Journal of Cryptographic Engineering manuscript No.
(will be inserted by the editor)

Trust Can be Misplaced

Noreddine El Janati El Idrissi · Guillaume Bouffard · Jean-Louis Lanet · Said El Hajji

Received: date / Accepted: date

Abstract Retrieving assets inside a secure element is a challenging task. The most attractive assets are the cryptographic keys stored into the non volatile memory (NVM) area. Most of the researches try to obtain cryptographic keys through side channel attacks or fault injection attacks. Such cryptographic objects are stored into secure containers. We demonstrate in this paper how one can use some characteristics of the Java Card platform to gain access to these assets. Such a smart card embeds a Firewall that provides isolation between applets from different clients (using the notion of security contexts). We exploit the client/server architecture of the intra platform communication to lure a client application to execute within its security context, a hostile code written and called from another security context: the server security context. This attack shows the possibility for a trusted application to execute within its security context some hostile code uploaded previously by the server.

Noreddine El Janati El Idrissi · Said El Hajji
LabMIA, Faculté des Sciences, Rabat, Morocco,
E-mail: janatinoreddine@gmail.com,
E-mail: elhajji@fsr.ac.ma

Guillaume Bouffard
Agence Nationale de la Sécurité des Systèmes d'Information
Secrétariat Général de la Défense et de la Sécurité Nationale
51, boulevard de La Tour-Maubourg, 75700 Paris SP, France
E-mail: guillaume.bouffard@ssi.gouv.fr

Jean-Louis Lanet
INRIA, LHS-PEC
263 Avenue Général Leclerc, 35042 Rennes, France
E-mail: jean-louis.lanet@inria.fr

Keywords Smart Card; Java Card; Software Attack; Shareable Interface; Key Extraction; Self Modifying Code

1 Introduction

Today, most of the smart cards embed a Java Card Virtual Machine (JCVM). Java Card is a type of smart card that implements the standard Java Card [22] in one of the two editions *Classic Edition* or *Connected Edition*. The Virtual Machine (VM) interprets application byte codes already romized with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by GlobalPlatform [13]. This protocol ensures that, the code owner has the required credentials to perform the particular action.

A smart card can be viewed as a smart and secure container which stores sensitive assets. Such tokens are often the target of attacks at different levels: pure software attacks, hardware based, *i.e.* side channel or fault injection attacks and mixed attacks. Security issues and risks of these attacks are ever increasing and continuous efforts to develop countermeasures against these attacks are sought. The main assets in a smart card are the sensitive data (*i.e.* the cryptographic keys) and the code of the program. Often, attackers perform cryptanalysis using side channel attacks to recover the keys, thus breaking their confidentiality. The difficulty to break the security properties of these assets are in the decreasing order:

- data confidentiality,
- data integrity,
- code integrity,
- code confidentiality.

Bouffard and Lanet [9] have shown that it was relatively easy to break the code confidentiality and if this attack succeeds, then the code integrity can be broken leading to the memory snapshot. Once the memory is read, it is possible to perform memory carving [20] to gain information on the data and in particular the key containers. Smart card manufacturers have increased the security of their JCVM implementation from years to years, in such a way that published attacks do not work anymore on recent cards. The current smart cards are now well-protected against software attacks with different mechanisms such as program counter bound checks, typed stack, separation of kernel and user data.

We evaluate here in a first step the robustness of the countermeasures and in particular the ability of an attacker to mitigate them for getting access to valuable assets. We have reversed the mechanism that reduces the capacity of a shellcode to execute loops. We externalize the control flow of the shellcode in such a way we keep the semantics of the shellcode of the remaining basic blocks (linear parts of code). This step needs a preliminary code transformation and data analysis to provide to the shellcode the data required to the basic block. We demonstrate first a proof of concept and then its application to the dump of a card memory. It is based on separating the control flow and the basic blocks of a program.

Then, we extend this idea by revisiting the concept of shared interface. Our approach allows us to bypass the Firewall mechanism implemented inside the Java Card. To the best of our knowledge, this is the first time that the Java Card Firewall can be bypassed. To succeed, we need to implement in the client/server architecture our attack at the server side. The client executes the code which is under the control of the server. The novelty of the approach relies on the fact that the code is executed under the security context of the client, allowing the server to get access to all objects belonging to the client. In particular, we have been able to get the secret keys in plain-text and to send them to the server. The contribution introduced in this article aims at proving that a Byte Code Verifier (BCV) must be used to protect the smart card assets.

The remaining of this paper is organized as follows: the first section introduces the Java Card security. The second section presents the state of the art both in term of attacks and published countermeasures. The third section presents an attempt to break secure container by brute force. Then, the fourth section introduces our contribution for mitigating the control flow countermeasure. Next, we propose to use this possibility in a client/server architecture to force the client to provide in plain-text the keys. Finally, the last section concludes this article.

2 Java Card Security

Smart cards security depends on the underlying hardware and the embedded software. Embedded sensors (light sensors, heat sensors, voltage sensors, *etc.*) protect the card from physical attacks. While the card detects such an attack, it has the possibility to erase quickly the content of the non volatile memory (NVM) preserving the confidentiality of secret data or blocking definitely the card (Card is mute). In addition to the hardware protection, software are designed to securely ensure that applications are syntactically and semantically correct before installation and also sometimes during execution. They also manage sensitive information and ensure that the current operation is authorized before executing it.

The BCV ensures type correctness of code, which in turn guarantees the Java properties regarding the memory access. For example, Java-language forbids to perform arithmetic on references. Thus, it must be proved that the two elements on top of the stack are of primitive types before performing any arithmetic operation. On the Java platform, byte code verification is invoked at loading time by the class loader. Due to the fact that Java Card does not support dynamic class loading, byte code verification is performed at loading time, *i.e.* before installing the application into the card. However, most of the Java based smart cards do not embed an on-card BCV as it is quite expensive in terms of memory consumption. Thus, a trusted third party performs an off-card byte code verification and signs the application and, on-card, during the installation, the digital signature is checked.

Moreover, the Java Card Firewall performs checks at runtime to prevent applets from accessing (reading or writing) data of other applets. When an applet is installed, the system uses a unique Applet IDentifier (AID) allowing to retrieve the name of the package in which the applet is defined. If two applets are instances of classes coming from the same Java Card package, they are considered to belong to the same context. The Java Card Firewall isolates the contexts such that a method which is executed in one context cannot access any attribute or method of objects that belong to another context. The only possibility to share functionality is *via* a Shareable Interface Object (SIO). When an object is created, this object is tagged with the security context of the owner applet. During the execution, the access control policy of the firewall checks that the

current program has the right to access a given object by comparing their security contexts.

The Java Card specification [22] introduces the notion of Shareable Interface, which defines a set of methods that an applet may export through the Firewall. For exporting methods, one must declare them in an interface that extends the tagging interface `javacard.framework.Shareable`. The interface must be implemented in a class, and an object should instantiate this class to obtain a SIO. When the client applet wants to access the specified methods, it first declares a reference of the type defined by the server's shareable interface, and then invoke the `JCSystem.getAppletShareableInterfaceObject()` method, indicating the AID of the server applet. The Java Card Runtime Environment (JCRE) invokes the server applet's `getShareableInterfaceObject` indicating the AID of the client applet. The server applet then decides, based on the client's AID, if the client is authorized to access to the required SIO, and returns either a reference to that SIO or `null`. This mechanism allows the server to implement its security policy concerning the access of its SIO and restricts the client of gaining access to fields or methods that do not belong to the interface.

Smart card security is a complex problem with different points of view, but products based on JCVM have passed successfully real-world security evaluations for major industries around the world. It is also the platform that has passed high level security evaluations for issuance by banking associations and by leading government authorities, they have also achieved compliance with FIPS 140-1 certification scheme. Nevertheless, implementations have suffered several attacks either on hardware or on software. Some of them succeeded in getting access to the data stored in the NVM area (code of the downloaded applets). Breaking the Java Card sandbox to read the Read Only Memory (ROM) area is a difficult task. The ROM area contains the operating system binary code and a part of the JCVM (Application Programming Interface (API), interpretor, etc.). To the best of our knowledge, two attacks bypass the Java Card sandbox to read the ROM area. On the first hand, Bouffard and Lanet [9] corrupted the indirection table. This table is used to invoke a native method from the Java-side. This attack is succeeding in executing the attacker's native-shellcode. On the other hand, Lancia and Bouffard [18, 19], after presented a bug in the Java Card BCV, where a check is not correctly done, exploited it to corrupt the execution flow. When a public Java Card method is invoked, the link resolution mechanism makes an overflow to execute native code contains in the Application Protocol Data Unit (APDU) buffer. Each of those attackers offers a way to execute native shellcodes and reading the ROM memory.

3 Accessing the content of a Secure Key Container

There is no specification which defines how to store securely keys inside the card. The Java Card specification [22] states when creating a key the method has a boolean parameter which request (when is set to `true`) to store the key in a secure way. In several implementations, the value of the boolean parameter is irrelevant: the key is always stored encrypted. Nevertheless, there are some standards for securely storing keys. The PKCS12 specification defines a container for private keys using a pair of asymmetric keys. They mainly rely on the fact that it exists a key for encrypting keys but located in a secure environment (smart cards, TPM, *etc.*). Of course, in a smart card it is possible to store a key in the NVM memory to encrypt a key container.

We can consider that the ROM or NVM areas, not accessible by the VM of the smart card, are probably where a key can be stored. In PKCS12, a secure container (PFX) includes a tag, a SafeBag and mac data for integrity. Each SafeBag holds one piece of information: a key, a certificate, *etc.* which is identified by an object identifier, then the value and potentially attributes.

For retrieving sensitive keys, the first idea is to snapshot the content of a smart card memories, to find the adequate pattern and try to brute force it. We made these experiments on a card that contains a triple-DES co-processor and a public key co-processor. The card embeds 32 KB of NVM with only 31 KB available for applications. It supports triple-DES and RSA algorithms.

3.1 Characterizing a key container

The first step consists in reading the memory using a classical EMAN2 attack as described in the next section. This allows us to search for a secure container. We had collected several memory dumps with different configurations of keys (values, initialization, type, *etc.*) and after the analysis of the differences between each snapshot, we have reached the conclusion that keys are stored as arrays in this card. The metadata of an array contains the size (two bytes), a type (one byte), a security context (one byte) and the data. A DES key, for example, is stored as two arrays as shown in Table 1.

A secure container is split in two parts, the first one describes the kind of key, while the second stores the key. The second array is referred by the first one

Size	Type	Security context	Data
00 0c	c0	12	0f 03 05 57 12 a4 00 01 00 03 00 80
00 0d	c9	12	01 0d 01 99 95 b1 5d d5 46 1e fa 5d f9

Table 1 Internal representation of a secure key container.

with the bytes `0x12` and `0xa4` which represent an offset from the beginning of the current package. In the second array, the three first bytes represent the status of the key, while the elements between the bytes `0x99` to `0xf9` represent the 8-byte key value and a 2-byte integrity checksum. Whatever the value of the boolean `keyEncryption` of the method `buildKey`[1] of the interface `KeyBuilder` is, the key is always stored encrypted. The specification states the key implementation returned may implement the `javacardx.crypto.KeyEncryption` interface even when the `keyEncryption` parameter is `false`. The Java Card specification does not define how the key is ciphered.

3.2 Guessing the keys

After the secure container identified in the dump, we had tried to brute force it. If the keys are encrypted through the DES algorithm with the key stored in the different snapshots it is possible to brute force. We looked for a common immutable part in different memory snapshots because for a given plain text (the key to store) the cipher value is always the same. Finally, we have not succeeded in brute forcing it. Several reasons can explain it:

- the key is not stored in the NVM area;
- the plain text is not the correct one, a function is applied on the key value before encryption (*e.g.* xoring the key);
- the ciphering algorithm is unknown;
- key is split in non continuous blocks.

Secure containers for sensitive objects like keys are well-implemented on the certified cards. We did not success within this card to brute force their secure container even if we succeeded in characterizing them. It seems that the only solution is to force the embedded software to decrypt itself the key and to provide it in plain text. This raised several challenges:

- **Secure download**: to load a shellcode, we need to upload an ill-typed applet. The BCV checks the correctness of the applet but recently [12, 18], flaws in this secure piece of code has shown the possibility to upload ill-formed applet and run some shellcodes. If the BCV used to check each applet and library to be installed is not up to date, it may contains some vulnerability exploitable from the literature.
- **Executing a shellcode**: most of the cards implement countermeasures to avoid illegal control flow transfer. We will focus on how to bypass it.
- **Bypassing the Firewall**: being able to execute an arbitrary code does not prevent to be blocked by the Firewall. We will demonstrate the possibility to gain access to an object that do not belongs to our security context.

We consider that the literature has solved the first issue, then we focus on the two lasts *i.e.* executing a shellcode and bypassing the Firewall.

4 Control Flow Attacks to Obtain Smart Card Secrets

The previous section has presented that snapshotting the smart card memory is not enough to obtain the sensitive data. Each of them is securely stored in the smart card. In fact, each applet sensitive data, ciphered in the NVM area, is quietly decrypted when it is legally read (accessed from a path allowed by Java Card security rules) by the JCVM. From the Java Card specification, a legitimate access is a reading by the applet which owns the sensitive asset. The owner properties are computed with the context information at execution time. If an attacker succeeds in accessing an object with the owner applet context, then the access to the object is granted else a security exception is raised. The best solution is to let the JCVM decrypt itself the content of the secure container using a shellcode which requires to execute a hostile code.

Executing malicious fragment of code on the JCVM is studied in the literature and it is mainly based on a software attack which corrupts the application control flow [7]. This approach is the cheapest solution to get access to sensitive information from the targeted cards. Recently, cards embed several verifications during the installation process. Those verifications are considered as a lightweight (partial) BCV embedded in the card. To succeed software attacks inside the card and to bypass this countermeasure, the combined attacks [25] enable software attacks *via* a physical perturbation (laser or electromagnetic fault injection) which brings on a logical fault.

[1] The function `buildkey` requires three parameters: `keyType`, `keyLength` and `keyEncryption`. The `keyType` parameter defines the type of key to generate, `keyLength` the key length in bits and `keyEncryption` is a boolean which requests to encrypt the key value.

4.1 Software Attacks against Java Card Platform

Mostowski and Poll [21] introduced attacks on smart cards where they presented a quick overview of the classical attacks available on smart cards and in particular the SIO. The idea to abuse shareable interfaces is interesting and can lead tricking the VM even in the presence of a BCV. The main goal is to obtain a type confusion without the need to modify the Converted APplet (CAP) files. To do that, they had to create two applets which will 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 during the load, there is no way for the BCV to detect it.

This attack aims at abusing the shareable mechanism thanks to the non-typed verification. In fact, they tried to pass a byte-array as a short-array. Thanks to this trick, when reading the original array, the authors are able to read after the last value due to the length confusion. To make this attack, two interfaces are required: one for the client (Listing 1) and one for the server (Listing 2).

```
public interface
  Interface extends
        Shareable {
  public byte[]
       giveArray();
  public short
       accessArray
       (byte[]
          MyArray);
}
```

Listing 1 Client interface.

```
public interface
  Interface extends
        Shareable {
  public byte[]
       giveArray();
  public short
       accessArray
       (short[]
          MyArray);
}
```

Listing 2 Server interface.

These two interfaces must have the same AID for package and applet of the server and client. Then, the server's interface is uploaded into the card. The byte-array is interpreted as a short-array from the client side. The two required methods are used to read values in the array and to share an array between the client and the server. From the client's side, the server's array is retrieved, which is a byte-array, by using the `giveArray` method. After, the array is passed as a parameter of `accessArray` method and they send the return reading short through the APDU. As a result, the authors succeed to pass a byte-array as a short-array in all cases, but when they exceeded the standard ending of the array, an error was checked by the card.

Other software attacks are based on the fact that the runtime relies on the BCV to avoid costly tests. An absence of a test during runtime leads to an attack path. An attack aims at confusing the applet's control flow upon a corruption of the Java card Program Counter (JPC) or perturbation of the data.

Misleading the application's control flow purposes to execute a shellcode stored somewhere in the memory. The aim of EMAN1 attack [17] is to abuse the Firewall mechanism with the unchecked static instructions (as `getstatic`, `putstatic` and `invokestatic`) to call malicious byte codes. In a malicious CAP file, the parameter of an `invokestatic` instruction may redirect the Control Flow Graph (CFG) of another installed applet in the targeted smart card. Such an attack leads for the first time to execute self modifying code in a Java Card. This attack has been mitigated through different countermeasures. EMAN2 [7] attack was related to the return address stored in the Java Card stack. The authors used the unchecked local variables to modify the return address, while Faugeron in [11] exploited an underflow on the stack to get access to the return address.

Since a perfect BCV is embedded or if the process requires its usage, installing an ill-formed applet becomes impossible. To bypass a code verification process, new attacks exploit the idea to combine physical and logical attacks. Barbu *et al.* introduced and performed several combined attacks such as the attack [4] based on the Java Card 3.0 specification leading to the circumvention of the Firewall application. Another attack [3] consisting of tampering the APDU that leads to access the APDU buffer array at any time. They also discussed in [2] about a way to disturb the operand stack with a combined attack. It also gives the ability to alter any method regardless of its Java context or to execute any byte code sequence, even if it is ill-formed. This attack bypasses the on-card BCV [5]. In [7], Bouffard *et al.* described how to change the execution flow of an application after loading it into a Java Card. Recently, Razafindralambo *et al.* [24] introduced a combined attack based on fault enabled viruses. Such a virus is activated by hitting with a laser beam, at a precise location in the memory, where the instruction of a program (virus) is stored. Then, the targeted instruction mutates one instruction with one operand to an instruction with no operand. The operand is executed by the JCVM as an instruction. They demonstrated the ability to design a program in such a way, that the modification of a given instruction can change the semantics of the program. Finally, a well-typed application is loaded into the card but an ill-typed one is executed.

Hamadouche and Lanet [16] described various techniques used for designing efficient viruses for smart cards. The first one is to exploit the linking process by forcing it to link a token with an unauthorized instruction. The second step is to characterize the whole Java Card API by designing a set of CAP files which are used to extract the addresses of the API regardless of the platform. The authors were able to design CAP files that

embed a shellcode (virus). As the author learned all the addresses of each method of the API, they could replace invocation token of any method.

4.2 Executing Arbitrary Code to Retrieve Smart Card Secrets

Retrieving sensitive keys stored in the smart card can be accessed by the owner applet. This section focuses on how, based on a control flow attack, we are able to read the secret keys of an applet.

4.2.1 EMAN1: Polymorphic code strikes again

The original EMAN1 attack allowed to invoke in the method stored inside the card to invoke an array instead of a method. It required to browse the internal representation of the CAP such that it detected the stored byte array and then modify the destination of the invocation. In [10], the authors presented a new logical vector attack that used of the API and in particular the method `ArrayCopyNonAtomic`. This attack has generated the concept of reference forgery. Understanding the metadata enabled the authors to forge their own metadata with applet fields. Then, they used these metadata to enforce and mislead the system to consider them as a header of an object. This behavior gives the authors an access to the NVM memory starting from their first forge. Thanks to these forges, they revisited the EMAN1 attack, where the original attack is mitigated by the counter measures of targeted card. The result of this revisited attack is a huge dump with a length `0xFFFF` bytes of the NVM, which gives access to memory areas that were unattainable with the forges.

The advantage of this revisited version of the EMAN1 attack is that it does not rely on any assumption regarding how the frame is built on the given card. The only assumptions are:

- the right to load applets inside the card with the appropriate keys;
- a way to bypass the type verification process or to rely on combined attack;

This attack allows to execute a method stored in an array which in turn can be easily filled at runtime with the adequate data.

4.2.2 EMAN2: Fooling the Control Flow

Introduced in the previous section, the EMAN2 attack [7] aims at changing the index of a local variable to confuse the applet control flow. For that purpose, we use two instructions: `sload` and `sstore`. As described in the JCVM specification, these instructions are used to transfer from or to the stack a short value from or to a local variable .

If someone changes the parameter of the `sstore` instruction, as for instance `sstore 4`, a short value will be stored into the local variable 4. Let us assume that the program stores a short value that corresponds to the first element of an array into the last local variable increased by an offset of 2. It means, that we try to store into a local that does not exist. Due to the fact that the BCV checks the range of the local variables, this overflow is detected during the conversion process. But, after this verification step, if one changes the value of the `sstore` parameter, it will not be detected during the runtime.

With the knowledge of the internal reference of an array[2], this manipulation changes the return address of the current method by the address of the first element of an array. When exiting from the current method, instead of returning to the caller, the JPC will jump into the array and the JCRE will execute the content of the byte-array. Of course, this data must be interpretable by the VM without any stack underflow or overflow. With this attack, an arbitrary shellcode can be executed.

The assumptions concerning the EMAN2 attack are:

- the right to load applets inside the card with the appropriate keys;
- a way to bypass the type verification process or to rely on combined attack;
- knowledge of the implementation details of the frame;
- the absence of countermeasures on the return address.

The last assumption requires that no integrity check is made at runtime while using the frame. We demonstrated in [10] that a split stack (user and kernel stacks are separated) is a weak countermeasure that can be bypassed.

4.2.3 Accessing Applet Secrets

The previous section has introduced the EMAN1 and EMAN2 attacks which fool the applet control flow to execute the content of a Java array. Whatever the attack path (array or return address) we will have to face the out of the bounds countermeasure while executing an array. For the sake of simplicity, we continue with the EMAN2 attack, knowing that the scheme of the attack against the out of the bounds countermeasure will be the same with the EMAN1 attack.

[2] This information can be disclosed *via* a characterization step as introduced in [1, 6].

This section focuses on how, by the EMAN2 attack, an attacker can obtain applet sensitive assets. The Listing 3 introduces an applet which uses sensitive assets. This applet aims, upon a control flow modification, at reading the content of the `private3DESKey` field. This field is only accessible from an `AppletSecretKey` class instance. The `private3DESKey` field is only readable from the same security context as the field owner.

```
class AppletSecretKey
        extends javacard.framework.Applet {
  // Secret 3-DES Key
  private DESKey private3DESKey;
  // Payload to execute
  private byte[] shellcode;
  // ...
 private3DESKey=(DESKey) KeyBuilder.buildKey
     (KeyBuilder.TYPE_DES,0x40, false);
 private3DESKey.setKey(keyArray, (short)0);
  // ...
}
```

Listing 3 An applet with sensitive assets.

In the `AppletSecretKey` class, the `private3DESKey` field is the first declared one. According to the JCVM specification, to access to this field, the `getfield_<T>`[3] instruction is used. As it is the first `AppletSecretKey` class field, the `private3DESKey` field is accessible upon the `getfield_a 1` instruction.

Confusing the applet control flow to read fields. To retrieve the content of the `private3DESKey` field, we assume that the code listed in the Listing 4 is executed. The `getArrayAddress()` function returns the address of the content of the byte-array given in parameter.

```
public void ConfusingControlFlow
   (byte[] apduBuffer, APDU apdu, short a) {
  short i = (short) 0;
  short j = getMyAddressByteArray
                        (SERVER_SHELLCODE);
  i = j;
  // returns to the byte code pointed out by
  // the return address register.
  return;
}
```

Listing 4 Function to confuse the applet control flow.

The compiled version, in Java Card byte code, is shown in the Listing 5.

Regarding to the JCVM implementation, succeeding EMAN2 attack aims at changing the return address register upon a stack overflow from the local variables area. The Figure 1(a) introduces a possible implementation of a Java Card stack.

[3] The type `T` must be either `a` for a type reference field, `b` for a type byte or type boolean field, `s` for a type short field or `i` for a type integer field.

```
public void ConfusingControlFlow
   (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  }
```

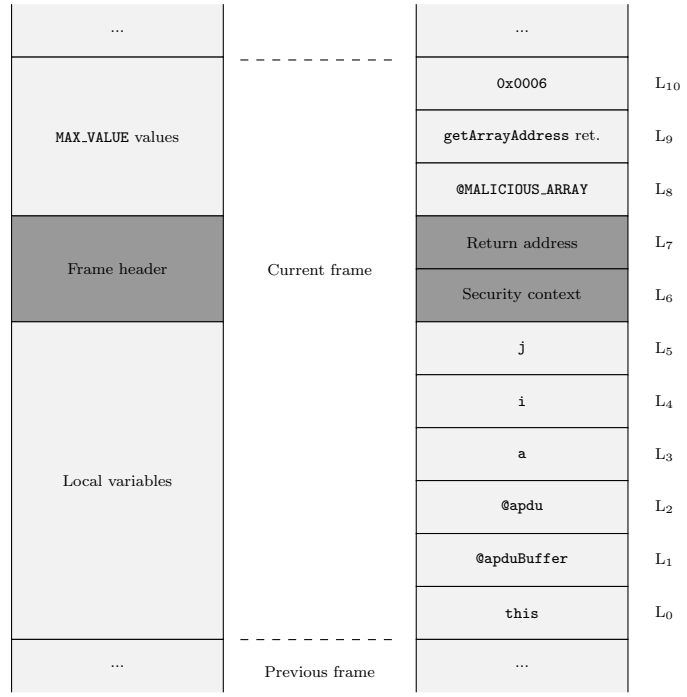
Listing 5 Byte code applet of the function 4.

The Figure 1(b) represents the state of the stack at the end of the method `ConfusingControlFlow()`. In this figure, the return address register is located 2 words after the last local variable. If the JCVM does not check the stack bounds access, the return address register is located in L_7. If the byte code listed in the Listing 5 is changed in such a way that the `sstore 4` (at the offset 0x13, in **bold**) is shifted to `sstore 7`, the return address register will be updated with the content of `j` (L_5) local variable. Since the `return` instruction is executed, the JPC jumps to the address pointed by the return address register. In this case, the JPC points the payload contained in the shellcode byte-array. The shellcode is executed within the caller's stack and the caller's security context.

Reading the `private3DESKey` field. To access `private3DESKey` field upon the shellcode, we use the payload listed in the Listing 6.

```
0x01: [18] sload_0          // push this
                            // reference.
0x02: [83] getfield_a 1     // push
                            // private3DESKey
                            // field reference.
0x04: [1A] aload_2          // Assume that it is
                            // the APDU buffer
0x05: [03] sconst_0
0x06: [8E] invokeinterface  03 02
                              0F 04 // getKey()
0x0C: [3B] pop
0x0D: [7A] return
```

Listing 6 Payload to read the content of a cryptographic key.



(a) Generic Java Card stack.

(b) A Java Card stack implementation.

Fig. 1 Java Card stack characterization.

In the shellcode, shown in the Listing 6, the `private-3DESKey` field is copied to the content of the local variable 2 by the `getKey()` function. In this discussion, the local variable 2 is a reference (pushed by the `aload_2` instruction) to a byte-array like the buffer APDU. When the key is copied to the buffer APDU, the deciphered key is sent out of the card. Indeed, in the memory, the key container ciphers the key value but, when a key owner accesses to it, the JCVM deciphers the key value.

4.2.4 Conclusion

The integrity of application data is often used in Java Card and is called secure storage. It consists of mainly a dual storage or a checksum to verify whether the modification of the field is only done through the VM. Another integrity check concerns the VM structure and in particular the frame context. By using the EMAN2 attack, it is possible to modify the return address in the frame using unchecked local variable indexes. Several old smart cards available on the web markets might be flooded by the modification of the CFG. Thus, it is possible to jump into an array which contains any shellcode.

For preventing the execution of a shellcode, there is the possibility to re-encode on the fly during the linking phase of the value of byte code. So, if someone tries to execute an arbitrary array, he will not be able to obtain the desired behavior. In such a method, the encoded value depends on a dynamic variable, using the JPC for example as a nonce is enough to avoid any brute force attack for guessing the scrambled value.

There are lot of possibilities to protect the data and the execution of a code into the VM. Unfortunately, if all of them are activated during the execution of an application, the performance of the smart card will drastically decrease reaching an unacceptable level. For that reason, most of the smart cards available on web market implement the bound check countermeasure which has been demonstrated efficient enough to mitigate any exploitable shellcode.

4.3 Checking the Jump Boundaries

An attack as the EMAN2, presented in the previous section, modifies the return address such that while it returns from method `f()` the control is transferred to the shellcode instead of the caller. But, the execution of the shellcode is done within the execution context of the caller as shown in Figure 2. In such a case, when the shellcode ends with its own `return` instruction, it goes back to the caller of the caller of the method `f()`. The shellcode cannot be embedded within the method `f()` and thus is implemented as an array stored in a

different area of the method. Then, the offset of the array where the shellcode is stored is different from the offset of the method.

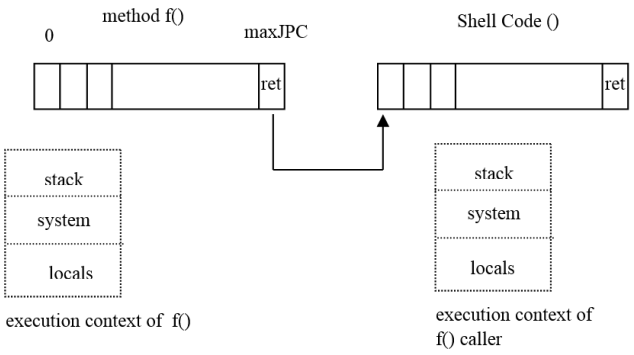


Fig. 2 Description of the execution context.

An affordable countermeasure against the execution of shellcode is to verify if the code is still executing within the boundaries of the current method. For each method, the system maintains several information as `maxJPC`. So, the domain of the JPC of a method belongs to [`minJPC`, `maxJPC`]. The countermeasure is implemented in each instruction concerned by a branch, *i.e.* `goto, ifeq, tableswitch`, etc. The operand following the instruction is checked if it belongs to the domain of the method.

This countermeasure does not prevent to jump to a shellcode but restrict the semantics of the shellcode to a linear code. In particular, no loop can be used, no condition evaluation and so on. As an effect, it becomes impossible to use a shellcode for dumping the memory. The first step is to bypass this countermeasure.

4.4 Bypassing the Embedded Countermeasure

Two ways are possible to bypass the countermeasure. Both of them are related to the non completeness of the countermeasure. The first one is to use the exception mechanism to transfer the control flow and data to the caller. It requires to rebuild in the caller the control flow using the catch mechanism of Java. Thus, the exception object is propagated to the caller. If a handler is present in the method, it can take decision using the `reason` embedded in the exception object.

The second possibility is to split the original code into several fragments representing each basic block adding a preamble and a postamble. The preamble is used to initialize each variable of the basic block. To provide input data to any of the basic blocks stored into the array, we use the caller context, *i.e.* the argument of the `dummy()` method. The number of arguments of the dummy method must be the maximum number of arguments of all the basic blocks for each type of data. The only constraint is that the order of the parameters of the `dummy()` method must be strictly the same as the `shellCodeLaucher()` method because they share the same execution context

The postamble of each basic block has two parts. An instruction `sspush value` is inserted and its value is the variable that is evaluated at the beginning of the next basic block. And secondly, an instruction `sreturn` finishes each of the basic block. All these basic blocks are stored consecutively into an array. The control flow is then assumed by a specific `controlFlow()` method that controls the correct sequencing of each basic block. The CFG is implemented into this method which contains only decision instructions and calls to the `dummy()` method. This intermediary method plays only the role of embedding the context execution of the shellcode and invokes the `shellCodeLauncher()` method. This latter is the one patched thanks to the EMAN2 attack.

Once the `shellCodeLauncher()` method ends its execution, it transfers the control flow to one of the basic block stored into the array. At the end of the shellcode, the `return` instruction is executed which transfers the control flow to the `controlFlow()` method as shown in the Figure 3. It is important to notice that the execution context of the shellcode is the `dummy()` method and not the `shellCodeLauncher()` method.

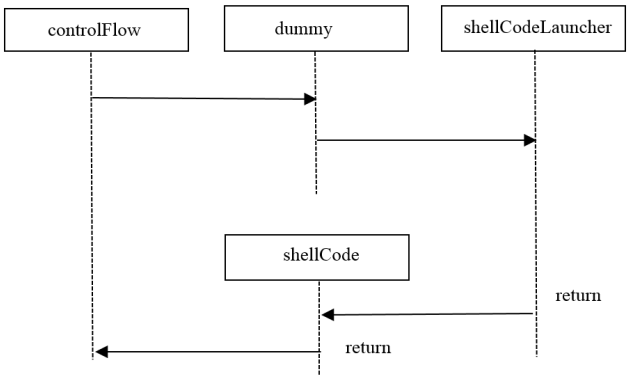


Fig. 3 Control flow derivation.

Within this method, we are able to execute in a shellcode only linear part of a code delegating the control flow part into the original method such that, no branching instruction is executed outside the domain of the method.

4.5 Automatic extraction

The proof of concept described here has been automated with a program transformation. Our tool requires as input a method into a CAP file. It automatically creates an array, extract the basic blocks of the method, stores them into the array and replace them by an invocation to the method `shellCodeLauncher()`. Thus, we have add to the CapMap tool the capability to represent a method with a linked list of basic blocks.

4.5.1 Basic Block extraction

The CapMap tool allows to manipulate the CAP file as a set of objects. We can define a basic block as a sequence of instructions with the properties:

- Only one entry point, meaning no code within it is the destination of a jump instruction anywhere in the program.
- Only one exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block.

We used a block boundaries algorithm to built the block. We scan over the code, marking block boundaries, which are instructions which may either begin or end a block because they either transfer control or accept control from another point. Exception handlers are included in the basic block extraction procedure. Two steps are necessary to built the blocks. With CapMap, we go through the entire method to obtain the list of offsets (destination of a jump). The second step builds the basic blocks starting with the first instruction which begins the first basic block. We search for the end of the current block which consists in an instructions that can be:

- Unconditional and conditional branches,
- Targets of jumps or branches
- Exception handlers

The list of basic block is chained with the previous and the next one. Then, each basic block is copied into the array. In the structure of each basic block, we store the value of the index where the block is stored.

4.5.2 Passing parameters

We can observe in Figure 3 that the execution context of the shellcode is the frame of the method `dummy()`. In that case, the stack and the local variables used by the shell code are those of the method `dummy` and not the original one `controlFlow`. We need to transfer each variable of the original method to the called `dummy` method. For that reason, the execution context of the method dummy is sized to be compatible with the execution context of `controlFlow`.

The method `controlFlow` does not change the state of the method *i.e.* all the assignments are done by the shell code except the initialization. The arguments of the method must be transferred from the caller to the `dummy` method at the beginning of the method. We transfer them in the first call to `dummy` to initialize the execution context. Local variables are initialized in the basic blocks and do not require to be synchronized.

The return value of the method and evaluation of the variables are done in the `controlFlow` method. We need to transfer the required data at the end of each basic block. We automatically insert at the end of each basic block a return instruction with the adequate type and the expected variable as parameter. There are two instructions that require to have two element on top of the stack to be executed: `if_acmp<cond>`, `if_scmp<cond>`. The first one compares two references and the second compares two shorts. But we can return only one value from the `dummy` method. For the reference value, the condition is either equality or inequality. We add in the shell code a subtraction of short values `ssub`, and we return a short. We do not care about the well-typeness of the code: there is no runtime test. In the `controlFlow` method, we replace the `if_acmp_eq` by a `ifeq` which compares if the value on top is equal to zero. For the short value, we use the same mechanism except that we have more cases to evaluate (lower, lower or equal, and so on).

4.6 Completeness of the Countermeasure

The countermeasure is inefficient due to its incompleteness. The objective of the countermeasure is to detect the execution of a shellcode outside its original position by checking the destination branch. Thus, the current countermeasure encompasses only the set of intra procedure instructions (*i.e.* `goto, if, jsr`). It must be extended to the set of inter procedure instructions which is more complicated (*i.e.* `invokestatic, return`). The VM has the information about the `minJPC` and the `maxJPC` which is enough to check destination branches within the boundaries.

For inter procedure instructions the VM requires to know while building or destroying the frame, if the JPC belongs to a valid method. A valid method's JPC depends on how methods are stored within the class. One can suggest to define the boundaries of the methods pool but. But if the method is inherited, then the check must be done with the mother class and not the current one. Moreover, the method must be allowed to be called according to the current instance.

This is threaten naturally by the `invoke` instruction while building the frame, no new check is required. The `return` instruction is more difficult to handle but one invariant at least must hold: at the destination the previous instruction must be an `invoke`. This is enough to ensure the completeness of the invariant check countermeasure.

We have shown in this section that we are able to execute shellcode even in presence of dedicated countermeasure. We got access to our own key in plain-text. We demonstrate now how to get access to objects that belong to another security context bypassing the Firewall.

5 The Server-based Attack

5.1 Client Server Architecture

As explained in Section 2, the Firewall enforces the embedded security which prevents to access an object that does not belong to the user. This is one of the differences between Java and Java Card. Even if a method or a field is `public`, it cannot be accessible from another package which prevents to cast illegally an object reference to gain access to the public methods of the instance.

Checking that an object belongs to the current applet is obtained with the security context. The security context segregates the objects such that all applets in a given package share the same security context and are prohibited from accessing objects having a different security context. The JCRE has root privileges to access objects in any security context. Some objects like the APDU buffer which belongs to the JCRE can be accessed by applets in any context. Once an instance is created, the current security context is attached to the object. Each access (read or write) is checked by the Firewall. It verifies that the object has the same security context than the current one.

To support cooperative applications on-card, the Java Card technology provides well-defined sharing mechanisms. The SIO mechanism is the system in the Java Card platform intended for applets collaboration. The `javacard.framework` package provides an interface called `Shareable` and any interface which extends the `Shareable` interface will be considered as shareable. Requests for services to objects implementing a shareable interface are allowed by the Firewall mechanism. Any server applet which provides services to other applets, within the Java Card, must define the exportable services in an interface tagged as `Shareable`. Then, a client applet can request a reference to that service.

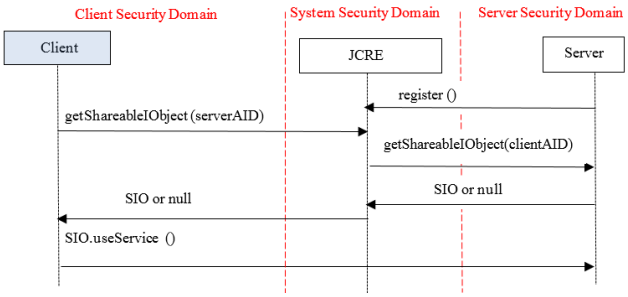


Fig. 4 Shareable interface protocol.

The server that implements the `Shareable` interface is registered within the JCRE as shown Figure 4. Then, the client requests the JCRE to obtain a SIO of a given server where the AID of the server is provided as parameter. The JCRE looks up the server associated with this AID and forward the request with the AID of the client. The called method of the server implements its policy regarding client's request. If the server accepts the request it provides the reference to the client, otherwise `null` is returned. Once the SIO is obtained, the client can invoke any shared method of the interface. At that time, a security context switch occurs and the execution is done under the security context of the server. If the server tries an access to an object of the client, a security exception will be raised. Once the method of the server finishes, the control is given back to the caller (*i.e.* the client) and the execution continues under the security context of the client.

5.2 Revisiting the Shareable Interface

Our contribution aims to abuse the client security context from the server to access the client assets. In this case, the server contains a malicious fragment of code which is executed under the client rights. For this purpose, the control flow is transferred through the EMAN2 attack to a shellcode. This later is contained in the server memory area but is executed with the client security context. This attack succeeded in executing malicious code under another context to read secrets.

5.2.1 Can you trust in a server applet?

Trust in a server applet which provides features is a difficult problem when an application is developed. In fact, a secure application must use evaluated libraries as the Java Card API which commonly evaluated from, for instance, the common criteria scheme during the platform certification.

Several others features may be out of the evaluation scheme. Some extended features may be downloaded on

the smart card developer website (like export file for biometry or vendor' specific API).

On the one hand, an unevaluated API might contain bugs exploitable by an attacker.

One the other hand, as pointed by Bouffard *et al.* [8], an attacker can access secure information using export file fraudulence. By a man-in-the-middle attack, an export file which defines the same features as the legitimate one is used in the standard compilation toolchain[4]. This fraudulent export file has only one difference as the legitimate one, the package AID. As the standard Java Card toolchain does not support file signature, the Java Card BCV is not able to detect an illegitimate export file. For this attack, we will use the this approach.

5.2.2 Linking a legitimate application with a malicious server applet.

A legitimate applet requires features provided by another applet inside the card. The chosen architecture by the Java Card technology is a client/server approach. Therefore, the first applet will be named client applet and the second one is named the server applet. To use the features provided by the server applet, the client applet should be linked with the server export file.

As explained in the section 5.1, the client must be granted by the Firewall to use specific features provided by the server application. This protocol was overviewed in the Figure 4. When a client applet uses features provided by a server applet embedded in the card, the client cannot prove that the server has a malicious behavior. From the Java Card security model point of view, the Firewall prevents any illegal behavior from the client application to the server applet (and *vice versa*).

5.2.3 Confusing the client execution flow from the server.

A method named `sharedM()` is shared by the server applet. This method is called by the client application. The client applet has the right to legally execute this method. During its execution, the `sharedM()` method exploits the EMAN2 attack to return in a malicious shellcode. This shellcode is located at the server side.

First, this method makes the feature required by the applet client. Next, the dark side of the server applet appears and performs an EMAN2 attack to set up the return address to jump inside the array `SERVER_SHELLCODE`.

[4] The export file contains information to translate the Java Class's classes, methods and fields names (encoded as a UTF16 string) to CAP file token value by the Java Card converter.

5.2.4 Executing malicious fragment of code from the server with the client privileges.

When the `return` instruction is executed, the control flow is confused and the JCVM updates the JPC register with the value contained in the return address register. As this value was moved to refer the content of the `SERVER_SHELLCODE`, the JCRE interprets the content of the `SERVER_SHELLCODE` array as the caller instructions with the context of the applet client. During the method returns, the JCVM pops the callee's frame, pushes the returned value on the stack of the caller's frame and continues the byte code execution within the incorrect instructions location.

5.2.5 Obtaining client assets.

Executing the server shellcode under the client applet context, with the client applet rights, gives the possibility to read each client instance fields as cryptographic keys. The aim of this attack is to extort client's assets and thus to jeopardize the client security.

To prove this attack, if we make the assumption that the field 5 of the client instance contains a cryptographic key stored in a byte-array, then the shellcode listed in the Listing 7 exports the key to the buffer APDU.

```
public void getSecretKey(APDU apdu) {
  getfield_a_this 5 // cryptographic key
  sspush 0
  aload  1 // pushing apdu reference
  invokevirtual @Apdu.getBuffer()
  sspush 0
  invokestatic @Util.arrayCopy()
  pop
  aload  1 // pushing apdu reference
  sspush 0
  sspush 50 // size of the cryptographic key
  invokevirtual @apdu.setOutgoingAndSend() }
```

Listing 7 A naive shellcode to read the content of an applet instance field.

The approach introduced in this section is a proof of concept. Indeed, to improve this attack, the client assets should be saved inside the server context.

5.2.6 Storing the client assets inside the server side.

To be closer to the real attacks, the attacker stores the client assets upon the shellcode in the server side. First, the executed shellcode should invoke hidden shared methods to save several client's assets. Second, the end of the shellcode should restore the caller context to continue the client applet execution flow. In fact, the server shellcode should not have effects on the client.

From the attacker's point of view, the internal references to call method inside the card are hidden. This information is a secret. Invoking methods from a malicious shellcode was studied by Hamadouche *et al.* [15]. In their paper, the authors fooled the on-card linker to perform code mutation and obtain the internal references of each method provided by the card API. Based on their approach, the shellcode is improved to invoke shared methods by the server application with a valid internal reference.

The pseudo-code described in the Listing 8 presents a fragment of the shellcode used to save the client's assets in the server context. At the end of the shellcode, the client caller context should be restored.

```
// Reading the client's asset.
invokeinterface @Server.saveAssets
                (/* client's assets */)
// Returning the JPC to the correct caller
// and restore the caller state.
```

Listing 8 Abstract shellcode to read and save the content of a client applet instance field.

Since the shellcode is executed under the client rights, it saves the content of each sensitive client's assets to the server context. Later in the card life cycle, the attacker might extract these client's assets upon the server applet.

5.3 Discussion

Our contribution introduces an attack that focuses on a malicious server. From the client point of view, it is difficult to ensure if the server is evil or not. The Java Card security model was designed to prevent this kind of application to be loaded and executed into the card. The BCV statically checks the application. Dynamically, the executed applet is controlled by the Firewall which checks each asked access. The standard Java Card toolchain architecture are not able to check if the export file use to convert a Java application to a Java Card one is correct (compliance with the features provided by the targeted Java Card implementation).

The requirements from the developer guidelines, associated to the linked Common Criteria evaluation report, force the usage of a BCV. Thus, most of JCVM implementations embed countermeasures that relies on the hypothesis that each installed applet and libraries had been verified by a BCV before loading on card. Moreover, embedding a BCV in the card or performing twice some BCV tests increase heavily the resources required into the card.

Secure applets are evaluated in a certification-like approach. In this case, interfaces and libraries used and imported by the evaluated applet are checked. In addition in the attack introduced in this article, if an unsecure applet is corrupted by a malicious interface, its own security context must be jeopardized. In this case, other (secure) applets or the platform based a sloppy unsecure applet can be corrupted.

To reproduce the attack presented in this paper, a malicious server should be installed in the card. Due to the starting hypothesis (for instance, the EMAN2 attack), this attack is detected by a BCV. To exploit this attack on a JCVM product, the attacker should install it without checking his server applet with a BCV. The attack may either due to a smart card developer who develops a product for a specific market or a smart card manufacturer which implements a back-door.

In the case of a wicked manufacturer, checking the target of a Common Criteria evaluation improves the confidence of the evaluated elements. The embedded elements out of the evaluation target can contain poor security parts and might embed bugs or back-doors.

From the developer's point of view, a server applet should be either evaluated, for instance, *via* the Common Criteria scheme or the applet source code should be checked by the entity that uses the feature provided by the server applet.

6 Conclusion and Future Works

In this paper, we set up an attack to break the confidentiality of assets of the card. In particular, we targeted the most difficult one: the cryptographic keys. For that purpose, we first demonstrated the ability to execute rich shellcode even in presence of dynamic countermeasures like address bound checks. We have been able to extract the control flow part, execute it as legal Java byte code, executing then only linear basic blocks stored into an array. We used code transformation algorithm to automate this part.

In a second step, we applied this mechanism on the client server architecture of the Java Card. Within this approach, the client request a service which is intended to be executed into the security context of the callee. Thanks to the shellcode execution paradigm, we are able to force the execution of the code under the security context of the caller, without any control of the system. The shellcode definition is under the control of the server, thus this approach forces an applet to execute arbitrary code. Thus, it becomes obvious to force

an applet to call the `getKey` method and to send the result to the server. As far as we know, it is the first time the Firewall of a Java Card has been bypassed. We go a step further than Faugeron in her paper where she modified the security context which had no integrity check. Her attack depends on hypotheses concerning the implementation. In the evaluated card, an integrity check is present and we can not use her attack, while the attack presented here does not rely on such an hypothesis.

This attack breaks the security model of the Java Card architecture relying on the separated security domain enforced by the Firewall. This attack relies on two hypothesis, we are able to load ill-typed code into the card, the BCV is not correctly use or it is not up to date and it contains some vulnerabilities, and the second one is the possibility to bypass the dynamic address bound check, an ill-formed applet/library is installed on card and can make an EMAN2 attack. We have explained how to mitigate the second pitfall with a complete implementation of the countermeasure.

References

1. G. Barbu. *On the security of Java Card platforms against hardware attacks*. PhD thesis, Télécom ParisTech, Sept. 2012.
2. G. Barbu, G. Duc, and P. Hoogvorst. Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In Prouff [23], pages 297–313.
3. G. Barbu, C. Giraud, and V. Guerin. Embedded Eavesdropping on Java Card. In D. Gritzalis, S. Furnell, and M. Theoharidou, editors, *SEC*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 37–48. Springer, 2012.
4. G. Barbu, P. Hoogvorst, and G. Duc. Application-Replay Attack on Java Cards: When the Garbage Collector Gets Confused. In G. Barthe, B. Livshits, and R. Scandariato, editors, *ESSoS*, volume 7159 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2012.
5. G. Barbu, H. Thiebeauld, and V. Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In Gollmann et al. [14], pages 148–163.
6. G. 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, Oct. 2014.
7. G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In Prouff [23], pages 283–296.
8. G. Bouffard, T. Khefif, J. Lanet, I. Kane, and S. C. Salvia. Accessing secure information using export file fraudulence. In B. Crispo, R. S. Sandhu, N. Cuppens-Boulahia, M. Conti, and J. Lanet, editors, *2013 International Conference on Risks and Security of Internet and Systems (CRiSIS), La Rochelle, France, October 23-25, 2013*, pages 1–5. IEEE, 2013.
9. G. Bouffard and J. Lanet. Reversing the operating system of a Java based smart card. *Journal Computer Virology and Hacking Techniques*, 10(4):239–253, 2014.
10. M. Farhadi and J.-L. Lanet. Chronicle of a java card death. *Journal of Computer Virology and Hacking Techniques*, pages 1–15, 2016.
11. E. Faugeron. Manipulating the Frame Information With an Underflow Attack. In A. Francillon and P. Rohatgi, editors, *CARDIS*, volume 8419 of *Lecture Notes in Computer Science*. Springer, 2013.
12. E. Faugeron and S. Valette. How to hoax an off-card verifier. *e-smart*, 2010.
13. GlobalPlatform. *Card Specification*. GlobalPlatform Inc., 2.2.1 edition, Jan. 2011.
14. D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, editors. *Smart Card Research and Advanced Application, CARDIS, Passau, Germany, April 14-16, 2010. Proceedings*, volume 6035 of *Lecture Notes in Computer Science*. Springer, 2010.
15. S. Hamadouche, G. Bouffard, J.-L. Lanet, B. Dorsemaine, B. Nouhant, A. Magloire, and A. 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 22rd to 25th 2012.
16. S. Hamadouche and J.-L. Lanet. Virus in a smart card: Myth or reality? In L. Cheng and K. Wong, editors, *Journal of Information Security and Applications*, volume 18 issues 2-3, pages 130–137. Elsevier, 2013.
17. J. Iguchi-Cartigny and J.-L. Lanet. Developing a Trojan applets in a smart card. *Journal in Computer Virology*, 6(4):343–351, 2010.
18. J. Lancia and G. Bouffard. Java Card Virtual Machine Compromising from a Bytecode Verified Applet. In *Smart Card Research and Advanced Applications, CARDIS, Bochum, Germany*, Nov. 2015.
19. J. Lancia and G. Bouffard. Fuzzing and overflows in java card smart cards. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, June 2016.
20. J.-L. Lanet, G. Bouffard, R. Lamrani, R. Chakra, A. Mestiri, M. Monsif, and A. Fandi. Memory forensics of a java card dump. In M. Joye and A. Moradi, editors, *Smart Card Research and Advanced Applications, CARDIS, Paris, France, November 5-7, 2014.*, volume 8968 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2014.
21. W. Mostowski and E. Poll. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In G. Grimaud and F. Standaert, editors, *Smart Card Research and Advanced Applications, CARDIS, London, UK, September 8-11, 2008. Proceedings*, volume 5189 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.
22. Oracle. *Java Card 3 Platform, Virtual Machine Specification, Classic Edition*. Number Version 3.0.4. Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065, 2011.
23. E. Prouff, editor. *Smart Card Research and Advanced Applications, CARDIS, Leuven, Belgium, September 14-16, 2011*, volume 7079 of *Lecture Notes in Computer Science*. Springer, 2011.
24. T. Razafindralambo, G. Bouffard, and J.-L. Lanet. A Friendly Framework for Hidding fault enabled virus for Java Based Smartcard. In N. Cuppens-Boulahia, F. Cuppens, and J. García-Alfaro, editors, *DBSec*, volume 7371 of *Lecture Notes in Computer Science*, pages 122–128. Springer, 2012.
25. E. Vétillard and A. Ferrari. Combined Attacks and Countermeasures. In Gollmann et al. [14], pages 133–147.