International Journal of Information Security (2019) 18:85–100
https://doi.org/10.1007/s10207-018-0401-9

REGULAR CONTRIBUTION

Reverse engineering Java Card and vulnerability exploitation: a shortcut to ROM

Abdelhak Mesbah[1] · Jean-Louis Lanet[2] · Mohamed Mezghiche[1]

Published online: 22 February 2018

Abstract
Secure elements store and manipulate assets in a secure way. The most attractive assets are the cryptographic keys stored into the memory that can be used to provide secure services to a system. For this reason, secure elements are prone to attacks. But retrieving assets inside such a highly secure device is a challenging task. This paper presents the process we used to gain access to the assets in the particular case of Java Card secure element. In a Java Card, the assets are stored securely, i.e., respecting confidentiality and integrity attributes. Only the native layers can manipulate these sensitive objects. Thus, the Java interpreter, the API and the run time act as a firewall between the assets and the Java applications that one can load into the device. Finding a vulnerability into this piece of software is of a prime importance. Finding a vulnerability into a software is often not enough to develop a complete exploit. Here, we demonstrate at the end that a Java Card applet can call the hidden native functions used to decipher the secure container that encapsulates a key. Some previous attacks have shown the ability to get access to the application code area. But the Java Card intermediate byte code detected in the dumps has shown several differences with regard to the specification, which prevents the reverse engineering of the applicative code. Thus, to avoid the execution of shell code by a hostile applet, a part of the byte code stored into the card is unknown. The transformation is done on-the-fly during the upload of an application. We present in this article a new approach for reversing the unknown instruction set of the intermediate byte code which in turn has led to reverse engineering of the Java classes of the attacked card. We discovered during the reverse that some method calls have an unusual signature. Without having access to the native code, we have inferred the semantics of the called methods and their calling convention. These methods have access to the assets of the card without being restricted by security mechanisms like the firewall. We exploit this knowledge to set up a new attack that provides a full access to the cryptographic material and allows to reset the state of the card to the initial configuration. We demonstrate the ability to call these methods at the Java level in an application to retrieve sensitive assets whatever the protections are. Then, we suggest several possibilities to mitigate these attacks.

Keywords Smart card · Java Card · Reverse engineering · Native calls · Vulnerability exploitation

1 Introduction

Smart card is a small tamper-resistant device with few memory. Since the size constraints restrict the amount of on chip memory, the majority of smart cards on the market have at most 4 KB of Random Access Memory (RAM) (for run-time data and OS stacks), 256 KB of Read Only Memory (ROM) (OS, and *romized* applications), and 256 KB of Non-Volatile Memory (NVM) (for persistent data). These constraints have a deep impact on software design. A smart card can also be viewed as an intelligent data carrier which can store data in a secured manner and ensure data security during transactions. Smart cards store several assets like PIN, keys and cryptographic algorithms.

✉ Abdelhak Mesbah
abdelhak.mesbah@univ-boumerdes.dz

Jean-Louis Lanet
jean-louis.lanet@inria.fr

Mohamed Mezghiche
mohamed.mezghiche@univ-boumerdes.dz

1 University of Boumerdes, Independence Avenue, 35000 Boumerdes, Algeria

2 INRIA, LHS-PEC, 263 Avenue Général Leclerc, 35042 Rennes, France

Most of the cards are based on the Java Card (JC) specification [22], and some of these cards are able to load and execute applets after post-issuance. The GlobalPlatfrom (GP) specification [13] defines the process to load an application into the card. Due to the possibility to load applets, these devices are prone to attacks in order to retrieve these assets using hostile applets.

Many efforts have been made by the smart card industry to increase the security of cards to mitigate such attacks. Guidelines [12], certification process [23] and test suites [9] have been provided for a safe design of such applications and their run time. As for many industries, an important part of the security relies on the obscurity [8]. The source code of the implementation is not public, and thus, the binary code is not available. This latter is one of the assets of the system.

One of the basic assumptions of such a card is that one cannot access to the native layers using only applicative programs written in Java. This restriction is valid under the assumption that such a program passed the secured loading process. Some papers [2,17,21] have shown the possibility to execute hostile Java programs even if they passed this secure loading process. The two first papers refer to security evaluation labs that have been able to execute arbitrary native code even in the presence of a secure loading process on new products. The last one refers to the 56-bit symmetric key used for Over The Air (OTA) applet loading process. In that case, the author brute-forced the key on real SIM product and demonstrated the ability to upload any applications.

The main contributions of this paper versus our prior work are the following:

– In a previous paper [20], we proposed a new attack to forge illegal references that allows us to dump the memory. Unfortunately, we have not been able to reverse the Java byte code due to the presence of unknown instructions. We propose in this paper a method to infer the semantics of these instructions.
– In a second paper [19], we proposed to reuse the concept of reference forgery to reverse the memory management algorithm without having access to the native code, using only the behavior of the data. We have been able to understand how the system data were used. In this paper, we have not yet get access to the native layers, but while reversing the JC Application Programming Interface (API) we discovered unusual method invocation. We infer the semantics of these methods by observing their behavior on the system objects. We also resolved the dynamic linking process. The discovered methods, of course, are not documented and provide an efficient and uncontrolled access to the assets of the card. We demonstrate that we are able to access to the native layers inside a JC application.
– Exploitation of the gained knowledge: we provide several exploitation of this access in particular regarding the ability to reverse the state of the card. The card manages a life cycle state machine defined by the GP specification [13]. They are several transitions before reaching the state `SECURED`. Once the card is in this state, there is no way to backtrack to the previous states, e.g., `OP_READY` or `INITIALIZED`. These card life cycle states are intended for use during the pre-issuance phases of the cards life. The non-reversibility of these states is strictly controlled by the system.
– Finally, our last contribution is a proposition in five steps to mitigate this attack.

The rest of the paper is organized as follows: The first section presents the different solutions to avoid the reverse of the embedded code. In the second section, we introduce our method to disassemble the JC API and to retrieve the semantics of the missing instructions. The third section is related to the reverse of the native layers and the discovering of sensitive methods. In that section, we exploit the reverse engineering information with several examples: read and write methods without any check, breaking the firewall, recovering secret containers and changing the status of the card life cycle automaton. Then, we propose a set of countermeasures related to each step of our methodology. Finally, we conclude in the last section.

2 Java Card specialization or obfuscation?

In a previous work [20], we have reversed the JC memory management algorithm and found a new attack vector. We called it auto-forges, and it provides an access to a memory fragment which belongs probably to the ROM area. We discovered the packages of the embedded JC API. The next step is to reverse this API in order to find all the predefined entry points and also to look for low-level functions, which could allow us to gain more access rights.

2.1 Dumping the memory

Dumping the NVM can be performed by using two different techniques. The first one needs to execute the byte codes `getstatic-putstatic` [4,16]. The argument of these byte codes is a token resolved at link time or dynamically either by an address, or an offset. If an attacker controls the token, he can read and write everywhere. It has been demonstrated in [15] that a simple alteration of the `Reference Location` component authorizes the attacker in case of a link time resolution to have read and write access to the whole NVM. The second technique consists in increasing the size of an array by modifying the meta-data of the array as shown in [5]. Modifying the meta-data often requires to use the `putstatic` instruction, which requires the first technique to be used.

In [20], we introduced the concept of auto-forge. It consists in parsing the memory for detecting a sequence of memory cells such that the Java Card Virtual Machine (JCVM) interprets them as valid meta-data and consequently as an array. Once the sequence is found, it is possible to read and write directly into the memory. Then, it becomes obvious to write new data in the memory, which in turn can be interpreted as meta-data to read more memory fragments.

The capacity to use data as meta-data relies only on the probability to find a correct sequence in the memory. The meta-data are often a sequence of four bytes. The first two bytes indicate the size of the array, the next byte the type and the fourth the security context. Suppose that the cardinal of the domain of the first byte $b1$ is the high part of the size, the constraint requires a nonnegative number, so we have a probability of 0.5 to have a correct value. The second byte $b2$ is the low part of the size, and there is no constraint (any value is valid). The third byte $b3$ represents the type and the cardinal of the domain is k, and the fourth byte $b4$ represents the security context and the cardinal of the domain says i corresponds to the number of packages the attacker wants to load.

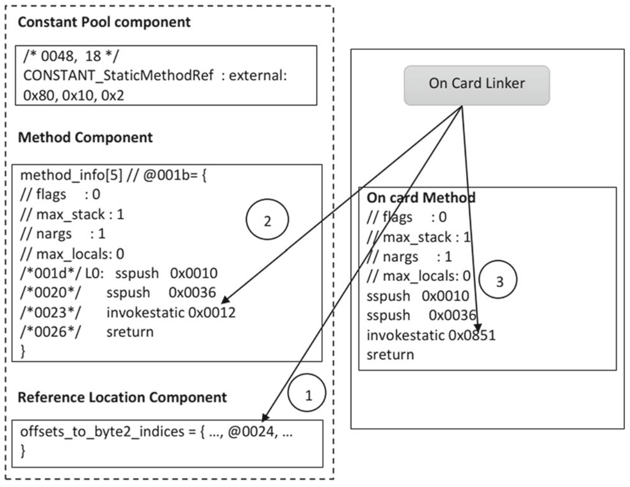
Then, the probability p to find a correct sequence in a memory cell is:
$p = P(b1b2b3b4) = P(b1) * P(b2) * P(b3) * P(b4)$
$p = (k * i)/(256^2 * 2)$.

This is the probability that a valid sequence can be found at one memory cell. We have to compute the probability to find this sequence on the whole memory segment. Let N be the size of the memory, and then the probability to have at least one valid sequence is:
$1 - (1 - p)^N$

If $k = 10$, $i = 10$, and the memory size $N = 100{,}000$, then the probability to find at least one valid sequence is 0.999999996 truncated to 1. Moreover, the probability to have such a pattern in the first 1000 bytes is around 0.67.

N represents the full size of the allocated memory. Non-allocated memory is sometime filled with zero, whereas for other cases, it can be filled with random values or the previous values. We do not have a true random distribution. Nevertheless, this parameter is still in the hand of the attacker. He can upload as many applications as possible, which leads to fill all the NVM. This attack runs well in practice; there are many opportunities to find correct meta-data inside the memory.

Then, with the auto-forge attack, it becomes obvious to read the content of the NVM or the ROM. In the NVM, we find the applications loaded after the issuance while in the ROM we should find the JC API.



```
/* 0048, 18 */
CONSTANT_StaticMethodRef : external:
0x80, 0x10, 0x2
```

```
method_info[5] // @001b= {
// flags    : 0
// max_stack : 1
// nargs    : 1
// max_locals: 0
/*001d*/ L0:  sspush  0x0010
/*0020*/      sspush  0x0036
/*0023*/      invokestatic 0x0012
/*0026*/      sreturn
}
```

```
offsets_to_byte2_indices = { ..., @0024, ...
}
```

```
// flags    : 0
// max_stack : 1
// nargs    : 1
// max_locals: 0
sspush  0x0010
sspush  0x0036
invokestatic 0x0851
sreturn
```

Fig. 1 Link edition while loading a CAP

2.2 Disassembling the dump

We expect to find in the NVM either applications or libraries and the romized API. The JC API provides a framework of classes and interfaces that hides the details of the underlying smart card interface.

The loading format is known as Converted APplet (CAP) file. It consists of eleven component, such as `Header, Directory, Applet, Import, Constant Pool, Class, Method, Static Field, Reference Location, Export, and Descriptor`. Each component describes an aspect of the CAP file contents, such as class information, executable byte code, linking information, verification information, and so forth.

It is of a paramount importance to locate each package's component in the dump, to analyze them, to reverse their implementations and to look for some alternative attacks. When a CAP file is loaded into the card, the JCVM provides a way to link this CAP, and especially a token translation is performed in the `Method component` and `Class component` with the installed JC API as shown in Fig. 1. The Reference Location component specifies the offsets (1) in the Method component where a token should be linked (2) to a card internal reference (3). In the model of card attacked in [6], the token used represents a direct physical address, but in our model of card, the token is resolved at run time. We consider this as another layer of protection. When a method is called in the card, the linker resolves the token, to point and execute the called method.

The specification [22] defines only the external representation of the CAP file to be loaded into the card and not the internal format which is often proprietary. In the early implementation of JC, we had a direct mapping between the external and internal representation of the code. Nowadays, it is usual to find on-the-fly byte code transformations in a smart card. The problem we have to face is to disassemble a binary code without knowing the instruction set of the targeted (virtual) processor. It is difficult to recover the logic of the original program because an examination of the executed code reveals only the structure and logic of the byte code interpreter. Existing techniques for reverse engineering of code protected by virtualization-obfuscation [25] first reverse engineer the VM interpreter; use this information to work out individual byte code instructions; and finally, recover the logic embedded in the byte code program. In case of compression, one byte code can represent a sequence of byte codes with several parameters. Thus, the length of the instruction is unknown, which adds difficulties while trying to resynchronize the byte code flow.

2.2.1 Unknown instruction set

In a previous work [18], we have already found unexpected instructions in the dumped memory. A JC byte code operation is composed of an instruction, encoded on one byte. The valid instructions are comprised between the range `0x00` and `0xB8` and potentially a set of bytes as argument. The two values `0xFE, 0xFF` are defined as implementation dependent and reserved for internal use only. These two instructions are intended to provide traps for functionalities implemented in software or hardware. The remaining values ranging from `0xB9` and `0xFD` (a set of 68 byte codes) are undefined and cannot be used in a valid CAP file.

In [18], we have detected in a dump some byte codes that belong to the undefined byte codes set. The transformation is made during the on-card linking step. Sometime, the translation is straightforward: we can compare the external code and the dumped one and infer the semantics of the unknown byte code. Sometime it is more complex: The pre- and post-conditions are not the same. But we have also encountered the case where one external instruction is translated into different unknown instructions depending on the context.

In such a case, the disassembling process must take into account the possibility to have unknown instructions. These instructions have unknown effects on the consumption and production of data into the memory. In a classic reverse method using the linear sweep algorithm [26], the process is to cancel a sequence if one encounters a non-valid instruction, which means that a sequence of data have been treated as sequence of code. In this case, we have to continue the process inferring the effect on the memory.

2.2.2 Code compression

In the previous section, we have presented the code translation possibility where one instruction is encoded with different byte codes according to different expected behavior. It simplifies the decoding of the instruction. But an opposite possibility is to encounter byte code compression which is used when recurrent sequence of instructions is found in a program. Code compression is a technique that uses extra byte codes to have a more complex instruction set using the aggregation of byte code into a single instruction. Code compression proposes to factorize some instruction sequences with a high occurrence into new instructions, yielding to a more concise program. It uses an extended instruction set. The specification only uses 184 byte codes over the 255 possibilities. By expressing the new instructions as macros over existing instruction as proposed in [7], the JCVM needs only to be extended to support generic macro instructions. Such an approach allows to accept programs with and without compression.

Another approach has been proposed by Bizzotto and Grimaud [3] by using global macro instead of standard macro. The advantage of this approach is to store the macro in the ROM area saving space in the NVM memory. These two approaches are valid only if the compressor is embedded inside the card or it cannot pass the byte code verification process, part of the secure loading procedure required for any certification.

2.2.3 Encoding the code

In his Ph.D. thesis, Barbu [1] proposed a counter measure that prevents the malicious byte code execution. His idea is to scramble each instruction during the installation step, such that the code is byte code verifiable before loading it. For that purpose, each JC instruction ins performs a xor with the K_{xor} key. The hidden instructions (and their parameters) perform the following operation:

$$ins_{hidden} = ins \oplus K_{xor}$$

If an attacker tries to interpret a dump, he cannot read the code without the knowledge of the K_{xor} key. Thus, to find the xor key, he just should change the Control Flow Graph (CFG) of the program to a `return` instruction. As defined by the JC specification, the associated opcode is `0x7A`. With a 1-byte xor key, this instruction may have 256 possible values. A brute force attack offers the way to find the xor key.

In [24], the authors suggest an improvement by adding the value of the Java Program Counter (JPC) to execute the hidden instruction, such that the coding of an instruction is variable up to its position into the byte array. The same values at different position do not have the same semantics. The computation becomes:

$$ins_{hidden} = ins \oplus K_{xor} \oplus \text{JPC}$$

The `JPC` value depends to where each instruction is stored in the smart card memory. Without the knowledge of where each instruction is stored in the NVM memory, an attacker has no possibility to decode the byte code stored.

2.3 Conclusion

We have seen in this section the possibility to both, dump the memory and the difficulty to disassemble its content. The NVM stores either data or programs used by the native processor but also the virtual processor having different semantics and different object layouts. The Instruction Set Architecture (ISA) of both native processor and virtual processor is either totally unknown or partially unknown. The challenge is to separate the two layers in terms of data and program and also infer the unknown instructions.

3 The reverse of the Java Card API

3.1 Reversing the binary dump

We discovered in [19] that the information about all the applets defined in an installed package which are element of the static part (Listing 1) is stored in a byte array, which we called `TabPackage`.

```
Structure static_part {
    PAID paid
    RefPAID refPAID
    TabPackage tabPackage
    SC8 sc8
    StaticHeap stHeap
}
```

Listing 1 Static part of the package

We analyze the content of this array. On the one hand, we find the different components kept by the card (Listing 2). Among these components, we can list the `Applet` component, which contains the Applet IDentifier (AID) of each defined applet, the `Class` component that contains all the classes of all the applet's classes, the `Method` component that contains all the methods of all applet's methods. On the other hand, we find some information added by the card that facilitate the manipulation of the internal structure, e.g., offsets toward the beginning of each component. We notice that the structure of `TabPackages` is very close to the CAP representation. However, the resolution of links between these components is different from the point of view of a CAP. It uses a dynamic approach for resolving the tokens.

```
Structure TabPackage {
    u2 applet_component_offset
    u2 class_component_offset
    u2 method_component_offset
    u1 package_minor_version
    u1 package_major_version
    union{
        u2 undefined
        export_component export_cp
    }
    applet_component applet_cp
    class_component class_cp
    method_component method_cp
    u11 undefined
    u1 static_field_count
    u1 undefined
    u1 static_field_ref_count
    u10 undefined
}
```

Listing 2 Structure of the `TabPackage`

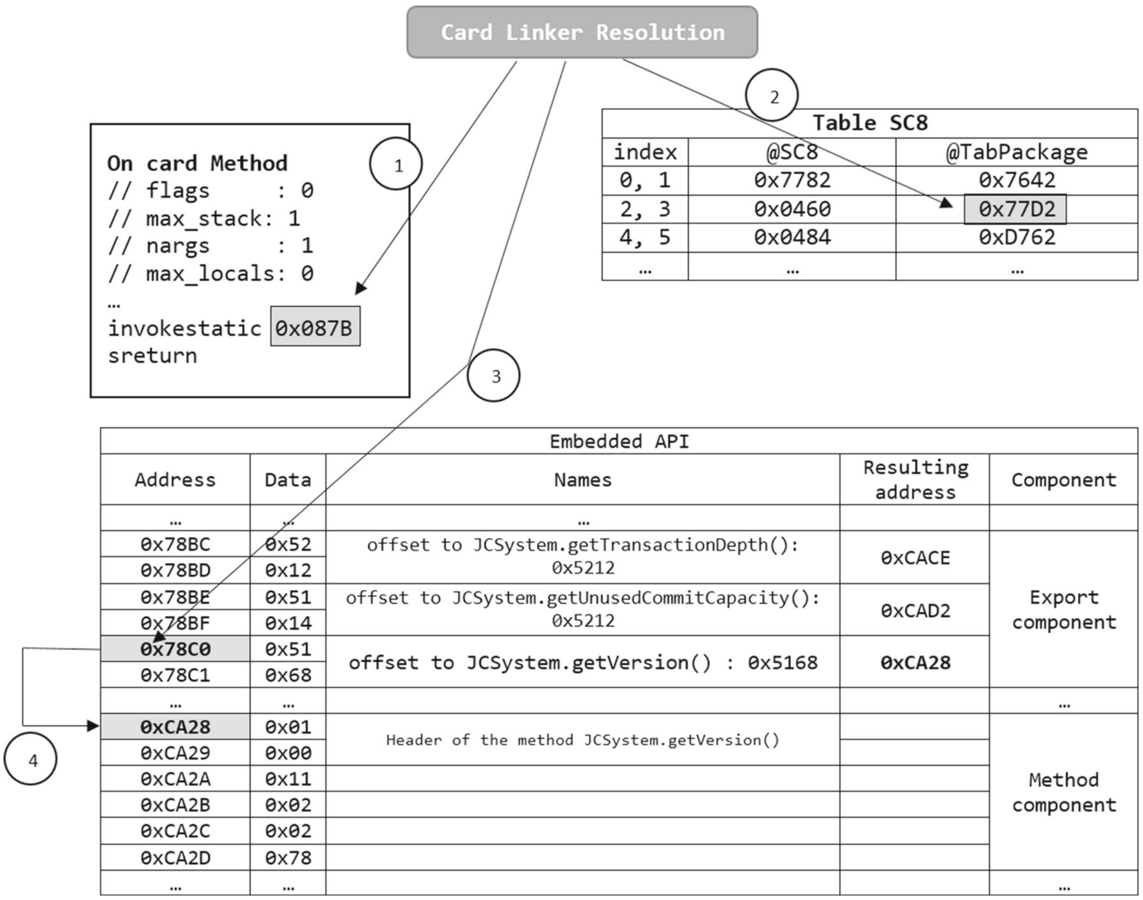
We use this knowledge of the internal structure of an installed package to reverse the embedded JC API. We can separate the code of each component. The more interesting is to characterize this API and to identify the real address of each class, interface and method. To do this, we have first to reverse the dynamic linking process used at run time by the card, to match each token to a physical address.

3.2 Reversing the dynamic linking resolution

The card involved uses tokens to call methods and to instantiate classes instead of direct physical addresses. In order to reverse the dynamic linking resolution, we install our own libraries and we analyze how the card manages the resolution of these tokens. The dynamic linking process of these tokens can have a direct relation with the `tableSC8` found in [19] and presented in Listing 3 or it can be an offset added to the current address. This table contains the couples of references for all pre-installed and installed packages (reference to the `SC8`, reference to the `TabPackage`). There are two ways to manage the access to all elements of the installed packages, and the instructions have to discriminate them.

```
01 00 84 02 02 01 00 58 //Header
//[@SC8    , @tabPackage]
[@0x7782, @0x7642], //preinst.: java/lang
[@0x042C, @0x77D2], //preinst.: **Csystem
[@0x0460, @0xD762], //preinst.: **Csecurity
[@0x0484, @0x0A33], //preinst.: **sd
[@0x09CC, @0x0508], //installed Package 1
[@0x145C, @0x0D98], //installed Package 2
[@0x0000, @0x0000], //unused entry
[@0x0000, @0x0000], //unused entry
  ...
```

Listing 3 TableSC8



```
On card Method
// flags     : 0
// max_stack: 1
// nargs     : 1
// max_locals: 0
…
invokestatic 0x087B
sreturn
```

Table SC8		
index	@SC8	@TabPackage
0, 1	0x7782	0x7642
2, 3	0x0460	0x77D2
4, 5	0x0484	0xD762
…	…	…

Embedded API

Address	Data	Names	Resulting address	Component
…	…	…		
0x78BC	0x52	offset to JCSystem.getTransactionDepth(): 0x5212	0xCACE	Export component
0x78BD	0x12			
0x78BE	0x51	offset to JCSystem.getUnusedCommitCapacity(): 0x5212	0xCAD2	
0x78BF	0x14			
0x78C0	0x51	offset to JCSystem.getVersion() : 0x5168	0xCA28	
0x78C1	0x68			
…	…			…
0xCA28	0x01	Header of the method JCSystem.getVersion()		Method component
0xCA29	0x00			
0xCA2A	0x11			
0xCA2B	0x02			
0xCA2C	0x02			
0xCA2D	0x78			
…	…			…

Fig. 2 Dynamic token resolution

To manage these differences, the card uses some additional instructions that are not specified in the JC specification to make the process more efficient. For instance, the card uses some additional instructions to call methods, e.g., `invokestatic (0xC6)`. These additional instructions are used by the card to manage internal[1] and external[2] methods calls. The token's resolution method changes according to the choice of the instruction. Then, for each kind of method invocation, there is a specific resolution.

3.2.1 The instruction `invokestatic`

This instruction is generated by the compiler with the opcode `0x8D`, but inside the card it can be represented by two different opcodes. These opcodes are changed during on-the-fly loading. If the invoked method is internal to the package, it uses a `0xC6` opcode, and if the invoked method is external to the package, the opcode remains unchanged. The token used for each representation is resolved in different manners.

[1] Internal call: the called method is in the same package as the caller.

[2] External call: the method called is in a different package that the caller method.

`invokestatic` external call format: `0x8D`

The resolution of the token (Fig. 2) has a direct relation with the `tableSC8`. It is used to extract two kinds of information. The first one is an index on the `tableSC8` where we can locate the `TabPackage's` reference of the package where belongs the called method. The second one is used as an offset which is added to the reference of the `TabPackage` table. The resulting address is a reference to the item table `static_method_offsets[]` in the export component of the targeted package. The entry in this reference represents a direct offset to the method, which is added to the current address where it is stored. The resulting reference represents the physical (direct) address to the header of the method.

`Format:` 0x8D byte1 byte2

`Resolution:` byte1 and byte2 are used to construct a short `token`. In Fig. 2, the instruction is `invokestatic 0x08 0x7B` which is an external call to the `getVersion()` method, of the `javacard.framework.JCSystem`. Thus, the `token` has the value `0x087B`. From this `token`, we extract the index in the `tableSc8` to the corresponding package using the inferred formula:

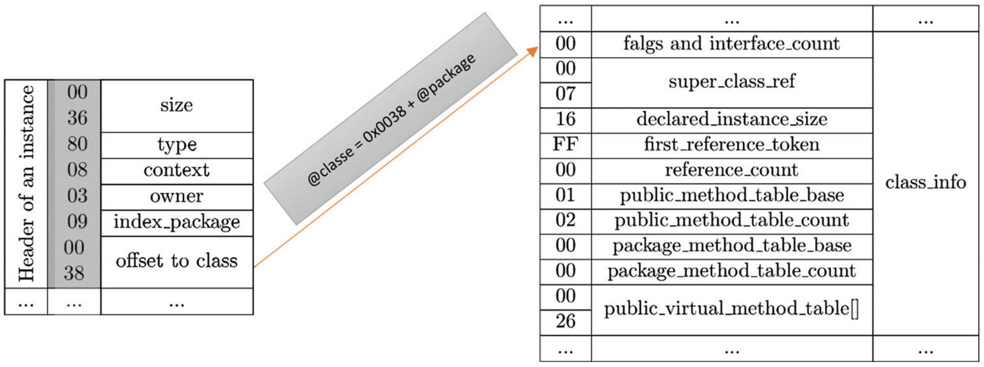


Fig. 3 Link instance to class using meta-data

(1) `index_package=((token/0x0800)*2)+1`. At that step, the index has the value 3. Then, the linker retrieves the address of the package.
(2) `@Package=tableSC8[index_package]-8`. We have retrieved the address of the package: `0x77CA` (`0x77D2-8`). With the already obtained `token`, we extract the offset to add to the package's address found using the formula: `offset_export_component=token % 0x0800`. In that case, the offset is `0x7B`.
(3) `@Offset_to_method=@Package+(2*Offset_export_component)`. These two information are used to find an entry in the item `static_method_offsets[]` in the `Export` component of the targeted package. The offset stored at the address `0x78C0` is added to this address to get the physical address of the static invoked method.
(4) `@method_invoked=@Offset_to_method+Offset_to_method;` In our example at `0x78C0`, we find the two bytes `0x51 0x68` which are concatenated to obtain a short value. Thus the method `getVersion()` is located at the address `0xCA28`.

`invokestatic` internal call format: `0xC6`
`Format:` 0xC6 byte1 byte2

`Resolution:` byte1 and byte2 are used to build a short `shortOffset`. This `shortOffset` represents a direct offset to the called method, from the current address of this `shortOffset` using the following formula:
`@method_invoked=@current address+1+shortOffset`.

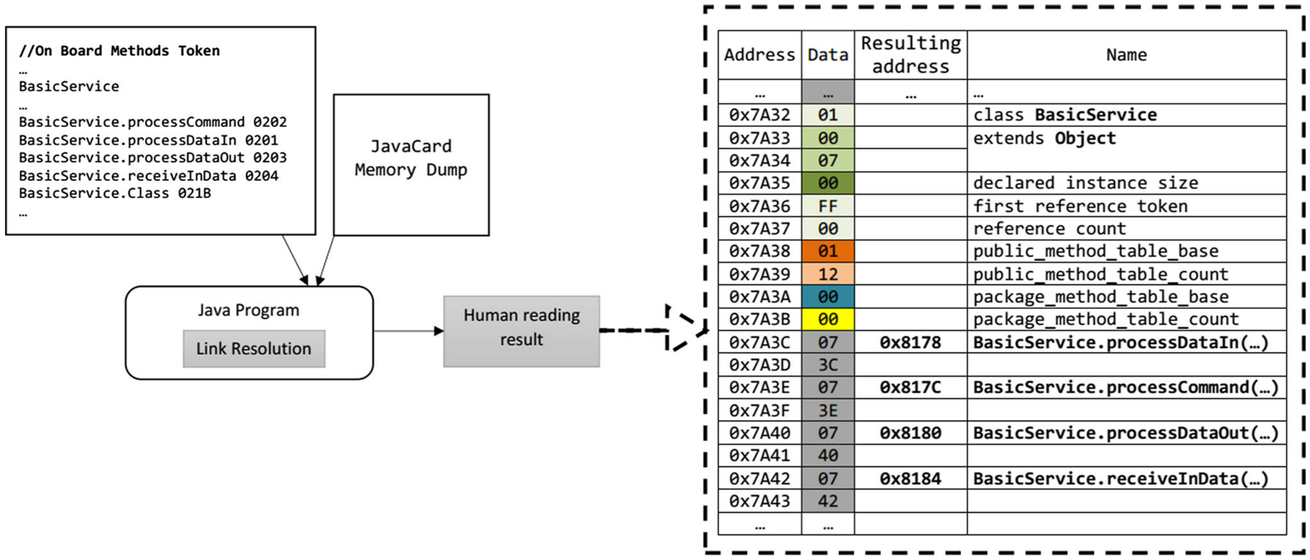
3.2.2 The instruction `invokespecial`

The external call format of this instruction is `0x8C`. It is transformed into several instructions depending on the targeted method. If the method refers to a private instance method, the internal call is changed to `0xCC` and is resolved in the same way as the `invokestatic` for internal call. If the method is an instance initialization method, we get two cases. If it is an internal method, it is changed to `0xCC` and treated like the previous case. If the method is external, the call of the `invokespecial` instruction is changed on-the-fly by the value `0x8D` and it is resolved in the same way as the external `invokestatic`. The last case concerns any super class method. In that case, there is no change for the instruction and the treatment is similar to `invokevirtual`.

3.2.3 The instruction `invokevirtual`

The `invokevirtual` is not changed, whether the call is internal or external. The `invokevirtual` instruction uses the object (instance class reference pushed in the stack) and its parameters to locate the method's class. It uses the first byte argument of the token as a number of arguments to pop from the top of the stack. The second byte argument is used to locate the index of the offset of the called method in the item `public_virtual_method_table[]` in the `class_info` of the `Class` Component of the targeted class, or hierarchical class.
`Format:` 0x8B byte1 byte2
`Resolution:` To locate the class of the called method, we use the meta-data of the class instance (objectref). In [20], we saw that the meta-data of an instance holds an index to the reference of the `TabPackage` in the `tableSC8`, but in our case this is not enough. We want to get the reference to class's instance. We analyze the meta-data of an instance and find that the last two bytes represent an offset that are added to the reference of the `TabPackage`. The resulting address represents a reference to an entry item `class_info` in the `Class` component of the targeted package (Fig. 3). We can use this understanding of the meta-data and the dynamic resolution to forge our own instances to get access to private classes and call private methods of the API.

The dynamic linking process collects and combines various pieces of information at run time. The reverse engineering of this process allows us to characterize the embedded API.



```
//On Board Methods Token
…
BasicService
…
BasicService.processCommand 0202
BasicService.processDataIn 0201
BasicService.processDataOut 0203
BasicService.receiveInData 0204
BasicService.Class 021B
…
```

Address	Data	Resulting address	Name
…	…	…	…
0x7A32	01		class **BasicService**
0x7A33	00		extends **Object**
0x7A34	07		
0x7A35	00		declared instance size
0x7A36	FF		first reference token
0x7A37	00		reference count
0x7A38	01		public_method_table_base
0x7A39	12		public_method_table_count
0x7A3A	00		package_method_table_base
0x7A3B	00		package_method_table_count
0x7A3C	07	**0x8178**	**BasicService.processDataIn(…)**
0x7A3D	3C		
0x7A3E	07	**0x817C**	**BasicService.processCommand(…)**
0x7A3F	3E		
0x7A40	07	**0x8180**	**BasicService.processDataOut(…)**
0x7A41	40		
0x7A42	07	**0x8184**	**BasicService.receiveInData(…)**
0x7A43	42		
…	…		

Fig. 4 Automating the characterization of the API

3.3 Reverse engineering of the API

Before starting this process, we have had to succeed with several preliminary steps that consisted of:

– Understanding of the internal structure of the packages;
– Reverse engineering the dynamic link resolution;
– Characterization of all the tokens of the JC API.

Then, we locate precisely the code of all classes, interfaces and methods defined in all the packages of the JC API in the various dumps we got. To complete this characterization, we apply the same process to the GP API to locate the implementation of all the classes, interfaces and methods of this API. In these pre-installed packages,[3] we find the following packages:

– `java/lang`, which contains the package: `java.lang`
– `**Csystem`, which contains the packages: `javacard. framework, java.io, java.rmi, javacard x.framework, org.globalplatform ...`
– `**Csecurity`, which contains the packages: `javacard. security, javacardx.crypto ...`
– `**sd`: System's applet.

We developed a Java program, which transforms the raw data of the dump into a structured view of the different methods. The inputs are a JC dump and the list of all the API tokens. This program generates the list of all packages and name's attribution for all public interfaces, classes and methods of the API. A human readable format is reported with the packages found in the JC dumped memory in Fig. 4; this figure shows the result of the program analysis for the class `BasicService` of the API. We have now access to the binary code of all the implementations of the API. This access allows us to:

`Get fake instance:` The embedded JC API uses some specific classes, which are not public. In order to instantiate these classes and have access to their methods, we use the knowledge of the dynamic resolution to calculate an instance with tailored meta-data (calculate the `size` of the instance, the `index_package` and the `offset to class` (Fig. 3)) which allows us to point directly the targeted class and invoke its virtual methods.

`Invoke specific and private methods:` In addition to specific classes, the JC API uses some specific and private methods. In order to access to these methods, we calculate the corresponding token toward the `tablSC8`, which allows us to invoke the targeted method. Moreover, we can now directly write a rich shell code that contains calls to all the methods available in the API, e.g., `0x8D 0x08C5` which invokes a specific method used by the card to create an array with key container feature.

3.4 Reverse engineering of the additional instructions

During the reverse engineering of the API, we noticed the presence of some extra opcodes in addition to those presented in Sect. 3.2. The JCVM specification indicates that the allowed opcodes are ranging from `0x00` to `0xB8`, and two reserved opcodes 0xFE and 0xFF for internal use by a

[3] Part of the package's name has been obfuscated.

JCVM implementation. All the other opcodes are undefined, and the specification does not define the behavior for this opcodes.

However, we discovered that in the targeted card 16 new opcodes are used; some of them have described previously as specialization of the invoke instructions. At that step, we have to infer the condition of use of the instruction (the pre-condition) and the memory production (post-condition) on the stack or in the memory. The JCVM is a stack-oriented machine which executes all operation on the stack. But here, we are in a situation where we compose several byte codes in one. This instruction can have memory effect. Once we know the pre and post of this instruction, we have to understand its semantics. For that purpose, we can either try to infer it by reading the specification of the function or generate an applet that uses this instruction and observe the effects.

3.4.1 Inferring the pre- and post-condition

The approach is to use the technique used by the JC byte code verifiers using abstract execution on types instead of values to analyze the raw data. The execution of the code is performed at the type level instead of the values as in normal execution. For each instruction, we check that the stack before the execution of the instruction contains enough entries and that these entries are of the expected types for the instruction. We simulate the effect of the instruction on the operand stack, popping the arguments, pushing back the types of the results. We explain this using the reversed function `isCommandChainingCLA()`. We provide in the Listing 4 a fragment of the method. The signature of the method is `void` for the parameter and `boolean` for the return. If one observes the header, he remarks that the function has one local variable and is non-static due to the implicit argument `this`.

At line 8, we have a branch to the label L1. A property of Java is that at each label, the stack is empty, which is the case at label L1. We push a short on top and the method returns a short value which is the boolean 0. At that point, we have inferred that the stack is empty at line 9. We then execute the instruction at line 9 that produces a reference, and the instruction at line 10 which produces a short on the stack.

We have the pre-condition for this unknown instruction at line 10; We have to infer its production. We backtrack from a known state, at line 15 after the instruction `sreturn` the stack is empty, but this instruction needs as pre-condition a short. The previous instruction at line 14 produces a short and does not require anything. At line 13, we have an `if_scmpne` which produces nothing but consumes two shorts. At line 12, we produce a short. This implies that the production of the unknown instruction is a short.

```
 1 public boolean isCommandChainingCLA(){
 2 02          //flags: 0 max_stack: 2
 3 20          //nargs: 1 max_locals: 1
 4 ...
 5 0x11 0x7412 sspush    0x7412
 6 0x2C            astore_1
 7 ...
 8 0x61 0x0E  ifne   L1        0x7E7E
 9 0x19       aload_1
10 0x03       sconst_0
11 0xBB 0x10  unknown   0x10
12 0x10 0x10  bspush    0x10
13 0x6B 4     if_scmpne   L2       0x7E7C
14 0x04       sconst_1
15 0x78       sreturn
16 ...
17 L1
18 0x0        sconst_0
19 0x78       sreturn
20 }
```

Listing 4 Type inference for `isCommandChainingCLA()` method

Stack
..., ref, short
⇒
..., short

We can refine this inference with a manual code analysis. At line 5, we have a strange sequence, the method pushes on top of the stack the short `0x7412` and store it in local 1 using an `astore_1` instruction. We have here clearly an illegal sequence, where a short is stored into a reference. This proves that this API has not passed the byte code verification process which would have detected the default. We know that `0x7412` is the fixed address of the Application Protocol Data Unit (APDU) buffer.

3.4.2 Inferring the semantics

We have now to infer the transformation done by this instruction. It uses three parameters: Two of them are passed through the stack, the address of the APDU buffer and a short having the value 0. The third parameter is provided as an argument, a short having the value 0x10. The specification expresses that this method *Returns whether the current APDU command is the first or part of a command chain. Bit b5 of the CLA byte if set, indicates that the APDU is the first or part of a chain of commands*.

The CLA byte has the position 0 inside the buffer, and the byte b5 can be logically coded as 0x10. According to the value produced by this instruction, the method returns `true` or `false`. To confirm our hypothesis that the short pushed on top of the stack is the position in the buffer, and the argument is the mask tested at this position, we write on our own application to check this instruction. This confirmed that this instruction is a combination of a `saload`, `bspush` and `sand`.

Stack
..., **ref, short**
⇒ saload
..., value ⇒ bspush 0x10
..., value, value
⇒ sand
.., **short**

This stack usage is the same as the one inferred in the previous section.

3.4.3 The set of unknown commands

We reverse all the additional instructions (Table1); some of them represent compressed instructions, for instance the instruction `0xBB`, some others are specialized instruction to access element on the stack, and others are used for a specified purposes, i.e., to call native methods.

4 Native calls

In the previous section, we described how we locate each class, interface and method of the API. We analyze and reverse the code of these methods. Some of the called methods use some specific headers (Fig. 5) and additional instructions as described in Listing 6. These headers diverge to those specified in [22]. We demonstrate hereafter that we are in the presence of native method headers. They can be distinguished from the other headers by their non-standardized flag values (the second least significant bit is set to 1), these native headers are used by the JC API to call native methods. Thus, we discover the mechanism to call native methods.

4.1 Specific headers

The JCVM specification [22] defines a method as a `method_header_info`, described in the Listing 5, and its associated byte code.

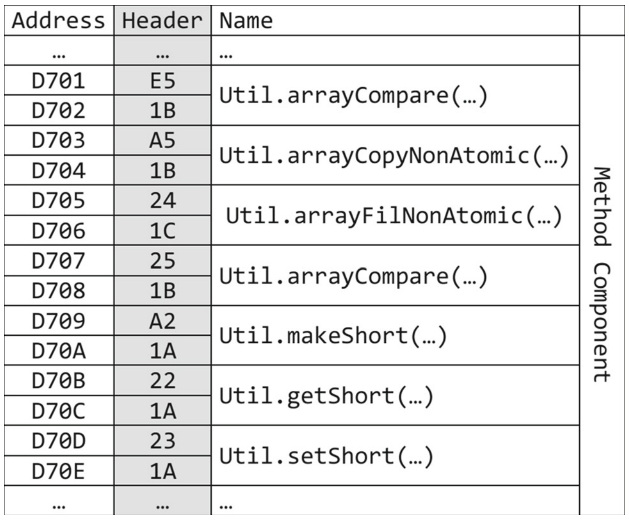
```
method_header_info {
ul bitfield {
     bit[4] flags
     bit[4] max_stack
}
ul bitfield {
     bit[4] nargs
     bit[4] max_locals
}
```

Listing 5 Java Card method header info

The `flag` item is a mask of modifiers valid for the current method. The `max_stack` defines the maximum number of elements required on the operand stack during the execution of the method. The `narg` item corresponds to the number of parameters passed to the function, and `max_locals` is the number of local variables declared by the method.

For the flag value, three defined possibilities are expected:

- 0x0 : It is a regular method
- 0x4 (ACC_ABSTRACT): The method represents an abstract method
- 0x8 : (ACC_EXTENDED): The method represents an extended method
- All other flag values are reserved



Address	Header	Name	
…	…	…	Method Component
D701	E5	Util.arrayCompare(…)	
D702	1B		
D703	A5	Util.arrayCopyNonAtomic(…)	
D704	1B		
D705	24	Util.arrayFilNonAtomic(…)	
D706	1C		
D707	25	Util.arrayCompare(…)	
D708	1B		
D709	A2	Util.makeShort(…)	
D70A	1A		
D70B	22	Util.getShort(…)	
D70C	1A		
D70D	23	Util.setShort(…)	
D70E	1A		
…	…	…	

Fig. 5 Headers of native methods

Each method listed in Fig. 5 represents a header of the API's methods found during the characterization process of the API, and it contains non-standardized flag values (0x2, 0xA, 0xE, ...). Moreover, there is no body for these methods. The card uses these headers to represent native methods.

4.2 Specific instructions

As we have seen previously, the card uses some specific headers to represent native methods. Moreover, it uses a specific instruction to call the native methods. Listing 6 depicts a reversed code of the method `getStatusWord()` of the `javacard.framework.service.BasicService` from the JC API class.

The instruction `0xCD` in the Listing 6 at line (7) is used by the card to call native method, and the low nibble of the high byte of the token used by the card `0x221A` represents the number of arguments of the called method. We notice that the token used in this native call `0x221A` is the same as the definition of the header of the method `getShort()` in Fig. 5. We hypothesize that this call represents a call to

```
public short getStatusWord(APDU apdu){
0x81CF 02          //flags: 0 max_stack: 2
0x81D0 20          //nargs: 2 max_locals: 0
...
0x81DC 11 74 12 // sspush 0x7412
0x81DF 05          // sconst_2
0x81E0 CD 22 1A // invoke native 0x221A
0x81E3 78         // sreturn
}
```

Listing 6 Call to native method

the `getShort()` method. To prove our assumption, we try to call directly the method `getShort()`, by replacing the original call, which was 0x8D 0x08E4, with a native call 0xCD 0x221A in our installed applet. The original opcode and the modified are described in Table 2.

We use the instruction `0xCD` with the right arguments to call the `getShort()` method, the card executes the instruction and returns the value from the given array, which confirms our assumption. We succeed to call a native method from the user applicative area.

As we successfully called native methods, we try to use the native calls as the `getShort()`, to have access to objects which do not belong to our context. Surprisingly, this does not give any privileges, because the verification of context is performed in the native layers.

4.3 Characterization of the native methods

In order to characterize the native methods, we analyze the methods and the headers of the API to construct Table 3 of all native calls. But, as we can see in that table, we cannot find all the names of all the native methods; some of them are not specified in the public API.

The challenge now is to find the function's names of the unknown native headers and tokens (Table 3). Our method consists in analyzing the external API, as it uses some classes that declare some native methods, for instance methods in the `com.sun.javacard.impl.NativeMethods` class, which are native methods called by the external API. Listing 7 gives an instance of these calls in the `JCSystem.getPreviousContextAID()` which calls the `NativeMethods.getPreviousContext()`.

To analyze the equivalent code used in the card, we reversed the embedded method `getPreviousContext AID()`, and we find that it uses a native call, `0xCD 0x2180`, which represents the call of the `NativeMethods.getCurrentContext()` method. To confirm it, we call this native method in our installed applet, and it returns the value of the previous context. This method is implemented in a different way, because as we can see in its `numarg`, it takes a value which is used to specify if we want to return the current context or the previous context. Theoretically, we can get the name of each unknown native function, but there

Table 1 Reverse of the illegal instructions

Illegal instruction	Arguments	Stack	Description
0xBA	/	..., arrayref, index $\Longrightarrow$ ..., value	baload bspush 0xFF sand
0xBB	byteArg	..., arrayref, index $\Longrightarrow$ ..., value	baload bspush byteArg sand
0xBC	/	...,value $\Longrightarrow$ ...	sstore 0x4
0xBD	/	...,value $\Longrightarrow$ ...	sstore 0x5
...	...	...	...
0xC0	/	... $\Longrightarrow$ ..., value	sload 0x4
0xC1	/	... $\Longrightarrow$ ..., value	sload 0x5
...	...	...	...

Table 2 Call native method from user area

Original opcode		Modified opcode	
...	...	...	...
19	aload_1	19	aload_1
03	sconst_0	03	sconst_0
8D 08 E4	invokestatic 0x08E4 //getShort(...)	CD 22 1A	invoke native 0x221A // getShort(...)
7A	sreturn	7A	sreturn

Table 3 Headers and tokens of native methods

Headers/token native	Function's name
0x6111	APDU.getBuffer()
0x6010	APDU.getInBlockSize()
0x6050	APDU.getOutBlockSize()
...	...
0xE51B	Util.arrayCopy()
0xA51B	Util.arrayCopyNonAtomic()
0x241C	Util.arrayFillNonAtomic()
0x251B	Util.arrayCompare()
0xA21A	Util.makeShort()
0x221A	Util.getShort()
0x231A	Util.setShort()
...	...
0x2180	?
0x2202	?
0x3302	?
0x2242	?
...	...

```
public static AID getPreviousContextAID()
{
    byte prevCtx = NativeMethods.getPreviousContext();
    if (prevCtx == 0)
    {
        return null;
    }
    return thePrivAccess.getAID((byte) (prevCtx & 0xF));
}
```

Listing 7 Call native methods in the external API

is a possibility that the card uses some specific function, for instance native call to switch the card mode (development or production mode).

4.4 Native read and write bytes method

In the class `com.sun.javacard.impl.NativeMethods`, we can find two interesting methods, `public static native byte readByte (int paramInt, short paramShort)` and `public static native void writeByte (int paramInt, short paramShort, byte paramByte)`. These methods are used to read and write anywhere in the memory without any control. We analyze the reverse engineered API codes. We find that several methods of the API calls these native methods, with the instructions `0xCD 0x2202` and `0xCD 0x3302`.

We use these two methods in our applets. We notice that the system does not make any check related to the security context. With the `readByte`, we have access to each byte

Table 4 Characterization native methods

Native Token	Function's name
0x2100	readByteRam(byte index)
0x2200	writeByteRam(short index, byte value)
0x2140	readShortRam(byte index)
0x2240	writeShortRam(short index, short value)
0x2202	readByte(short address, short index)
0x3302	writeByte(short address, short index, byte value)
0x2242	readShort(short address, short index)
0x3342	writeShort(short address, short index, short value)
0x2180	readByteVMSTACK(byte index) or readByteVmStack(byte index)
0x2280	writeByteVMSTACK(byte index, byte value)
0x21C0	readShortVMSTACK(byte index)
0x2280	writeShortVMSTACK(byte index, short value)
0x25AB	encrypt(short value1, byte[] tabPlain, short offset, byte[] tabEncrypt, short offset2)
0x252B	decrypt(short value1,byte[] tabEncrypt, short offset, byte[] tabPlain, short offset2)
0x7342	xorify(short address, short index, byte value)
0x6242	dexorify(short address, short index)
0xA11D	isAppletActive(byte owner)
...	...

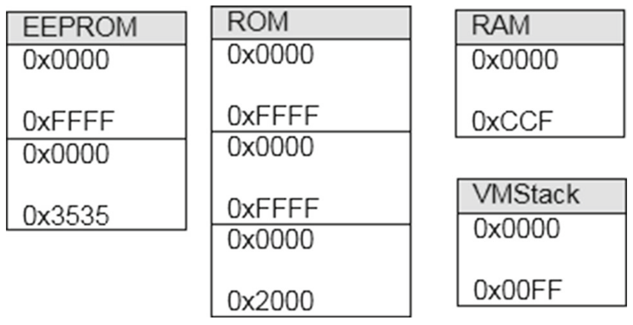


Fig. 6 Memory layout

of the NVM and ROM memories. This provides an access to the part of the ROM that was not accessible with the auto-forges. In addition, with the method `writeByte`, we can write at any address in the NVM memory. This provides us the privileges of the system for reading and writing in memory. We also inferred some others native methods (Table 4). Using these native methods, we can have access to all of the memories presented in Fig. 6.

5 Exploitation of the vulnerabilities

In this paper, we found several vulnerabilities and particularly the possibility for an ill-typed applet to call directly native methods. The most important question is about a potential exploitation by a legal applet of these vulnerabilities. We present hereafter three use cases of this exploitation. They all rely on the classic hypothesis of bypassing the secure download. The last one can be used to put the card in a predefined state (`OP_READY`) which allows to gain access to new resources.

5.1 Breaking the firewall: security context

A JCVM frame contains a set of local variables, an operand stack and all needed data (frame header) to reconstruct the previous frame. A frame header usually contains the security context. Many underflow attacks can gave us an access to this context and maliciously modify it. This is the reason why designers choose to separate the frame header from the stack to prevent the access to these assets.

However, despite this separation, the authors in [10] have found a way to get access to these assets using a frame overflow, which leads to a collision between the separated parts and once again with an underflow attack the frame's header is changed.

We have observed that in the involved card the context used in the Java stack can be modified and does not correspond to the one checked at run time. We analyze all the RAM, and we do not find any value which acts like a context. To understand how the system manages the security context, we analyze the implementation of the method `javacard.framework.JCSystem.getAppletShareableInterfaceObject()` in the embedded API.

This method performs an explicit context switch to return the server applet's shareable interface object in four steps :

- Recover the context and the owner of the server's applet;
- Save the current context and owner;
- Replace the current context and owner by those of the server's applet and call the `getShareableInterface()` on the server;
- Resume the original context and owner.

This method uses a specific memory (VM stack), where the current owner and the context are stored. This memory is accessible only by a native call. The API uses the native call `0xCD 0x2180` to read from this memory and the call `0xCD 0x2280` to write on it (see Table 4). We can find in this memory other important information as the assigned channel, the reference of the current applet's instance and the nested contexts. We use these methods to modify the context and access to any applet objects and methods.

The access to the native layers breaks the segregation provided by the firewall.

5.2 Secure containers for key and OwnerPin

The JC API provides different classes to store in a secure way the sensitive data. These classes are containers for cryptographic keys and Personal Identification Number (PIN) codes. Authors in recent publications [27], Farhadi and Lanet [11] demonstrated that some cards do not implement any integrity and confidentiality protection for these containers. This is not the case in the targeted card, where the containers are properly implemented (confidentiality and integrity). Reversing the API allowed to characterize these containers, particularly the key container (e.g., 3DES, AES, KoreanSEEDKey).

5.2.1 PIN container

An `ownerPIN` instance must contain several fields that we have to retrieve :

- The PIN value stored securely;
- The Try limit which is the maximum number of times an incorrect PIN can be presented before the PIN is blocked;
- The Max PIN size, the maximum length of PIN allowed;
- The Try counter, the remaining number of times an incorrect PIN presentation is permitted before the PIN becomes blocked;
- The Validated flag set to true if a valid PIN has been presented during the current session, must be implemented into the volatile memory.

The method is to create an applet with such an object and a call to all the methods of the API. Then, we can reverse engineer both the methods of the API and understand the data structure used to store the secure container as shown in Listing 8.

```
Structure ownerPin{
u2 reference to the PIN container
u2 reference to transient array of the flag
u2 max Try
u2 Try left
u2 max Pin size
u2 Pin size
}
```

Listing 8 Characterization of the `OwnerPIN`

This structure contains a reference to two objects: the secure container and the volatile array. This later contains the flag indicating that the PIN has been validated, and it is implemented using a transient array. The secure container (see Listing 9) is an array with a particular type `0x89`, which is different from all of the predefined types. This header is a weakness because a simple search of this specific pattern allows an attacker to find it in a dump. In fact, this type is not only used for PIN but for all the secure container including the key containers. After these two references, an object contains the remaining fields. Two fields `max Try` and `Try left` are protected for integrity using a simple XOR. An arbitrary change in these values kills the card.

```
Structure Pin container{
u1 constant
u1 pin size
u1[pin size] encrypted Pin
u2 checksum
}
```

Listing 9 Secure container

The PIN value is stored encrypted to be protected against confidentiality with a checksum to be protected against integrity.

5.2.2 Key container

We used the same approach to reverse engineer key container shown in Listing 10. The structure is quite similar to the `ownerPin`, and it starts with a reference on a secure container and then the expected fields. None of these fields are protected for integrity.

```
Structure key{
u2 reference to key container
u1 type of algorithm
u1 key length
u2 key type
}
```

Listing 10 Key container

The secure container for keys (Listing 11) shares similarity with the secure container for `ownerPin`. Its type is also `0x89`, and any array in a dump with this value refers to such a sensitive data. Then, the key is stored encrypted and a checksum protects the container against any illegal changes.

```
Structure Key container{
u1 key encryption
u1 container length
u1 key length
u1[] encrypted key
u2 checksum
}
```

Listing 11 Key container

5.2.3 Deciphering the secure container

We have in the previous step reverse engineered the data, and now we analyze the code of the API in order to understand the different methods used. With this analysis, we find that the API uses native calls to encrypt the container using the native call `0xCD 0x25AB` for ciphering and `0xCD 0x252B` to decrypt these containers. It uses the same method's call for all type of encryption algorithm and for PINs container. The only difference relies on the first argument of the native calls, which represents the type of the used algorithm for key container.

We use this knowledge to search for containers in the memory dump. We find a Cardholder Verification Method (CVM) object, and we can change its PIN value or get its plain text value without having the relevant privilege and this can be done for any installed applet. We have also found the Secure Channel key set that contains the 3 double length DES Keys (S-ENC, S-MAC and DEK). We believe that, whatever the security degree of these containers is, once an attacker has the hand on the API, he can find a way to decrypt these containers and figure out more exploitation.

This attack offers the ability for an attacker to break the confidentiality and the integrity of the secure containers even object that does not belong to its security context.

5.3 Change card life cycle state

The GP defines a card life cycle state which allows to control access to some functionalities. The main purpose is to increase the security for each state by reducing the available functionalities. The specification states that some of the transitions between the states are not reversible (Fig. 7a). For instance, the transition from `OP_READY` to `INITIALIZED` is irreversible.

The GP provides two methods to change the card's state, i.e., `GPSystem.lockCard()` and `GPSystem.terminateCard()` which lock and terminate the card. But it does not provide a method to manipulate the state. We still can use the `APDU SET STATUS COMMAND` to change the life cycle, but obviously this throws an exception. The state is stored in a secure way. It is protected basically against integrity using a XOR.

Thanks to the characterization of the GP's API, we find the implementation of all the methods' classes of the `GPSystem`. We reverse these methods, and we find where the card stores this state. We cannot modify it directly since

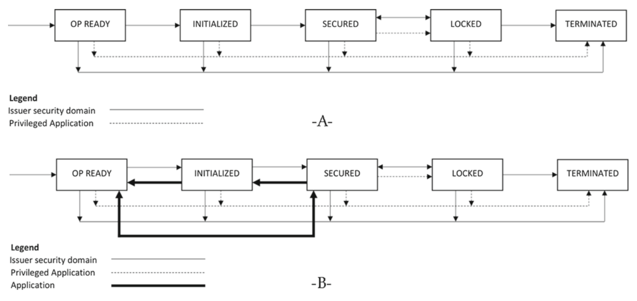


Fig. 7 Card life cycle state. **a** Original Card life cycle. **b** Hijacked Card life cycle

this value is XORed. To do that, we use the native call `0xCD 0x7342` (see Table 4) which is a call to XORing method. Thanks to this call, we can change the state (Fig. 7b) such that an irreversible state can be reached. We are able to go back to the initial state `OP_READY`. If the state value is not XORed, an attacker can use a fault injection to modify it.

The ability to reverse the state of a card to reach an initial state offers the possibility to execute commands that are only allowed in the `OP_READY` state

5.4 Countermeasures

To exploit all these vulnerabilities, we had to go through multiple steps. Each of them could be mitigated by one or more countermeasures. We propose hereafter different possibilities related to the step a vulnerability is used, to avoid such exploits.

Step 1 The first vulnerability is related to the possibility to get an access to the NVM's code. This has been possible due to the use of the forges as presented in [20]. It is based on the illegal use of the meta-data as applet fields in an ill-typed applet. This is basically mitigated by forcing the use of an Byte Code Verifier (BCV). If this scheme, known as secure download process, is effectively used for uploading applet in a product, there is currently no possibility to force its usage on development cards. The only way is to embed the verification process into the card or to add enough tests that could detect all the ill-typed applets.
Step 2 It corresponds to reverse engineering the NVM's code. It allows us to find the memory layout of this card, through the usage of Table `tableSC8` as discovered in [19]. This gives us access to all the references of all installed or pre-installed packages. The analysis of this table allows us to understand how the card accesses these different memory areas. A countermeasure would be to hide it using a lightweight mechanism like a `XOR` of this table. This would only make this discovery more difficult.
Step 3 We got an access to the various memory areas of the card. This raises a new issue by exploiting an attack based on the concept of `auto-forges` presented in [19]. It consists of search for an exploitable meta-data through all the memories. Then, we can use it as a valid object to read from the targeted memory. To counter this attack, the card must ensure that the object (object or array) that is about to be read is part of the linked list of all the objects of the card. The cost is related to the number of objects contained in the different applets and the way they are managed. Using a linked list of different kind of object could reduce its cost.
Step 4 It consists of reverse engineering the code contained in the different memories, especially the ROM. We have been able to find the JC and GP API's. This allows us to analyze their implementations. A mitigation technique should be to obfuscate the code of these API's using the technique described in [24].
Step 5 This step exploits the different vulnerabilities and allows a privilege escalation. In this step, we look at how the card manages the native calls. As a result, we can exploit this knowledge to perform privilege escalation (by changing the security context or retrieving the cryptographic keys in plain text,. . .). One possibility to mitigate it should be to verify the origin of the native call. In particular, it should not be possible if the call to these native methods is made from a user applet. This requires a stack introspection to check the security context of the caller and thus to identify the origin of the call.

It is surprising that none of these techniques have been used in this development card. Obviously, the ultimate countermeasure is to refuse to sell development cards to academic researchers, which is the case for most of the major European smart card manufacturers.

6 Conclusion and future works

In this paper, we evaluate the security of a development smart card. We demonstrate the possibility to retrieve assets, to access to the native code which should not be possible even in the case where the secure download process is not respected. In particular, we obtain a full access to the native layer from the application layer which seems to not be acceptable.

The targeted system uses object-oriented mechanisms. From the attacker point of view, this adds a difficulty due to the dual semantics of the embedded programs [native and Java Virtual Machine (JVM)]. The attacker has to differentiate between these two languages. But we demonstrated here that the attacker can also take advantage of the OO paradigm. We used the Java meta-data of the instance to perform pattern matching in the dump data that could be interpreted as meta-data. This new attack is general and can be applied to every device that uses meta-data for representing objects.

Our approach is to discover several vulnerabilities, which individually do not provide access to the targeted resources, but combined, they give a complete access to every objects in the system, whatever the current countermeasures are. In particular, we discovered the complete ISA used by the JVM. This knowledge allows to reverse the JC API. In turn, this reverse engineering phase allows to understand the native call mechanism. Then, we get access to most of the native methods of the card. Finally, this last step allows to get a full access to most of the assets and resources of the card.

We propose a set of countermeasures to mitigate all these scenarios. Some of these proposals are obvious to integrate into a card development (e.g., stack introspection).

The lesson learn from this Ph.D. student work is that securing a device is a difficult task. The evaluation labs try to attack these devices in a limited amount of time (a couple of weeks) and can prove its resistance having a complete knowledge of the internals. We have worked in a complete black box approach, with a couple of cards and with a proper methodology. Thus, we have been able to get a complete access to assets of the card. This raises the question about the time an attacker could spend on a device to break its security.

This approach can be used for any secure element which allows secure download of applets. For example, the eSIM (Embedded-SIM) is mostly used in connected objects or automotive application for eCall capability. These tokens are based on secure cores like the smart cards. The GSMA (GSM Association) protection profile [14] suggests that the embedded system could be a Java Card. In that case, this kind of devices should also be an interesting target for our approach.

References

1. Barbu, G.: On the security of Java Card^{TM} platforms against hardware attacks. PhD thesis, Grant-funded with Oberthur Technologies and Télécom ParisTech (2012)
2. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 combining fault and logical attacks. In: Gollmann, D., Lanet, J.L., Iguchi-Cartigny J. (eds.) Smart Card Research and Advanced Application, Lecture Notes in Computer Science, vol. 6035, pp. 148–163. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-12510-2_11
3. Bizzotto, G., Grimaud, G.: Practical Java Card bytecode compression (2002)
4. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined software and hardware attacks on the Java Card control flow. In: Prouff, E. (ed.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 7079, pp. 283–296. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-27257-8_18
5. Bouffard, G., Lackner, M., Lanet, J.L., Loinig, J.: Heap... hop! heap is also vulnerable. In: Smart Card Research and Advanced Applications, pp. 18–31. Springer, Berlin (2014)
6. Bouffard, G., Lanet, J.: Reversing the operating system of a java based smart card. J. Comput. Virol. Hack. Tech. **10**(4), 239–253 (2014). https://doi.org/10.1007/s11416-014-0218-7. http://link.springer.com/article/10.1007/s11416-014-0218-7/fulltext.html
7. Clausen, L.R., Schultz, U.P., Consel, C., Muller, G.: Java bytecode compression for low-end embedded systems. ACM Trans. Program. Lang. Syst. **22**(3), 471–489 (2000). https://doi.org/10.1145/353926.353933
8. Courtois, N.T.: The dark side of security by obscurity and cloning Mifare Classic Rail and building passes, anywhere, anytime. IACR Cryptology ePrint Archive (2009)
9. EMV: EMV Testing and Certification White Paper (2013). http://www.emv-connection.com/downloads/2013/05/EMV_Testing_Certification_V1.0-080213.pdf. Accessed 7 Feb 2018
10. Farhadi, M., Lanet, J.L.: Chronicle of a java card death. J. Comput. Virol. Hack. Tech. (2016). https://doi.org/10.1007/s11416-016-0276-0
11. Farhadi, M., Lanet, J.L.: Paper tigers: an endless fight. In: International Conference for Information Technology and Communications, pp. 40–62. Springer, Berlin (2016)
12. gemalto: Java Card and STK Applet Development Guidelines. http://developer.gemalto.com. Accessed 7 Feb 2018
13. GlobalPlatform, C.S.: Version 2.2. Mars (2006)
14. GSMA: Embedded UICC Protection Profile (2015). https://www.gsma.com/newsroom/wp-content/uploads/SGP_05_v1_1.pdf. Accessed 7 Feb 2018
15. Hamadouche, S., Bouffard, G., Lanet, J.L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: Subverting byte code linker service to characterize Java Card API. In: Seventh Conference on Network and Information Systems Security (SAR-SSI), pp. 75–81 (2012)
16. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan applets in a smart card. J. Comput. Virol. **6**(4), 343–351 (2010)
17. Lancia, J., Bouffard, G.: Java Card virtual machine compromising from a bytecode verified applet. In: Smart Card Research and Advanced Applications, pp. 75–88. Springer, Berlin (2015)
18. Lanet, J.L., Bouffard, G., Lamrani, R., Chakra, R., Mestiri, A., Monsif, M., Fandi, A.: Memory forensics of a Java Card dump. In: Joye, M., Moradi, A. (eds.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 8968, pp. 3–17. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-16763-3_1
19. Mesbah, A., Lanet, J.L., Mezghiche, M.: Reverse engineering a Java Card memory management algorithm. Comput. Secur. **66**, 97–114 (2017)
20. Mesbah, A., Regnaud, L., Lanet, J.L., Mezghiche, M.: The hell forgery, polymorphic codes shoot again. In: 15th Smart Card Research and Advanced Application Conference (2016)
21. Nohl, K.: Rooting SIM cards. In: BlackHat Conference, Las Vegas, July 31, 2013, Security Research Labs blog. https://srlabs.de/rooting-sim-cards/
22. Oracle: Java Card 3 Platform, Virtual Machine Specification, Classic Edition. Version 3.0.4. Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065 (2011)
23. oracle: Java Card Protection Profile Open Configuration (2012). https://www.commoncriteriaportal.org/files/ppfiles/ANSSI-CC-profil_PP-2010-03en.pdf. Accessed 7 Feb 2018
24. Razafindralambo, T., Bouffard, G., Thampi, B.N., Lanet, J.L.: A dynamic syntax interpretation for java based smart card to mitigate logical attacks. In: International Conference on Security in Computer Networks and Distributed Systems, pp. 185–194. Springer, Berlin (2012)
25. Rolles, R.: Unpacking virtualization obfuscators. In: 3rd USENIX Workshop on Offensive Technologies.(WOOT) (2009)
26. Schwarz, B., Debray, S., Andrews, G.: Disassembly of executable code revisited. In: Proceedings of the Ninth Working Conference on Reverse Engineering, 2002. pp. 45–54 (2002)
27. Volokitin, S., Poll, E.: Logical attacks on secured containers of the Java Card platform. In: Smart Card Research and Advanced Applications. Springer, Berlin (2016)