J Comput Virol Hack Tech (2014) 10:239–253
DOI 10.1007/s11416-014-0218-7

ORIGINAL PAPER

Reversing the operating system of a Java based smart card

Guillaume Bouffard · Jean-Louis Lanet

Received: 5 February 2014 / Accepted: 12 June 2014 / Published online: 5 July 2014

Abstract Attacks on smart cards can only be based on a black box approach where the code of cryptographic primitives and operating system are not accessible. To perform hardware or software attacks, a white box approach providing access to the binary code is more efficient. In this paper, we propose a methodology to discover the *romized* code whose access is protected by the virtual machine. It uses a hooked code in an indirection table. We gained access to the real processor, thus allowing us to run a shell code written in 8051 assembly language. As a result, this code has been able to dump completely the ROM of a Java Card operating system. One of the issues is the possibility to reverse the cryptographic algorithm and all the embedded countermeasures. Finally, our attack is evaluated on different cards from distinct manufacturers.

1 Introduction

A smart card is a small tamper-resistant device with few memory. Size constraints restrict the amount of on chip memory and the most of smart cards on the market embed at most 4 kB of RAM (for runtime data and OS stacks), 256 kB of ROM (OS, API and applications), and 256 kB of EEPROM (for persistent data). These constraints have a deep impact on software design. A smart card can also be viewed as an intelligent data carrier which stores data in a secured manner and ensures data security during transactions. Several means have been used to retrieve sensitive information: side channel analysis or fault injection appear to be the most efficient. Having a precise knowledge of the software and in particular the control flow graph should be helpful for new attacks or understanding the algorithms.

1.1 Java based smart card

A Java Card is a smart card that implements the standard Java Card 3 [30] in one of the two editions Classic Edition or Connected Edition. Such a smart card embeds a virtual machine, which interprets codes already *romized*, with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by GlobalPlatform [18] (GP). The GP specification supports the implementation and management of tamper-resistant chips such as smart cards. Being a subset of Java technology, Java Card runs by the same principles: compile the Java code to get the Java class file format, convert it into CAP (for *Converted APplet*) file, a more compact format. It reduces the size of the applet image downloaded into the card and minimizes run-time memory requirements. Then the program is executed using the Java Card Virtual Machine (JCVM). The Java Card platform is a multi-application environment where the sensitive data of an applet must be protected against malicious access from another applet or from the external world. To enforce protection from the software level, Java technology uses type verification, class loader and security managers to create private name-spaces for applets. In a smart card, complying with the traditional enforcement process is not possible. The type verification is executed outside the card due to memory constraints. The Java Card platform provides further security enhancements, such as transaction atomicity, cryptographic classes and the applet Firewall. This later replaces the class

G. Bouffard (✉) · J.-L. Lanet
University of Limoges, 123 Avenue Albert Thomas,
87060 Limoges, France
e-mail: guillaume.bouffard@unilim.fr

J.-L. Lanet
e-mail: jean-louis.lanet@unilim.fr

loader and security manager for enforcing the sandbox security model. Due to the limited resources of the platform, the Java Card architecture is split in two parts. One part of the JCVM is executed off-card on a workstation. It converts and optimizes the executable code that is ran on the on-card part. The JCVM consists of a Converter and a Byte Code Verifier (BCV). The Converter takes the Java class files as input for a Java Card applet, and replaces all the method and field names by tokens. The BCV allows to assert whether the content of the applet meets the Java Card specification [30] or not. The on-card part of the virtual machine consists of the JCVM, the Java Card Runtime Environment (JCRE), and the Java Card APIs.[1]

The Java Native Interface (JNI) is an interface that enables Java code running in a Java Virtual Machine to call and to be called by native applications and libraries written in other languages. The Java Card specification expresses that no native access is defined in a JCVM. Thus, there is no way to access or execute native code in a Java Card.

1.2 Smart card security

Smart card security depends on the underlying hardware and the embedded software. Embedded sensors (light, heat, voltage sensors, etc.) protect the card from physical attacks. While the card detects such an attack, it has the possibility to erase quickly the content of the EEPROM preserving the confidentiality of secret data or blocking definitely the card (Card mute). In addition to the hardware protections, softwares are designed to securely ensure that application are syntactically and semantically correct before installation and also sometimes during execution. They also manage sensitive information and ensure that the current operation is authorized before executing it. The BCV guarantees type correctness of code, which in turn guarantees the Java properties regarding memory access. For example, it is impossible in Java to perform arithmetic on reference. Thus, it must be proved that the two elements on top of the stack are of primitive types before performing any arithmetic operation. On the Java platform, BCV is invoked at load time by the loader. Due to the fact that Java Card does not support dynamic class loading, byte code check is performed during installation time, i.e., before storing the CAP onto the card. However, most Java Card smart cards do not embed an on-card BCV as it is quite expensive in term of memory consumption. Thus, a trusted third party performs an off-card byte code verification and sign it, and on card its digital signature is checked.

Moreover, the Firewall performs checks at runtime to prevent applets from accessing (reading or writing) data of other applets. When an applet is created, the system uses an unique applet identifier (AID) from which it is possible to retrieve the name of the package. If two applets are instances of classes from the same Java Card package, they are considered belonging to the same context. The Firewall isolates the contexts in such a way that a method executing in one context cannot access any attribute or method of objects belonging to another context unless it explicitly exposes functionality via a Shareable Interface Object.

Smart card security is a complex problem with different points of view but products based on JCVM have passed successfully real-world security evaluations for major industries around the world. It is also platform that has passed high level security evaluations for issuance by banking associations and by leading government authorities, they have also achieved compliance with FIPS 140-1 certification scheme [39]. Nevertheless implementations have suffered from several attacks either hardware or software based. Some of them succeeded into getting access to the EEPROM (code of the downloaded applets) but nobody succeeded into reversing the code, i.e., having access to the code of the VM, the operating system and the cryptographic algorithm implementations. These latter are protected by the interpretation layer which denies access to other memories than the EEPROM.

This paper is organized as follow. First, the Sect. 2 presents some approaches to obtain a snapshot of the EEPROM memory. Next, in the Sect. 3, the memory snapshot is analyzed to understand how the targeted smart card works. In this snapshot, the approach related to native code execution is discovered. To prove the power of our attack, we evaluate it on cards from different manufacturers (Sect. 4). Since the smart card memories snapshot (RAM, EEPROM and ROM) is obtained, it have been analyzed in the Sect. 5. This part was done by our tool, the Java Card Disassembler and Analyser (Sect. 6), which offers the reverse of the JCVM and the smart card operating system in the Sect. 7. Finally, we conclude this article by the related works.

2 Dumping the EEPROM

Confidentiality of the EEPROM covers for example applet confidentiality, i.e., disclosure of already stored code and data confidentially. Often the sensitive data are stored into a secure container, even if sometime keys are stored in clear text in memory [23]. There are several ways to dump the EEPROM according to the attacker hypotheses. The card can be locked, i.e., post-issuance download is not permitted (like banking card). In such a case, the only way is to use Side Channel Attack (SCA). If the card is open (some (U)SIM cards, for example), so it allows post-issuance code download. This scenario is protected by the GP protocol that needs a mutual authentication before loading any code. In this scenario, the operator who is the only one allowed to load code into the card, checks systematically new applica-

[1] Application programming interface.

tion using of course the BCV but also rules checkers (e.g. never use `getKey()` nor `setValidatedFlag()`, etc.) and code reviews. Using the BCV denies the right to load ill-formed applications. But Barbu et al. [5] and Bouffard et al. [9] demonstrated that this step is useless. They loaded a well-formed code but executed ill-formed one. The last scenario concerns development cards. With such cards, the developer has the right to load code (he has the authentication keys) and he can load ill-formed applications. This latter requires less knowledge and means from the attacker but he can only retrieves information related to a development card. This scenario is not valid for a product, the BCV will always check the applet before being loaded into the card.

2.1 Product card with no post-issuance allowed

In order to gain access to services or assets stored in a product, several means have been used to retrieve valuable information, and SCA appears to be the most efficient. The idea is to use information leakage from the processor which could be power consumption or electromagnetic emanations. Power analysis involves interpreting power traces, or graphs of electrical activity over time. For example [40], it is used to get knowledge on RSA keys during the "Square and Multiply" step of modular exponentiation. A naive and vulnerable implementation of the algorithm such as the binary exponentiation algorithm could be used. Each key bit is processed with a modular square operation followed by a conditional modular multiplication. Thus, the power consumption trace shows explicitly the difference between each round of the loop the value of each bit of the key. The power consumption gives a global representation of the card activity while a EM probe gives a local information (memory, register, etc.). But this technique [40] has also been used to reverse the code of an application and thus provides a means to have a collection of traces representing different executions. There is no possibility for the attacker to be sure to fully cover all the code of the application he wants to reverse. He can only obtain traces according to the input data he provides to the card.

2.2 Product card with post-issuance allowed

In that case, only well-formed application can be loaded into the card. This scenario is suitable for cards that are modifiable after delivery to the customer. Attacks are based either on a misunderstanding of the specification by the developers or the use of a perturbation attack.

2.2.1 Specification misunderstanding

The idea of [22] is to abuse shareable interfaces to obtain a type confusion without the need to load an ill typed CAP files. To do that, the authors created two applets which communicate using the shareable interface mechanism. To create a type confusion, each applet uses a different type of array to exchange data. During compilation, there is no way for the BCV to detect a problem. But, this attack is rather old and now it seems that every tested card, with an on-card BCV, refuses to allow applets using shareable interface. As it is impossible for an on-card BCV to detect this kind of anomaly, Hubbers et al. emitted the hypothesis that any use of shareable interface on card can be forbidden with an on-card BCV.

The other way is to abuse the transaction mechanism. The purpose of transaction is to execute a set of atomic operations. Of course, it is a widely used concept, for instance in databases, but still complex to implement in a card. By definition, the rollback mechanism should also deallocate any objects allocated during an aborted transaction and reset references to such objects to `null`. However, Hubbers et al. found some cases where the card keeps the references to objects allocated during transaction even after a rollback. Moreover, the authors described the easiest way to make and exploit a type confusion to gain illegal access to protected memory. Their example was to get two arrays with different types, a byte array and a short array. If a byte array of 10 bytes is declared and it exists a reference to a short array, it is possible to read 10 shorts, so 20 bytes. With this method, they can read the 10 bytes stored after the array. If Hubbers et al. increase the size of the array, they will able to read as much memory as they want.

2.2.2 Using perturbation

Hardware based attack with perturbation is explained in [3]. A modification of the input current may modify the execution flow as the card is not self-powered [1,26]. We also have attacks, explained by Skorobogatov et al. [38], that use light (LED, laser, etc.) and focus on a specific part of the chip. This light provides enough energy in the memory cells to change their values. Electromagnetic attacks [29,31,37] like inducted current, provide a way to modify the memory value, and it also help in characterizing the chip area used during a critical operation.

The idea is to combine software and physical attacks. Barbu et al. presented and performed several combined attacks such as the attack [7] based on the Java Card 3.0 specification leading to the circumvention of the Firewall application. Another attack [6] consisting of tampering the Application Protocol Data Unit buffer (APDU) that leads to access the APDU buffer array at any time. They also discussed in [5] about a way to disturb the operand stack with a combined attack. It also gives the ability to alter any method regardless of its java context or to execute any byte code sequence, even if it is ill-formed. This attack bypasses the on-card BCV [8]. In [9], Bouffard et al. described how to change the execution flow of an application after loading it into a Java Card. Recently, Razafindralambo et al. [33] introduced a combined attack based on fault enabled viruses. Such a virus is activated by hitting with a laser beam, at a precise location in the memory, where the instruction of a program (virus) is stored. Then, the targeted instruction mutates one instruction with one operand to an instruction with no operand. Then, the operand is executed by the JCVM as an instruction. They demonstrated the ability to design a code in a such way that a given instruction can change the semantics of the program. And then a well-typed application is loaded into the card but an ill-typed one is executed.

2.3 Development card

Logical attacks are based on the fact that the runtime relies on the BCV to avoid costly tests. Then once someone find an absence of a test during runtime, there is a possibility that it leads to an attack path. The idea is to be able to execute a shellcode stored somewhere in the memory. The aim of EMAN1 attack [23], explained by Iguchi-Cartigny et al., is to abuse the Firewall mechanism with the unchecked static instructions (as `getstatic`, `putstatic` and `invokestatic`) to call malicious byte codes, this behavior is allowed by the Java Card specification. In a malicious CAP file, the parameter of an `invokestatic` instruction may redirect the Control Flow Graph (CFG) of another installed applet in the targeted smart card. The EMAN2 [9] attack was related to the return address stored in the Java Card stack. They used the unchecked local variables to modify the return address, while Faugeron in [14] uses an underflow on the stack to get access to the return address.

Hamadouche et al. [19] described various techniques used for designing efficient viruses for smart cards. The first one is to exploit the linking process by forcing it to link a token with an unauthorized instruction. The second step is to characterize the whole Java card API by designing a set of CAP files which are used to extract the addresses of the API regardless of the platform. The authors were able to develop CAP files that embed a shellcode (virus). As they know all the addresses of each method of the API, they could replace instructions of any method. In [34], they abuse the on board linker in such a way that the application is only made of tokens to be resolved by the linker. Knowing the mapping between addresses to tokens thanks to the previous attack, they have been able to use the linker to generate itself the shellcode to be executed.

Since the card embeds an on-card BCV, Savary et al. [36] proposed to generate vulnerability tests to discover failures in the BCV implementation. This approach is based on the use of formal methods, model transformation and model-based testing. They shown in [35] the feasibility of their approach on a smart card product.

Logical attacks are the simplest (few knowledge and no specific hardware) and thus affordable. Unfortunately, it gives only access to the EEPROM, thus the implementations of cryptographic algorithms and their associated counter measures still remain confidential. Nowadays, all sensitive data are stored into cryptographic containers and such an attack can affect only confidentiality of the code of the post-issuance applets.

2.4 Dumping the EEPROM memory

The EMAN2 [9] attack allows to modify the value of the return address of a method by storing a short into a local. By choosing the right value for the local number, the return address can be overwritten. In a given, card the return address register is stored at `MAX_LOCAL + 2`. The value stored in this register will be the address where Java Program Counter (JPC) will be updated while returning from the current method to the caller. We just need to define a static array which is stored close to the method area. Then, after returning from the method, the JCVM will execute the content of the array. Due to the fact that `getstatic` and `putstatic` instructions are not checked by the Firewall [23], we are able to read the content of the memory. The shellcode is presented in Listing 1.

```
7C 01 00   getstatic_b   0x0100
78         sreturn
```

Listing 1 Executing the basic shellcode

This piece of code pushes on the top of the stack, the content of the memory from the address `0x0100` and returns this value. The caller has just to store it into the APDU buffer and the value is sent to the terminal. Then, the third byte of the static array (the low byte of the `getstatic_b` instruction parameter) must be incremented and the next call will return the value of the address `0x0101`. We just need to manage the carry from the low byte to the high byte representing the address. Unfortunately, this method stresses the memory and will need more than 65,000 writing to the same cell. So 10 or 20 executions of this shellcode will kill the card reaching the stress threshold of the EEPROM. We need to have a smarter shellcode. For that, we purpose to use transient array.

A transient array is an array where the data are stored in RAM and its descriptor is stored in EEPROM. Thus a transient array lost its content during power off but not the reference, there is no *natural* garbage collection. Unlike the EEPROM, one can write indefinitely in RAM area. So, using a transient array is better to dump RAM and EEPROM parts to avoid memory stress. To understand how a transient array is stored in the smart card, we created a simple applet which gets the transient array address and reads data at this address.

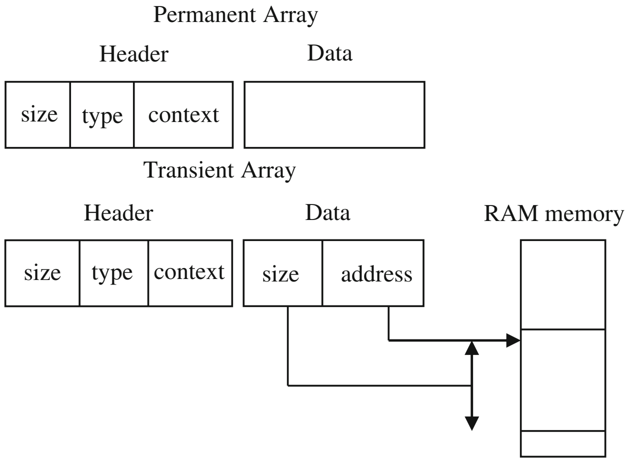

Fig. 1 Structure of transient and permanent arrays

The header of a transient array is the following at the EEPROM area address:

```
0x8E85: 0x00 0x04 0x5B 0x30 0x6C 0x88
        0x00 0x0A 0x05 0xB9 ...
```

Where `0x0004` is the size of the structure without the meta data corresponding to the header. In the header part the byte `0x5B` corresponds to the array type. Here, its data is of type byte. The three next bytes corresponding probably to the security context `0x30 0x6C 0x88`. It remains the four last bytes as pseudo data. After several trials, we understood that `0x000A` represents the size of the data in RAM and `0x05B9` its address as shown in the Fig. 1.

Since we are able to write wherever we want on the card, we could edit the size and the address of our array in order to cover from `0` for `0xFFFF` bytes, i.e., the whole memory. This is perfectly accepted by the card and we can use such a transient array to read the complete memory. This manipulation seems to work while the size of the array is less than `0x00FF`. The array length should be coded on a byte.

Using a transient array is more efficient than the regular EMAN2 attack: we need to write only few bytes in memory to obtain an array which can be read normally.

```
18 FF      sspush 0x00FF
80 8E 9B   putstatic_s 0x8E9B //size: 0x00FF
18 00      sspush 0x00FE
80 8E 9D   putstatic_b 0x8E9D //address: start from 0x00FE
7A         return
```

Listing 2 Executing the basic shellcode

Once this shellcode is executed, we have to copy the array in the APDU buffer slicing it into slots of 255 bytes to fit the size of the APDU buffer. Unfortunately these arrays could not be used to read the ROM. The values returned at these addresses are filled with 0. With the EMAN2 attack, the dumping shellcode needs to write around 65,000 times into a particular cell. Now, we have improved the dump with only one write into each cell, which improved greatly the execution time[2] (the time needed to dump is divided by 7) and minimized the memory stress.

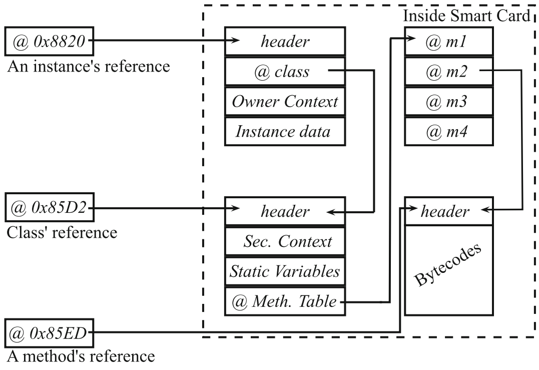

Fig. 2 Representation of an applet stored in a Java Card memory

3 EMAN 3: all roads lead to the ROM

The main attack that we have used was the EMAN2 attack [23] improved by using transient arrays instead of permanent arrays. It gave us the ability to execute arbitrary code in the EEPROM and to dump the memory. As we succeeded this step, it is necessary to analyze and to determine the exact nature of each element contained in the memory, in order to display the memory dump in a more understandable way. Since there is no existing tool, we developed the disassembler for Java Card memory dump. Indeed, the Java Card byte codes are different from Java byte code.

For rest of this paper, a memory dump of a smart card with Java Card 2.1, GP 2.0 and 8-bit processor will be analyzed. This targeted card runs with 5MHz CPU and has 32 KB of ROM, 32 kB of EEPROM and 2 kB of RAM.

3.1 Storing a Java card applet into the smart card

The CAP file format is based on the notion of interdependent components. It is specified in [30] and they consist of eleven standard components: Header, Directory, Import, Applet, Class, Method, Static Field, Export, Constant Pool, Reference Location and Descriptor. Moreover, the targeted JCVM may support user Custom components. Tokens resolution is performed in the Method component. Each component is stored in memory by the Java Card Linker. As explained in [23], the smart card memory can be structured like in the Fig. 2.

[2] Writing in EEPROM needs to erase which is time consuming.

Table 1 Array with methods call

0xDBC4	21 01 32	0xE681	21 00 3F
0xDBC7	24 00 33	0xE684	21 00 40
0xDBCA	24 00 34	0xE687	21 00 41
0xDBEA	22 01 35	0xE68A	21 00 42
0xDBF9	22 00 36	0xE024	22 00 43
0xDF7D	21 02 37	0xE69C	21 02 44
0xE66C	24 01 38	0xE69F	21 03 45
0xE66F	24 00 39	0xE6A2	22 01 46
0xE672	21 00 3A	0xF251	24 01 47
0xE675	24 00 3B	0x96BC	23 00 48
0xE678	24 00 3C	0xF32D	22 01 49
0xE67B	22 00 3D	0xF330	22 02 4A
0xE67E	24 04 3E	0xF7B5	24 02 4B

3.2 Unexcepted memory behavior

During the analysis of each linked applets into the smart card memory, a method with a unexcepted call has been noticed at the address 0xDBE6 listed in Listing 3.

```
0xDBE6: 01 // flags: 0 max_stack : 1
0xDBE8: 00 // nargs: 0 max_locals: 0
0xDBE9: 8D DB C7   invokestatic DB C7
0xDBEC: 67 08      ifnonnull    08
0xDBEE: 11 6F 00   sspush       6F 00
          // ISOException.throwIt(0x6F00);
0xDBF0: 8D 6F 05   invokstatic  6F 05
0xDBF3: 7A         return
```

Listing 3 Calling a method stored in EEPROM at the address 0xDBE6

At 0xDBE9, the invokestatic instruction refers to a method in the EEPROM area. This method is located[3] at 0xDBC7. At this address several non specified methods have been found out and are given in the Table 1.

The JCVM specification [30] defines a method as a method_header_info, described in the Listing 4, and its associated byte code.

For the flag value, three defined possibilities are expected:

- 0x0: it is a normal method;
- 0x8 (ACC_EXTENDED): the method represents an extended method;
- 0x4 (ACC_ABSTRACT): the method represents an abstract method;
- All other flag values are reserved.

```
method_header_info {
  u1 bitfield {
    // a mask of modifiers defined
    // for the method
    bit[4] flags
    // max cells required during execution
    // of the method
    bit[4] max_stack
  }

  u1 bitfield {
    // number of parameters passed to
    // the method
    bit[4] nargs
    // number of local variables declared
    // by the method
    bit[4] max_locals
  }
}
```

Listing 4 Java Card method header info

Each method listed in the Table 1 contains non standardized flag value (0x2). Moreover, the associated byte code (1-byte) cannot be an instruction. On the other side, we also have a set of interesting values in the EEPROM part which are given in Table 2. We assumed that all these values are addresses that refer to ROM area, except the one in bold font which refers to the EEPROM.

To prove our hypothesis, we checked the data contained at the address 0xFF5C to understand the meaning of this table. As we have seen in the Lisiting 13 (Appendix A), this method is associated to a 8051 assembler language which corresponds to the native code for the targeted card.

3.3 The indirection table

In order to access to the JNI or to execute updated functions, cards have vendor's implementation of an indirection table. This kind of table is used to switch between the Java Card land and native land. For that, we supposed to have an invokestatic instruction to a method with a flag value 0x2. This non standardized method has an offset to the indirection table which gives the address of the associated native method and is described in Fig. 3.

3.4 Dumping the ROM area

We explained in the previous section, how the JCVM executes native code using an indirection table and a non standardized method with a flag value 0x2. In this section, we explain a way to abuse this mechanism to execute our native code and an associated countermeasure. To perform this attack, we assume that we are able to write in the memory

[3] The targeted card has not hidden mechanism for address.

Table 2 List of addresses in the EEPROM

	0x0	0x2	0x4	0x6	0x8	0xA	0xC	0xE
0x8060								0x5800
0x8070	0x7E84	0x6ADC	0x6AED	0x6AFE	0x1800	0x7F08	0x7F29	0x7F02
0x8080	0x7F24	0x7EFC	0x5E79	0x47AD	0x6732	0x6B85	0x49DD	0x68BD
0x8090	0x5F9F	0x5DC9	0x631D	0x4638	0x5EA5	0x7E01	0x0FEB	0x6915
0x80A0	0x6C22	0x68A7	0x5FEF	0x6B0F	0x6B20	0x6B31	0x6B42	0x6B53
0x80B0	0x7EF0	0x38BE	0x62D9	0x5767	0x6B64	0x4EA3	0x55DA	0x7F31
0x80C0	0x7F46	0x5208	0x378F	**0xFF5C**	0x6515	0x5CE5	0x7EF6	0x67C7
0x80D0	0x7F1A	0x63A0	0x6732	0x49DD	0x7EA2	0x61E4	0x641F	0x67AF
0x80E0	0x37FB	0x692A	0x6732	0x17FB	0x7E7A	0x487C	0x1686	0x7DE9
0x80F0	0x7F35	0x7EEA	0x7F1F	0x5AEA	0x2AAC	0x7D9C	0x7EE4	0x7D8F
0x8100	0x61E4	0x67F7	0x2797	0x64D9	0x6000	0x009C	0x009E	0x0000

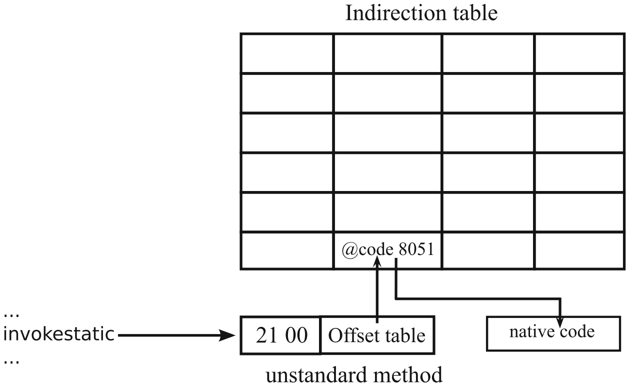

Fig. 3 Indirection table usage

and we can install an applet on the targeted card. For that, we used a development card.

As described in the Fig. 3, a *fake method* (a method with a flag value equals to `0x2`) contains an offset to an address in the indirection table. Each element in the indirection table refers to a native function. At this offset we store the address of our shellcode. Without integrity check, the Java Card runtime will execute the malicious code. According to the card architecture, the native code is written in 8051 language.

To do that, we split this modification into three steps. First, we create two Java Card arrays. The first array has a *fake method* and an offset to the indirection table. It will refer to the address of the shellcode. The other array contains our malicious native code. Then, in the indirection table, we write the address of the native shell code. We should know the offset associated to the shellcode location in order to shift it into the *fake method*. To know the array address of the smart card memory, an attack like [23] can be done. Finally, to execute the native shellcode the parameter of an `invokestatic` instruction, or another kind of call instruction should be changed by the address of our *fake method*.

Table 3 Cards used during this evaluation

Reference	Java Card	GP	Characteristics
a-21a	2.1.1	2.0.1	
a-22b	2.2	2.1	72kB EEPROM
b-22a	2.2.1	2.1.1	36kB EEPROM, RSA
b-22b	2.2.2	2.1.1	72kB EEPROM, RSA
b-21c	2.1.1	2.1.2	16kB EEPROM, RSA
c-21a	2.1	2.0.1	32KB EEPROM, RSA
c-22b	2.2.1	2.1.1	16kB EEPROM

Thus, the faulty instruction provides a way to execute our shellcode which set as the Listing 14 in the Appendix B. This piece of 8051-code reads each byte contained in an address range. This range shall be in the ROM or EEPROM memory. Each read byte is copied into the APDU buffer. With this shellcode, we are able to do a memory dump for ROM.

4 Evaluation: to a generic exploit?

To evaluate our approach, we try our attack on different smart cards from different manufacturers. The evaluated cards are available on public internet shops. We evaluated seven cards from three distinct manufacturers (a, b and c). Each card name is associated with the manufacturer reference and its Java Card specification. Moreover, we hypothesized that each evaluated card supports the native methods execution through the JCVM. The list of evaluated Java Card is presented in the Table 3.

At each card, we sent a CAP file which contains the method shown on the Listing 5. This method contains its FLAG value set to `0x2` and 1-byte. We supposed that he byte is interpreted as an offset into the indirection table, so the first native method will be called.

```
methodNative() {
  21 // flags: 2 max_stack : 1
  01 // nargs: 0 max_locals: 1
// the offset value used to the indirection table
  00 NOP
}
```

Listing 5 A fragment of a dumped Java Card memory.

Table 4 Each tested card's return value for the native method execution

Reference	Return value
`a-21a`	Status word: `0x6F00`
`a-22b`	PCSC error: `SCARD_E_NOT_TRANSACTED`
`b-22a`	Status word: `0x9000`
`b-22b`	Status word: `0x9000`
`b-21c`	Status word: `0x6F00`
`c-21a`	Status word: `0x6F00`
`c-22b`	PCSC error: `SCARD_E_NOT_TRANSACTED`

We executed the code contained in the Listing 5. The return value of each card is presented into the Table 4.

In this table, three different values are returned:

- The status word `0x9000`: the execution was done without error.
- The status word `0x6F00`: a Java exception was thrown and never caught.
- A PCSC error (`SCARD_E_NOT_TRANSACTED`): The smart card has an unexpected behavior during the execution of a command and the exchange is stopped by the card.

Since a error value is returned, we cannot conclude anything.

On the Table 4, two cards (`b-22a` and `b-22b`) return the status word `0x9000`. We focused on the card `b-22b`. To improve our analysis, we call the `arrayCopy()`[4] function. This function is provided by the class `Util` from the package `javacard.framework`. It may be developed in C language. With a script, we searched the offset value required to call the native method which returns the expected value. After testing each value of the offset range `0x00` to `0xFF`, the expected return was not be found. So, we investigate and discover that the card uses a different strategy to call native methods.

After an investigating step, we discovered that the card `b-22a` encodes the native method offset into a `0x02 flag` method header, inside the fields `nargs` and `max_locals` (cf Listing 4). Inside this card, we succeed to call a native method which changes the smart card's life cycle. Since the native method refereed by the offset `0x0F` is called, the smart card shifts to the production mode without specially right.

In this card, we have a proof that we are able to call native piece of code. However, we had not succeed to locate where the indirection table is stored. It may be masked by a xor operation.

5 Java Card memory forensics

Once we obtained the ROM, we have to face a challenge, defining the area that are related to native code and the area related to Java Card byte code. Currently, the reverse of the dumped smart card memory is manually analyzed. Of course, it's a difficult task (human errors are frequent) and long (an exhaustive result needs some days or weeks). To automatize this analyze, the Java Card Disassembler and Analyser (JCDA) based on the index of coincidence has been developed to reverse the Java Card memory. A *dump file* contains a set of binary values which represents a fragment of the smart card memory. Into the smart card memory, the program's code either Java or native and data, object's instance information can be found and some sensitive information can be retrieved.

5.1 A memory dump

In the Listing 6, a fragment of a dumped Java Card memory is presented which corresponds to 88 bytes fragment of the EEPROM and start from the logical address `0x13F8`.

```
/*0x13F8*/ 00 0B 81 00 0A 48 65 6C
/*0x1400*/ 6C 6F 57 6F 72 6C 64 00
/*0x1408*/ 00 02 80 00 00 03 04 02
/*0x1410*/ 0C 34 00 00 01 BE 81 08
/*0x1418*/ 00 0a 00 19 00 25 00 01
/*0x1420*/ 2E 00 01 0D 48 65 6C 6C
/*0x1428*/ 6F 57 6F 72 6C 65 41 70
/*0x1430*/ 70 01 71 00 02 34 04 00
/*0x1438*/ 04 06 02 00 00 01 73 01
/*0x1440*/ 75 00 05 42 18 8D 08 97
/*0x1448*/ 18 01 87 06 18 01 87 07
/*0x1450*/ 18 08 91 00 07 87 08 18
/*0x1458*/ 01 87 09 1e 29 04 03 29
```

Listing 6 A fragment of a dumped Java Card memory

A reverse version of the dump is listed in the Listing 7. The first part of the analyzed dump contains metadata information (package and class information). The second part describes the byte code of each class methods.

[4] This function returns a value depending to the start offset of the output buffer plus the length of the copied data.

```
0x13F8: 000B 81 00 // Array header:
                   // data size: 0x000b,
                   // type: 0x81, owner: 0)
           // PACKAGE_AID
           0A 48 65 6C 6C 6F 57 6F 72 6C 64
        // Unknown data
0x1408: 00 0002 8000 0003 0402 0C34 0000
0x1414: 01BE 81 08 // Array header:
                   // data size: 0x01be,
                   // type: 0x81, owner: 08)
        // Undefined values
        000a 0019 0025 0001 2E00 010D
        // APPLET_AID
      48 65 6C 6C 6F 57 6F 72 6C 64 41 70 70
        // Undefined values
        01 71 00 02 34 04 00 04 06 02 00 00
        01 73 01 75 00
0x1442:   /* method 00: */
          // Method's header
          05 // flags: 0 max_stack: 5
          42 // nargs: 4 max_locals: 2
          // Method's byte code
          18       // aload_0
          8D 08 97 // invokestatic 0x0897
          18       // aload_0
          01       // aconst_null
          87 06    // putfield_a 06
// To be continued ...
```

Listing 7 The reverse of the values listed in the Listing 6

In this sample, the byte codes are used as defined into the Java Card specification [30]. Depending on the card model, the program's code may be modified or scrambled [4,34]. In this case, the card hides each instruction's value to mask the code. But, even with any masking or encryption, the program stored in the memory always keeps the same semantics. On the other side, the way to store the data depending to a proprietary implementation. Due to the fact that disassembling is done without knowledge of the card dumped, reversing was difficult.

5.2 Index of coincidence

In 1922, Friedman [16] invented the notion of Index of Coincidence (IC) to reverse ciphered messages. In cryptography, this technique consists to counting the times that identical letters appear in the same position in both texts. This count, either as a ratio of the total or normalized by dividing by the expected count for a random source model. The IC is computed as defined in the Eq. 1.

$$IC = \frac{\sum_{i=1}^{c} n_i(n_i - 1)}{N(N-1)/c} \tag{1}$$

where N is the length of the analyzed text and n_i is the frequencies of the c letters of the alphabet (c = 26 for a Latin alphabet). The IC is mainly used in the analysis of natural-language text and in the analysis of ciphered message (as cryptanalysis). Even when only ciphered message is available, the coincidences in the ciphered text can be caused by coincidences in the plain message. This cryptanalysis technique is mainly used to attack the Vigenere cipher, for instance. For a natural-language, the IC for the French language is 0.0778, for the English language, 0.0667 and for German language is 0.0762.

5.3 Finding Java Card byte codes

In a Java Card dumped memory, it is very difficult to find where are the program's data and code. The program's byte code can be assimilated to a language where each instruction has a precise location into the language's grammar.

The Java Card toolchain ensures that the compiled Java Card byte codes is compliant with the rules of Java language. To determine the IC for the Java Card byte codes, we tested a set of Java Card byte code compiled by the Oracle's toolchain. An acceptable index of coincidence for Java Card byte codes is located between 0.009 and 0.025. In addition to the IC, we defined two counters that will be used during the research of Java Card byte codes. The first one counts the incorrect Java Card instructions. A Java Card instruction is specified [30] inside the range `0x00` to `0xB8`. Outside this range, the byte read is not a standard byte code value. The second counter counts the `NOP` instruction. The `NOP` instruction does nothing and is never used by the Oracle's compiler. So, except a handy-written byte code, a Java Card program has no `NOP` instruction.

In the Listing 6, a fragment of a dumped memory is analyzed to determined if there is Java Card bytes and where. Our analysis shown that the IC equals 0.00927, 1.87 % bytes are unknown instructions and 3.30 % of the byte corresponds to a `NOP` instruction. We are 90 % sure that this piece of the dumped memory contains Java Card byte codes.

```
/*0x85E0*/ 86 85 86 8C 85 ED 85 F8
/*0x85E8*/ 86 04 86 04 00 03 21 10
/*0x85F0*/ AA 31 19 03 02 39 1E 78
/*0x85F8*/ 01 10 7C 00 02 78 01 00
/*0x8560*/ 7C 00 02 78 01 21 1D 00
/*0x8568*/ 80 87 7D 78 01 21 1D 00
 ...
/*0x8690*/ DA 13 83 21 95 A7 83 21
/*0x8698*/ 87 F8 88 9E 00 00 10 00
/*0x8700*/ 00 02 41 FE 6D C5 91 BB
/*0x8708*/ 00 02 46 FE 6C 88 DA 42
```

Listing 8 A fragment of a dumped Java Card memory.

5.4 How to find data in a Java Card dumped memory?

Instead of the byte code search in a dump memory, searching data cannot be predictable without a pre-analysis realized by the attacker. By a learning process based on the pattern matching, the attacker should discover himself how the card stores each element in its memory. In the smart card world, any kind of data stored in the smart card contains a header (or metadata) which describes the data type, data owner and, sometimes, the size of the data. Into a Java Card, the data can be:

- A package information. A package AID and all contained classes should be saved to verify the ownership context during the applet execution.
- A class information. A class contains the initialization value of each field, the method's byte code and an AID. A fragment is presented, with its package information, in the Listing 7.
- The class instance which referees each field sets to the current value (regarding to the instance's life). The class instance is linked with an instance AID which is different compared to the class AID.
- An array is the last element which contains a header. Empirically, we have discovered that an array header is structured by the size of the array data, the data type and the owner context. A example of Java Card array found in a dumped memory is shown in the Listing 9.

```
0010   // Data size
81     // type of the data,
       // it's a byte array
08     // applet's owner context
/* Data */
CA FE CA FE CA FE CA FE CA FE CA FE CA FE CA FE
```

Listing 9 A Java Card array found in the dumped memory

6 JCDA: Java Card disassembler and analyzer

To disassemble Java Card dumped memory, we have developed a Java-tool, named JCDA, which need to be adapted to each Java Card memories architecture. To reverse of a Java Card dumped memory, the JCDA (Fig. 4) requires a card model and a *dump file* to work. The first one defines the structure of the data contained in the dumped smart card memory. This model is a high level abstraction of how the smart card stores objects and associated instances, array, etc. in the memory dump. This file should be created by the attacker. The second parameter, is a piece of an attacked Java Card memory.

In our tool, the reverse a Java Card dump memory is done though two steps. In a first time, the Java Card Analyzer parses the Java Card dumped memory to locate Java information: data and applet's code. As described previously, to find Java Card instructions, an automatic process based on the index of coincidence is done. For the Java Card data, the card memory model is used to search by pattern matching classes information, tables and other data.

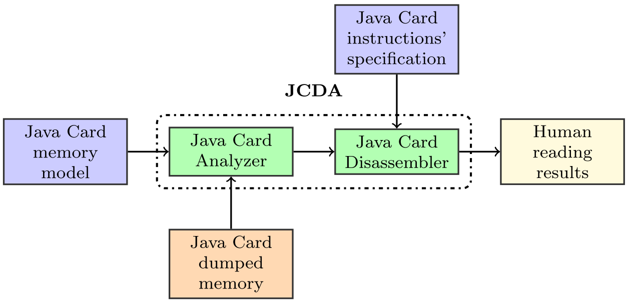

Fig. 4 JCDA architecture

Table 5 Java Card instructions specification

Byte Code	Parameters number	Stack's state
`0x1d sload_1`	0	`.. -> value_1`
`0x78 sreturn`	0	`value -> [empty]`
`0x8d invokestatic`	1 (2-byte parameter)	`[args] -> ..`
`0x96 sinc_w`	2 (1-byte parameter and 2-byte parameter)	`.. -> ..`

Since the applet's code and the data located in the dumped Java Card memory have been identified, the second part of the JCDA starts. The Java Card disassembler performs a reverse of each applet installed in the dumped memory. A Java Card instruction's specification is required to correctly disassemble the program's byte code. Into this specification, each instruction is defined as its byte code value, its parameters number, the parameter's size and the stack's state of each instruction. A part of Java Card instructions specification is presented in the Table 5.

Next, the disassembler step realizes a symbolic execution of each detected method. This way allows to determine the Java Card method's CFG and its stack's state for each Java-instruction. Finally, the human reading results are reported with the elements found in the Java Card dumped memory. Currently, the JCDA outputs a HTML file which reports tables, applet's AIDs, and the reverse of the program's byte codes. A report fragment is shown in the Table 6. This report corresponds to the memory dump refereed in the Listing 8.

To improve our tool, we will integrate it in IDA Disassembler [20]. IDA is a software which has been developed by the Hex Rays company and it implements all features required to reverse a computer application. This software is mainly used by security laboratories.

Table 6 A fragment of the JCDA output

Address	Instructions			Operand stack
0x85ED	0x03	*// flags: 0, max_stack: 3*		
0x85EE	0x21	*// nargs: 2, max_locals: 1*		
0x85EF	0x10AA	bspush	0xAA	[]
0x85F1	0x31	sstore_2		['0xAA']
0x85F2	0x19	aload_1		[]
0x85F3	0x03	sconst_0		[ref]
0x85F4	0x02	sconst_m1		[ref,'0']
0x85F5	0x39	sastore		[ref,'0','0xFF']
0x85F6	0x1E	sload_2		[]
0x85F7	0x78	sreturn		[value_2]=>[empty]

7 Reversing the code

In the previous sections, we have shown how to obtain an efficient dump of the EEPROM. Then, we explained how to obtain the dump of the ROM. We have developed a tool to help us in retrieving some of Java objects, how to distinguish native code and Java code. In this section, we try to reverse some functions of the ROM.

7.1 Getting the entry points of the API

To reverse the smart card operating system, the JCDA can exploit the way described by Hamadouche et al. [19]. Their attack was described in the Subsect. 2.3. Thus, for a specific card model, the JCDA can reverse the API functions based on the API addresses given as input. An example of the kind of input is shown in the Table 7.

Table 7 `javacard.framework` functions addresses

Function's name	Address
...	...
JCSystem.abortTransaction	0x6FC9
JCSystem.beginTransaction	0x6FCC
JCSystem.commitTransaction	0x6FCF
JCSystem.getAID	0x6FD2
JCSystem.getAppletShareableInterfaceObject	0x6FE2
JCSystem.getMaxCommitCapacity	0x701C
JCSystem.getPreviousContextAID	0x701F
JCSystem.getTransactionDepth	0x702F
JCSystem.getUnusedCommitCapacity	0x7032
JCSystem.getVersion	0x7035
JCSystem.isTransient	0x703B
JCSystem.lookupAID	0x703E
JCSystem.makeTransientBooleanArray	0x7053
JCSystem.makeTransientByteArray	0x7060
JCSystem.makeTransientObjectArray	0x706D
JCSystem.makeTransientShortArray	0x707B
...	...
Util.arrayCompare	0x7358
Util.arrayCopy	0x735B
Util.arrayCopyNonAtomic	0x735E
Util.arrayFillNonAtomic	0x7361
Util.getShort	0x7364
Util.makeShort	0x7372
...	...

This table gives the address of some functions contained into the package `javacard.framework`. The functions with a gray foreground are stored in the 8051-assembler area into the analyzed dump.

7.2 Reversing Java based methods

Based on the Table 7, we reverse each Java function provided by the Java Card API. As example, we will reverse the `makeTransientShortArray` function. From the address `0x707B`, the bytes corresponding to the method. Next, we reverse it. As we seen in the Listing 10, it's a Java method.

```
static short[] makeTransientShortArray
    (short length, byte event) {
/*0x707B*/ 03 // flags: 0 max_stack: 3
/*0x707C*/ 20 // nargs: 2 max_locals: 0
/*0x707D*/ 1C      sload_0
/*0x707E*/ 1D      sload_1
/*0x707F*/ 07      sconst_4
/*0x7080*/ 8D 6FC6 invokestatic 0x6FC6
/*0x7083*/ 94 0C00 checkcast    0x0C00
/*0x7085*/ 77      areturn
}
```

Listing 10 Byte code of function `makeTransientShortArray`

This function corresponds to the Java code listed into the Listing 11.

```
static short[] makeTransientShortArray
    (short length, byte event) {
  short[] ret =
     createTransientArray(length, event, 4);
  return ret;
}
```

Listing 11 Source of `makeTransientShortArray` function

In this method, an external function is called. This function, located at address `0x6FC6`, corresponds to the following bytes: `0x240305`. As previously explained, a method flagged with 0x02 is a native method. So, the `createTransientArray` function will be executed in the native side.

7.3 Reversing native methods

Reversing the native part is a more difficult task. Indeed, we do not know the initialization vector of the 8051 processor.

The best way that we found is to start the reverse by the Java-functions listed into the indirection table.

On the previous example, the function listed in the Listing 10 calls a native method with the token `0x05`. The fifth element into the indirection table refers to the address `0x6FCE`. At this address, the code presented into the Listing 12 is stored.

```
/*0x6AFE*/ E4          clr    a
/*0x6AFF*/ FF          mov    r7, a
/*0x6B00*/ 126C04      lcall  0x6C04
/*0x6B03*/ AC          mov    r4, 0x06
/*0x6B04*/ 06AD        inc    @r0
/*0x6B06*/ 07          inc    @r1
/*0x6B07*/ 7FD9        mov    r7, 0xd9
/*0x6B09*/ 7E          mov    r6, 0x6d
/*0x6B0A*/ 6D          xrl    a, r5
/*0x6B0B*/ 1265C7      lcall  0x65C7
/*0x6B0E*/ 22          ret
```

Listing 12 `createTransientArray()` function

This native function calls two other code fragments to create a transient array. The first method computes where the transient array will be stored in the RAM memory. The second one, reserves, in the memory, the space for the array.

8 Related works

8.1 Side channel origins and analysis

In order to gain access to services or assets stored on the card, several means have been used to retrieve valuable information, and side channel analysis or fault injection appears to be the most efficient. The major difficulty is to clearly identify operations or data manipulated. Side channel analysis can be used in many situation for the same. Side channel leakage is correlated with electrical devices power consumption.

8.1.1 Power analysis attack

Power analysis involves interpreting power traces, or graphs of electrical activity over time.

Timing attack [24]. For example, it was used to get knowledge on RSA keys during the "Square and Multiply" step of modular exponentiation. It was based on the analysis of the time difference between two same subroutine executions using two different data entry. Attackers were able to easily read key bits during exponentiation.

Simple and differential power analysis [25]. These two well-known attacks were developed to exploit the differences between two signals in order to extract information. The first one shows that data processing is influenced by computation as in power consumption and the other one improves the signal clarity by characterizing it.

Correlation power analysis [10]. It is based on statistical models and uses a consumption model based on the Hamming distance or weight. A correlation coefficient value is used to evaluate a guessed proposition.

8.1.2 Electromagnetic attacks [17,28,32]

The magnetic traces are obtained by positioning an electromagnetic probe at a precise location, and also by varying the value of the data stored at this location. This is more precise and gives better results than the power analysis methods.

8.1.3 Temperature analysis [21,27,41]

The thermal difference gives a spacial information on where specific operations are executed as any resistive electrical system generate heat. For instance, to retrieve where the static tables used by the encryption algorithms are stored.

8.2 Reverse engineering methods

Side channel analysis for revere engineering (SCARE) method [11,13]. It manages to understand and highlight some specific implementation depending on the monitored system. It insists on the fact that observing the power consumption by clock cycle can provide priceless information on the targeted system.

Fault injection for reverse engineering (FIRE) attacks are using faults injected in embedded systems by various methods (light flashes, electromagnetic impulses, etc.) in order to gather informations on the secure algorithm implemented in the device. There are some studies of partial data recovering by means of faults but, as far as we know, there is only one reverse engineering of a full algorithm [12].

This method is a combination of an efficient classification tool and a convex form comparison, which is able to evaluate the probability for an instruction, executed on a CISC[5] architecture, correspond to a given power trace. It offers a complete solution for reverse-engineering code even if it seems to be complicated to adapt it on other architectures.

[5] Complex instruction set computer (CISC) is an architecture where each instruction can be executed with several low-level operations.

9 Conclusion

In this paper, we presented an attack to execute native code into a Java Card based smart card. For that purpose, we described how to obtain a smart card with a new attack which exploits the transient array. On the obtained dump, we are able to identify the indirection table, a table used to redirect the called Java Card method to its native implementation. Fool this element offers us the way to execute our native code.

After the evaluation of our attack with other cards, we discovered that the attacked element is sometime implemented in the same way for the different card manufacturer. Because the indirection table was not found, we are not able to exploit this attack on other cards.

This attack offers the ability to read the content of the ROM area. When a snapshot of the memory was obtained, we have reversed the ROM area in order to obtain the operating system code.

Acknowledgments The authors would thank to Julien Boutet for his contribution during this work.

10 Native code in the EEPROM area

```
FF5C               lcall  code_5A46
FF5F               jnc    code_FF9A
FF61               clr    C
FF62               mov    A, RAM_3F
FF64               subb   A, #0x80 ; 'Ç'
FF66               jnc    code_FF76
FF68               mov    R7, RAM_40
FF6A               mov    R6, RAM_3F
FF6C               mov    R4, RAM_76
FF6E               mov    R5, RAM_77
FF70               mov    R3, RAM_44
FF72               lcall  code_3F6B
FF75               ret

FF76 code_FF76:    ; CODE XREF: code:FF66
FF76               setb   RAM_20.0
FF78               jnb    RAM_20.2, code_FF8B
FF7B               clr    A
FF7C               mov    R7, A
FF7D               lcall  code_6C04
FF80               mov    RAM_45, R6
FF82               mov    RAM_46, R7
FF84               mov    R7, #0x45 ; 'E'
FF86               lcall  code_3CE3
FF89               mov    RAM_20.0, C

FF8B code_FF8B:    ; CODE XREF: code:FF78
FF8B               mov    R7, RAM_40
FF8D               mov    R6, RAM_3F
FF8F               mov    R4, RAM_76
FF91               mov    R5, RAM_77
FF93               mov    R3, RAM_44
FF95               mov    R2, RAM_43
FF97               lcall  code_2961

FF9A code_FF9A:    ; CODE XREF: code:FF5F
FF9A               ret
```

Listing 13 Native code found in the EEPROM

11 Native code to dump ROM area

```
mov    DPTR,  @tab      ; Address of the transient array
                        ; which contains the dump of the ROM
;; Save the A, R0 and R1 registers
movx   @DPTR, A         ; A is saved
mov    DPTR,  @tab+1    ;
mov    A,     R0
movx   @DPTR, A         ; R0 is saved
mov    DPTR,  @tab+2
mov    A,     R1
movx   @DPTR, A         ; R1 is saved
mov    R0,    1         ; R0 is used as offset

LOOP1:    ;; main loop which dump the ROM
   mov    A,     R0
   mov    R1,    A              ; Copy A in the R1 register
   mov    DPTR,  @ROM           ; Copying the first address of
                                ; ROM to dump
   movc   A,     @A + DPTR      ; Increasing the first ROM
                                ; address by A
   mov    DPTR,  @BUFFER_APDU   ; Moving the address of the
       APDU
                                ; buffer in the DPTR register

   LOOPINC:   ;; Set DPTR as an index into the APDU buffer
     inc   DPTR              ; Increase DPTR register
     djnz  R1, LOOPINC       ; Decrease R1 while it is > 0

   movx    @DPTR, A          ; the value read in ROM is put
                             ; in the APDU buffer

   inc    R0                 ; increasing the R0 register
   mov    A,     R0          ; Setting the offset of the byte
                             ; to read ROM in the next loop
   jz LOOP1                  ; Jump to LOOP1 if A ≠ 0

;; Restore the CPU registers
mov    DPTR,  @tab+2
movx   A,     @DPTR
mov    R1,    A         ; R1 is reloaded
mov    DPTR,  @tab+1
movx   A,     @DPTR
mov    R0,    A         ; R0 is reloaded
mov    DPTR,  @tab
movx   A,     @DPTR     ; A is reloaded
ret
```

Listing 14 Dump 255-byte of the ROM in 8051

References

1. Agoyan, M., Dutertre, J.M., Naccache, D., Robisson, B., Tria, A.: When clocks fail: on critical paths and clock faults. In: Gollmann, D., Lanet, J.L., Iguchi-Cartigny, J. (eds.) Smart Card Research and Advanced Application, Lecture Notes in Computer Science, vol.

6035, pp. 182–193. Springer, Berlin Heidelberg (2010). doi:10.1007/978-3-642-12510_213

2. Aranda, F.X., Lanet, J.L.: Smart card reverse-engineering binary code execution using side-channel analysis. Thorie des Nombres, Codes, Cryptographie et Systmes de Communication (NTCCCS) (2012)
3. Aumller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.P.: Fault attacks on RSA with CRT: concrete results and practical countermeasures. In: Kaliski, B., Ko, E., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems—CHES 2002, Lecture Notes in Computer Science, vol. 2523, pp. 81–95. Springer, Berlin Heidelberg (2003). doi:10.1007/3-540-36400-5_20
4. Barbu, G.: On the security of Java Card™ platforms against hardware attacks. Ph.D. thesis, Grant-funded with Oberthur Technologies and Télécom ParisTech (2012)
5. Barbu, G., Duc, G.: Java Card operand stack: fault attacks, combined attacks and countermeasures. In: Prouff, E. (ed.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, pp. 297–313. Springer, Berlin Heidelberg (2011). doi:10.1007/978-3-642-27257-8_19
6. Barbu, G., Giraud, C., Guerin, V.: Embedded eavesdropping on Java Card. In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) Information Security and Privacy Research, IFIP Advances in Information and Communication Technology, vol. 376. Springer, Berlin Heidelberg (2012). doi:10.1007/978-3-642-30436-1_4
7. Barbu, G., Hoogvorst, P., Duc, G.: Application-replay attack on Java Cards: when the garbage collector gets confused. In: Barthe, G., Livshits, B., Scandariato, R. (eds.) Engineering Secure Software and Systems, Lecture Notes in Computer Science, vol. 7159, pp. 1–13. Springer, Berlin Heidelberg (2012). doi:10.1007/978-3-642-28166-2_1
8. 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.) Smart Card Research and Advanced Application, Lecture Notes in Computer Science, vol. 6035, pp. 148–163. Springer, Berlin Heidelberg (2010). doi:10.1007/978-3-642-12510-2_11
9. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined software and hardware attacks on the Java Card control flow. In: Prouff, E. (ed.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 7079, pp. 283–296. Springer, Berlin Heidelberg (2011). doi:10.1007/978-3-642-27257-8_18
10. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, M., Quisquater, J.J. (eds.) CHES, Lecture Notes in Computer Science, vol. 3156, pp. 16–29. Springer, Berlin Hidelberg (2004). doi:10.1007/978-3-540-28632-5_2
11. Clavier, C., Isorez, Q., Wurcker, A.: Complete SCARE of AES-like block ciphers by chosen plaintext collision power analysis. In: Paul, G., Vaudenay, S. (eds.) INDOCRYPT, Lecture Notes in Computer Science, vol. 8250, pp. 116–135. Springer, berlin Hidelberg (2013). doi:10.1007/978-3-319-03515-4_8
12. Clavier, C., Wurcker, A.: Reverse engineering of a secret AES-like cipher by ineffective fault analysis. In: Fischer and Schmidt [15], pp. 119–128. doi:10.1109/FDTC.2013.16
13. Daudigny, R., Ledig, H., Muller, F., Valette, F.: SCARE of the DES. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) Applied Cryptography and Network Security, Lecture Notes in Computer Science, vol. 3531, pp. 19–33. Springer, Berlin Heidelberg (2005). doi:10.1007/11496137_27
14. Faugeron, E.: Manipulating the frame information with an underflow attack. In: CARDIS 2013 (2013)
15. Fischer, W., Schmidt, J.M. (eds.): 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013. IEEE (2013)
16. Friedman, W.F.: The index of coincidence and its applications in cryptography. Cryptographic Series. Aegean Park Press (1996)
17. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: concrete results. In: Ko, C.C., Naccache, D., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems " CHES 2001, Lecture Notes in Computer Science, vol. 2162, pp. 251–261. Springer, Berlin Heidelberg (2001). doi:10.1007/3-540-44709-1_21
18. GlobalPlatform: Card Specification. In: GlobalPlatform, 2.2.1 edn. GlobalPlatform Inc. (2011)
19. Hamadouche, S., Bouffard, G., Lanet, J.L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: Subverting Byte Code Linker service to characterize Java Card API. In: Seventh Conference on Network and Information Systems Security (SAR-SSI), pp. 75–81 (2012)
20. Hex Rays, S.: IDA Pro Disassembler and Debugger
21. Huang, H., Quan, G., Fan, J.: Leakage temperature dependency modeling in system level analysis. In: ISQED, pp. 447–452. IEEE (2010). doi:10.1109/ISQED.2010.5450539
22. Hubbers, E., Poll, E.: Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours. University of Nijmegen (2004)
23. Iguchi-Cartigny, J., Lanet, J.L.: Developing a trojan applets in a Smart Card. J. Comput. Virol. **6**, 343–351 (2010). doi:10.1007/s11416-009-0135-3
24. Kocher, P.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) Advances in Cryptology - CRYPTO'96, Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer, Berlin Heidelberg (1996). doi:10.1007/3-540-68697-5_9
25. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) Advances in Cryptology - CRYPTO'99, Lecture Notes in Computer Science, vol. 1666, pp. 789–789. Springer, Berlin Heidelberg (1999). doi:10.1007/3-540-48405-1_25
26. Kömmerling, O., Kuhn, M.G.: Design principles for tamper-resistant smartcard processors. Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology. WOST'99, pp. 2–2. USENIX Association, Berkeley, CA, USA (1999)
27. Meterelliyoz, M., Kulkarni, J.P., Roy, K.: Analysis of SRAM and eDRAM cache memories under spatial temperature variations. Comput. Aided Design Integrated Circuits Syst., IEEE Trans. On **29**(1), 2–13 (2009). doi:10.1109/TCAD.2009.2035535
28. Circuits, O., Ral, D., Guilley, S., Flament, F., Danger, J.L., Valette, F.: Characterization of the Electromagnetic Side Channel in Frequency Domain. In: Lai, X., Yung, M., D, D. (eds.) Information Security and Cryptology, Lecture Notes in Computer Science, vol. 6584, pp. 471–486. Springer, Berlin Heidelberg (2011). doi:10.1007/978-3-642-21518-6_33
29. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic fault injection: towards a fault model on a 32-bit Microcontroller. In: Fischer, W., Schmidt, J.M. (eds.) FDTC. Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013, pp. 77–88. IEEE (2013). doi:10.1109/FDTC.2013.9
30. Oracle: Java Card 3 Platform, Virtual Machine Specification, Classic Edition. Version 3.0.4. Oracle, Oracle America Inc, 500 Oracle Parkway, Redwood City, CA 94065 (2011)
31. Quisquater, J., Samyde, D.: Eddy current for magnetic analysis with active sensor. In: Proceedings of E-Smart (2002)
32. Quisquater, J.J., Samyde, D.: Electromagnetic analysis (EMA): measures and counter-measures for Smart Cards. In: Attali, I., Jensen, T. (eds.) Smart Card Programming and Security, Lecture Notes in Computer Science, vol. 2140, pp. 200–210. Springer, Berlin Heidelberg (2001). doi:10.1007/3-540-45418-7_17
33. Razafindralambo, T., Bouffard, G., Lanet, J.: A friendly framework for hidding fault enabled virus for Java based smartcard. In: Nora Cuppens-Boulahia Frédéic Cuppens, J.G.A. (ed.) Data and Applications Security and Privacy XXVI, Lecture Notes in Com-

puter Science, vol. 7371, pp. 122–128. Springer, Berlin Heidelberg (2012). doi:10.1007/978-3-642-31540-4

34. Razafindralambo, T., Bouffard, G., Thampi, B.N., Lanet, J.L.: A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks. In: Thampi, S.M., Zomaya, A.Y., Strufe, T., Calero, J.M.A., Thomas, T. (eds.) SNDS, Communications in Computer and Information Science, vol. 335, pp. 185–194. Springer, Trivandrum (2012). doi:10.1007/978-3-642-34135-9_19
35. Savary, A., Frappier, M., Lanet, J.: Automatic Generation of Vulnerability Tests for the Java Card Byte Code Verifier. In: Network and Information Systems Security (SAR-SSI), 2011 Conference on, pp. 1–7 (2011). doi:10.1109/SAR-SSI.2011.5931379
36. Savary, A., Frappier, M., Lanet, J.L.: Detecting Vulnerabilities in Java-Card Bytecode Verifiers Using Model-Based Testing. In: Johnsen, E., Petre, L. (eds.) Integrated Formal Methods, Lecture Notes in Computer Science, vol. 7940, pp. 223–237. Springer, Berlin Heidelberg (2013). doi:10.1007/978-3-642-38613-8_16
37. Schmidt, J., Hutter, M.: Optical and EM fault-attacks on crt-based RSA: Concrete results. In: Proceedings of the Austrochip, pp. 61–67. Citeseer (2007).
38. Skorobogatov, S.P., Anderson, R.: Optical Fault Induction Attacks. In: Kaliski, B., Ko, E., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2002, vol. 2523, pp. 31–48. Springer, Berlin Heidelberg (2003). doi:10.1007/3-540-36400-5_2
39. Standard, S.H.: Federal information processing standard publication# 180. US Department of Commerce, National Institute of Standards and Technology **56**, 57–71 (1993)
40. Vermoen, D.: Reverse engineering of Java Card applets using power analysis. Master's thesis, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Computer Engineering, Mekelweg 4, 2628 CD Delft, The Netherlands (2006).
41. Viraraghavan, J., Amrutur, B., Visvanathan, V.: Voltage and Temperature Aware Statistical Leakage Analysis Framework Using Artificial Neural Networks. IEEE Trans. on CAD of Integrated Circuits and Systems 29(7), 1056–1069 (2010). doi:10.1109/TCAD.2010.2049059