Paper Tigers: An Endless Fight

Mozhdeh Farhadi[1] and Jean-Louis Lanet[2](✉)

[1] Tehran, Iran
[2] INRIA, LHS PEC, 263 Avenue Général Leclerc, 35042 Rennes, France
jean-louis.lanet@inria.fr

Abstract. Recently, researchers published several attacks on smart cards. Among these, software attacks are the most affordable, they do not require specific hardware (laser, EM probe, *etc.*). To prevent such attacks, smart card manufacturers embed dedicated software countermeasures to protect the sensitive system elements. They design countermeasure to mitigate an existing attack with global view of the security. An affordable countermeasure must have a high coverage with a low footprint. For that reasons the design of a mitigation technique is often a trade off between the memory usage and the efficiency of a countermeasure. We present here a survey bringing to the fore the countermeasures used to mitigate the attacks. We use the formalism of attack defense tree to have a synthetic and graphical view of the attack scenario.

Keywords: Java Card platform · Security · Attacks · Countermeasures · Attack tree

1 Introduction

In the 70's decade, the idea of Smart Card made a high level of security available to our everyday life [1]. This small device can keep securely sensitive data of the card holder, such as fingerprint or bank credit. Smart Card can be used in extremely diverse applications such as access control systems, digital signature, electronic purse and identity.

The Smart Cards can be divided into two main categories from their operating system point of view. One is the classic Smart Card operating system which prohibits load of new applications into the card after card issuance. On the other hand, there is another type of card operating system which is categorized as Open Platform. Java Card, Multos and Dot net Card are examples of Open Platforms. Open Platform cards which provide multi-application in a single card, allow loading new applications to the card even after card issuance. In the classic smart cards, code of the application which is part of the proprietary operating system is masked into the Read Only Memory (ROM) and thus loading new application code is prohibited. But in the Open Platform cards new applications can be loaded and installed into the Electrically Erasable Programmable ROM (EEPROM).

In 1997, Java Card with the main idea of Java which is "Write once, Run anywhere" was born. This Open Platform card provides application independence from the hardware. The Java Card manufacturer companies develop their

I. Bica and R. Reyhanitabar (Eds.): SECITC 2016, LNCS 10006, pp. 40–62, 2016.
DOI: 10.1007/978-3-319-47238-6_3

Java Card Runtime Environment (JCRE) for a Smart Card chip according to the Java Card specification published by Oracle [2]. On the other hand, the Java Card application providers develop their own applications using the Java Card Application Programming Interface (APIs) published as part of the specification. This standardized APIs provides a uniform interface to various Smart Card chips [5]. Thus, the Java Card applications can be installed on any Java Card platform only concerning the version competency of the Java Card specification. Java Card applications (applets) are loaded as Converted APplication (CAP) files into the card.

The Java Card platform provides security to the smart card world by its security features. But as it is an Open Platform card which provides loading new applications after issuance it increases the possibility to be target of attacks. Attackers can easily do experiments by loading test applets on the card and analyzing the results. Thus, the fight between attackers and Java Card platform designers starts. Whenever an attack is hindered by a countermeasure, another new attack is proposed by another attacker. The attackers use software, hardware or a combination of both to get access to the assets of the cards. In this paper we describe the software attacks and their countermeasures on the Java Card platform.

The logical attacks can be divided into three categories: attacks due to the specification, attacks with ill-typed code and attacks against bad implementation. Each kind of attack uses different hypotheses which are pointed out in this document.

The rest of the paper is organized as follows: first we give an overview of the security features of the Java Card platform. Second, we describe the attacks due to the specification. Third, we focus on attacks with Ill-typed code and fourth a description on the attacks against bad implementation will be presented. Finally, we conclude in the last section.

2 Security Features of the Java Card

Java Card uses a subset of the Java language and Java is considered as a secure language. This language pays attention to the security by blocking type mismatches, eliminating pointer use by the developers and control of array boundaries [3]. The following features of the Java Card platform play main roles in providing security:

Byte Code Verification: Byte Code Verification (BCV) ensures that the code is compliant with the Java Card specification rules. It verifies all the operations regarding the type system of the Java language.

Firewall mechanism: As Java Card is a multi-application card, the firewall takes care of applet isolation and controls the interaction between the applets [5]. In the Java Card, access to elements of other applets in different security contexts (packages) are not allowed. But the Java Card specification defined an interface which provides sharing services to the other applets. If an applet wants to share its services to other applets, it should define an interface which inherits from the `shareable` interface of the Java Card API.

Transaction mechanism: The Java Card specification defines the *transaction* mechanism to protect persistent data against events such as a power loss in the middle of a transaction operation. The Java Card applet developer can group some update operations into a transaction block. Thus, the atomicity of these updates is guaranteed. It means that in this block of code, all of these update operations are bounded together. Either all of the updates will be successfully done or none of these updates will be done.

In order to provide a synthetic view of each kind of attack, we present them using the graphical attack tree methodology. It has been introduced by Schneier [18] to analyze the different ways a system can be attacked. In this method, an undesirable event is defined and then the system is analyzed to represent the combination of basic events with AND and OR gates, that can lead to the undesirable event.

The root node of the tree represents the undesirable event. The nodes are refinements of this event, and leafs are the initial causes. It should be mentioned that an attack tree does not represent all possible cases of failure but a restricted set. A path from a leaf to the root represents an attack scenario. An event with a NON gate represents a countermeasure. If the countermeasure is not present, then the attack will succeed. One can remark that the closer to the root the countermeasure is, the better is the coverage of this countermeasure.

In analyzing a smart card, four undesirable events are accounted as system failure: code or data integrity and code or data confidentiality. The code integrity is the most important property among others. Because a failure in code integrity can lead to data and code confidentiality and also to data integrity. In this paper, the representation within the attack tree will focus on the code integrity as the root of each tree.

3 Attacks Due to the Specification

The Java Card specification [2], specifies the necessary behavior and environment that a Java Card implementation should provide. In the Java Card specification, there are some points related to the card behavior which were not clearly defined. These points were firstly showed by Erik Poll at `Cardis 2004`. In his paper [4], he shows that in some implementations of the Java Card specification, serious security wreckages can be found. These problems resulted from the ambiguity of the specification and the interpretation of the designers. In this section, we describe these ambiguities in more details.

3.1 Abusing the Transaction Mechanism

The *transaction* mechanism aims to provide atomicity to the operations on persistent data elements. In the Java Card API, a method is also defined to give the ability to cancel the transaction operations by the developer. If the applet encounters an internal problem, the `JCSystem.abortTransaction` can be called to abort the transaction. It should also be mentioned that the transaction mechanism only applies to persistent data and not to the transient data.

The transaction mechanism abuse, exploits the creation of an object inside a transaction and the mis-operation of the platform to completely clean up object references after the `JCSystem.abortTransaction` call. The clean up task of the platform was not clearly mentioned in the old Java Card specs (objects created during a transaction must be garbage collected) and it caused ambiguity in the implementation of the Java Card platforms (a reference to this object should remain, becoming a dangling pointer). The transaction mechanism abuse exploits this ambiguity.

They define two arrays: aPers and aLoc which refer to the same `short` array. When the transaction aborts, the aPers reference which refers to a persistent object is set to null. In some implementations, the Java Card platform does not also sets the reference of aLoc to null. After the transaction abortion, this reference can still be used but it is a dangling pointer. If the attacker defines a new `byte` array exactly after the `JCSystem.abortTransaction`, the memory manager allocates the previously canceled area. Thus it returns the reference of `aPers` to this recently created `byte` array (because the platform supposes that this reference is released). Thus, the attacker can get access to the new `byte` array as a `short` array.

In this example, the attacker can access to the `newByteArray` with accessing 10 cells of aLoc which is of type `short`, so it leads to get access to 20 bytes. As it can be seen, it is two times bigger than the allowed length for the `byte` array. By defining the `byte` array big enough, the amount of un-allowed data retrieval can be extended.

3.1.1 Countermeasures

To mitigate this attack, one effective countermeasure is to forbid load of applets that have used `JCSystem.abortTransaction` method in their source or binary code. This solution is often encountered on old cards.

Most of the recent Java Cards perform a rigid clean up of the objects inside a transaction after an `JCSystem.abor-Transaction` call. Thus, all the references of the objects used in a transaction should be equal to null after an abort event.

3.2 Abusing Shareable Interface Objects

The Java Card platform protects installed applets and packages from access by other applets in different security contexts by the firewall. In some cases, it is needed that applets in different security contexts communicate and use services of each other in the presence of the firewall. Thus, the Java Card specification has defined a sharing mechanism to share services of one applet to other applets belonging to different security contexts.

The mechanism which provides access to objects in different security contexts is called Shareable Interface Objects (SIO). To use this mechanism, one needs to create an interface by inheriting the `javacard.framework.Shareable` class of the Java Card. This interface defines which services are shared with other applets. The client applets refers to this interface and call shared methods of the server applet which implements this interface.

In [6], the authors use the sharing mechanism to create a type confusion and get access to non allowed memory areas. They create a server and client applet with a fine difference in the interfaces that they use. As the compilation and loading of the server and client applets are done in separate steps, the Java Card platform does not notice that the client and server applets are using different interfaces. When the applets are installed on the card, the client gets access to a data with a different type than the server applets get access to it. Thus, the server applet is accessing its own byte array as a short array, which is a type confusion. As explained in the previous section, this type confusion can lead to access to the array data twice the array data length.

3.2.1 Countermeasures

To mitigate the SIO abuse, the platform can count the number of bytes that each entity tries to get access to it. In the above type confusion, suppose that the server applet's array has a length equal to `N` bytes. In a non-secure implementation of the platform, accessing this byte array as a short array can lead to access `N*2` bytes. This scenario is depicted in Figure 1.

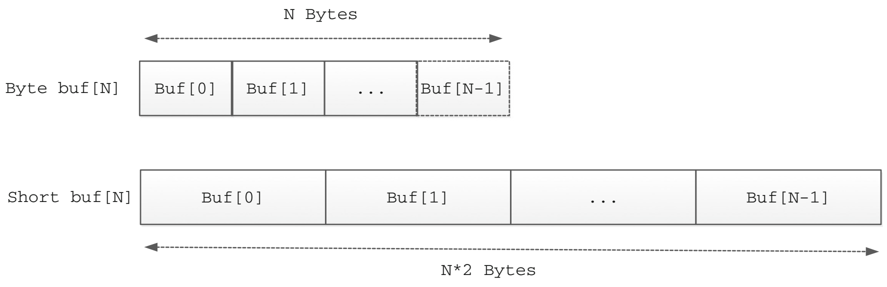

Fig. 1. Type confusion in accessing arrays

In a secure implementation of a Java Card platform, even if the attacker is able to get access to a byte array as a short array, he is only able to get access to the array with its defined length. The platform gives access to the arrays regarding the number of bytes and not regarding the number of elements of the array that occupies the memory.

As another countermeasure, the platform builds an internal table which stores definition of the methods in the interface, while loading the CAP into the card. On the other hand, it checks the methods that they call the interface methods regarding the definition of the methods in the Table. If there is a conflict, the platform will notice.

3.3 The Export File Fraudulence

The Java Card uses a two-step linking process [7]. The first step is done outside the card and the second step is performed inside the card.

In the off-card step, the Java Class files are converted to a CAP file. Each CAP file consists of at least eleven components. These components are: `Header`, `Directory`, `Import`, `Applet`, `Class`, `Method`, `Static Field`, `Export`, `Constant Pool`, `Reference Location` and `Descriptor`. There are also two optional components: `Debug` and `Custom` components.

In the process of conversion of a Java Class file into a CAP file, the necessary linkage information is stored in both off-card and on-card sides. In the off-card side, an export file carries this information which can be used for creating other CAP files afterward. The export files are publicly available files which are used to translate Java items into tokens in the off-card linking process.

The CAP file itself carries on-card linkage information in these three components: `Constant Pool`, `Reference Location` and `Import` components. The `Constant Pool` component contains the linking information between each tokens value and the corresponding reference in the method, class and/or package which needs to execute the byte code from the component. The Reference Location component specifies the offsets in the `Method` component where a token should be linked to a card internal reference [7]. The used packages by the applet is listed in the `Import` component of the CAP file. In the on-card linkage process, each token is converted to an internal reference.

In [7], Bouffard *et al.* use a manipulated export file to create a CAP file. In this fake CAP file, a malicious representation of an API method is provided. In their example they replace a malicious implementation of the `buildKey` API. This implementation stores a copy of the key it builds in a place of memory which can be retrieved by the attacker later. They insert the fake export file which contains malicious linking information into the export files' path and use it to create the CAP file. As the Java Card off-card linker uses a first match algorithm to find the export file for the linkage operation, it uses the fake export file. They also load the corresponding fake implementation of the `buildKey` API into the card. The attacker also loads the victim CAP file, which is the CAP file that has used the fake export file during creation. Thus, when the API is called inside the applet, the fake API will be called.

3.3.1 Countermeasures

To prevent this attack, the developer should protect the export files when provided by a third party. They must carefully check if the method name inside the export file can be confused with API method name.

3.4 Specification Ambiguity Attack Tree

The Fig. 2 depicts attack tree of the attacks due to specification ambiguity. The property that we expect to protect is the integrity of the code. And thus represents the root of the tree. A A_i label indicates an attack while a D_i represents a

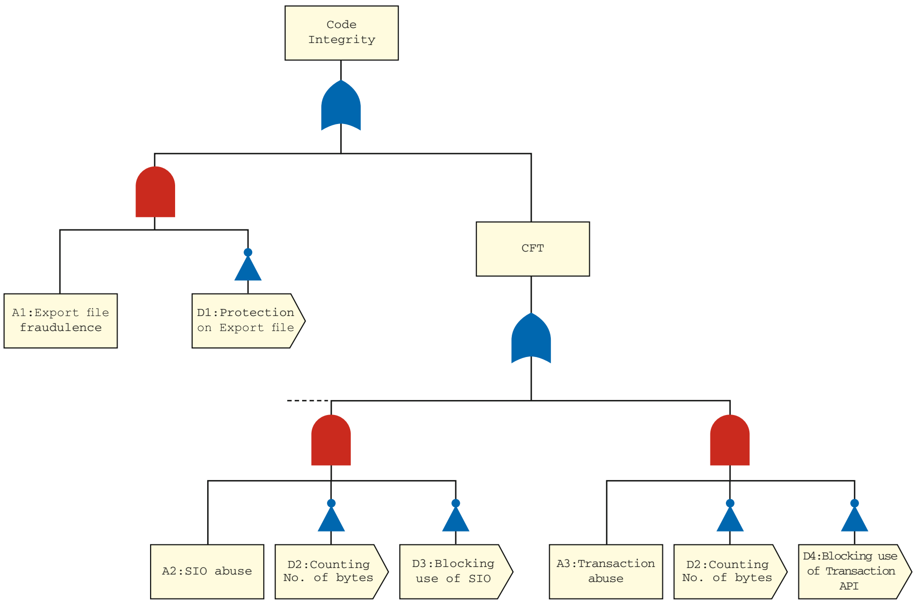

Fig. 2. Attack tree of the attacks due to the specification ambiguity

countermeasure. If one wants to setup a SIO abuse attack (A_2) it requires to neither count the number of elements (D_2) nor to block the use of shareable (D_3). The nodes in its hierarchy are OR gates, so the scenario is enough to succeed.

4 Attacks with Ill-Typed Code

Both Java and Java Byte Code languages are strongly typed languages. Type mismatches in the Java source code are detected at compile-time. To detect type mismatches at byte code level, a BCV, either off-card or on-card is required.

The byte code verification process is considered as a costly process, in terms of execution time and memory usage. Thus, currently most of the cards are not equipped with an on-card BCV. Recently, the Java Card specification 3 mandates the use of on-card BCV in the Java Card platforms for the connected edition. It is due to the fact that, in the off-line BCV approach, a verified CAP file can be manipulated by an attacker before loading it into the card. In [8], the authors developed a tool that can manipulate a cap file easily (for example after the CAP file is verified by an off-card BCV).

In the Global platform (GP) compliant Java Cards, off-card verified CAP files can be protected by Data Authentication Pattern (DAP) mechanism. In this mechanism, the CAP file is signed by a trusted authority and the resulted signature should be verified by the on-card representative of the signer before allowing the installation of the applet into the card. This mechanism needs to create Security Domain with DAP verification support and also inserting the required keys into the card. This scenario ensures integrity and authenticity of the CAP file.

The on-card BCV checks the CAP file when the applet is going to be loaded on the card. A manipulated CAP file can be detected by an on-card BCV. As the on-card BCV checks the CAP file in a static manner, some vulnerabilities that are related to the dynamic characteristic of the code, such as an overflow still remain undetected. To defeat such vulnerabilities, a defensive virtual machine can be used to check every operation before execution. But the performance issues of such a virtual machine should be taken into account leading to a trade off between performances and security.

In this section, we review attacks that are exploiting an ill-typed applet. The hypotheses are the ability to load applets on the card and the possibility to bypass the BCV mechanism.

4.1 The EMAN2

The Java Card Virtual Machine (JCVM) is a stack based machine. It creates frame in the stack for each method call and destroys it after method completion. Each frame comprised of two main parts: a data stack as a temporary place for the method's calculations and a locals area to store input parameters of the method and the method's local variables. To manage program flow, the Java Card virtual machine also keeps the return address of the caller method in an area with Last In, First Out (LIFO) structure like stack. The return address is considered as a system data and should be kept in a safe place far from access of attackers. There is a category of attacks which focus on changing the return address of the methods, called Control Flow Transfer (CFT) attacks.

In these attacks, the attacker is able to redirect program flow by changing the return address of a method to a desired address, usually to an easily editable area such as an array. The attacker changes program flow when the current method execution finishes and the control flow is resumed to the caller method. But if this address is modified by the attacker, the program flow will go to the desired address of the attacker and not to the caller's method. In some Java Cards, the return address of each method is stored in an area between locals and data stack. In such cards, if the attacker performs an overflow of the locals or an underflow from the stack, then he gets access to the return address of the method. The structure of the method's frame in such platforms is depicted in Fig. 3. This attack uses an ill-typed code to access the return address of the methods in a Java Card platform. The attacker modifies the return address of a method and redirects program flow to the address of a desired array. Then, he executes any malicious byte code by updating the array. It is supposed that the card is not equipped with an on-card BCV and the attacker has the card keys to load and install his own applet onto the card.

The EMAN2 attack introduced in [9] by Bouffard *et al.* at 2011. The authors characterize the frame by misuse of `sload` op-code and locate the return address

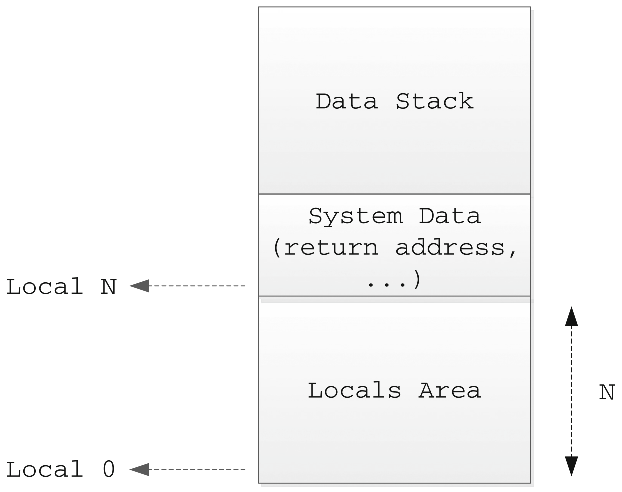

Fig. 3. A frame structure design in a Java Card platform

in the frame. As depicted in Fig. 3, if a method's frame has `N` locals, its local variables can be accessed using `sload-0` to `sload N-1` op-code. If the attacker uses `sload N` op-code and the platform does not throw any exception, it means that the platform does not detect overflow.

Listing 1.1. Method to retrieve address of an array

```
public short getMyAddressByte (byte[] tab) {
        short dummyValue = (byte)0x0A;
        tab[0] = (byte)0x12;
        return dummyValue;
}
```

In the cards that the system data is stored just above the locals area, the frame overflow may effectively lead to access to system data. In this case, the attacker gets access to the system data by increasing argument of `sload` op-code from `N` to whatever the length of these area is. After characterizing the system data area, and locating the return address in the system data, he updates the return address using `sstore X`, where `X` is the location of the return address in the frame. To complete this attack, the attacker needs to know the address of the array to redirect the program flow into it.

In the Listing 1.1, the targeted array is sent to the `getMyAddressByte` method. The corresponding op-code of the method `getMyAddressByte` is presented in the Listing 1.2.

Listing 1.2. Original code of `getMyAddressByte` method

```
Public short
   getMyAddressByte
       (byte[] tab){
03 //flags: 0 max_stack:3
21 //nargs: 2 max_locals:1
10 0A bspush 0x0A
31    sstore_2
19    aload_1
03    sconst_0
10 12 bspush 0x12
39    sastore
1E    sload_2
78    sreturn
}
```

Listing 1.3. Modified code of `getMyAddressByte` method

```
Public short
   getMyAddressByte
       (byte[] tab){
03 //flags: 0 max_stack:3
21 //nargs: 2 max_locals:1
10 0\,A bspush 0x0A
31    sstore_2
19    aload_1
78    sreturn
10 12 bspush 0x12
39    sastore
1E    sload_2
78    sreturn
}
```

The Listing 1.3 shows the manipulated version of the Listing 1.2. As it can be seen in the Listing 1.3, when the address of the array is pushed on top of the stack, the attacker returns this address by replacing the next op-codes with `sreturn`. Thus, the value on top of the stack, which is the address of the targeted array is returned by `sreturn` op-code.

4.1.1 Countermeasures

The EMAN2 attack, needs some conditions to succeed. The attacker must get access to the area above the locals, where the system data is stored. Thus, as a countermeasure a Java Card platform can check if an op-code is accessing an area out of the locals boundary. But these checks add an extra cost to each instruction while accessing locals. It has never observed.

Another countermeasure is to store system data in another area of RAM, in order to harden access of the attacker to this valuable data. The *Separate Stack* is to define another stack to store return addresses of the methods. This approach is discussed in more detail in Sect. 4.7 *Stack Underflow and Frame Overflow* and is often used.

The most implemented countermeasure is to check if a jump destination belongs to the minPC (minimum Program Counter) and maxPC (maximum Program Counter) value of the method. If the control flow has been transferred to a an array, this array is stored outside the bounds of the method and any execution of non linear code will be detected as illegal operation.

4.2 Subverting BC Linker Service to Characterize JC API

In [13], Hamadouche `et al.` introduced a method to characterize a Java Card platform in order to design rich shell code for that platform. To write a rich shell code, it is required to know the reference of the needed API methods and then directly calling the methods by their actual reference in the targeted platform.

Because most of the time the attacker hides his shell code inside an array, this code fragment is not subject to any linking process.

In the off-card linking step, while converting class files into a CAP file, the converter does not have any information about the reference of the methods or fields in the targeted card, so it uses tokens instead. At the loading step, the JCVM transforms these tokens in the `Method` component to their reference with the help of the `Constant Pool` component and the `Reference Location` component. Thus, as the reference of the objects, methods and other elements is not known for the attacker, he needs to firstly find these references in the targeted Java Card platform.

The authors in [13] create a set of malicious CAP files, whereas each of these CAP files contains a call to one of the JC API methods. They replace the instruction that precedes the token of the API method call in the `Method` component of the CAP file with a desired instruction. In the on-card linking step, this token will be translated to the reference of that specified API method in the card. Thus, if the attacker replaces the preceding instruction of that token to an instruction which prepares sending data out of the card, he can use this technique to find references of the methods in the targeted Java Card platform.

If the attacker wants to retrieve the address of the `setOutgoingAndSend` method of the Java Card API, he only needs to write an applet which uses this method and then replaces the `invokevirtual` instruction with the `sspush` instruction. This pushes the address of the `setOutgoingAndSend` method on the top of the operand stack. Then he can send it out of the card.

The authors used the on-card linker to retrieve reference of API methods in the targeted platform and send it out of the card. The corresponding CAP files are general and can be used to retrieve API references in other Java Cards without sufficient countermeasures.

4.2.1 Countermeasures

This attack is based on replacing an instruction that invokes a method with a desired instruction such as `sspush`. A countermeasure is to only resolve the token if the instruction requires a token (like `invokevirtual, invokestatic, getstatic, setstatic`, *etc.*) during the applet loading phase. Few cards implement such a policy.

4.3 The Stack Underflow Attack by Misuse of `dup_x` Instruction

In [10], Faugeron introduced a technique to get access to the data below the stack of the current executed method. She misuses the `dup_x` instruction to create an underflow and get access to elements stored below the current method. In some Java Cards, the system data such as context identifier are stored below the stack. Thus, in such cases the attacker can modify this data and gain privileged access to the card's resources.

The `dup_x` instruction takes two parameters which the first parameter indicates number of elements from the top of the stack that should be duplicated and inserted in the stack. The second parameter indicates the distance of the top of the stack that the duplicated elements should be inserted. The author misuses this operation by performing this instruction on a stack without enough data on it. In this case, the JCRE uses the system data below the current method as elements of the operand stack and the attacker gets access to them.

4.3.1 Countermeasures

This attack can be bypassed by a defensive JCVM which checks valid boundaries of the stack access for each operation that pops elements from the stack [10]. Few cards implement such a countermeasure.

4.4 The JSR/RET

The `jsr` and `ret` are two op-codes in the Java Card specification. The old Java compilers generated them while the `finally` statement was used after a `try-catch` statement [11]. Recent Java compilers do not generate the `jsr` and `ret` op-codes, but these two instructions must be supported by the JCVM only for backward compatibility.

The `jsr` and `ret` op-codes were used for executing subroutines [2]. The operand of the `jsr` instruction specifies the address of the subroutine that the program execution should be continued from there. Before jumping to the subroutine address, the JCVM pushes the return address (the address of the next instruction after the `jsr` instruction) into the stack. The compiler inserts an `astore` op-code as the first instruction of the subroutine to keep the return address in the locals area.

Using `ret` instruction, the return address from the local variables area is retrieved and pushed into the Java Program Counter (JPC) register. Thus, the program flow continues from the address of the next instruction of the `jsr` instruction.

The interesting point about this couple of instructions is that it provides easy access to the return address despite of where and how the return addresses are stored in the targeted Java Card platform. In [11], the authors misuse these couple of instructions to cause a CFT attack in the cards with no on-card BCV. The code in 1.4 shows how this attack is performed.

Listing 1.4. Byte code representation of `jsr/ret` abuse

```
short exploitJSRInstructionWithoutBCV () {
flags: 0 max_stack : 1 ;   nargs: 0 max_locals: 1
/*0053*/ L0: jsr L1
/*0056*/ sspush 0xCAFE
/*0059*/ sreturn
/*005a*/ sspush 0xBEEF
/*005D*/ sreturn
/*005E*/ L1: astore_1
/*005F*/ sinc 0x1, 0x4
/*0062*/ ret 0x1}
```

In the above code, after executing the code at line `0x53`, the program flow jumps to the address `0x5E` where the address of the next instruction (`0x56`) is stored in the local variable 1. But there is an instruction which manipulates the content of the local variable 1: `sinc 0x1, 0x4`. This line of code, adds 4 to the return address. Thus, when the subroutine execution finishes, the `ret` instruction pushes the manipulated return address (`0x5A`) into the JPC register. By executing the above code, the user will receive `0xBEEF` instead of `0xCAFE` at the output. Any other op-code which manipulates the content of the local variables can be used instead of `sinc` op-code to change the return address which is stored in the locals area.

This attack is based on performing arithmetic operations on the return address which is of reference type. Thus, if a Java Card platform is equipped with a typed stack, this attack can not be performed successfully, because in such Java Card platforms the operations on the stack are checked against their type. The authors of the article [11], proposed a method to successfully perform their attack on Java Cards with typed stack. They took advantage of `putfield_<t>_this` and `getfield_<t>_this` op-codes.

These two instructions manipulate instance data stored into the heap memory. The authors used `putfield_a_this` to store the return address into the heap area, and then used `getfield_s_this` op-code to retrieve and manipulate the return address as a `short` variable in a typed stack.

4.4.1 Countermeasures

The `jsr/ret` attack can be mitigated using a combination of a typed stack and inserting type checking in the heap memory. Inserting a type checker in the heap memory can stop misuse of the `putfield_<t>_this` and `getfield_<t>_this` op-codes to cause a type confusion and thus changing the return address as a `short` variable.

4.5 Stack Underflow by Abusing the Frame Creation Mechanism

In [17], the authors describe their attack method to get access to the (system) data below the operand stack. They use an ill-typed applet to cause a stack underflow. Each method in the Java Card has these following attributes which are determined at compile time: the size of the local variables (*nargs* + *max_locals*) and the size of the operand stack (*max_stack*). The *nargs* determines the number of arguments of a method whereas the *max_locals* determines maximum number of local variables for a method. These attributes are stored in the `method_header_info` structure of the corresponding method in the `method` component of the CAP file.

While the JCVM reaches to an `invoke<>` operation, it pops the arguments of the targeted method from the operand stack of the caller method and pushes them into the local variables area of the newly created frame for the targeted method of the `invoke<>` operation. The start of the newly created frame is equal to the value of the stack pointer (before invoking the method) minus the size of the local variables of the invoked method. Thus if an attacker illegally extends size of the local variables of a method, he will be able to get access to an illegal area as the area of the newly created frame.

The authors illegally extend the `nargs` value of the method and thus the frame allocation of the method is compromised. If the increase in the `nargs` be as much as it places the new frame below the start address of the stack, a stack underflow occurs. Thus the attacker will be able to get access to an undetermined memory area using `aload` op-code. This memory area might contain system data. The attacker might get system information by analyzing system data.

4.5.1 Countermeasures

This attack which uses ill-typed cap file can be blocked by the on-card BCV. Thus, presence of on-card BCV is the main countermeasure to this attack.

4.6 The ArrayCopyNonAtomic API Attack

The Java Card platform provides an API to copy the content of an array into another array. The `Util.arrayCopyNonAtomic` method has the following method signature: `arrayCopyNonAtomic(byte[] src, short srcOff, byte[] dest, short destOff, short length)`

In [12], the authors used this method to create a type confusion and get access to the meta-data of objects as data of array. The idea behind this attack is the fact that, while the type checking is the task of the BCV, if a card is not equipped with an on-card BCV, creating a type confusion might be possible.

In the Java Card's type system, an array is inherited from `object`. Arrays of different types (`byte`, `short`, *etc.*) are separated as branches of the type tree. In this attack, we created an ill-typed CAP file, where the source array were replaced by an object in separate branch than the `byte` type. The authors used reference of an instance of `key` class instead of the source array in `Util.arrayCopyNonAtomic` method and the card did not noticed this type confusion.

In the targeted platform, the card allowed to copy the content of the key object into the destination array. Thus, the authors were able to get access to a key object as an array and read its meta-data at the output. The code in Listing 1.5 is used to perform this attack.

Listing 1.5. Method used to retrieve key

```
public short CopyObject(byte[] dummyArray, DESKey deskey,
        APDU apdu){
Util.arrayCopyNonAtomic(dummyArray, (short)0, dummyArray,
        (short)0, (short)16);
apdu.setOutgoing();
apdu.setOutgoingLength((short)(16));
apdu.sendBytesLong(dummyArray,(short)0, (short)16);}
```

The code in Listing 1.6 shows the modification on the original code presented in the Listing 1.5. In the modified code, the source array is replaced by a key object.

Listing 1.6. Code snippet of `CopyObject` method

```
19 aload_1 -> 1A aload_2
03 sconst_0
19 aload_1
03 sconst_0
... ...
```

The reference of the *dummyArray* is replaced by the reference of the targeted key which is stored as Local variable 2. This type confusion attack can be repeated using other data types. The authors use `This` object to get more information about the platform's internals. They get access to the code of their applet in the EEPROM and also the reference of object's defined in their applet.

Moreover, they are able to write from an array directly into the memory by swapping the `src` and `dest` in the `arrayCopyNonAtomic` method. This gives them the capability to change their own code just using an API call.

4.6.1 Countermeasures

To mitigate this attack, it is needed to carefully check if the types of the data in the `arrayCopyNonAtomic` method is compliant with the type specified for each parameter. While performing this attack on the targeted platform, the authors noticed that the card raises an exception if the type of the `src` or `dest` array is `short`. Thus, it proves that the platform has implemented a type checker but it is not comprehensive. The platform designers have forgot that other types might be used for `src` and `dest` array.

This attack also uses arithmetic operations on the reference type. In the targeted Java Card platform, the authors had to increase the key reference by one to get access to the key value and its meta-data. Thus, if the platform be able to detect arithmatic operations on references, the `arrayCopyNonAtomic` attack can not be successfully exploited.

4.7 The Stack Underflow and Frame Overflow

As described earlier in Sect. 4.1, the JCVM is a stack based machine. In some Java Card platforms, the return address of a method with other useful information such as context of the method execution, number of local variables and *etc.* which we refer to them as system data, are stored below operand stack of each frame. An example of these type of stack implementation represented in Sect. 4.1. In some recent cards, two separate stacks are designed. One of them as the operand stack and the other one as the storage for the system data of each frame. We refer to this type of stack design as *Separate Stack*. In the *Separate Stack* design, the two stacks are growing as depicted in Fig. 4.

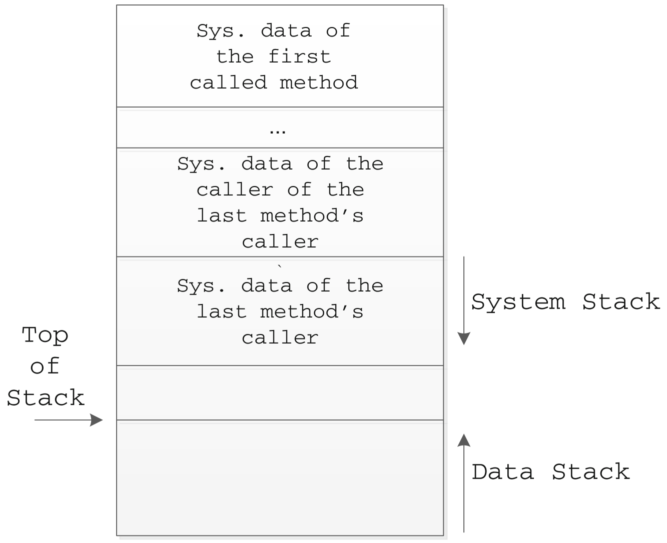
Sys. data of the first called method
...
Sys. data of the caller of the last method's caller
Sys. data of the last method's caller
System Stack
Top of Stack
Data Stack

Fig. 4. *Separate Stack* scheme

In this section, we discuss a CFT attack in the cards that have implemented *Separate Stack*. This attack, which is described at [12], is based on creating a frame overflow and using underflow in the data stack to access and modify return addresses stored in the system stack. In this attack, the attacker uses an ill-typed applet to characterize system stack and then he changes the return address of a method in the system stack to direct program flow to his desired address.

The authors created a frame overflow by calling a recursive method 31 times in a Java Card which uses *Separate Stack*. Thus, They find how many calls of that recursive method creates a frame overflow. Then, they call the recursive method until one call before overflow occurrence (30 times). In the recursive method, they insert some specific byte codes which are only executed in the last call of the recursive method. These byte codes use `sload X` op-code to create an underflow in the stack. They also insert op-codes to send the value stored on the top of the stack to the terminal. Listing 1.7 shows this recursive method code.

Listing 1.7. The recursive method

```
private void exploreFrame(byte numberofCalls){
    if(numberofCalls==0)  return;
    else
        if((numberofCalls==(byte)1))    {
                //an arbitrary code, we will change it
                //to a malicious code before loading
                //the CAP file into the card    }
        exploreFrame(--numberofCalls);
}
```

In the targeted Java Card platform, the authors created 256 CAP files which had used different operands for `sload` op-code, from 0 to 255. As the card did not raise any exception while executing these CAP files, they could experience frame overflow. Each `sload X` op-code in the different 256 CAP files returns a cell of the system stack. They analyzed the result of execution of these different CAP files. As they used a recursive method, they expected to find 30 equal values as the return address.

Analysis of the data of the system stack leaded to find the data pattern in the system data of the targeted Java Card platform as depicted in Fig. 5.

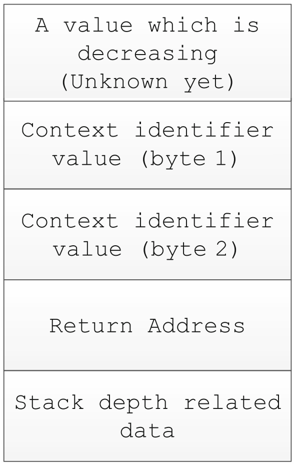
A value which is decreasing (Unknown yet)
Context identifier value (byte 1)
Context identifier value (byte 2)
Return Address
Stack depth related data

Fig. 5. General pattern for system stack

After characterizing system stack, the attacker are able to change the return addresses using `sstore` op-code. In the targeted Java Card platform, the authors could successfully change the program flow using this attack.

4.7.1 Countermeasures

This attack uses frame overflow and stack underflow. If one of these operations (frame overflow or stack overflow) is blocked, then the attack will not succeed. The following countermeasures mitigate this attack: A precise frame management be implemented to detect all violations of the boundaries, in such a way that the stack underflow can not generate a frame overflow [12].

The applet used in this attack is an ill-typed applet. If the card is equipped with an on-card BCV, the malicious op-codes is detected and the applet can not be installed on the card.

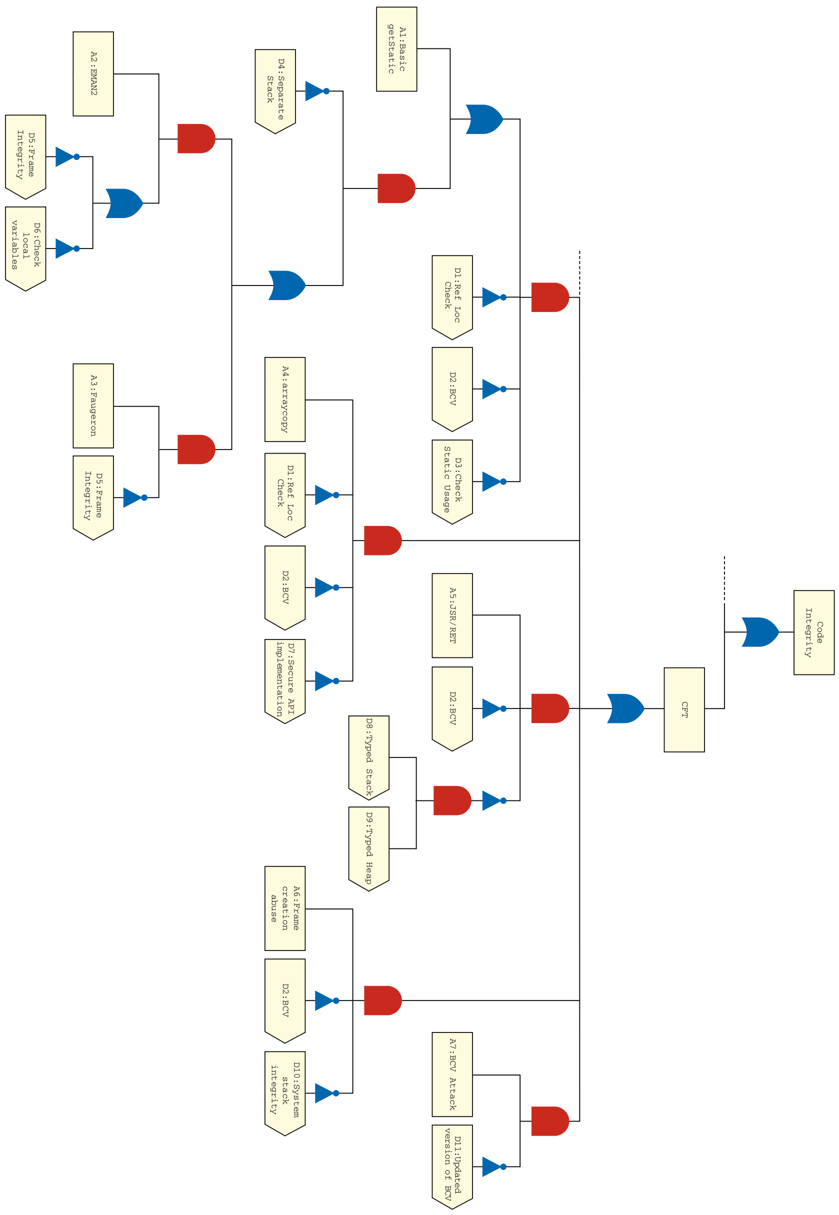

Fig. 6. Attack tree of the ill-typed code attacks

4.8 Ill-Typed Code Attack Tree

The Fig. 6 depicts the corresponding attack tree of the ill-typed code attacks. One can remark that the attack tree representation is closed to the model depicted with the Common Criteria [19] methodology: security concepts and relationship. It represents an instantiation of the general model for a Java Card.

5 Attack Against Bad Implementation

In the final category of the Java Card attacks, we describe attacks based on bad implementation of the Java Card platforms. The Java Card specification provides features and behavior that should be supported by a Java Card platform but the specification does not give any design direction. So there might be some bugs in the design and implementation of the produced Java Card platform.

In this section we describe attacks exploiting these bugs.

5.1 The BC Verifier Attack

This attack is particularly sensible due to the single point of failure represented by the BC Verifier software. There is only one implementation of the specification and the certification procedures require to use at least the implementation of Oracle. Of course, a failure in this program can allow any hostile applet to gain access to the assets of the card. The last bug found in this critical piece of code has been described by Bouffard *et al.* in [15,16].

The authors discovered that one verification was missing in the token-based linking scheme. This scheme allows downloaded software to be linked with API already embedded on the card. As we described previously, each externally visible item in a package is assigned a public token that can be referenced from another package. There are three kinds of items that can be assigned public tokens: classes, fields and methods. When the CAP file is loaded on the card, the tokens are linked with the API and are resolved to the internal representation used by the JCVM. The linking process resolves tokens into the JCVM internal representation. For a method invoke, the class token identifies a `class_info` element in the Class component. In the `class_info` element, the `public_virtual_method_table` array stores the methods internal representation. The method token refers through an index into this array to an absolute offset into the Method component. This offset points to the header of the refereed method.

This offset is a redundant information which is already stored in the CAP file in the `method_descriptor_info` elements in the Descriptor component. On most cards, the offset information in the Descriptor component is used by the BCV before loading, while the offset information in the Class component is used by the JCVM linker on card. Some cards perform a test to check the coherence between this redundant information. An ill-formed offset information in the Class component remains undetected by the BCV checks, but it is still used by some

JCVM linker on card. If the last entry of the `public_virtual_method_table` is removed then the on card linker will use the next bytes to form its offset. The Method component is the component loaded after the Class component and this is under the control of the attacker.

The authors have developed a proof of concept where they remove the last entry, design the header of the first method to be understandable either as a correct method header but also as an absolute offset allowing to jump in a dedicated fragment of code. This allowed them to execute any arbitrary shell code. The last version of the BCV corrects this flaw.

5.2 Stack Overflow and Changing the Security Context of a Method

In the Sect. 4.7, the authors used an ill-typed applet to get access to the system stack of a Java Card while the card has implemented *Separate Stack*. In [14], Dubreuil uses a well-typed applet to create frame overflow and get access to the system stack in a card equipped with *Separate Stack*. He uses the frame overflow to get access to all objects of the targeted Java Card platform and change the security context of a method to JCRE context. In his paper, he uses a verified Java Card applet which contains a method with some specified local variables and objects in it. He also calls other methods of his applet in order to fill the stack of the targeted Java Card platform. The targeted card, has implemented *Separate Stack*, thus its stacks are growing as depicted Fig. 4.

In his experiments on the targeted Java Card platform, he fills the stack until only two bytes of the data stack are available. Upper than these two bytes, the system data of the caller method is stored.

Then he calls a method which creates overflow in the data stack that causes to update system data of the previously called method (the caller of current method). This situation is depicted in Fig. 7. In the targeted Java Card platform,

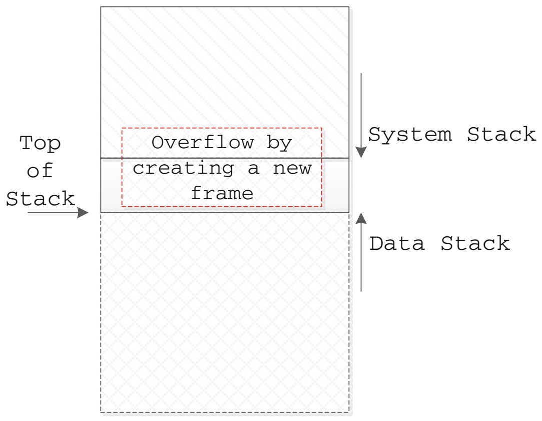

Fig. 7. Stack overflow in a card which uses *Separate Stack*

the platform does not notice that a frame overflow is occurred and the attacker misuses this vulnerability to change system data of the methods.

In the system data of each frame, the number of available local variables is also stored. Thus, he designs his last called method in a way to decrease number of local variables of the caller method to a desired number. For example if he changes the number of local variables of a method from 9 to 3, different data in the stack will be interpreted as the real data of that particular frame. Moreover, type of these data is also changed and type confusion can occur. This issue is showed in Fig. 8.

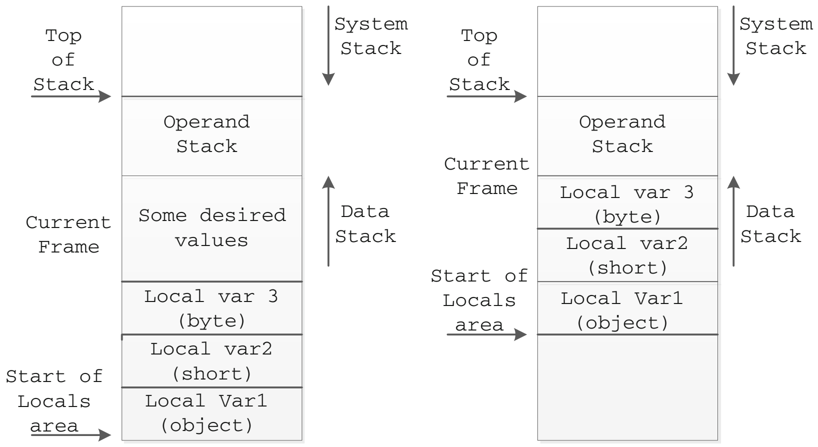

Fig. 8. Changing size of local variables in a frame

As it can be seen in Fig. 8, after changing the size of the local variables area, other bytes of the stack are interpreted as the original local variables. For example the bytes stored upper than the first local variable are interpreted as the first local variable. Suppose that for the first local variable which its type is `reference`, we can update it by a desired value with other types like `short`. Using this technique, the attacker can receive input data, for example as `short` type and update an area where a reference is stored. This can also happen to change the execution context of a method.

In this paper, the author could successfully change the context of a method to the JCRE context. The JCRE context is a privileged context, so the attacker can access to all objects despite of the existence of firewall mechanism.

The applet used for this attack is a completely verified applet and no modification after the conversion has been made on it. Only mismanagement in the stack pointer of this platform, leaded to change the security context of a method in the targeted Java Card. The author exploited this bug in the platform to dump all the objects of the targeted Java Card.

5.3 Bad Implementation Attack Tree

The Fig. 9 depicts attack tree of the bad implementation of the platform. Of course, this attack tree is only relevant to the published weaknesses and is probably larger.

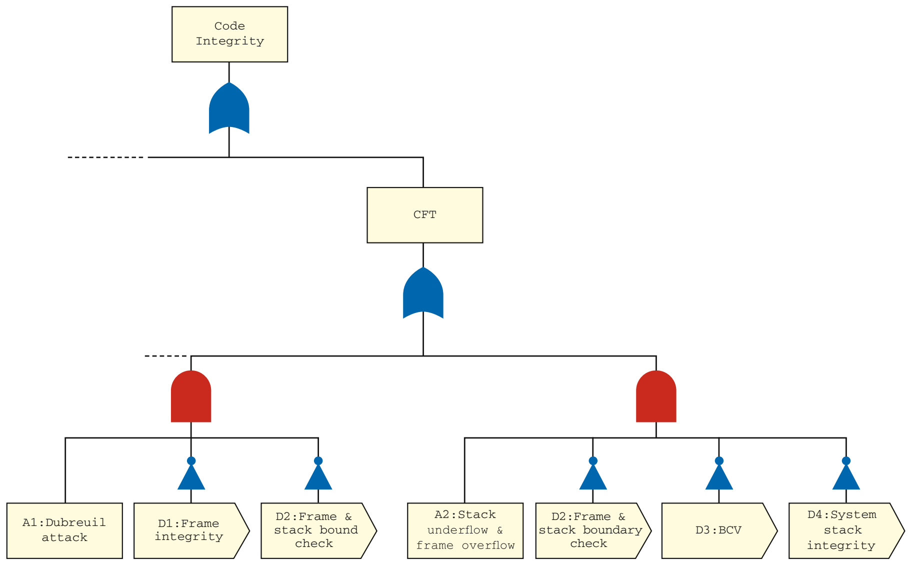
Code Integrity
CFT
A1:Dubreuil attack
D1:Frame integrity
D2:Frame & stack bound check
A2:Stack underflow & frame overflow
D2:Frame & stack boundary check
D3:BCV
D4:System stack integrity

Fig. 9. Attack tree of the bad implementation attacks

6 Conclusion

In this paper, we pointed out the different categories of attacks against the most secure device: the smart card and in particular the Java Card. Researchers focused on hardware attacks while more simple attacks were still possible. Specification ambiguities, bugs in the implementations, characterization of platforms have led to pure software attacks against this platform. Most of the attacks can be mitigated either by correct implementations but also by implementing adequate countermeasures. We presented all this attack with the formalism of Attack Tree which provides a synthetic and graphical view of both the attack scenario but also the set of available countermeasures.

We verified that several implementations did not implement these countermeasures and thus are subject to attacks. Of course, the publicly available products are often old products and do not reflect the state of the art of current Java Card products. Nevertheless, some recent publications of Security Evaluation centers try to demonstrate that even recent products are subject to bad design.

References

1. Rankl, W., Effing, W.: Smart Card Handbook. Wiley, Hoboken (2004)
2. Oracle: Java Card Platform Specification. http://java.sun.com/javacard/specs.html
3. Sun Microsystems, Java Card Platform Security, Technical White Paper, October 2001
4. Hubbers, E., Poll, E.: Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviors. Department of Computer Science NIII-R0438, Radboud University Nijmegen (2004)
5. Witteman, M.: Java Card security. Inf. Secur. Bull. **8**, 291–298 (2003)
6. Mostowski, W., Poll, E.: Malicious code on Java Card smartcards: attacks and countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85893-5_1
7. Bouffard, G., Khefif, T., Lanet, J.-L., Kane, I., Salvia, S.C.: Accessing secure information using export file fraudulence. In: CRiSIS, pp. 1–5 (2013)
8. Noubissi, A., Séré, A., Iguchi-Cartigny, J., Lanet, J.-L., Bouffard, G., Boutet, J.: Cartes puce: attaques et contremesures. In: MajecSTIC 16.1112 (2009)
9. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.-L.: Combined software and hardware attacks on the Java Card control flow. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011). doi:10.1007/978-3-642-27257-8_18
10. Faugeron, E.: Manipulating the frame information with an underflow attack. In: Francillon, A., Rohatgi, P. (eds.) CARDIS 2013. LNCS, vol. 8419, pp. 140–151. Springer, Heidelberg (2014). doi:10.1007/978-3-319-08302-5_10
11. Bouffard, G., Lanet, J.-L.: The ultimate control flow transfer in a Java based smart card. Comput. Secur. **50**(2015), 3346 (2015). doi:10.1016/j.cose.01.004
12. Farhadi, M. , Lanet, J.L.: Chronicle of Java Card death. J. Comput. Virol. Hacking Tech. 1–15 (2016). doi:10.1007/s11416-016-0276-0
13. 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)
14. Dubreuil J.: Java Card security, software and combined attacks. In: SSTIC (2016)
15. Lancia, J., Bouffard, G.: Java Card virtual machine compromising from a byte code verified applet. In: 14th CARDIS, Bochum, pp. 75–88 (2015)
16. Lancia, J., Bouffard, G.: Fuzzing and overflows in Java Card smart cards. In: SSTIC Conference, Rennes, France, June 2016
17. Laugier, B., Razafindralambo, T.: Misuse of frame creation to exploit stack underflow attacks on Java Card. In: Homma, N., Medwed, M. (eds.) CARDIS 2015. LNCS, vol. 9514, pp. 89–104. Springer, Heidelberg (2016). doi:10.1007/978-3-319-31271-2_6
18. Schneier, B.: Attack trees. Dr. Dobb J. **24**(12), 21–29 (1999)
19. Common Criteria, Common Criteria for Information Technology Security Evaluation, version 3.1, July 2009