Combined Software and Hardware Attacks on the Java Card Control Flow

Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet

Smart Secure Devices (SSD) Team – XLIM Labs, Université de Limoges
83 Rue d'Isle, 87000 Limoges, France
{guillaume.bouffard,julien.cartigny,jean-louis.lanet}@xlim.fr

Abstract. The Java Card uses two components to ensure the security of its model. On the one hand, the byte code verifier (BCV) checks, during an applet installation, if the Java Card security model is ensured. This mechanism may not be present in the card. On the other hand, the firewall dynamically checks if there is no illegal access. This paper describes two attacks to modify the Java Card control flow and to execute our own malicious byte code. In the first attack, we use a card without embedded security verifier and we show how it is simple to change the return address of a current function. In the second attack, we consider the hypothesis that the card embeds a partial implementation of a BCV. With the help of a laser beam, we are able to change the execution flow.

Keywords: Java Card, control flow, laser, Java Card Stack, attack

1 Introduction

Java Card is a kind of smart card that implements one of the two editions, "*Classic Edition*" or "*Connected Edition*", of the standard Java Card 3.0 [8]. Such smart cards embed a virtual machine (VM) which interprets codes already romized with the operating system or downloaded after issuance [1]. In fact, Java Card is an open platform for smart cards, *i.e.* able of loading and executing new applications after issuance. Thus, 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) are safe, *i.e.* the loaded application is not hostile to other applications in the Java Card. Furthermore, the Java Card firewall checks application permissions and access in the card, enforcing isolation between them.

Java Cards have shown improved robustness compared to native applications regarding many attacks. They are designed to resist to numerous attacks using both physical and logical techniques. Currently, the most powerful attacks are

[1] Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [3]. This protocol ensures that the owner of the code has the necessary authorization to perform the action.

hardware based attacks and particularly fault attacks. A fault attack modifies parts of memory content or a signal on internal bus, which can lead to deviant behavior exploitable by an attacker. A comprehensive consequence of such attacks can be found in [6]. Although fault attacks have been mainly used in the literature from a cryptanalytic point of view (see [1,4,9]), they can be applied to every code layers embedded in a device. For instance, while choosing the exact byte of a program the attacker can bypass counter-measures or logical tests. We called *mutant* such modified application.

2 State of the Art

2.1 Java Card Security

The Java Card platform is a multi-application environment where critical data of an applet must be protected against malicious access from another applet. To enforce protection between applets, classical Java technology uses the type verification, class loader and security managers to create private namespaces for applets. In a smart card, complying with the traditional enforcement process is not possible. On the one hand, the type verification is executed outside the card due to memory constraints. On the other hand, the class loader and security managers are replaced by the Java Card firewall.

To allow code to be loaded into the card after post-issuance raises the same issues as the web applets. An applet not built by a compiler (handmade byte code) or modified after the compilation step may break the Java sandbox model. Thus, the client must check that the Java-language typing rules are preserved at the byte code level. Java is a strongly typed language where each variable and expression has a type determined at compile-time, so that if a type mismatches from the source code, an error is thrown. The Java byte code is also typed. Moreover, local and stack variables of the VM have fixed types even in the scope of a method execution but no type mismatches are detected at run time, and it is possible to make malicious applets exploiting this issue. For example, pointers are not supported by he Java programming language although they are extensively used by the Java VM (JVM) where object references from the source code are relative to a pointer. Thus, the absence of pointers reduces the number of programming errors. But it does not stop attempts to break security protections with unfair uses of pointers.

The BCV is an essential security component in the Java sandbox model: byte code alteration contained in an ill-typed applet may induce a security flaw. The byte code verification is a complex process involving elaborate program analyses using a very costly algorithm in time consumption and memory usage. For these reasons, lot of cards do not implement this kind of component and rely on the responsibility of the organization which signs the code of the applet to ensure that they are well-typed.

The separation of different applets is enforced by the firewall which is based on the package structure of Java Card and the notion of context. When an applet is created, the Java Card Runtime Environment (JCRE) uses an unique Applet IDentifier (AID) whose it is possible to retrieve the name of the package which defines it. If two applets are an instance of classes of the same Java Card package, they are considered in the same context. There is also a super user context, called the JCRE context. Applets associated with this context can access to objects from any other context on the card.

Each object is assigned to an unique owner context which is the context of the created applet. An object method is executed in the owner object context. This context provides information allowing, or not, to access to another object. The firewall prevent a method executing in one context from access to any attribute or method of objects to another context.

2.2 The CAP File

The CAP (for Convert APplet) file format is based on the notion of components. It is specified by Oracle [8] 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 VM (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 it has linked between them. A modification, voluntary or not, of a component is difficult and may provide unmeaning file. An invalid file is often detected during the installation step by the target JCVM.

2.3 Logical Attacks

The Hubbers and Poll's Attack Erik Hubbers *et al.* made a presentation at CARDIS 2008 about attacks on smart card. In their paper [5], they present a quick overview of the classical attacks available and gave some counter-measures. 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 not particular physical protection,

this attack is efficient but quiet 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 create two applets which communicate using the shareable interface mechanism. To create a type confusion, each applet use 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 tried, 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 able to read as much memory as they want. The main problem is more *how to read memory before the array?*

The other 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?*

Barbu *et al.*'s Attack: Combined Physical & Logical Attack At CARDIS 2010, Barbu *et al.* described a new kind of attack in their paper [2]. This attack is based on the use of a laser beam which modifies a runtime type check (the `checkcast` instruction) while running. This applet was checked by the on-card BCV, considered as valid, and installed on the card. The goal 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 1.1.

```
public class A {
   byte b00, ..., bFF
}
```

```
public class B {
   short addr
}
```

```
public class C {
   A a;
}
```

Listing 1.1. Classes used to create a type confusion.

The cast mechanism is explained in the JCRE specification [8]. When casting an object to another, the JCRE dynamically verifies if both types are compatible, with a `checkcast` instruction. Moreover, an object reference depends of the card architecture. The following example can be used:

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

The authors want cast an object `b` to an object `c`. If `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 his `AttackExtApp` applet (listing 1.2) an illegal cast at line 9.

```
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 1.2. `checkcast` type confusion

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 with an injection by a laser based fault the `checkcast` to be thrown. When the cast is done, the references of `c.a` and `b.addr` link 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.* can apply their combined attack to inject ill-formed code and modify any application on Java Card 3.0, such as EMAN1 [6].

3 EMAN2: A Stack Underflow in the Java Card

3.1 Genesis

The aim of this attack is to modify the register which contains the method return address by the address of an array which contains our malicious byte code. To succeed, the target smart card has no BCV and we know its loading keys. For this work, we have used two tools developed in the Java-language. The first one, the CFM [11] (for CAP File Manipulator) provides a friendly way to parse and full-modify the CAP files. The other one is the Java library OPAL [10] used to communicate with the card. So, to perform this attack, we must:

1. find the array address which contains the malicious byte code;
2. find where is located, in the Java Card stack, the address of the return function;
3. change this address by the address of the byte codes contained in our malicious array.

We will explain each step in the next subsections.

3.2 How to obtain the Address of our Malicious Array?

In a previous work [6], we explained how to execute auto-modifiable code in a Java Card. This malicious byte code was stored in an byte-array and called by ill-formed applet. We also have to remember the way to obtain the array address.

```
1 public short getMyAddressByteArray (byte[] array) {
2   short foo=(byte)0x55AA;
3   array[0] = (byte)0xFF;
4   return foo;
5 }
```

Listing 1.3. method to retrieve the address of an array

In order to retrieve the address of an array, we have implemented the method `getMyAddressByteArray` described in the listing 1.3. In its unmodified version, it returns the value contained in `foo`. The instruction in line 3 uses an array given in the function parameter. As seen in listing 1.4, the JCVM needs first to push a reference to the array `tab`[2]. Finally the function returns the previously

[2] In our tested card, all references are performed in a `short` type

pushed value of `foo`.

If an event changed our byte code like described in the listing 1.5, our function directly returns the reference of the array given as parameter. To make this modification, we use the CFM to "`nop`" each instruction between *push the array reference* and *return the short pushed value*. These instructions are written in a bold font in the listing 1.5. Using a card without BCV, a applet containing this function provides address of each array given in its parameter. The returned address is locate in the EEPROM area.

```
public short
getMyAddressByteArray
                (byte[] array) {
03 // flags: 0 max_stack : 3
21 // nargs: 2 max_locals: 1
10 AA       bspush      -86
31          sstore_2
19          aload_1
03          sconst_0
02          sconst_m1
39          sastore
1E          sload_2
78          sreturn
}
```

Listing 1.4. The Java byte code corresponding to the function 1.3

```
public short
getMyAddressByteArray
                (byte[] array) {
03 // flags: 0 max_stack : 3
21 // nargs: 2 max_locals: 1
10 AA       bspush      -86
31          sstore_2
19          aload_1
00          nop
00          nop
00          nop
00          nop
78          sreturn
}
```

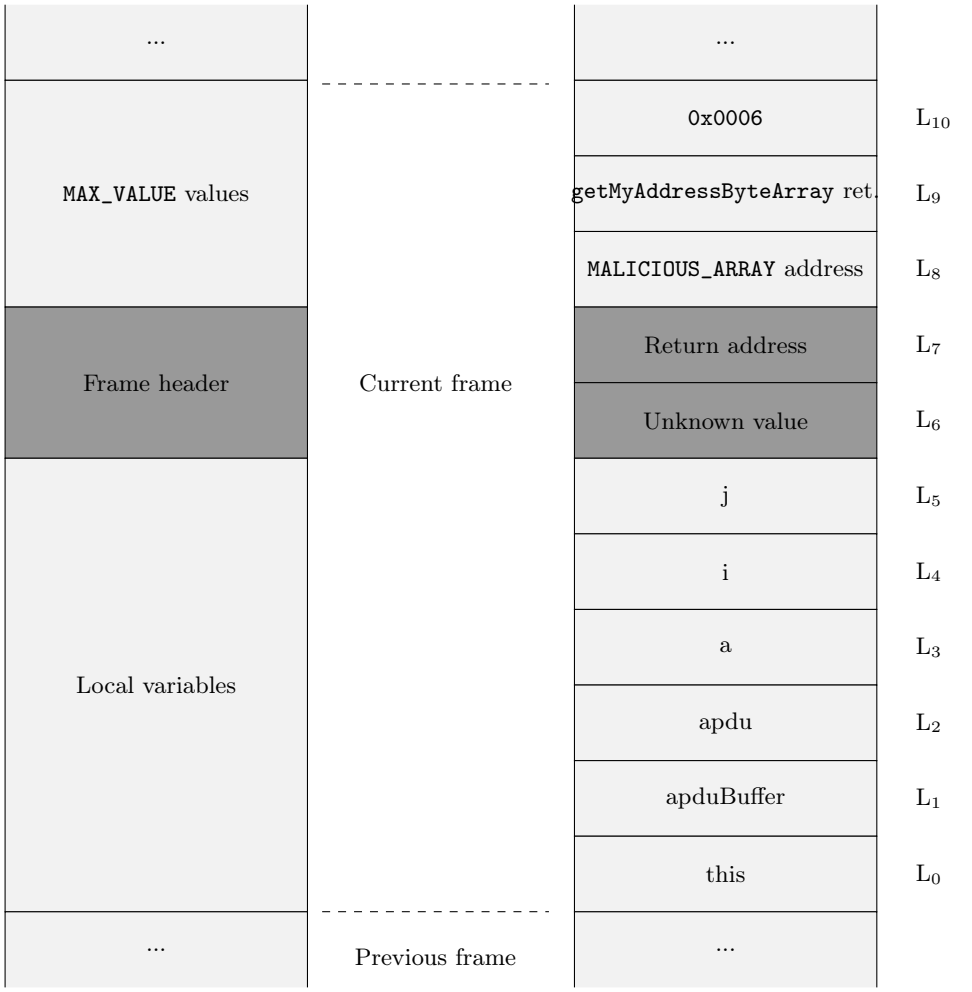
Listing 1.5. The function 1.3 with the modified return

On the targeted JCVM, the returned address by the malicious function `getMyAddressByteArray` does not refer to the array data. It is a pointer on the array header which is structured by 6 bytes that include the type and the number of contained elements. if the array is transient, the RAM array address follows the header. Else, the array data is stored after the 6-byte header.

3.3 Java Card Stack

To perform this attack, we should understand the Java Card stack. In fact, a Java Card contains two stacks, the native and the JCVM stack. The first one is used by the smart card operating system. The second one, is used by the JCVM to execute some Java applets value pushed in the Java Card stack.

To characterize the Java Card stack, We used the method `ModifyStack`, listed in 1.6. This method has three parameters: `apduBuffer`, a reference to a byte array; `apdu`, a reference to an instance of the `APDU` class; and `a`, a short value. The figure 1(b) represents the Java Card stack where each method parameter, variable and a reference to the class instance (`this`) are stored in the local variables area. Next, the information present in the frame header (in L_6 and L_7) are important data which hold the method return address. Finally, the



(a) Generic Java Card stack

(b) An Example of the Java Card Stack

Fig. 1. Java Card stack characterization

stack contains the data pushed while the method runs[3].

The BCV must checks several points. In particular: it should prevent any violations of the memory management (illegal reference access), stack underflow or overflow. This means these checks are potentially not verified during runtime and thus can lead to vulnerabilities. The Java frame is a non persistent data

[3] The maximum number of values to push is defined in the field `MAX_STACK` included in each Java Card method header.

```
1 public void ModifyStack(byte[] apduBuffer, APDU apdu, short a)
2 {
3   short i=(short) 0xCAFE;
4   short j=(short) (getMyAddressByteArray(MALICIOUS_ARRAY)+6);
5   i = j ;
6 }
```

Listing 1.6. Function to modify the Java Card stack

structure implemented in different manners and the specification gives no design direction for it.

3.4 Our Attack

Our attack aims to change the index of a local variable[4]. We purpose to use two instructions: `sload` and `sstore`. As described in the JCVM specification [8], these instructions are normally used in order to load a short value from a local variable and to store a short value in a local variable. The CFM allows us to modify the CAP file in order to access the system data and the previous frame. As example, the code in the listing 1.6, line 4, stores the value returned by `getMyAdressByteArray()` and adds 6 into variable `j`. Then, it loads the value of `j`, and stores it into variable `i` (line 5).

So, if we change the operand of `sload` (`sload 5`, at the offset `0x0F` of the listing 1.7) we store information from a non-authorized area into the local 5. Then, this information is sent out using an APDU. We tried this attack using a +2 offset and we retrieved the short value `0x8AFA` which was the address of the caller. Thus, we able to read without difficulty in the stack after our local variables. Furthermore, we can write anywhere into the stack below: there is no counter-measures. The targeted smart card implements an interpreter that relies entirely on the byte code verification process.

Next, we modified the CAP file to change the return address by our malicious array address, this step was explained in the section 3.2. When this modification is performed, the exception `0x1712` will be throw. So, we proved within this applet that we can redirect the control flow of such a JCVM.

3.5 Counter-measure

As we said, no important knowledge are needed in Java Card security and the simple modifications of an CAP file, with the tool [11], may perform these at-

[4] The specification says that the maximum number of variables that may be used in a method is 255. It includes local variables, method parameters, and in case of an instance method invocation, a reference to the object on which the instance method is being invoked.

```
      public void ModifyStack
             (byte[] apduBuffer, APDU apdu, short a) {
0x00:    02 // flags: 0   max_stack:  2
0x01:    42 // nargs: 4   max_locals: 2
0x02:    11 CA FE   sspush          0xCAFE
0x05:    29 04      sstore          4
0x07:    18         aload_0
0x08:    7B 00      getstatic_a     0
0x0A:    8B 01      invokevirtual   1
0x0C:    10 06      bspush          6
0x0E:    41         sadd
0x0F:    29 05      sstore          5
0x11:    16 05      sload           5
0x13:    29 04      sstore          4
0x15:    7A         return
      }
```

Listing 1.7. Malicious byte code applet of the function 1.6

tacks.

The principle of the stack underflow is to get access to memory area normally used by the system to the previous frame. A simple counter-measure would consist in checking the number of locals and arguments provided in the header of the method. With this simple check one cannot gain access to the system area where the JPC (previous Java Program Counter) and SPC (previous Stack Pointer) are stored. In order to avoid parsing the previous frame; the implementation can use the linked frame approach like in the simple RTJ VM references. This approach implies to create a new frame, to copy the argument of the current frame into the new, instead of the implemented method which uses the current stack as the beginning of the new frame. Desynchronizing the frames will avoid simply a stack underflow attack.

4 EMAN4: Modifying the Execution Flow with a Laser Beam

4.1 Description of our attack

In the section 3, we supposed there is no BCV. This hypothesis allowed us to modify the CAP file before loading it on the card. For the following, the targeted card has an improved security system based on a partial implementation of a BCV. This component statically checks the byte codes during the loading step and dynamic byte code checks are done during the runtime.

To perform this attack, we provide an external modification, such as the Barbu *et al.*'s attack, with a laser beam to change the control flow to execute

our own malicious byte code. Furthermore, we have the smart card loading keys.

In order to modify the execution flow, we will use the `for` loop properties. Next, after having understood how this kind of loop works, we modify it to change the control flow.

4.2 How Re-loop a For Loop

The `for` loop is probably the most widely loop used in the imperative programming languages. A classic `for` loop, such as in the listing 1.8, may be split in three parts. The first one is the declaration of the loop with the preamble (the initialization of the loop), followed by the stop condition and a function executed at each iteration. Next, the loop body contains the executed instructions for each iteration. Finally, a jump-like instruction re-loop to the next iteration if the stop condition is not satisfied.

```
for (short i=0 ; i<n ; ++i){
 foo = (byte) 0xBA;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 // Few instructions have
 // been hidden for a
 // better meaning.
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
}
```

Listing 1.8. A `for` loop

```
0x00: sconst_0
0x01: sstore_1
0x02: sload_1
0x03: sconst_1
0x04: if_scmpge_w       00 7C
0x07: aload_0
0x08: bspush            BA
0x0A: putfield_b        0
0x0C: aload_0
0x0D: getfield_b_this   0
0x0F: putfield_b        1
// Few instructions have
// been hidden for a
// better meaning.
0xE3: aload_0
0xE4: getfield_b_this   1
0xE6: putfield_b        0
0xE8: sinc              1 1
0xEB: goto_w            FF17
```

Listing 1.9. Associated byte codes of the loop 1.8

According to the amount of instructions contained in the loop body, the re-loop instruction has relative offset on 1 or 2-byte (± 127 or ± 255 bytes). In the Java Card byte code, the re-loop instruction may be a `goto` or `goto_w`. For our attack, we are interested in the `goto_w` statement at the offset `0xEB` (listing 1.9).

4.3 Our Attack:

To begin, we install into a Java Card an applet which contains the loop `for` described in the listing 1.8. The function which contains this loop is compliant

with each security rule of Java Card and the embedded smart card BCV allows its loading.

An external modification based on a laser beam against the `goto_w` instruction, at the offset `0xEB` in the listing 1.9, may change the control flow of the applet. We would like to redirect this flow in the array `MALICIOUS_ARRAY` to execute our malicious byte code. Thus, changing the `goto_w` parameter `0xFF17` to `0x0017` involves a relative jump to the 17^{th} byte after this instruction. To success attack, our array must locate after the modified function in the EEPROM area.

Smart Card memory management The main difficulty regarding this attack is the memory management. Indeed, the static array `MALICIOUS_ARRAY` must physically put after our malicious function. For that, we analyzed how our targeted smart card stores its data. In order to understand the algorithm used by the card to organize its memory, we did the following method:

1. first, few chosen applets are installed on the card within a careful dump of the EEPROM memory between each install,
2. next, the card is stressed by installing and deleting different applets size. A dump is done at each step.

For each analyzed smart card, we have obtained the same algorithm used to manage the memory. These Java Cards have a *first fit* algorithm where the applet data are stored after its byte code. If the smart card have managed few applets without causing fragmentation, it is likely than the applet data is stored before the corresponding applet byte code.

In our case, when our applet is firstly installed on the card the dump obtained is listed in 1.10.

```
0x0A7F0:  18AE 0188 0018 AE00 8801 18AE 0188 0018
0x0A800:  AE00 8801 18AE 0188 0018 ae00 8801 18ae
0x0A810: 0188 0059 0101 A8FF 177A 008A 43C0 6C88
0x0A820:  abcd ef00 0000 0000 0000 0000 0000 0000
0x0A830:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A840:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A850:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A860:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A870:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A880:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A890:  0000 0000 0000 0000 1117 1200 0000 8D6F
0x0A8A0:  C000 0000 0000 00FE DCBA
```

Listing 1.10. Memory organization of our installed applet

As may be seen in the dump 1.10, the function to fault precedes the array `MALICIOUS_ARRAY` in light-gray. This dump is a linked byte code on the contrary of the byte code listed in 1.9.

The Goto redirection Before injecting our fault, the function returns `0x9000` (status without error).

After precisely targeted the high-byte parameter of the `goto_w` instruction located at `0xA817` in the listing 1.10, a laser beam attack swaps `0xFF17` to `0x0017`. This fault allows to redirect the execution flow. Indeed, the `goto_w` jumps forward to go in the array `MALICIOUS_ARRAY`. A land up area of `nop` catches up the instruction pointer which will execute our malicious code, here an exception thrown the value `0x1712`. This result proves that we succeed to change the control flow of our applet.

Moreover, event if the memory is encrypted, this kind of attack have fifty percent to change the `goto_w` instruction statement to redirect towards the front.

4.4 Counter-measures

Create a mutant application is the same way that changing an applet after its loading. To protect the JCVM against this attack, voluntary or not, we have developed some counter-measures described in [7]. We are going to present a brief resume of these counter-measures.

The XOR Detection Mechanism This protection is based on basic blocks. It allows code integrity and application control flow checking. A basic block is a sequence of instructions with a single entry point and a single exit point[5]. For each basic bloc, a checksum is computed by using the XOR operation on all the bytes composing a basic block. Then this table is stored in the CAP file as a Java Card custom component. The interpreter has to be modified to exploit and verify the checksum information. During runtime, the interpreter computes again the checksum and compares it with the stored values.

The Field of Bit Detection Mechanism This counter-measure checks the nature of the element stored in the byte array of the CAP file. A tag (bit) is associated to each byte of the bytecode. The tag has the value 0 if the bytecode is an opcode, and it has the value 1 if the byte code is a value (a parameter of an *opcode*). During an attack, the following situations can appear:

1. An increase of operands number for the instruction, it is the case when `add` (no operand) is replaced by `icmpeq` (one operand).

[5] The execution of a basic block starts only at an entry point, and leaves a basic block only at an exit point.

2. A decrease of operands number for the instruction, it is the case when `aload` (one operand) is replaced by `athrow` (no operand).
3. No change on operand number: it is the case when an `iload` (one operand) is replaced by a `return` (one operand).

This method can detect when the changing `1` and `2` happen. During the compilation, a field of bit is generated representing the type of each element contained in the method byte array. It is stored also as a Java Card custom component in the CAP file. The interpreter checks before executing an *opcode* that its byte was scheduled to be executed or not.

The Path Check Mechanism This method computes the control flow graph of the method by extracting the basic blocks from the code. The list of paths from the beginning vertex is computed for each vertex of the control flow graph. This computed paths are encoded using the following convention:

1. Each path begins with the tags `0` and `1` to avoid a physical attack that changes it to `0x00` or to `0xFF`.
2. If the instruction that ends the current block is an unconditional or conditional branch instruction when jumping to the target of this instruction, then the tag `0` is used.
3. If the execution continues at the instruction that immediately follows the final instruction of the current block then the tag `1` is used.

If the final instruction of the current basic block is a switch instruction, the is made by any number of bits that are necessary to encode all the targets. When interpreting the byte code, the VM computes the path followed by the program using the same convention; for example, when jumping to the target of a branch instruction it saves the tag `0`. Then prior to the execution of a basic block, the VM checks that the followed path is an authorized path, *i.e.* a path that belongs to the list of path computed for this basic block. In the case of a loop (backward jump) the interpreter checks the path for the loop, the number of references and the number of values on the operand stack before and after the loop, to be sure that for each round the path remains the same.

5 Conclusion

In this paper we have described two ways to change the execution flow of an application after loading it into a Java Card. The first method, EMAN2, provides a way to change the return address of the current method contained in its frame stack. This attack is possible because there is no check during the stack operations. The second method, EMAN4, uses a laser beam to modifies a well-formed applet loaded and installed on the card to become mutant, even with the on-board BCV.

These two attacks allow to execute malicious code in the JCVM without to be detected by the firewall component. In the case of EMAN2, we have proposed two counter-measures. To opposite, EMAN4 needs a good knowledge of the targeted JCVM and to find the faulted area with the laser beam.

References

1. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.: Fault attacks on RSA with CRT: Concrete results and practical countermeasures. Cryptographic Hardware and Embedded Systems-CHES 2002 pp. 81–95 (2003)
2. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on java card 3.0 combining fault and logical attacks. In: Gollmann, D., Lanet, J.L., Iguchi-Cartigny, J. (eds.) CARDIS. Lecture Notes in Computer Science, vol. 6035, pp. 148–163. Springer (2010)
3. Global Platform: Card Specification v2.2 (2006)
4. Hemme, L.: A differential fault attack against early rounds of (triple-) DES. Cryptographic Hardware and Embedded Systems-CHES 2004 pp. 170–217 (2004)
5. Hubbers, E., Poll, E.: Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours. Dept. of Computer Science NIII-R0438, Radboud University Nijmegen (2004)
6. Iguchi-Cartigny, J., Lanet, J.: Developing a Trojan applet in a Smart Card. Journal in Computer Virology (2010)
7. Lanet, J.L., Bouffard, G., Machemie, J.B., Poichotte, J.Y., Wary, J.P.: Evaluation of the ability to transform sim application into hostile applications. Cardis (2011)
8. Oracle: Java Card Platform Specification
9. Piret, G., Quisquater, J.: 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)
10. Smart Secure Devices (SSD) Team – XLIM, Université de Limoges: OPAL: An Open Platform Access Library. `http://secinfo.msi.unilim.fr/`
11. Smart Secure Devices (SSD) Team – XLIM, Université de Limoges: The CAP file manipulator. `http://secinfo.msi.unilim.fr/`