This could be for instance data having led to a successful authentification or granted authorization.

Listing 1.3. APDU analysis on event notification when the attacking applet implements `CLApplet`

```
public class MyApplet extends Applet, implements CLApplet {
    ...
    public void notifyCLEvent(short event) {
        analyseAPDU(); // using the stored reference
    }
}
```

In this section we have shown how the access to the APDU buffer could lead to gain sensitive information relative to the security of both the platform and the hosted applications. The following section describes how the privacy of the card holder can also be threatened.

4.2 Attacking through the CAT/(U)SAT

Mobile communication is in constant evolution since the early 90s. With the well known (U)SIM card and UICC (resp. for (Universal) Subscriber Identity Module and Universal Integrated Circuit Card), it is today the most important market in the smart card industry. The CAT [25] and (U)SAT [26] are standards from the mobile communication. Their main goal is to define how the smart card should interact with the outside world and initiate commands independently of both the handset and the network. We show in the following that these can be misused by an attacker with the APDU buffer access privilege.

The CAT Runtime Environment and the (U)SAT Framework. As part of these toolkits, the CAT and (U)SAT Application Programming Interfaces (APIs) for Java Card are respectively specified in [28] and [29]. Java Card toolkit applets are then likely to control access to the network, displaying menus on the handset, etc...

These features are achieved thanks to the *Toolkit Registry*, which similarly to the GP registry defined in the previous section, allows a *Toolkit Applet* to register to events fired by the runtime environment.

Eavesdropping and Corrupting the Short Message Service. As for the attack described in the previous section, event notifications are operating through calls to a shareable interface method. In order to register to some events, an applet must implement the interface `uicc.toolkit.ToolkitInterface` and call the `setEvent(short event)` method of its ToolkitRegistry instance (available by a call to `uicc.toolkit.ToolkitRegistrySystem.getEntry()`). Subsequently, the implemented `processToolkit(short event)` will be triggered each time an event to which the applet is registered occurs.

In the context of the attack we describe, the attacker has then all the reasons to register to all possible events, in order to have her eavesdropping method called as often as possible. In particular, we study the case of events associated to the reception of a short message through the Short Message Service (SMS). The attacker's toolkit applet is described in Listing 1.4.

Provided, short messages are located in the APDU buffer, the attacker is able to intercept them and to either redirect them to the outside world or modify their content as she pleases. This is indeed one of the many ways the attacker can use the APDU buffer in this context. Other potential applications can also take advantage of the pro-active capability of the CAT environment to redirect outgoing messages or calls to taxed services for instance.

The two previous case studies were targeting the Java Card 3.0 *Classic* Edition and earlier. The next one described how the so-called eavesdropping restartable task can be exploited on a Java Card 3.0 *Connected* Edition.

4.3 Attacking through the Eavesdropping Restartable Task

In this section we depict a scenario where the attacker is able to eavesdrop or tamper with the communication (even if secured by a secure channel) and

Listing 1.4. Eavesdropping and corrupting the SMS

```
public class MyApplet    extends Applet,
                         implements ToolkitInterface {
  ToolkitRegistry r;
  public MyApplet() {
    r = ToolkitRegistrySystem.getEntry();
    ...
    r.setEvent(ToolkitConstants.EVENT_UNFORMATTED_SMS_PP_UPD);
  }
  public void processToolkit(short ev) {
    if(ev == ToolkitConstants.EVENT_UNFORMATTED_SMS_PP_UPD){
      analyseAPDUSMS(); // using the stored reference
    }
    ...
  }
}
```

temporary data. In both cases we can use the result presented in [21], exploiting I/O flooding to force a thread scheduling at a specific time. As a consequence, the attacker finds herself in the situation described in Section 2.1 where she can access the APDU buffer almost whenever she pleases. In the following, we detail a case study proving the potential threat of such a situation.

Breaking the Secure Channel. As stated in Section 2.1, a Secure Channel is a mechanism provided by GP to ensure both the confidentiality and integrity of the terminal-card communication through cryptographic mechanisms. We show here that the restartable task we have introduced in Section 3.3 can be use to break a Secure Channel.

The initialisation of a Secure Channel is made thanks to two APDU commands, namely INIT_UPDATE and EXT_AUTHENTICATE, with specific `CLA` and `INS` bytes (respectively `80 50` and `84 82`). Therefore, the eavesdropping task can detect the beginning of a Secure Channel session by detecting these commands. Subsequently, the deciphering (resp. ciphering) and MAC checking (resp. computing) of an incoming (resp. outgoing) APDU is operated thanks to a call to the method `unwrap(byte[] baBuffer, short sOffset, short sLength)` (resp. `wrap(byte[] baBuffer, short sOffset, short sLength)`) of the `SecureChannel` interface. Our point is that if this method is called with the APDU buffer array as parameter, the attacker owning the restartable task will be able to both eavesdrop and corrupt the communication. The attack scenario is depicted in Figure 1.

The SecureChannel is indeed used in numerous application in all smart card fields of application, from finance to health-care. If deemed successful, the described attack would have serious consequences regarding security and privacy.

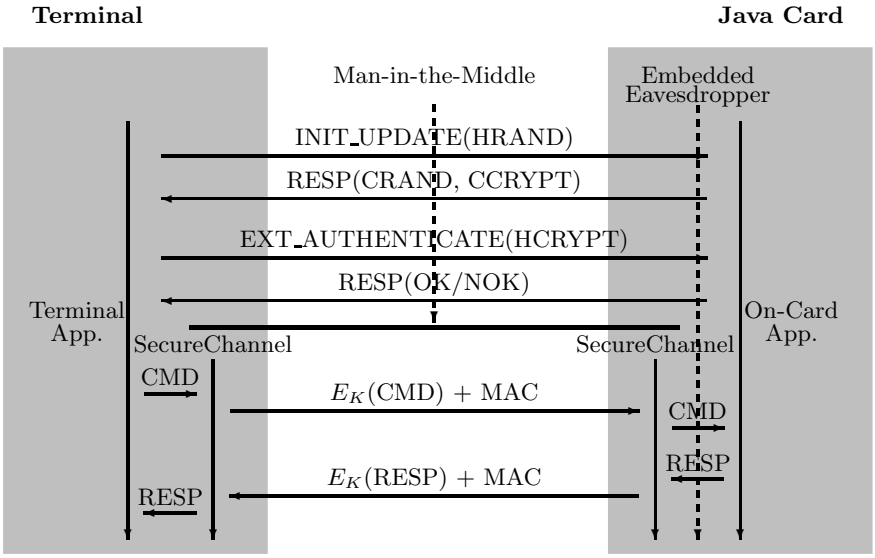

Fig. 1. Breaking the SecureChannel with the Eavesdropping Restartable Task

5 Conclusion

In this article, we have introduced a novel Combined Attack tampering with the Application Protocol Data Unit buffer. This attack leads to an outstanding privilege: accessing the APDU buffer array at any time. This attack was motivated by the fact that the APDU buffer is indeed far from being only the communication channel between the card and the terminal.

In order not to limit the range of the attack, we have described different ways to take advantage of this privilege on platforms implementing both the *Classic* and *Connected* Editions of the Java Card 3.0 specifications. Finally, we have exhibited practical exploitations of the attack on both platforms using widely spread frameworks (the GP API and the CAT/(U)SAT API) for the first platform and the multithreading facility for the second.

These exploitations highlight the crucial necessity to protect the access to the APDU buffer array, by taking into account especially attackers with fault injection capability.

References

1. Witteman, M.: Java Card Security. In: Information Security Bulletin. Vol 8. (2003) 291–298
2. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In CARDIS 2008. Vol. 5189 of LNCS, Springer (2008) 1–16
3. Sere, A.A., Iguchi-Cartigny, J., Lanet, J.L.: Automatic Detection of Fault Attack and Countermeasures. In WESS '09 (2009) 1–7
4. Hogenboom, J., Mostowski, W.: Full Memory Attack on a Java Card. In: 4th Benelux Workshop on Information and System Security (2009)

5. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan Applet in a Smart Card. Journale on Computers and Virology **6** (2010) 343–351
6. Lindholm, T., Yellin, F.: Java Virtual Machine Specification. 2nd edn. Addison-Wesley, Inc. (1999)
7. Sun Microsystems Inc.: Virtual Machine Specification – Java Card Plateform, Version 3.0.1 (2009)
8. Sun Microsystems Inc.: Application Programming Interface, Java Card Platform, Version 3.0.1 Connected Edition (2009)
9. Sun Microsystems Inc.: Java Servlet Specification, Java Card Platform, Version 3.0.1 Connected Edition (2009)
10. Sun Microsystems Inc.: Runtime Environment Specification, Java Card Platform, Version 3.0.1 Connected Edition (2009)
11. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In CRYPTO '99. Vol. 1666 of LNCS, Springer (1999) 388–397
12. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic Analysis: Concrete Results. In CHES 2001. Vol. 2162 of LNCS, Springer (2001) 251–261
13. Giraud, C., Thiebeauld, H.: A Survey on Fault Attacks. In CARDIS 2004, Kluwer Academic Publishers (2004) 159–176
14. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. IEEE **94** (2006) 370–382
15. Skorobogatov, S., Anderson, R.: Optical Fault Induction Attack. In CHES 2002. Vol. 2523 of LNCS, Springer (2002) 2–12
16. Quisquater, J.J., Samyde, D.: Eddy Current for Magnetic Analysis with Active Sensor. In: e-Smart 2002. (2002)
17. Barbu, G.: Fault Attacks on Java Card 3 Virtual Machine. In: e-Smart'09. (2009)
18. Barbu, G., Duc, G., Hoogvorst, P.: Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In CARDIS 2011. Vol. 7079 of LNCS, Springer (2011) 297–313
19. Barbu, G., Hoogvorst, P., Duc, G.: Application-Replay Attack on Java Cards: When the Garbage Collector Gets Confused. In ESSoS 2012. Vol. 7159 of LNCS, Springer (2012)
20. Vétillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In CARDIS 2010. Vol. 6035 of LNCS, Springer (2010) 133–147
21. Barbu, G., Thiebeauld, H.: Synchronized Attacks on Multithreaded Systems - Application to Java Card 3.0 -. In CARDIS 2011. Vol. 7079 of LNCS, Springer (2011) 18–33
22. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In CARDIS 2010. Vol. 6035 of LNCS, Springer (2010) 148–163
23. GlobalPlatform Inc.: GlobalPlatform Card Specification 2.2.1 (2011)
24. GlobalPlatform Inc.: GlobalPlatform Card Specification 2.2, Amendment C, Contactless Services (2010)
25. European Telecommunications Standards Institute: Card Application Toolkit (CAT) (Release 10) (2011)
26. European Telecommunications Standards Institute: Universal Subscriber Identity Module (USIM) Application Toolkit (USAT) (Release 10) (2011)
27. GlobalPlatform Inc.: Java Card Contactless API and Export File for Card Specification v2.2.1 (org.globalplatform.contactless) v1.0 (2011)
28. European Telecommunications Standards Institute: UICC Application Programming Interface (UICC API) for Java Card (Release 9) (2011)
29. European Telecommunications Standards Institute: (U)SIM Application Programming Interface ((U)SIM API) for Java Card (Release 10) (2011)