Manipulating the Frame Information with an Underflow Attack

Emilie Faugeron[(✉)]

Thales Communications and Security, 18 Avenue Edouard Belin, BPI 1414
31401 Toulouse Cedex 9, France
emilie.faugeron@thalesgroup.com

Abstract. This paper presents an underflow attack performed on Java Card platforms. This underflow is based on the *dup_x* instruction that can be used in order to read and modify the current context of execution of the attacker's application. We first detail the theoretical and practical attack path by describing the method that can be used to characterize the platform and exploit the obtained information. Secondly, we show how it is possible to set up this underflow attack in a way that makes it bypass the current concept of Byte Code Verifier. Finally, we describe some countermeasures that can be implemented to prevent this kind of attack.

Keywords: Malicious application · Underflow · Java Card Open Platform

1 Introduction

Java Card technology allows loading and executing a set of applications in a secure way on a small device. This technology is widely used by smart card industry today and has been proved to reach a high level of security in the common context of use, i.e. single issuers mastering their production of Java Card platform and related applications. Nowadays, the use of those Java Card platforms is becoming more complex. In the field of telecommunication applications, for instance, the context is moving to multi-applications provided by different issuers for different Java Card platforms. Platforms refer to the combination of a secure hardware device and a secure Operating System including the Virtual Machine, the Runtime Environment and APIs. The concern is to check how multiple applets, loaded on a Java Card platform by multiple actors, can be handled in a secure way and maintain the security of the product over its whole lifecycle.

Open Java Card platforms, enabling post-issuance applet loading and induce a new actor that is responsible of application validation. Indeed, the Verification Authority is in charge of verifying the basic application against the platform guidance. It shall include at least an off-card verification of the application. If the application is invalid, it is rejected and cannot be loaded onto the targeted platform. Therefore, an attacker has two possibilities to bypass the concept of Byte Code Verification: either developing a malicious application in a way that cannot be detected by the Off-Card Verifier, or implementing a combined attack in order to perturb the application behaviour during its execution using a laser or ElectroMagnetic pulse device. In the first case, all logical attacks using application file format manipulation are to be discarded otherwise they will be detected by the Off-Card Verifier. The attacker needs so to identify weaknesses on the Java Card platform at JCRE (Java Card Runtime Environment) level or at JCVM (Java Card Virtual Machine) level that could allow performing purely software attack. It can be a weakness in the platform implementation, or a known weakness regarding Java Card platform specification as explained in [1]. For instance, the Shareable Interface mechanism can be abused in order to perform a type confusion attack that will not be detected by the Off-Card Verifier.

The Java Card platforms are sensitive to several types of malicious applications. It can be address forging attacks by modifying specific CAP component [2, 3], type confusion attacks [1, 9] or underflow attacks [8, 10]. The first and second kinds of attacks are not relevant in that context: the first one is detected by the Off-Card Verifier, and the second one does not allow reading and modifying the context of execution of the application directly. On the other hand, the third one enables an attacker to manipulate the system information.

In this paper, we are going to focus our analysis on the underflow attack that allows manipulating the execution frame of a method associated to the current executed application. In the first part of this work we describe the theoretical and the practical attack path with a particular focus on the dup_x instruction that will be used to read and modify the frame information. In the second part, we detail the means that can be used by an attacker in order to bypass the current concept of Byte Code Verifier. Indeed, the attack described in this paper can be performed by an attacker without privilege. The attacker just needs to be able to develop an application. Finally, we present the countermeasures that can be implemented by the developer to prevent these attacks.

2 Underflow Attack: State of the Art

The underflow attack presented in this paper differs from previous works. Our hypothesis is that the malicious application is verified by Off-Card-Verifier and it uses a new type of potential vulnerability in the platform implementation.

To go back to previous work, the underflow attacks have been introduced in [8] and in [10]. The thesis [10] describes underflow attacks at a high level and is focused on countermeasures to protect a platform against such attack. The aim of an underflow attack described in [8] is to find the position of the return address onto the stack and then modify it in order to execute a code located inside an array. This underflow is performed by using non-existing local variables in order to access information located below the stack bottom. The purely software attack takes the hypothesis that there is no bytecode verification performed on the application (off-card verification or on-card verification).

Two different methods that can allow an attacker bypassing the Off-Card Verification are described in [1]. The first attack method aims abusing the transaction mechanism in order to create a type confusion. This attack is now detected by most of the platforms and cannot be applied to underflow anymore. The second attack method aims to abuse the Shareable Interface mechanism. The goal is to create type confusion using two definitions of interfaces, one for the Client and one for the Server. Actually, the attack methods described in [1] only focus on type confusion.

The aim of our paper is to describe a new way of exploiting the underflow attack despite off-card verification. Indeed, this paper describes an underflow using the instruction dup_x that is usually not checked by on card countermeasures due to the fact that the stack pointer is not decreasing at the end of the instruction processing (this kind of verification is dependent of the platform implementation). The final goal of our attack is to replace the context of the attacker's method with the JCRE context in order to gain access to out-of-context data to be able to dump and modify information link to the platform or to a sensitive application.

This attack considers that the malicious application is verified by Off-Card-Verifier. Indeed, we have extended the attack described in [1] in order to create an underflow. We have implemented two different ways of bypassing the off-card verification: (1) abusing the Shareable Interface mechanism to create an underflow, (2) abusing the library versioning to create an underflow. All steps of the attack will be further described in this paper.

3 Underflow Attack: Theoretical Attack Path

The aim of the underflow attack is to retrieve and modify the elements located before the stack of the current executed method.

All the instructions that pop elements from the stack can be used in order to perform a stack underflow attack. There are two kinds of instructions, those that lead to a modification of the stack pointer (*sp*) and those that pop elements from the stack without decreasing the stack pointer at the end of their processing. In the first case, if the operation is performed on an empty stack, the stack pointer will be located below the stack bottom at the end of the instruction treatment. This kind of attack can be done, for instance, with the instruction *putstatic* (Fig. 1).

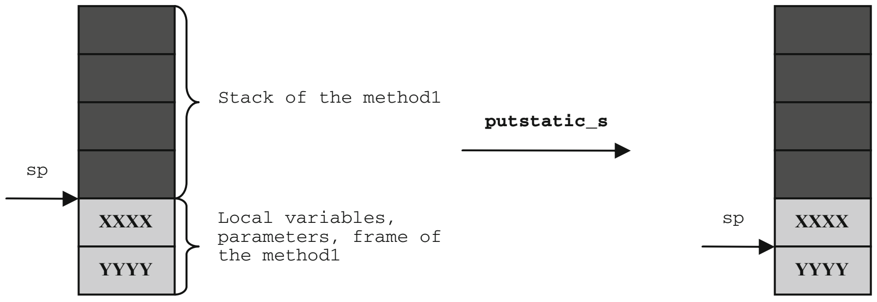

Fig. 1. *putstatic_s* instruction on empty stack

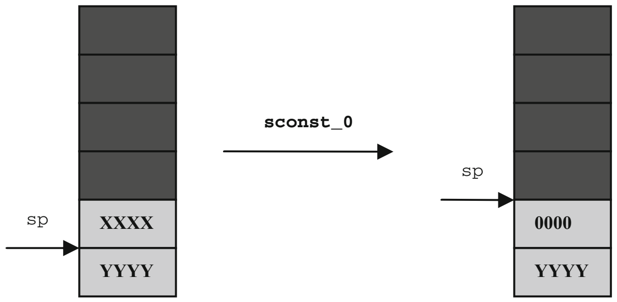

Fig. 2. Modification of the frame information thank to *sconst_0*

Once the stack pointer has been corrupted, an attacker can update any information located between the stack pointer and the stack bottom (Fig. 2):

In the second case, the stack pointer is not decreased at the end of the instruction processing but during the processing. It is for instance the case of the instruction *dup_x*. The instruction *dup_x* takes two parameters coded on 1 byte:

- m, the high nibble, that is in the range 1 to 4.
- n, the low nibble, that is in the range 0 to m + 4.

If n has a value different from 0, the top m words of the operand stack are duplicated and the copied words are inserted n words down in the operand stack. When n equals 0, the top m words are copied and placed on top of the stack [4].

Figure 3 shows the impact of a dup_x 32 on an empty stack (m is equal to 2).

This instruction can also be misused in order to update information located below the stack bottom. In this case, the attacker needs to provide a "n" different from 0 (Fig. 4).

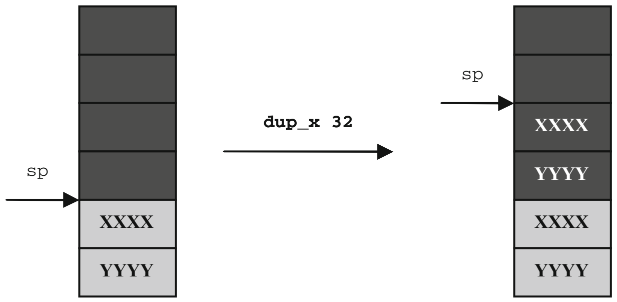

Fig. 3. *dup_x* instruction in order to read data located below the stack bottom. The two short at the top of the stack (m equal to 2) will be duplicated at the top of the stack (n equal to 0).

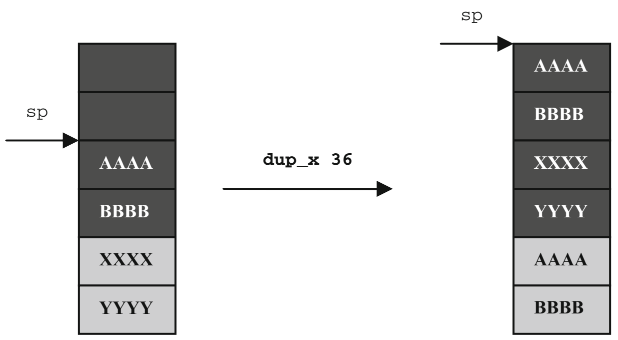

Fig. 4. *dup_x* instruction in order to modify data located below the stack bottom. The two shorts on the top of the stack (m equal to 2) will be duplicated at 4 shorts down the stack top (n equal to 4).

By using the underflow of the stack, an attacker will be able to manipulate the following information (the order of this information depends on the platform implementation):

- The local variables of the executed/caller method.
- The parameters of the executed/caller method.
- The frame information of the executed/caller method. This structure contains the context of execution of the executed or of the caller method.

In most implementation, the frame is located just before the stack. An attacker will then be able to modify the context of execution of his method.

4 Underflow Attack: Practical Attack Path

An attacker can characterize each bytecode that manipulates the stack in order to identify those that are not subject to security verification regarding underflow attacks. Each instruction can be invoked on an empty stack and then the platform behaviour is analysed for each case. In this paper, we focus our analysis on the byte code *dup_x*.

4.1 Underflow Attack Using *dup_x*

Characterisation of the Underflow Data. The first step of the attack aims reading the data located below the stack, and then to analyse and characterize each byte reading. The *dup_x* instruction allows reading 8 bytes located below the stack bottom (m equal to 4 and n equal to 0).

Depending on the platform implementation, the attacker may localize

- the frame information of the current/caller method,
- the stack number of the current/caller method,
- the stack of caller method, the number of local variables of the current/caller method,
- the local variable of the current/caller method.

The attacker needs to characterize the frame information in order to find the position of the context.

The identification of information related to the attacker's method (stack, local variable, system information) can be done by performing an underflow inside different methods of the same applet. To be efficient, these methods need to have different local variable numbers and different stack sizes. Moreover, the parameters used for each method need to be initialized with identifiable patterns:

```
public void local_method1 (short foo)
{
short var1 = (short) 0xBAB1;
short var2 = (short) 0xDED1;
short var3 = (short) 0xFEF1;
short var4 = local_method2((byte)0xDE,(byte)0xED);
return;
}
public short local_method2 (byte foo, byte bar)
{
short var1 = (short) 0xBAB2;
short var2 = (short) 0xDED2;
short var3 = local_method3();
return (short)0xDDFF;
}
public short local_method3 ()
{
//Perform the underflow attack
attr1 = (short)0x3333;
return (short)0xCDCD;
}
```

The following dump is obtained when an attacker performs an underflow using the instruction *dup_x* on an open Java Card platform:

`0x01 0x0C 0x00 0x01` **0xDE 0xD2 0xBA 0xB2**

The state of the stack is the following (Fig. 5):

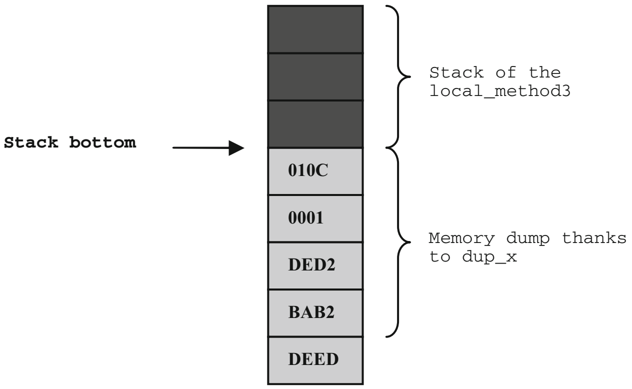

Fig. 5. State of the stack after an underflow attack using *dup_x* instruction

By analysing the dump obtained thanks to the instruction *dup_x* on an open Java Card platform, we can notice that the 3rd and the 4th words correspond to the local variables of the *local_method2*.

The identification of the context of execution of the attacker's applet can be done by loading two underflow malicious applications having different AIDs but identical code. In this case, the two applications will have the same local definition but differ on the context ID. As an example, the following data can be read when an attacker performs an underflow in an internal method of his applet:

- Underflow attack with *dup_x 64* instruction on an applet APP1 with a context APP1_context:
  0x01 0x0C `0x00 0x01 0xDE 0xD2 0xBA 0xB2`
- Underflow attack with *dup_x 64* instruction on an applet APP2 with a context APP2_context:
  0x01 0x18 `0x00 0x01 0xDE 0xD2 0xBA 0xB2`

The first two bytes are different for the two applets: it is linked to the context of the current executed applet. The second byte needs to be fixed to 0x00 in order to take the JCRE context.

Exploitation of the Underflow. Once the frame information has been localized and in particular the context of the method of the attacker, the *dup_x* instruction can be used with n, different from zero, in order to modify the execution context (as described in the Fig. 4). Indeed, this instruction allows modifying 8 bytes located below the stack (m equal to 4 and n equal to 8).

The attacker can then update the context of his own method with the identifier of the JCRE's context (equal to 0x00) to gain access to the whole card content. Indeed, there is no firewall restriction for the JCRE [5] and as long as JCRE's context is granted to a method then it can read and modify any defined object in memory.

The instructions *baload*, *saload* or *getfield* can be used in order to read specific address in the memory. Indeed, these instructions will allow accessing different types of objects in the memory: byte array, short array and class. An address forging operation needs to be performed inside the application in order to be able to access to the targeted address (push the targeted address onto the stack).

The attacker needs then to reverse the memory access process. To perform this analysis, he can dump his application code and data in order to understand object representation into the memory:

- package/applet/instance (AIDs, CAP components,…)
- code
- standard objects (byte array, class,…)
- sensitive objects (OwnerPIN, Keys,…)

Once the characterisation has been done, the attacker is able to identify all these parts for other applications loaded onto the card. The instructions *bastore*, *sastore* and *putfield* can then be used in order to modify all objects read in memory.

By targeting the code of a sensitive application, he will be able to modify it. For instance, he can replace, directly in memory, sensitive checks by NOPs in order to avoid security/error detections. He can also modify the code of the Owner PIN object inside the memory by replacing the ciphered PIN representation of the sensitive application by the ciphered PIN representation of the attacker (if the representation of objects is not diversified by object).

4.2 Byte Code Verification

Off-Card Verifier detects classical underflow attack. Nevertheless, an attacker has several means to bypass this verification:

- Abuse the Shareable Interface mechanism as published in [1]: we have extended and adapted the attack described in [1] in order to create an underflow.
- Abuse the Library mechanism.
- Use combined attack as published in [6].

Abusing the Shareable Interface Mechanism. The Shareable Interface mechanism is used to share services between applications in different contexts. An Interface is defined and contains all methods that will be shared. A Server implements these methods and builds the Shareable Interface Object (object instance of the class that implement the Shareable Interface).

A Client uses these methods by obtaining the Shareable Interface Object thanks to the method *getAppletShareableInterfaceObject(AID serverAID, byte parameter).*

The Shareable Interface mechanism can be abused in order to create a type confusion attack as described in [1]. Indeed, the Client is generated using one definition of the interface I1 with a function F that take, for instance, a byte array as parameter. The Server is generated using another definition of the interface I2 with a function F that takes a short array as parameter. During the application validation, the Client will be verified with I1 and the Server with I2, the verifications are done at two different times. That's why no error will be detected during the validation. Regarding application installations, only the interface I2 will be loaded onto the card. During the Client applet execution, the type confusion is created and can be exploited by the attacker (byte array read as a short array).

This principle can be applied to the underflow attack. Indeed, the method definition will be the following for the two interfaces:

- The Client is generated using the definition of the interface I1 (the Client contains the underflow attack exploitation part):

```
//creation of the Underflow onto the card
public int myShareableMethod_underflow(short S1);

//Address forging onto the card
public byte[] myShareableMethod_shortToByteArray();
public short[] myShareableMethod_shortToShortArray();
public myClass myShareableMethod_shortToMyClass();
```

- The Server is generated using another definition of the interface I2:

```
//creation of the Underflow onto the card
public void myShareableMethod_underflow(short S1);

//Address forging onto the card
public short myShareableMethod_shortToByteArray ();
public short myShareableMethod_shortToShortArray ();
public short myShareableMethod_shortToMyClass ();
```

The function *myShareableMethod_underflow* is called just before performing the underflow attack as illustrated in the following code extract:

```
sspush frame_1;
sspush frame_2;
myShareableMethod_underflow();//returns INT in I1
dup_x 36;//Underflow of 4 bytes
        //because it returns void indeed
```

The instructions *sspush* are used to push the new value of the frame on the top of the stack (*frame_1* and *frame_2*). Once the underflow is performed, the *dup_x* instruction allows assigning the new frame information.

Then the functions *myShareableMethod_shortToByteArray*, *myShareableMethod_shortToShortArray* and *myShareableMethod_shortToMyClass* are used to create address forging. The aim is to read a short as a byte array, a short array or a class object. The short used in order to forge address is the one given as parameter of *myShareableMethod_underflow*.

During the off-card verification of the Client with the Interface I1, no error will be detected. Nevertheless, during on-card execution with the Interface I2:

1. No int will be pushed onto the stack by the method *myShareableMethod_underflow*. The underflow will be created.
2. The underflow is exploited by the attacker: he is able to modify the current context by the JCRE context that is equal to 0.
3. A short will be returned by *myShareableMethod_shortToByteArray*, *myShareableMethod_shortToShortArray* and *myShareableMethod_shortToMyClass* and will be assigned as a reference to byte array, short array and class object. The address will be forged. The attacker will be able to access to the targeted address.

Abusing the Libray Mechanism. A Java Card platform can contain some libraries (applications that are not applets). A library is never instantiated; it contains only methods that can be used by other application loaded onto the card.

As for the Shareable Interface mechanism, an attacker can abuse the Library mechanism. The concept of the attack path is the same. Indeed, an attacker develops a library in two versions:

- Library L1 v1.0, this version of the library will be used for the verification of the application:
  ```
    public int myShareableMethod()
  ```
  As the method *myShareableMethod* returns an int, the underflow attack is not detected by the tool.
- Library L1 v1.1, this version of the library will be loaded onto the card:
  ```
  public void myShareableMethod()
  ```
  During the execution of the malicious application, the method *myShareableMethod* that return void is called. The underflow is activated and can be exploited by the attacker.

Creating an Underflow with Combined Attack. A combined attack [6] is a combination between a logical attack and a physical attack.

A combined attack can be used to create a mutant application. A mutant application [7] is an application that is well-formed and that becomes malicious during its execution by injecting a fault using a laser or an electromagnetic pulse in order to modify transiently a specific bytecode execution. Indeed, an attacker develops a well formed applet (successfully verified by an Off-Card Verifier) that is designed such that the modification of one byte by a NOP allows him to execute a malicious code, in our case the underflow attack. The applet of the attacker is loaded onto the card. The attacker then modifies the interpretation of specific instruction during the code execution using fault injection. The instruction is interpreted as a NOP and consequently, the instruction's parameters are not processed and are interpreted as new instructions.

A combined attack can also be performed in order to avoid on-card security checks or to bypass on-card countermeasures. An attacker can use it in order to bypass verification made during application loading. Indeed, the application of the attacker uses a library L2 that declares the following method: *public* ***int*** *myShareableMethod()*. The version of the library is 2.0. The application of the attacker is well-formed and will be verified with success. Nevertheless, the platform contains a library L1 with the following method: *public* ***void*** *myShareableMethod()*. The version of the library onto the card is 1.0. During the application loading, the platform will ensure that each imported package has the same major version than the one loaded onto the card. An attacker can perform a fault injection in order to bypass this specific security check. In this case, the application will be loaded successfully and the underflow can be exploited during the application execution.

5 Countermeasures

The underflow attack can be covered by organisational measures or by technical countermeasures.

5.1 Organisational Measures

The developer can add specific mandatory requirements in the guidance. Indeed, requirements related to versioning and imported packages can be sufficient to cover the purely software attack abusing the Shareable Interface or Library mechanism. In such case, the attack will be detected during the application verification process by the Verification Authority and the application will be rejected.

Nevertheless, this countermeasure does not cover combined attacks. Only technical measures can be used to cover that kind of attacks.

5.2 Technical Countermeasures

The developer can implement dedicated countermeasures onto the Java Card Virtual Machine in order to defend against the underflow attack. Indeed, he needs to add security checks upon the processing of each instruction that pop elements from the stack in order to ensure that the stack pointer is valid, during and after the instruction processing.

Nevertheless, an attacker could perform a combined attack to bypass this countermeasure: the attacker develops his malicious application, loads it onto the card, and finally performs a fault injection attack upon the execution of the application in order to avoid the underflow countermeasure. Therefore, in order to implement an efficient underflow countermeasure, the code must also be protected against faults injection attacks.

6 Conclusion

Open Java Card platforms, enabling post-issuance applet loading, induce a new type of attackers having privileges. These attackers are untrusted application developers or application loaders that are able to choose the application that will be loaded onto the card. In such context, the platform with its guidance needs to be protected against malicious applications.

We have presented, in this paper, an underflow attack that exploits the *dup_x* instruction in order to read and modify the current context of execution of the attacker's application. Once this modification is done, the attacker is able to acquire the context of the JCRE and so to read and modify out-of-context data. This attack can be developed in such a way that the malicious application will bypass the concept of Byte Code Verifier. Indeed, the validation of application is done in a specific time and the validation of the library or of the Shareable service is done at another time. This underflow attack can also be exploited through other instructions that pop elements from the stack. This attack has been performed with success on several Java Card platforms.

Several solutions exist to protect the platform against this kind of attacks, either organisational - if the guidance includes specific requirements -, or technical - if the platform implements dedicated security checks upon instructions processing -.

Finally, this paper shows that the current concept of Byte Code Verification is not sufficient to prevent all kinds of malicious applications. During a platform evaluation, the overall malicious application attack paths need to be taken into account. A specific care is to be applied on the platform guidance in order to ensure that it will contain all the necessary requirements to cover logical attack path.

References

1. Mostowski, W., Poll, E.: Malicious code on java card smartcards: attacks and countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008)
2. Lanet, J.L., Faugeron, E., Dessiatnikoff, A.: EMAN: Un cheval de Troie dans une carte à Puce. Computer & Electronics Security Applications Rendez-vous (CESAR 2008), p. 198 (2008)
3. Lanet, J.L., Iguchi-Cartigny, J.: Évaluation de l'injection de code malicieux dans une Java Card (SSTIC 09) (2009)
4. Java Card Virtual Machine Specification - Java Card Platform, Version 2.2.2, March 2006
5. Java Card Runtime Environment specification - Java Card Platform, Version 2.2.2, March 2006
6. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on java card 3.0 combining fault and logical attacks. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg (2010)
7. Vetillard, E., Ferrari, A.: Combined attacks and countermeasures. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 133–147. Springer, Heidelberg (2010)
8. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.-L.: Combined software and hardware attacks on the java card control flow. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011)
9. Karsten Nohl: Rooting SIM cards. BlackHat (2013)
10. Pierre Girard thesis: Contribution à la sécurité des cartes à puce et de leur utilisation. University of Limoges (2011)