Subverting Byte Code Linker service to characterize Java Card API

Samiya Hamadouche
UMBB/FS/LIMOSE,
5 Avenue de l'indépendance, 35000 Boumerdès, Algeria
Email: hamadouche-samiya@umbb.dz

Guillaume Bouffard, Jean-Louis Lanet
SSD Team – XLIM/Université de Limoges
83 Rue d'Isle, 87000 Limoges, France
Emails: guillaume.bouffard@xlim.fr, jean-louis.lanet@xlim.fr

Bruno Dorsemaine, Bastien Nouhant
Alexandre Magloire, Arnaud Reygnaud
Students in Bachelor of Computer Science at Université de Limoges
83 Rue d'Isle, 87000 Limoges, France
Emails: bruno.dorsemaine@etu.unilim.fr, bastien.nouhant@etu.unilim.fr,
alexandre.magloire@etu.unilim.fr, arnaud.reygnaud@etu.unilim.fr

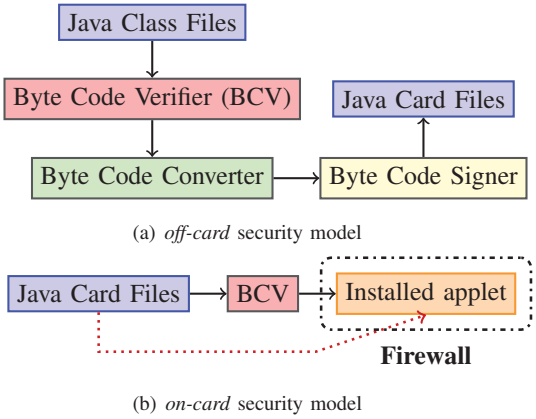
***Abstract*—Smart card is the safest device to execute cryptographic algorithms. Recent cards have the ability to download programs after issuance. These applications are verified before being loaded in the card. Recently, the idea of combining logical attacks with a physical attack in order to bypass byte code verification has emerged. For instance, correct and legitimate Java Card applications can be dynamically modified inside the card using a laser beam. Such applications become mutant applications. We propose here to go a step further in designing explicit viruses for smart cards. In order to generate efficient viruses we need to retrieve information concerning the linking process in the card. We have developed and experimented on most of the Java Card publicly available this generic attack. We present an example of virus using the result of this attack.**

***Index Terms*—Byte Code Linker, Java Card API, Characterization, Logical Attack.**

I. Introduction

Java Card is a kind of smart card which represents today most of delivered cards on the field and implements one of the two editions, "*Classic Edition*" or "*Connected Edition*", of the standard Java Card 3.0 [1]. Such a smart card embeds a virtual machine which interprets code 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 Global Platform [2]. This protocol ensures that the owner of the code has the necessary authorization to perform the action. Java Card being an open platform for smart cards, different applications from different providers run in the same smart card. Thanks to type verification, byte codes delivered by the Java compiler and the converter (in charge of giving a compact representation of class files called the CAP file) are safe, *i.e.* the loaded application is not hostile to other applications in the Java Card. Furthermore, the Java Card firewall checks access permissions between applications in the card, enforcing isolation between them.

Smart card manufacturers have issued SIM cards with NFC capability that allow downloading applications (called Basic Application) under the responsibility of the network operator. It opens new approaches to gain illegally access to secret information.



(a) *off-card* security model

(b) *on-card* security model

Fig. 1. Java Card Security Model

A. The Java Card Security Model

To install an applet on a Java Card, your applet is developed with the Java language. After being buit by the java compiler, the java `class` file is obtained. Next, to translate the `class` files into files to sent to the card, the Java Card toolchain is composed by the Byte Code Verifier (BCV), which checks the compliance to the Java Card security rules, the Byte Code Converter which translates your `class` files to the Java Card files (*i.e.* the CAP file) which may be signed. The *on-card* Global Platform layer verifies the applet signature (Figure 1(a)).

When the CAP file (Figure 1(b)), which contains the translated Java applet, is sent to the card the BCV component which

verifies its the compliance. A firewall component prevents the installed applet from access to unauthorized context.

B. The CAP File

The CAP (for Convert APplet) file format is based on the notion of components. It is specified by Oracle [1] as consisting of ten standard components: `Header`, `Directory`, `Import`, `Applet`, `Class`, `Method`, `Static Field`, `Export`, `Constant Pool` and `Reference Location` and one optional: `Descriptor`. Moreover, the targeted Java Card Virtual Machine (JCVM) may support user `custom` components. We except the `Debug` component because it is only used on the debugging step and it is not sent to the card.

Each component has a dedicated role and is linked to each other. A modification, intentional or not, of a component is difficult and may provide a meaningless file. An invalid file is often detected during the installation step by the target JCVM.

C. Mutant Applications

Java Cards have shown an improved robustness compared to native applications regarding many attacks using both physical and logical techniques. Currently, the most powerful attacks are hardware based attacks and particularly fault attacks. A fault attack modifies parts of memory content or signal on the internal bus and leads to a deviant behavior exploitable by an attacker. A comprehensive consequence of such attacks can be found in [3]. Although fault attacks have been mainly used in the literature from a cryptanalytic point of view (see [4], [5], [6]), they can be applied to every code layers embedded in a device. For instance, the attacker may change the exact byte of a program to bypass counter-measures or logical tests. We called such modified application a *mutant*.

Mutant applications are the result of the modification of the original code, but it can also turn into viruses if the attacker can hide its hostile application into a well-formed one. If such an application is transformed by a fault attack it will have the expected deviant behavior. In order to design a useful virus the attacker must have access to the methods of the API. Thanks to the design of Java Card the embedded linking process never provides information about the internal addresses of the these methods. The attack presented here aims at using the internal linking process to leak this information. Our contribution concerns mainly a generic framework to retrieve all the method addresses of any Java Card platform. The second point presents how to use this information to obtain a precise static analysis on an ability of an application to become hostile with the SmartCM framework [7].

The rest of the paper is organized as follows. First we present the state of the art concerning the logical attacks and the combined attacks. Then, we present the concept of code mutation in presence of a fault attack and how we can transform a mutant into a virus. After that, we introduce the SmartCM framework to prevent loading a fault attack enabled mutant. The next paragraph reviews some known logical and combined attacks.

II. Virus and anti-virus in the Java Card world

The design of a Java Card virtual machine cannot rely on the environmental hypotheses of Java. In fact, physical attacks have never been taken into account during the design of the Java platform. To fill this gap, card designers had developed an interpreter which relies on the principle that once the application has been linked to the card, it will not be modifiable again. The trade-off is between a highly defensive virtual machine which will be too slow to operate and an offensive interpreter that will expose too much vulnerabilities. The know-how of a smart card design is in the choice of a minimal countermeasure set with high fault coverage.

A. Logical Attacks

In this section, we will explain how to inject logical attacks into a Java Card platform. The aim of an attacker is to generate malicious applications that may bypass firewall restrictions, byte code verification and modify other applications, even if they do not belong to the same security package.

1) The Hubbers and Poll's Attack: Erik Hubbers *et al.* made a presentation at CARDIS 2008 about attacks on smart cards. In their paper [8], they presented a quick overview of the classical available attacks and gave some countermeasures. They described four methods:

1) CAP file manipulation,
2) Fault injection,
3) Shareable interfaces mechanisms abuse and
4) Transaction Mechanisms abuse

The goal of (1) is to modify the CAP file after the building step to bypass the BCV. The problem is that, like explained before, an on-card BCV is an efficient system to block this attack. Using the fault injection in (2), the authors succeed to bypass the BCV. Even if there is no particular physical protection, this kind of attack is efficient but quite difficult to perform and expensive.

The idea of (3) to abuse shareable interfaces is really interesting and can lead to trick the VM. The main goal is to obtain a type confusion without the need to modify the 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 or on loading, there is no way for the BCV to detect a problem. But it seems that every card we tried it on, with an on-card BCV, refused 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-board BCV.

The last option left is the transaction mechanism (4). The purpose of transaction is to make a group of atomic operations. Of course, it is a widely used concept, for instance in databases, but still complex to implement. 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 reference 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 otherwise protected memory. A first example is to get two arrays with different types, a byte 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 be able to read as much memory as they want. The main problem is more *how to read memory before the array?*

The other usual used confusion is between an array of bytes and an object. If Hubbers *et al.* put a byte as first object attribute, it is bound to the array length. Then, it is really easy to change the length of the array using the reference to the object. With this attack, the problem becomes *how to give a reference to an object for another object type?*

2) Barbu et al.'s Attack: Combined Physical & Logical Attack: At CARDIS 2010, Barbu *et al.* described a new kind of attack in their paper [9]. This attack is based on the use of a laser beam which modifies a runtime type check (the `checkcast` instruction). This applet was checked by the on-card BCV, considered as valid, and installed on the card. The aim is to cause a type confusion to forge a reference of an object and its content. We consider three classes `A`, `B` and `C`. They are declared in the listing 1.

```
class A {                  class B {          class C {
   byte b00, ..., bFF         short addr         A a;
}                          }                  }
```

Listing 1. Classes which create a type confusion.

The cast mechanism is explained in the JCRE specification [1]. When casting an object to another, the JCRE verifies dynamically if both types are compatible, with a `checkcast` instruction. Moreover, an object reference depends on the card architecture. The following example can be used:

```
T1 t1;                    aload @t1
T2 t2 = (T2) t1;   <=>    checkcast T2
                          astore @t2
```

The authors want to cast an object `b` to an object `c`. If

```
1  public class AttackExtApp extends Applet {
2      B b; C c; boolean classFound;
3      ... // Constructor, install method
4      public void process(APDU apdu) {
5        ...
6        switch (buffer[ISO7816.OFFSET_INS]) {
7          case INS_ILLEGAL_CAST:
8            try {
9              c = (C) ( (Object) b );
10             return; // Success, return SW 0x9000
11           } catch (ClassCastException e) {
12             /* Failure, return SW 0x6F00 */
13           }
14           ... // more later defined instructions
15 }   } }
```

Listing 2. `checkcast` type confusion

`b.addr` is modified to a specific value, and if this object is cast to a `C` instance, you may change the referenced address by `c.a`. But the `checkcast` instruction prevents from this illegal cast.

Barbu *et al.* use in their `AttackExtApp` applet (listing 2) an illegal cast at line 9. This cast instruction throws a `ClassCastException` exception. With specific material (oscilloscope, *etc.*), the thrown exception is visible in the consumption curves. With a time-precision attack, the authors prevent the `checkcast` from being thrown with the injection of laser based fault. When the cast is done, the references of `c.a` and `b.addr` link to the same value. Thus, the `c.a` reference may be changed dynamically by `b.addr`. This trick offers a read/write access on smart card memory within the fake `A` reference. Thanks to this kind of attack, Barbu *et al.* could apply their combined attack to inject ill-formed code and modify any application on Java Card 3.0, such as EMAN1 [3].

B. Code mutation

The mutant generation and detection is a new research field introduced simultaneously by [9] and [10] using the concepts of combined attacks and the detection process has been developed in [11]. Fault injection is a powerful mean to attack a smart card. Fault can be done into the chip by the environment perturbation. Consequences of fault attacks can be perturbation of the chip registers (*e.g.*, the program counter, the stack pointer, *etc.*), or the writable memories (variables and code modifications). These perturbations can have various effects, and in particular, they can allow an attacker to gain illegal access to data or services if not detected. Modifying the memory is a way to generate mutant code. A mutant can be defined as a piece of code that passes successfully the byte verification, audit process and any static analysis. But while the card is hit by the laser beam, a byte in the memory is modified and then the semantics of the code is changed and can become hostile.

We use the following example on a debit method (listing 3) that belongs to the wallet applet to demonstrate how a well-formed code can become hostile. In this method, the user's

PIN (Personal Identification Number) must be validated prior to the debit operation.

```
private void debit(APDU apdu) {
  if ( pin.isValidated() ) {
    ... make the debit operation
  } else {
    ISOException.throwIt
        (SW_PIN_VERIFICATION_REQUIRED);
  }
}
```

Listing 3. Original Java code

TABLE I
BYTE CODE REPRESENTATION BEFORE ATTACK

Byte	Byte Code
00 : 18	00 : aload_0
01 : 83 87 04	01 : getfield @8704
04 : 8B 05 23	04 : invokevirtual @0523
07 : 60 00 3B	07 : ifeq 00 59
10 : ...	10 : ...
...	...
59 : 13 63 01	59 : sipush 25345
63 : 8D 08 0D	63 : invokestatic @080D
66 : 7A	66 : return

In table I resides the corresponding byte code representation. If an attacker wants to bypass the PIN test, he injects a fault on the cell containing the conditional test byte code. Thus, the `ifeq` instruction (byte `0x60`) changes to a `nop` instruction (byte `0x00`). The obtained Java code follows (listing 4) with its byte code representation in table II. The exception throw becomes an unreachable code loading to call the debit method, whatever the PIN has been verified or not.

```
private void debit(APDU apdu) {
  ... make the debit operation
  // ISOException.throwIt
  // (SW_PIN_VERIFICATION_REQUIRED); // Dead code
}
```

Listing 4. Mutant Java code

TABLE II
BYTE CODE REPRESENTATION AFTER ATTACK

Byte	Byte code
00 : 18	00 : aload_0
01 : 83 00 04	01 : getfield @8704
04 : 8B 00 23	04 : invokevirtual @0523
07 : 00	07 : nop
08 : 00	08 : nop
09 : 3B	09 : pop
10 : ...	10 : ...
...	...
59 : 13 63 01	59 : sipush 25345
63 : 8D 00 0D	63 : invokestatic @080D
66 : 7A	66 : return

We can remark two points. First it is an example of what can be obtained thanks to a laser beam illumination. But we can also modify the control flow and jump to an arbitrary code *e.g.* an array. It becomes possible to design viruses that can be activated using a laser beam. The second point concerns the linking process. The byte code represented here is a linked code. This process occurred during the loading phase in the card. This hides the addresses of the API functions avoiding the design of viruses using function of the API.

C. Virus Example

Often, an attack tries to retrieve the crypto keys thanks to differential power analysis (DPA), simple power analysis (SPA), *etc.* attacks. It is impossible to retrieve the key by observing simply the memory because key containers are encrypted there. If one can write the following code (listing 5):

```
DES_Key.getKey (apduBuffer, (short) 0x00);
apdu.setOutgoingAndSend((short) 0x00,
                        DES_Key.getSize());
```

Listing 5. The `getKey` virus

And hide it, in a such a way that it becomes only reachable while a byte is hit by a laser beam, it opens the possibility to retrieve easily the cryptographic keys thanks to the API that will decrypt the key and put its value in clear text in the input output buffer. Such a virus, which is an active research field due to its difficulty, can be embedded into an array as shown in the EMAN 4 attack [12]. Thus the shell code to be embedded in an array and executed is the following (listing 6):

```
[AD] getfield_a_this  05 30 // push DES_Key
     reference
[19] aload_1
[8B] invokevirtual    04 02 // Get APDU buffer
[03] sconst_0
[8E] invokeinterface  03 09 30 04 // getKey()
[3B] pop
[19] aload_1
[03] sconst_0
[AD] getfield_a_this  05 30 // push DES_Key
     reference
[8E] invokeinterface  01 80 BA 01 // getSize()
[8B] invokevirtual    03 05 // setOutgoingAndSend
[7A] return
```

Listing 6. The virus shell code

A laser modification against a `goto_w` instruction in the main program, will change the control flow of the applet and transfer it to this array.

For enabling the design of such a virus we need to retrieve the linked address of the key object of the method `getKey` and the method address of `setOutgoingLength`.

D. A static analysis to detect laser beam enabled virus

As we present a way to build viruses in the card we also propose a way to detect them. The simulation tool *SmartCM* [7] aims to analyze the fault effect on a Java Card program [13]. Two different programs are used in this analysis. The first one is the mutation engine which takes as input a model of the card and the applicative program at the byte code level. It emulates the fault effect on the program according to a fault model and generates the mutant code. The mutant code is symbolically interpreted by the interpreter and if an embedded countermeasure detects the

deviant behavior the mutant is rejected, else it is stored as a mutant. The second tool is a risk analysis one. If a mutant is generated we need to evaluate the impact of its behavior to decide if it is a hostile or not.

The mutation engine is a brute force process which modifies the memory where the byte code is stored. The fault effect on the program is evaluated with an abstract interpreter that includes the management of the Java annotations. If the byte that has been impacted by the fault is an opcode, then according to the kind of memory (encrypted or not) the value of the new opcode is either `0x00` or `0xFF` or any value in this range. Then the mutation engine uses the smart card model to evaluate the execution of this new code for each value of the opcode and propagates the error until a countermeasure detects it (stack underflow, overflow, wrong local variable, wrong expected type, *etc.*). If a return of the method `ProcessAPDU()` (which can be considered as a main for a Java Card applet) of the applet is reached or an exception is never caught, then we consider that the mutant cannot be detected. We generate a class file that corresponds to the mutant code which is stored for further analysis. The less secure is the card, the more we must interpret the code and the longer is the simulation.

The mutation of an application can generate several mutants according to the security of the platform. To help the programmer to understand the error effect, it outputs the original Java Code and the Java perspective (if possible) of the mutant code, it highlights the area where the code has been modified. Often the mutants are harmless, but a security officer must check all of them. In order to facilitate this task we developed a risk analysis module that verifies a set of security properties on the mutant and decide to tag the mutants as dangerous or not. In order to have a more accurate analysis we need the internal map of the method addresses. Unfortunately this mapping is kept secret and cannot be obtained.

For two different reasons we need to obtain all the addresses of the Java Card API for a given platform. And of course a virus will be platform dependant, but as a card is produced in hundreds of thousands of units it becomes relevant to try to obtain this secret information.

III. How to Characterize the Card?

A. The Attack

The Java Card Specification [1] defines the linking step, which is done during the loading of CAP file. When the software is loaded into the card, the Java Card Virtual Machine provides a way to link the CAP file to install with the installed Java Card API. This step is done thanks to a token link resolution references in the `Constant Pool` component. To friendly find where each token is used, the `Reference Location` component keeps a list of offsets, in the `Method` Component.

So, in this loading step, the JCVM translates, with the help of the `Constant Pool` component and the `Reference Location` component, each reference to methods or fields used in the CAP file. Each offset for the fields and methods used in the `Method` component, are referred in the `Reference Location` component. So, the `Reference Location` component make a link between each token to link and the `Constant Pool` component. For the following, we will not modify the `Reference Location` and the `Constant Pool` components.

To characterize the embedded Java Card API in a smart card, we abuse the linking mechanism with the modification of the *natural* instructions, as `invokestatic`, which are followed by a token. If the card does not have an embedded BCV, a modification may push the linked reference on the stack and return it at the end of the current function. Remember that at the end of each Java method, the operand stack must be empty.

B. CAP File Modifications

In order to abuse the linking mechanism, we used a tool developed by the team. Thus, the *Cap Map* [14] was developed with the Java language. It provides an easy way to modify the CAP file.

As explained in the section III-A, we modify the instruction which follows a token to link. To understand it, we use the method `getSendOutGoingAndSendAddress` which sets the APDU buffer to send (listing 7).

```
public short getSendOutGoingAndSendAddress
                                        (APDU apdu){
   apdu.setOutgoingAndSend(FOO_BEGIN, FOO_LENGTH);
   return (short) 0xCAFE;
}
```

Listing 7. Java-function to get the `getSendOutGoingAndSend` address.

```
// flags: 0
// max_stack : 3
// nargs: 2
// max_locals: 0
00D8: [19] aload_1
00D9: [03] sconst_0
00DA: [03] sconst_0
00DB: [0B]
     invokevirtual 00 08
00DE: [11] sspush CAFE
00E1: [78] sreturn
```

Listing 8. Byte code associated to method listed in 7.

```
// flags: 0
// max_stack : 3
// nargs: 2
// max_locals: 0
00D8: [00] nop
00D9: [00] nop
00DA: [00] nop
00DB: [11] sspush 00 08
00DE: [00] nop
00DF: [00] nop
00E0: [00] nop
00E1: [78] sreturn
```

Listing 9. Modified byte code of the listing 8.

The associated unlinked byte code is described in the listing 8. In this byte code, the instruction `invokevirtual` is followed by a token (here `8`) updated by the address of the called method (`setOutgoingAndSend`) during the loading step. Moreover, the `Reference Location` component (listing 10 and the `Constant Pool` component are not

modified (they are the same as the original CAP file).

```
tag :9 size : 2b
byte_index_count : 8
offsets_to_byte_indices = {
  @0067(->@0067) @002a(->@0091) @0002(->@0093)
  ...
}
byte2_index_count : 31
offsets_to_byte2_indices = {
  ...
  @0006(->@00B1) @0014(->@00C5) @0017(->@00DC)
  ...
}
```

Listing 10. `Reference Location` component of the malicious applet.

In the `Reference Location` (listing 10), the offset `0x00DC` corresponds to the offset to link in our malicious method (listing 8) (here the parameter of the `sspush` instruction). When the Byte Code Linker updates the references in the `Method` component, it uses this information to update correctly each token as in the original CAP file.

To obtain the linked address, we modify the opcode `invokevirtual` by `sspush` as explained in the listing 9. So, when executing the linked byte code, the function address is pushed on the stack and returned at the end of the function.

C. Characterizing the complete API

Using this approach we are able to use the on board linker to generate the correct information, to store it on top of the stack and to send it back to the reader. Thanks to this information leakage we are able to obtain all the linked addresses of the Java Card API for a given card. For retrieving one address we need to build one CAP file. Retrieving the complete API, needs to generate 98 test cases for the methods of the classes and 60 test cases for the interfaces. All the test cases are validated whatever the tested card is. It means that the effort to design the test cases for retrieving the addresses will be reusable on all the cards. This attack is completely generic and independent of the platform. It depends only on the implementation of the specification *e.g.* 2.1.1, 2.2.1 or 2.2.

Moreover it allows us to characterize the level of implementation of a specification. All the methods must be implemented for a given specification but surprisingly some cards do not implement completely the specification and thus cannot be considered compliant with it.

D. Experimental Results

For the evaluation of this attack, we tried on the same cards, used at SSTIC'09 [3], plus a new one. Each card, listed in the table III, has a different manufacturer or is a different model.

On these cards, an applet with the modifications explained in the section III-B is sent. The `process` method contains the following code (listing 11):

Reference	Java Card	GP	Characteristic
a-21a	2.1.1	2.0.1	
a-22a	2.2	2.1	64k EEPROM
a-22c	2.2.1	2.1.1	36k EEPROM, RSA
b-21a	2.1.1	2.1.2	16k EEPROM, RSA
c-22a	2.1.1	2.0.1	RSA
c-22c	2.2	2.1.1	72k EEPROM, dual interface, RSA
d-21a	2.1	2.0.1	32K EEPROM, RSA
d-22b	2.2.1	2.1.1	16k EEPROM
e-21a	2.2	2.1	72k EEPROM

TABLE III
CARDS USED DURING THIS EVALUATION

```
switch (apduBuffer[ISO7816.OFFSET_INS]) {
  case INS_GET_GET_KEY_ADDRESS :
    ret = this.getGetKeyAddress();
    Util.setShort(apduBuffer, (short) 0x00, ret);
    apdu.setOutgoingAndSend((short)0x00,
                            SHORT_LENGTH);
    break;
  case INS_SEND_OUTGOING_AND_SEND_ADDRESS :
    ret = this.getSendOutGoingAndSendAddress(apdu);
    Util.setShort(apduBuffer, (short) 0x00, ret);
    apdu.setOutgoingAndSend((short)0x00,
                            SHORT_LENGTH);
    break;
  default:
    ISOException.throwIt
                (ISO7816.SW_INS_NOT_SUPPORTED);
```

Listing 11. Our `process` method.

Moreover, the object `Key` is a `DESKey` type as described in the listing 12.

```
this.DES_Key = (DESKey) KeyBuilder.buildKey
      ( KeyBuilder.TYPE_DES_TRANSIENT_DESELECT,
        KeyBuilder.LENGTH_DES3_2KEY, false);
this.DES_Key.setKey(INIT_KEY, (short) 0x00);
```

Listing 12. Applet Key initialization.

When the CAP file has been modified by the Cap Map [14], it is sent using OPAL [15] which provides the way to automate the evaluation of the targeted cards. We succeeded to install our malicious applet into each card listed in table III. We described, in the table IV, the value returned by the `getGetKeyAddress` and `getSetOutGoingAndSend` methods.

Reference	`getKey` address	`setOutgoingAndSend` address
a-21a	0x8C08	0x0308
a-22a	0x080A	0x0308
a-22c	0x020F	0x0308
b-21a	0x3267	0x0308
c-22a	0x810B	0x0803
c-22b	0x810B	0x0803
d-21a	0x0003	0x0008
d-22b	0x80BA	0x0803
e-21a	0x142F	0x0308

TABLE IV
RETURNED VALUES

The first observation concerns the address of `setOutGoingAndSend` method. This method may be

full-java implemented and may have the same address on each card (depending to little or big-endian memory organization).

On the other hand, the obtained address, for the card with the reference d-21a, corresponds to the token referred in the `Constant Pool` component. As we work in a black box model, the implemented countermeasures may provide a way to not link if the token is preceded by an illegal instruction. Another counter-measure may be the dynamically link during the applet execution.

Finally, some `getKey` methods are implemented in the EEPROM memory (like a-21a, c-22a, c-22b and d-22b).

To prevent the Byte Code Linker from giving some information about the Java Card API, an interesting countermeasure is, during the applet loading, to resolve just the token preceded by the specific instruction (like `invokevirtual`, `invokestatic`, `getstatic`, `setstatic`, etc.).

IV. Conclusion

In this paper, we presented a novel and generic method to lure the embedded linker of a smart card forcing it to link a token with a non authorized instruction, then this information is stored in the input output buffer and sent to the reader. It becomes possible to characterize completely the Java Card API. For that purpose, we built a complete set of CAP files which can be used to extract the addresses of the API whatever the platform is. This allows us to build very efficient viruses to be uploaded into the card and in particular to retrieve the container of crypto keys. We have shown with the experimental results that most cards do not have any counter measures against this attack. We will continue this evaluation on the most recent cards in order to detect if some of them embedded countermeasures.

The next step will consist in hiding the virus into a regular application, which is the subject of a thesis. For that purpose we will need to inject such a code with caring to the known countermeasures.

References

[1] Oracle, "Java Card Platform Specification," http://www.oracle.com/technetwork/java/javacard/.

[2] Global Platform, "Card Specification v2.2," http://www.globalplatform.org/, 2006.

[3] J. Iguchi-Cartigny and J.-L. Lanet, "Evaluation de linjection de code malicieux dans une java card," in *Symposium sur la Sécurité des Technologies de lInformation et de la Communication, SSTIC*, 2009.

[4] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on RSA with CRT: Concrete results and practical countermeasures," *Cryptographic Hardware and Embedded Systems-CHES 2002*, pp. 81–95, 2003.

[5] L. Hemme, "A differential fault attack against early rounds of (triple) DES," *Cryptographic Hardware and Embedded Systems-CHES 2004*, pp. 170–217, 2004.

[6] G. Piret and J.-J. Quisquater, "A differential fault attack technique against SPN structures, with application to the AES and KHAZAD," *Cryptographic Hardware and Embedded Systems-CHES 2003*, pp. 77–88, 2003.

[7] J.-B. Machemie, C. Mazin, J.-L. Lanet, and J. Cartigny, "SmartCM A Smart Card Fault Injection Simulator," *IEEE International Workshop on Information Forensics and Security - WIFS*, 2011.

[8] E. Hubbers and E. Poll, "Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours," Radboud University Nijmegen, Dept. of Computer Science NIII-R0438, 2004.

[9] G. Barbu, H. Thiebeauld, and V. Guerin, "Attacks on java card 3.0 combining fault and logical attacks," *Smart Card Research and Advanced Application*, pp. 148–163, 2010.

[10] E. Vetillard and A. Ferrari, "Combined attacks and countermeasures," *Smart Card Research and Advanced Application*, pp. 133–147, 2010.

[11] A. Séré, J. Iguchi-Cartigny, and J.-L. Lanet, "Checking the paths to identify mutant application on embedded systems," *Future Generation Information Technology*, pp. 459–468, 2010.

[12] G. Bouffard, J. Iguchi-Cartingy, and J.-L. Lanet, "Combined software and hardware attacks on the java card control flow," *CARDIS*, september 2011.

[13] J.-B. Machemie, J.-L. Lanet, G. Bouffard, J.-Y. Poichotte, and J.-P. Wary, "Evaluation of the ability to transform sim applications into hostile applications," *CARDIS*, september 2011.

[14] Smart Secure Devices (SSD) Team – XLIM, Université de Limoges, "The CAP file manipulator," http://secinfo.msi.unilim.fr/.

[15] A. Bkakria, G. Bouffard, J. Iguchy-Cartigny, and J.-L. Lanet, "OPAL: an open-source Global Platform Java Library which includes the remote application management over HTTP," *e-smart*, september 2011.