Attacks on Java Card 3.0
Combining Fault and Logical Attacks

Guillaume Barbu[1,2], Hugues Thiebeauld[1], and Vincent Guerin[1]

[1] Oberthur Technologies - France
`http://www.oberthur.com/`
{`g.barbu, h.thiebeauld, v.guerin`}`@oberthur.com`
[2] Telecom ParisTech, Dep. ComElec, Groupe SEN - France
`http://www.telecom-paristech.fr/`
`guillaume.barbu@telecom-paristech.fr`

Abstract. Java Cards have been threatened so far by attacks using ill-formed applications which assume that the application bytecode is not verified. This assumption remained realistic as long as the bytecode verifier was commonly executed off-card and could thus be bypassed. Nevertheless it can no longer be applied to the Java Card 3 *Connected Edition* context where the bytecode verification is necessarily performed on-card. Therefore Java Card 3 *Connected Edition* seems to be immune against this kind of attacks. In this paper, we demonstrate that running ill-formed application does not necessarily mean loading and installing ill-formed application. For that purpose, we introduce a brand new kind of attack which combines fault injection and logical tampering. By these means, we describe two case studies taking place in the new Java Card 3 context. The first one shows how ill-formed applications can still be introduced and executed despite the on-card bytecode verifier. The second example leads to the modification of any method already installed on the card into any malicious bytecode. Finally we successfully mount these attacks on a recent device, emphasizing the necessity of taking into account these new threats when implementing Java Card 3 features.

Key words: Java Card 3, Combined Attack, Fault Injection, Logical Attack.

1 Introduction

Nowadays Java Card technology is widely spread over the smartcard market for a large spectrum of applications, such as banking, identity or GSM. According to [19], more than 3.5 billion of Java Cards have been deployed worldwide so far, proving the needs in inter-operability, post-issuance loading, multi-application capability and security.

Fundamentally, smartcards are devoted to play a key role in secure transactions operating potentially in hostile environments. They are designed to resist to numerous attacks using both physical and logical techniques. Today fault attacks represent certainly the most powerful threat for smartcards. They consist in inducing a fault during a code execution as explained in [5] and then in exploiting either a faulty computation result or an erroneous behavior to obtain information on secrets stored in the card. Although fault attacks have been mainly used in the literature from a cryptanalytic angle [2, 6, 3], their strength is to potentially stress every code layers embedded in a device. Practical details and comprehensive consequences could be found in [8].

Thanks to the inherent structure of the Java language, Java Cards have shown an improved robustness compared to native applications regarding fault attacks. However a device offering post-issuance loading and multi-application capability must also face to new threats associated to these features. Thus, the so-called malicious applets which are specifically developed by an attacker aiming at tampering with a Java Card device, should then be taken into consideration. Until now, all attacks based on malicious applet only used logical techniques to defeat the Java Cards security [24, 17, 13].

In this paper, we will introduce for the first time how a fault injection and a logical attack using a malicious applet can be combined to defeat a Java Card 3. The novelty of this paper is twofold. Firstly, such a combination has never been exploited until now. It turns out to be a very efficient way to tamper with a device like a Java Card. Secondly, we will show that even if the Java Card 3 standard appears to be very well designed with a real concern for security, it is still possible to attack devices embedding straightforward Java Card 3 implementations. We will also demonstrate that our attack is not purely theoretical, as it was successfully put into practice on a recent chip.

This paper is organized as follows : In Section 2, a brief reminder of Java Card 3 is exposed, with a special interest on the new features introduced by this standard. In Section 3, after a brief description of already published logical attacks on Java Card, we analyse if they are still relevant in the Java Card 3 context. In Section 4, we introduce a new kind of attacks combining fault injection and logical tampering. Two case studies are then exposed in Section 5 revealing how the security of a Java Card 3 device can be defeated. Finally, we discuss in Section 6 how to protect a platform against that kind of attack and how to handle the security to anticipate any weakness.

2 Brief description of Java Card 3

This section aims at describing the context of the Java Card 3, with a special interest on the newly introduced features.
To follow the still growing requirements in embedded security, the Java Card 3.0 specification has been released [1]. For a best suitability, two editions are available: the *Classic* and the *Connected*.

2.1 Classic Versus Connected Java Card 3.0 Editions

The *Classic Edition* stands for a moderate evolution of the previous Java Card 2.2.2 standard [22, 20] and ensures a regular compatibility with classical applets and Java Card platforms deployed so far. Therefore all previous vulnerability analyses [24, 17, 13] applied on previous Java Card products remain valid on devices implementing *Classic Edition* Java Card 3.0.
The major evolution of Java Card 3 concerns the *Connected Edition* which represents a significant breakthrough compared to the previous Java Card standards. This latest release offers a myriad of new features and then opens up new opportunities for possible applications. This standard addresses the high-end range of devices and mainly targets the network field, where the smartcard plays a vital part in security.

The most interesting features introduced by Java Card 3.0 *Connected Edition* are:

- a strong evolution of the language, now closer to the standard java,
- a multi-threading capability,
- a connectivity adapted to the most common network standards (TCP/IP, HTTP(S)),
- an on-card class loader and linker.

For the sake of interest only the *Connected* version of Java Card 3.0 will be taken into consideration in the remainder of this paper.

2.2 Java Card 3.0: A Set of New Security Features

The complete specifications of the Java Card 3 *Connected Edition* have been revisited and enriched by new requirements to ensure a very high level of security. The main security features concern:

- a context isolation mechanism, more commonly called the *application firewall*, ensuring that objects created in an application are not accessed by any other application,
- a mandatory On-Card Bytecode Verifier (OCBV) to prevent any ill-formed applet to be loaded and installed,
- a code isolation mechanism,
- optional security annotations to specifically define in the code particular sequences requiring a stronger security,
- secure communications such as Transport Layer Security,
- a security policy enforced by role-based and permission-based rules.

Why would the OCBV influence the Java Card security?

2.3 The ByteCode Verification

The bytecode verification [21] consists in checking the coherence of the CAP file before installing the applet on the card. Such operation turns out to be costly in term of code size, which can be critical for resources-restricted devices like smartcards. For this reason, up to the Java Card 3 standard, the specifications [21] allowed the possibility to execute this bytecode verification outside the card, which is the case for a majority of the Java Cards deployed so far.
To force this verification, Global Platform (GP) has specified the notion of Data Authentication Pattern (DAP). According to the Java Card configuration set by the issuer, an optional DAP verification can be requested during each new applet installation, consisting in checking the signature validity. The issuer or the application provider has then to sign the CAP file after processing the bytecode verification.

Considering the Java Card 3 *Connected Edition*, the loading and installation of ill-formed applets is now compromised since the bytecode verifier is now on card. Therefore it could be interesting to analyse the consequences of this new feature on the previously published attacks.

3 State of the Art of Java Card Software Attacks

In literature, previous works [24, 17, 13] attempting to attack Java Card devices assumed systematically that the attacker has the right to load an applet. Firstly, the context describing how one can load and install his own applet is recalled. And then, previous attacks are briefly described in order to analyse how they can be applied on Java Card 3 *Connected Edition*.

3.1 Loading an Applet: in the Jungle of Permission

The right to load an applet is not obvious in the reality of the field. To have this capability, an attacker has several possibilities:

1. The attacker owns the card manager key set, which means he plays the role of the issuer. No additional conditions are requested, the attacker is then able to load any package or install an applet without any further restrictions. It is the case for instance of white cards that anyone can buy on several web stores. Such cards are commonly provided with the card manager key set.
2. The attacker is in condition to load a package or install an applet by *Delegated Management* (only available from GP 2.1.1 [9]). That means the issuer has created on the card a kind of loading environment, called a *Security Domain*, providing loading, installation and extra secure communication features with certain privileges. The use of *Security Domains* requires ownership of some secret keys to achieve both authentication and communication through a secure messaging. Furthermore, loading and installing the applet by *Delegated Management* requires the INSTALL commands to be signed by the card's issuer.

3. The attacker is in condition to load a package or install an applet by *Authorized Management* (only available from GP 2.2 [10]). To do so, it is necessary that a *Security Domain* has been created beforehand by the issuer. The access to this *Security Domain* requires being authenticated and then owning the corresponding key set.

In conclusion, having the opportunity to load an applet on a Java Card is not obvious. This is simple indeed for everyone using white cards. However in that case, few assets are at stake, rendering the reach of any attack limited. Otherwise we can consider very unlikely the possibility for a basic attacker to load his own applet in normal conditions of use.
Nevertheless this assumption is necessary to consider any software attack. Therefore, in the following of this paper, we will assume:

H_0: ***The attacker is able to load and install applications on card.***

3.2 Ill-formed Applets: The Threat Number One

Until now, attacks threatening Java Cards are mainly divided into two categories:

– They exploit a weakness in the Virtual Machine (VM) implementation [24], or even a bug in the atomic operations [17]. These attacks are still valid in Java Card 3 *Connected Edition*. However, the number of such attacks is limited and can be easily addressed by developers without affecting the product performances.
– Another kind of attack concerned the execution of ill-formed applets. Such techniques are now considered as a classical attack (a brief description could be found in [7]). Their principle is to modify the CAP file in order to defeat security controls ensured typically by the firewall. They can be extremely powerful, as it was shown in [24], [17] and [13].

Why would the security of a Java Card be threatened by a CAP file modification?

Firstly, [24] and [17] have exposed how the type confusion could eventually lead to either Non Volatile Memory dump possibilities or to firewall circumventing. The success of their attacks was nevertheless conditioned by the absence of dynamic controls embedded in the platform, explaining why the attacks failed on some cards.
Secondly, [13] has described how changing a bytecode could provide the knowledge of manipulated references. This information was then exploited in a second step of the attack, consisting in changing the *Method Component* contained in the CAP file, improving an attack proposed by [12]. Finally [13] showed how a malicious applet's method could be replaced with a data array playing so the rule of a trojan horse, and giving the access in reading and writing to a large part of the memory space. Once again, the viability of this attack is mainly dependent on the implementation choices of the different cards tested.

Moreover, every ill-formed attack requires that the CAP file has not passed the bytecode verifier. As explained in Section 2.3, this assumption is not longer applicable on Java Card 3 *Connected Edition*. The direct consequence seems to be an apparent protection face to ill-formed applet attacks, and to most of others attacks published in literature so far.
However, this bytecode verification remains static, as it is executed once when the applet is being installed. The attack described in the next section will show why this "static" protection is not sufficient. We will demonstrate that logical attacks are still possible by combining them with a single fault injection during the application execution.

4 Combined Attack on Java Card 3.0, Theory and Practice

As explained in Section 3, it is now admitted that loading ill-formed application in order to get a type flaw is not an option anymore. In this section, we will firstly demonstrate that a combined attack can be an alternative to an ill-formed application loading. Then, we will show how this can be particulary dangerous for a Java Card 3.0 platform.

4.1 Attack Step 1 : Combining Fault and Logical Attacks to Forge References

In this section we will firstly recall some basics about type conversion in Java. Then, we will demonstrate how a single physical disturbance of the code execution will enable us to induce a type confusion. This idea has been suggested in [17] and presented in [4] but neither any theoretical nor any practical examples have been published. Finally, we will show that the type confusion will permit to forge references and even possibly read and write their content, and this until the deletion of the application.

Recalls on Type Conversion

It is common knowledge that Java objects' types (classes) organization forms a hierarchy. Each class is a subclass of another class, except for the `Object` class on top of the hierarchy. This hierarchy enforces the principle of conversion which allows an object of type `T1` to be used as if it were an object of type `T2`. A type conversion can be explicitly requested in the source code by the use of the cast operator: `()`.
For type safety reason, such conversions must be checked. Conversions proven incorrect at compile time result in a error. But, in most case, the check will happen at runtime *via* the `checkcast` instruction produced by the compiler and executed by the VM. Such an example is given below:

```
T1 t1;                  aload X
T2 t2 = (T2) t1;   ⇔    checkcast Y
                        astore Z
```

where `X`, `Y` and `Z` point respectively to `t1`, `T2`'s class and `t2`.
The `checkcast` instruction takes as parameter the class into which the object (on top of the stack) is being converted. Its execution will merely consist in checking that the object is convertible into the given class regarding the class hierarchy. If the conversion is correct, the object reference is still on top of the stack at the end of the `checkcast` execution. Otherwise, a `ClassCastException` is thrown and the stack is cleared.

How a Faulty Conversion Leads to Reference Forgery

In [11], *Govindavajhala et al.* proposed a way to achieve type confusion and reference forgery on a virtual machine thanks to memory errors. Our approach, although slightly different, is inspired by their attack.

We consider the following classes[3]:
- `public class A {byte b00,...,bFF;}`
- `public class B {short addr;}`
- `public class C {A a;}`

Let us focus on the internal representation of instances of `B` and `C` classes (*cf.* Fig. 1). It is important to notice that both objects have the very same internal structure.

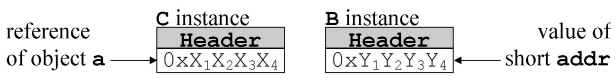

Fig. 1. Internal representation of instance of `B` and `C`

Imagine we can access an object either as an instance of B or C. Treating this object as a `B` instance, it is possible to set the value of its short field (`b.addr = 0x1234;`). And due to the internal structures of `B` and `C` classes, we have set the reference of the `a` field of this very object seen as an instance of `C`, as illustrated in Fig. 2.

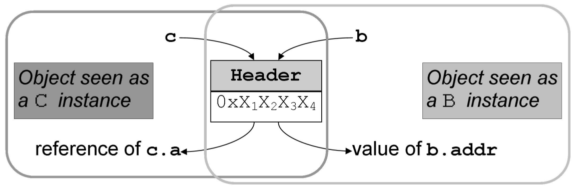

Fig. 2. Access to the same object either as `B` or `C` instance

[3] The size of a reference is implementation specific. Therefore the type of field `addr` in `B` could be either `short` or `int`.

Now let an application module containing the following extended applet (as well as classes A, B and C) be loaded, and this applet installed, on a recent chip embedding a straightforward implementation of the Java Card 3 *Connected Edition* specifications.

```
 1. public class AttackExtApp extends Applet {
 2.  B b; C c; boolean classFound;
 3.  ... // Constructor (objects initialization), install method
 4.  public void process(APDU apdu) {
 5.   byte[] buffer = apdu.getBuffer();
 6.   ...
 7.   switch (buffer[ISO7816.OFFSET_INS]) {
 8.    case INS_ILLEGAL_CAST:
 9.     try {
10.      c = (C) ( (Object) b );
11.      return; // Success, return SW 0x9000
12.     } catch (ClassCastException e) {/*Failure, return SW 0x6F00*/}
13.    ... // more later defined instructions
14. } } }
```

Obviously, this application is well-formed and the OCBV will allow its loading and installation. However, the reader may have noticed the incorrect cast conversion of a B instance into a C instance (step 10)[4]. Checking the correctness of cast conversions is not in the scope of the OCBV, it is left to the checkcast execution, at runtime, which will prove this one incorrect.

In [23], *Vermoen et al.* successfully applied the principle of Power Analysis (PA) [14, 15] to Java Cards in order to reverse engineer an applet. In our case, we will only rely on PA to monitor our application's execution, which is much less difficult.

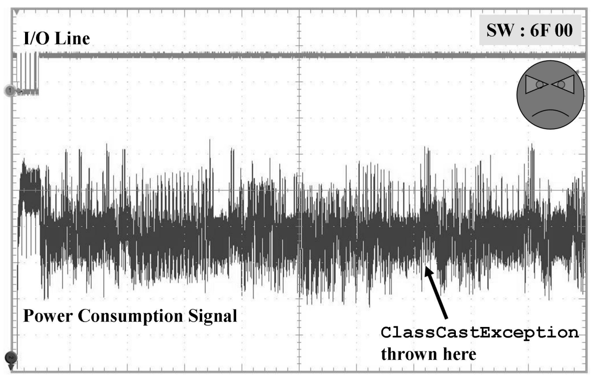

Fig. 3. Execution of the applet's INS_ILLEGAL_CAST instruction

[4] The Object conversion is only meant to fool certain compilers. This conversion will probably not even be checked as each class is a subclass of Object.

By analysing this power consumption curve, we are able to determine the moment when the `ClassCastException` is thrown and thus when the `checkcast` is executed.

We are now going to disturb the execution at the precise moment when the `checkcast` is executed. For this purpose, we will use a laser equipment, targeting the back side of the chip. The following figure (Fig. 4) depicts the faulty execution of the same instruction.

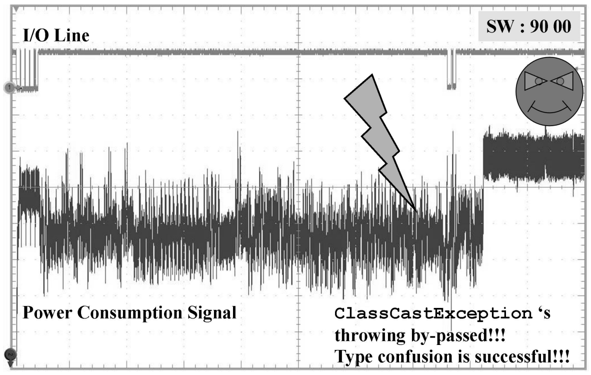

Fig. 4. Disturbed execution of the applet's INS_ILLEGAL_CAST instruction

We can see the instruction's execution is a little shorter than the regular execution in Fig. 3 (because the VM doesn't have to treat an exception raising) and the returned status word is the one expected when no error has occurred (`90 00`). The attack succeeded.

We are then able to access an actual `B` instance either as a `B` or a `C` object. Thus we can forge `a`'s reference to any value (*via* `b.addr`), which in turn may let us read and write as many bytes as declared byte fields of class `A` (*cf.* Fig. 5.).

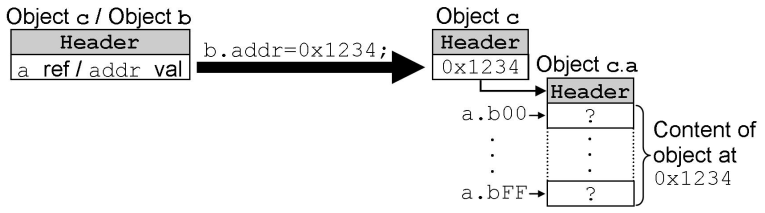

Fig. 5. Forgery of object `a`'s reference

This can be done without requiring any additional disturbance. The sole security check we encountered has been permanently[5] neutralized by one single fault injection.

Can we then dump the whole VM heap? Surely not. Even forging references, access to an object that is not owned by our application and not shared must be forbidden by the application firewall, as specified in [21] and result in a `SecurityException` being thrown. Also, the behavior of the platform when trying to access bytes beyond the object's size, thanks to forgery, is not specified and is then typically implementation dependent.

Nevertheless, we are able to assign any reference to `c.a` and possibly read and write bytes `c.a.b`XY within the boundaries fixed by the application firewall and the VM implementation. This is roughly equivalent to the type-confusion-based attacks presented in Section 3 on Java Card 2.x platforms. However, we do not need ill-formed application loading nor specification/implementation flaws, the fault injection being the type confusion's cause.

4.2 Attack Step 2 : Using our Reference Forgery Tool against a Java Card 3 Platform

We are now going to expose how the reference forgery tool presented in Section 4 can jeopardize a Java Card 3 *Connected Edition* platform thanks to the new dynamic features. We will start by setting our working hypothesis. Then we will explain how we can access and modify `Class` objects on the platform.

An Assumption about the `Class` Object

One of the features introduced by the Java Card 3 *Connected Edition* platform is on-card class loading. This can be used within an application thanks to the `java.lang.Class` class that has been added to the standard API [18] which specifies that *"Instances of the class* `Class` *represent classes and interfaces in a running Java application"*. `Class` objects are constructed by the class loading process, as defined in the standard Java VM specification [16], from the binary representation of these classes (the `.class` file). Working in a constrained environment, we cannot expect the `Class` object to be the exact copy of the `.class` file. Nevertheless we venture the following hypothesis:

H_1: ***The bytecode of a class is stored in its*** `Class` ***instance.***

This assumption appears quite natural as the `Class` object aims at representing a running or ready-to-run class. Besides, if the `.class` file format can be optimized in some ways, the bytecode array itself cannot be modified without modifying the behavior of the methods it represents.

[5] As long as our application is not deleted from the card.

REMARK. *Another interesting point concerning `Class` object is that the specification requires root classes (applet, servlet, filter and listener classes), dynamically loadable classes and shareable interface classes of an application module to be loaded and linked during this application module loading. Thus we know that these instances of the class `Class` are constructed as soon as an application is loaded.*

Searching and Accessing `Class` Objects

Section 4 shows how we can forge reference of an `A` instance and access memory using its byte fields. This access will be executed on the platform respectively by the `getfield` and `putfield` instructions wether we try to get (read) or set (write) the byte value. A particularity of Java Card 3 *Connected Edition* is that its specification [21] allows access to implicitly transferable objects *via* `getfield` and `putfield` instructions. An implicitly transferable object is an object that is not bound to a specific java context.

Therefore, when an application requests access to such an object, the application firewall will grant the access instead of checking that the java context of the application matches the requested object's java context. In other words, such objects are not protected by the application firewall. The list of specified implicitly transferable classes [21] contains an interesting element: `java.lang.Class`.

To access a `Class` object (*i.e.* to forge `b`'s reference to that of a `Class` object instance), we need to know the fully qualified name of this class and have the following instruction in the process method of our attack application:

```
 1. case INS_SEARCH_CLASS:
 2.   while (!classFound) {
 3.     try {
 4.       // Increment the forged reference
 5.       b.addr++;
 6.       // Convert the bytes given in APDU command into String
 7.       String name = bytesToString(buffer, ISO7816.OFFSET_CDATA);
 8.       // Is it a Class instance ?
 9.       if (((Object) (c.a)) instanceof Class) {
10.         // Is it the Class instance we're looking for ?
11.         // Let us check its name
12.         if (((Class)((Object) (c.a))).getName().equals(name))
13.           classFound = true;
14.       }
15.     } catch (SecurityException se) {}
16.   }
```

REMARK. *In this instruction, we already take advantage of the implicitly transferable property of `Class` objects by using type conversion, the `instanceof` instruction and the `getName()` method on the forged reference (steps 9 and 12).*

Another way to achieve this could be to use the `hashCode` method of the `Object` class provided it is typically implemented as per [18] :

> "As much as is reasonably practical, the `hashCode` method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java^{TM}programming language.)"

Under H_1, and provided we can forge `a`'s reference to a `Class` object's reference, then we can modify this class's bytecode array, regardless of the application it belongs to.
We expose in the following section two case studies of such an attack.

5 Applications of our Combined Attack

In this section, we will present two case studies based on our combined attack proving we can execute ill-formed code despite the OCBV and modify any application installed on the card.

5.1 Case Study 1 : Ill-Formed Code Injection

The OCBV prevents ill-formed applications from being loaded. This case study will show that our reference forgery tool enables an attacker to execute any sequence of bytecode instructions.

Imagine the attacker's application module contains an additional class with dummy methods filled with instructions ment to produce an easy to detect (and to modify) bytecode within the corresponding `Class` instance.

To access his dummy `Class` object, he only has to use the `INS_SEARCH_CLASS` instruction with the proper class name (he obviously knows) to forge `a`'s reference. He can then easily read the content of the `Class` object and detect the bytes corresponding to his dummy method. He can finally write the bytecode he wants, in disregard for any rule (Fig. 6).

This proves that under hypotheses H_0 and H_1, one can use the reference forgery tool to eventually have ill-formed code loaded on card despite the OCBV and without any additional fault injections.

REMARK. *Considering the state of the art, an application containing erroneous bytecode will not be more hazardous than the type-confusion we already got. Actually, we have the same chances to dump memory than with the attacks published in [24] and [17]. With a good knowledge of the `Class` object's structure we could also try to modify the static field resolution, as proposed in [12] and used in [13], to circumvent the application firewall. Nevertheless, this may enable future ill-formed-application-based attacks to target platform protected by OCBV.*

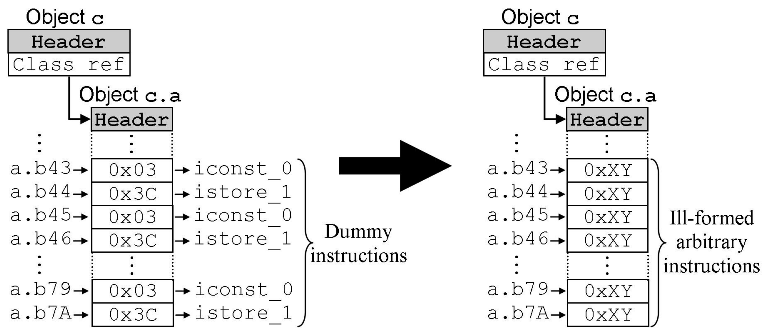

Fig. 6. Identification of `dummyMethod` and ill-formed code injection

5.2 Case Study 2 : Modifying any Application Behavior

Unlike the previous case study, we will now fully take advantage of the implicitly transferable property of `Class` objects. Thanks to our reference forgery tool, the attacker is also able to modify any other applications regardless its context, endangering thus the whole platform integrity.

To illustrate how dangerous this can be, we study here the case of an application whose security relies on user/client authentication based on a signature scheme. The designers of this application being totally confident in the embedded signature scheme, since its implementation has been certified resistant against all kinds of side channel and fault attacks.

Therefore, somewhere in this application's code, the following lines will appear:

```
1. if (sig.verify(inBuff, inOff, inLen, sigBuff, sigOff, sigLen)) {
2.   ... // Success, access granted.
3. } else {
4.   ... // Failure, access denied.
5. }
```

Consider now an attacker who wants to access this application's assets. Without the knowledge of the signature's private key he cannot be successful. But the forgery tool will allow him to circumvent this obstacle.

Thanks to the transferability of `Class` objects, under H_1 and provided the attacker knows the fully qualified name of the class containing the call to the `verify` method, he is then able to forge a reference to the corresponding `Class` instance (still using the `INS_SEARCH_CLASS_OBJECT` instruction). He will then have access to its bytecode array.

Knowing the `verify` method's descriptor, he can deduce that a call to this method will consist in pushing the `sig`'s reference and all the arguments on the stack (`inBuff`, `inOff`, ...).

He can then identify the bytes involved in the call to the `verify` method in the bytecode array (Fig. 7).

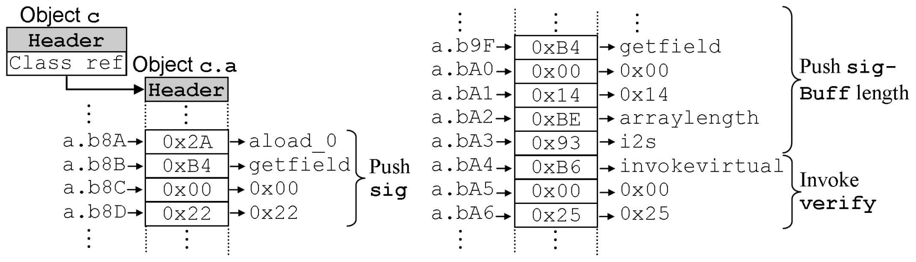

Fig. 7. The call of the `verify` method

Finally, he has just to set all these bytes to `0x00`, which corresponds to the `nop` instruction (*i.e.* `no operation`), except the last one, to which he assigns the value corresponding to `iconst_1`, pushing the value `1` on the stack (Fig. 8).

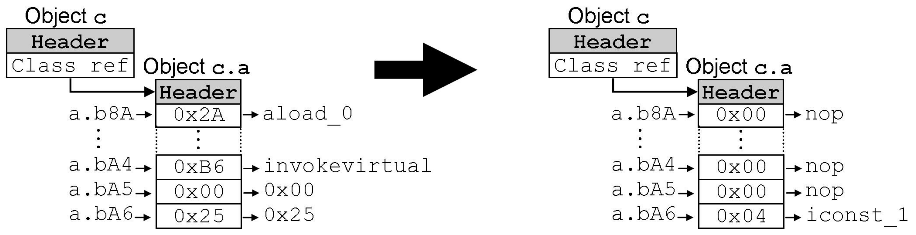

Fig. 8. Making the signature verification always successful

Operating this modification, the attacker changes the application's code as if its Java source code were the following:

```
1. if (true) {
2.   ... // Success, access granted.
3. } else {
4.   ... // Failure, access denied.
5. }
```

He has then granted access to the application's assets whatever the value of the signature.

This simple case study shows the potential threat such attacks can represent. Our application becomes a trojan horse capable of modifying other applications from the inside (as suggested in [13]). The number of possible attack *scenarii* is only limited by the attacker's imagination. Besides, although a good knowledge of the target application will ease the attack, we can eventually consider it as not necessary. If an attacker is able to read the content of all `Class` objects, identifying his target amongst those should not require huge efforts.

6 Security Concerns

The attacks described in Sections 4 and 4.2 reveal that some weaknesses in open platforms can defeat the whole security of a device. In a same vein, the attack described in [13] has shown that once a trojan horse had been setup, it could lead, in certain implementation conditions, to an access in both writing and reading to a large part of the memory, opening then plenty of possibilities to affect seriously the device reliability.

These examples illustrate that the open platform security relies on the implementation quality. Our attacks show that the Java Card 3 *Connected Edition* is not an exception. This new standard with all his additional features is not less secure than the previous versions, quite the reverse, and does not seem to contain any weakness. However to achieve a high level of security, even if a specification has been designed with a real concern for security, it must be associated to an appropriate security-oriented implementation. Therefore we can raise the question, how implementing the Java Card 3 specifications taking into account those threats?

Even if the chips appear to be more and more resistant with regard to fault injections, this risk should systematically be taken into consideration in the software, as an adequate complement. To achieve a resistant implementation, well-known countermeasures could be inserted, like execution flow controls, doubling sensitive operations and checking their coherency, variable redundancies, etc (*cf.* [8]). In the context of our attacks, the `checkcast` bytecode should then be considered as sensitive and handled accordingly.

Moreover, the success of the case studies described in Section 4.2 relies on an implementation choice: for optimization reason, as explained in Section 4.2, it was natural to associate the bytecode to the `Class` objects. However, it turns out to be a potential vulnerability regarding the logical attacks. It explains why the developer should often find the accurate balance between performance and security. A good know-how of the potential risks is then necessary to prevent any implementation weakness.

Last but not least, this attack has revealed that static controls could still be circumvented. We mean by static controls the controls performed for instance during the bytecode verification, ensuring the `.class` file is well-formed. However, once this verification achieved, the bytecode coherency is not checked during its execution. In other words, the installation of ill-formed applet is nearly impossible, but not its eventual execution. In normal conditions of use, the application needs obviously to be loaded and installed beforehand, the platform then looks secure. Nevertheless, for an attacker, once this first static verification bypassed, the platform does not ensure anymore an adequate protection, leading to a potential vulnerability. Finally the mandatory OCBV is a consequent security improvement, but it does not take the place of efficient dynamic controls performed during the VM execution.

7 Conclusion

In this paper, a new attack combining fault injection and logical tampering has been presented and applied on the Java Card 3 *Connected Edition*. We have demonstrated that this kind of attack is very efficient. It is thus possible to either alter any method regardless its java context or even to execute any bytecode, even ill-formed, bypassing the OCBV. Such vulnerabilities successfully applied on a device would affect dramatically its security.
Our attack is specific to Java Card 3.0 for two reasons. Firstly it held in a context of a mandatory OCBV, which is a specificity of the last Java Card specification. Secondly the possibility to handle `Class` objects, a newly introduced feature, was exploited.
Its practicability has been successfully demonstrated on a recent micro-controller, with a straightforward Java Card 3 implementation. These results have revealed the necessity of a secure implementation, even when specifications are designed to resist to the current state of the art attacks on smartcards.

Acknowledgement

The authors would like to thank Nicolas Bousquet, for his valuable suggestions during the design of the case studies presented in this paper, and Nicolas Morin, for his contribution to this work, putting the theoretical attack into practice. Additionally, thanks to Christophe Giraud and Philippe Hoogvorst for their helpful comments during this paper writing.

References

1. Allenbach, P.: Java Card 3 : Classic Functionality Gets a Connectivity Boost. http://java.sun.com/developer/technicalArticles/javacard/javacard3/ (2009)
2. Anderson, R., Kuhn, M.: Tamper Resistance – a Cautionary Note. In: Proceedings of the 2nd USENIX Workshop on Electronic Commerce, USENIX Association (1996) 1–11
3. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.P.: Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In: Cryptographic Hardware and Embedded Systems (CHES'02). Volume 2523 of LNCS., Springer (2002) 260–275
4. Barbu, G.: Fault Attacks on Java Card 3 Virtual Machine. In: e-Smart'09. (2009)
5. Bauduin, R.: Fault Attacks, an Intuitive Approach. In: Fault Diagnosis and Tolerance in Cryptography (FDTC´06). (2006) Invited talk.

6. Boneh, D., DeMillo, R., Lipton, R.: On the Importance of Checking Cryptographic Protocols for Faults. In: Advances in Cryptology - EUROCRYPT'97. Volume 1233 of LNCS., Springer-Verlag (1997) 37–51
7. Common Criteria: Application of Attack Potential to Smartcards - Version 2.7, Rev.1. (2009)
8. Giraud, C., Thiebeauld, H.: A Survey on Fault Attacks. In: Smart Card Research and Advanced Application Conference (CARDIS'04). LNCS, Springer-Verlag (2004) 159–176
9. GlobalPlatform Inc.: GlobalPlatform Card Specification 2.1.1. (2003)
10. GlobalPlatform Inc.: GlobalPlatform Card Specification 2.2. (2006)
11. Govindavajhala, S., Appel, A.: Using Memory Errors to Attack a Virtual Machine. In: IEEE Symposium on Security and Privacy (SP'03). (2003)
12. Hyppönen, K.: Use of Cryptographic Codes for Bytecode Verification in Smartcard Environment. Master's thesis, University of Kuopio, Finland (2003)
13. Iguchi-Cartigny, J., Lanet, J.L.: Évaluation de l'injection de code malicieux dans une Java Card. In: Symposium sur la Sécurité des Technologies de l'Information et de la Communication (SSTIC'09). (2009)
14. Kocher, P., Jaffe, J., Jun, B.: Introduction to Differential Power Analysis and Related Attacks. Technical report, Cryptography Research Inc. (1998)
15. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Advances in Cryptology - CRYPTO'99. LNCS, Springer-Verlag (1999) 388–397
16. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification (2nd Edition). Addison-Wesley (1999)
17. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In: Smart Card Research and Advanced Application Conference (CARDIS'08). LNCS, Springer-Verlag (2008) 1–16
18. Sun Microsystems Inc.: Application Programming Interface, Java Card Platform Version 3.0.1 Connected Edition. (2009)
19. Sun Microsystems Inc.: Java Card Portal. (http://java.sun.com/javacard/)
20. Sun Microsystems Inc.: Runtime Environment Specification, Java Card Platform Version 2.2.2. (2006)
21. Sun Microsystems Inc.: Runtime Environment Specification, Java Card Platform Version 3.0.1 Connected Edition. (2009)
22. Sun Microsystems Inc.: Virtual Machine Specification, Java Card Platform Version 2.2.2. (2006)
23. Vermoen, D., Witteman, M., Gaydadjiev, G.: Reverse Engineering Java Card Applet Using Power Analysis. In: Proceedings of the 1st Workshop on Information Security Theory and Practice (WISTP'07). (2007)
24. Witteman, M.: Java Card Security. In: Information Security Bulletin. Volume 8. (2003) 291–298