Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures

Guillaume Barbu[1,2], Guillaume Duc[2], and Philippe Hoogvorst[2]

[1] Oberthur Technologies, Innovation Group,
Parc Scientifique Unitec 1 - Porte 2,
4 allée du Doyen George Brus, 33600 Pessac, France

[2] Institut Télécom / Télécom ParisTech, CNRS LTCI,
Département COMELEC,
46 rue Barrault, 75634 Paris Cedex 13, France

Abstract. Until 2009, Java Cards have been mainly threatened by Logical Attacks based on ill-formed applications. The publication of the Java Card 3.0 Connected Edition specifications and their mandatory on-card byte code verification may have then lead to the end of software-based attacks against such platforms. However, the introduction in the Java Card field of Fault Attacks, well-known from the cryptologist community, has proven this conclusion wrong. Actually, the idea of combining Fault Attacks and Logical Attacks to tamper with Java Cards appears as an even more dangerous threat. Although the operand stack is a fundamental element of all Java Card Virtual Machines, the potential consequences of a physical perturbation of this element has never been studied so far. In this article, we explore this path by presenting both Fault Attacks and Combined Attacks taking advantage of an alteration of the operand stack. In addition, we provide experimental results proving the practical feasibility of these attacks and illustrating their efficiency. Finally, we describe different approaches to protect the operand stack's integrity and compare their cost with a particular interest on the time factor.

Key words: Java Card, Fault Attack, Logical Attacks, Combined Attack, Countermeasures.

1 Introduction

Java Card systems are generally considered as intrinsically safer than native ones due to the security brought by the Java Card Runtime Environment (JCRE). Indeed the strongly-typed Java language and the abstraction layer provided by the Java Card Virtual Machine (JCVM) thwarts many Logical Attacks, such as stack overflow for instance. Therefore numerous attacks against Java Cards consist in corrupting the binary representation of Java Card applications in order to bypass the inherent security of the platform [1–3]. Nevertheless, the recent Java Card 3.0 Connected Edition has rendered such Logical Attacks (LA) unpracticable by making on-card bytecode verification mandatory. That is to say,

it should not be possible to load an application that is not conform to the Java Card specifications [4–7].

However, as every embedded system, Java Cards are sensitive to attacks based on physical phenomena, amongst which fault-injection-based attacks. The principle of a fault injection on a smartcard is to modify the physical environment of the card in order to provoke an abnormal behavior of the component. It can target either the processor, the data/address bus or even the memory cells [8]. Since their publication in 1996, Fault Attacks (FA) have been mainly tackled in the literature with regards to embedded cryptographic implementations [9–11]. However these attacks can potentially target any function of an embedded system [12].

Two years ago, the idea of combining FA with LA has emerged [13]. Such attacks, called *Combined Attacks* (CA), use a fault injection to allow a malicious application to bypass the security mechanisms of the system. CA have turned out to be very efficient against improperly secured platform [14, 15]. This highlights the need to neglect none of the components of an embedded system when dealing with fault detection, and with security in general.

In Java-based systems, the *operand stack* appears as a central element. However its behaviour when targeted by fault injection has never been studied in the literature. In this paper we investigate this path and describe both FA and CA against Java Cards by disturbing the operand stack. We present practical results and detail two case-studies leading to the corruption of an application execution flow and an unduly granted authentication. These case studies prove the necessity of carefully ensuring the integrity of the operand stack. To reach this goal, we present and compare the efficiency of three different countermeasures.

The rest of this paper is organized as follow. In Section 2, we relate the uttermost importance of the operand stack in a Java Card environment. We also introduce the notions relative to FA and CA and present the fault model we consider in this work. Section 3 describes several Fault Attacks targeting the operand stack and leading to abuse Java Card applications. Section 4 presents how an attacker can threaten the platform and other applications with Combined Attacks focused on the operand stack. Finally, Section 5 describes different countermeasures against such attacks and compares their respective costs.

2 Basics of Operand Stack, Fault and Combined Attacks

In this section we give an overview of the operand stack in a Java-based environment. Then we detail the principles of FA and CA. Finally, we define the fault model we have chosen in the context of our work and discuss this choice.

2.1 The Operand Stack, a Central Element of the JCVM.

The JCVM, and more generally, Java Virtual Machines (JVMs) are known as stack-based machines, in opposition to register-based machines. Actually, several stacks are described in the JVM specification [16]. We focus our interest on one kind of these: operand stacks.

A Java *frame* is created on each Java method *invoke* to store temporary VM-specific data. The operand stack is the part of this frame in charge of holding the operands and results of the VM instruction. Most of these instructions consist in popping a certain number of operands, executing a specific process and pushing a returned value. For instance, the execution of an `iadd` (adding two integer values: *value1* and *value2*) is specified as follows:

QUOTE. *"Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is value1 + value2. The result is pushed onto the operand stack." [16]*

The integrity of the values passing through the operand stack appears then crucial. In this paper, we focus our attention on this central element of the JCRE and study its robustness with regards to fault injections.

2.2 Fault and Combined Attacks

In this section we intend to introduce the notions relative to FA and CA.

FA and the notion of fault model. Embedded systems are subject to the laws of physics. The impact of physical phenomena on such systems has been widely studied by the scientific community, with a particular interest on secure systems. This interest has led to the conclusion that without particular protections, sensitive information can be retrieved from the so-called *side channel leakages* such as execution timing, power consumption or electromagnetic radiation. But another conclusion that has been drawn is that by modifying the physical environment of the system, one can alter its behaviour. Such physical perturbation can be caused by various tools such as a laser beam or a glitch generator. This is the basis of the *perturbation* or *fault-injection attacks*.

In order to evaluate the possible consequences of a fault injection, it is then necessary to provide a model of the possible errors induced by the perturbation. Different fault models are commonly considered in the literature which mainly depends on:

- the impact of the fault:
  - whether it corrupts a bit, a byte, a data-word.
  - whether the value is:
    - set to a random value.
    - *stuck-at* all-0 or all-1.
- the precision of the fault.

Combined Attacks. Until 2009, the majority of the literature dealing with Java Card security remained focused on the effects of Logical Attacks (LA) [1, 2, 17, 18]. These attacks are generally based on the corruption of the binary representation of a Java Card application (*.cap* or *.class* file) into a so-called ill-formed application before it is loaded on-card. Such modifications aim at circumventing certain controls enforced by the JCVM. But in most cases, they also make the application illegal with regards to the Java Card specifications. Therefore the modified application should not be able to pass static analysis tools such as the Java bytecode verifier. The bytecode verification being a costly process, it is generally executed off-card on Java Card 2.2.2 and earlier, as a part of the application development tool chain. The usual philosophy of LA is then to skip this step and to directly load unverified applications on platforms that allow it.

The recently released Java Card 3.0 Connected Edition specifications, has made mandatory the on-card execution of the bytecode verification. Therefore loading ill-formed application is not possible anymore. This statement has given a push to the introduction of the combination of LA with FA into the Java Card field and practical applications have been published over the last two years. In these works FA are used to bypass certain security mechanisms in order to allow a LA. The so-called Combined Attacks allow then to take the benefits of both FA and LA. Indeed, they are more realistic than LA since they do not rely on an unverified application loading and potentially more powerful than FA since the malicious application can make permanent changes and act like a trojan inside the card.

2.3 The Selected Fault Model

In the scope of this work, we only consider FA targeting a JCVM. This has led us to define a fault model allowing the attacker to modify the value pushed onto the operand stack into a predetermined value or even to a chosen value, with some limitations. Indeed we consider two different fault models: the common *stuck-at* fault model and a model taking into account the value previously pushed onto the stack. This model is detailed below.

In the constrained context of single-threaded Java Cards, optimization may lead to use a single global operand stack. However, according to the specifications, an operand stack is allocated within a Java frame, on a method *invoke*. In both cases, this allocation is most likely done in RAM. Therefore, pushing an operand on the stack consists in writing this operand at a given address that only depends on the number of elements already on the stack.

In our fault model, the fault injection targets the execution of the JCVM. More precisely, we assume that the perturbation allows to prevent (at least partially) the updating of the operand stack during a push operation. The resulting erroneous value would then be either all-0, all-1 or a value resulting from an

incomplete writing. As a consequence, and assuming the attacker knows the values previously pushed onto the operand stack, we can conclude that she is able to predetermine the erroneous value. Furthermore, assuming she can run and attack her own application on the platform, she can choose the value previously pushed onto the stack and therefore control the resulting erroneous value. The experimental validation of this fault model is shown in Appendix A.

The following sections details possible exploitation of fault injections following this fault model through both FA and CA.

3 Fault Attacks on the Operand Stack

In this section we explore some potential consequences of a successful fault injection on an integral value pushed onto the operand stack. In the rest of this section, we assume that the attacker can only execute applications already loaded on-card. We start with a brief description of an attack on the instruction byte of the APDU (Application Protocol Data Unit) buffer. Then we raise the issue of boolean values in a Java environment with regards to fault injection and put into practice FA on a conditional branching instruction of the VM.

3.1 Taking Advantage of Erroneous Integral Values

As any other smartcard, a Java Card follows the ISO 7816 specifications [19]. Particulary, Java Card applets receive their command through APDUs, according to the specified format (Fig. 1).

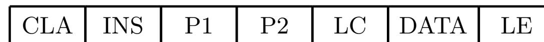

Fig. 1. Format of an APDU command.

The Java Card API provides a class representing APDUs and a virtual method to access the data sent within this APDU through a byte array. This byte array defines then the behaviour of the applet. For instance to select a specific instruction in the process method, an applet execute typically the following lines:

```
byte ins = apduBuf[ISO7816.OFFSET_INS];
switch (ins) {        // push ins on the stack and execute the
                      // appropriate switch instruction.

  case INS_A: processInstructionA(apdu); break;
  case INS_B: processInstructionB(apdu); break;
  ...
  default: ISOException.throwIt(ISO7816.SW_INS_UNKNOWN);
}
```

The value of `ins` is pushed onto the operand stack before executing the `switch` instruction. Therefore, a successful fault injection during the pushing of the value is likely to totally change the behaviour of the applet. However, the consequences are then dependent on the applet itself, but considering an e-wallet applet, turning a payment into a credit operation is definitely interesting from the attacker's point of view. The chances of success are nevertheless quite low regarding our fault model since the previous value on the operand stack (`apduBuf`'s reference in this case) is not known and there is no reason why it should be in relation with the different instructions.

3.2 The Case of Boolean Values

In this section, we discuss the particular cases of the boolean type and of conditional branching instructions. Then we describe FA on such instructions and give experimental results proving the efficiency of our fault model.

Booleans and conditional branching in Java Card. Amongst the basic types of the Java language, we find different types of integral values differing by their size or sign (byte, char, short, ...). But we also find a specific boolean type, which supports only two values: `true` and `false`. Indeed, the Java language forbid the use of any other type than boolean in `if` statement, unlike C language for instance.

Nevertheless, there is no such thing as a boolean type at the bytecode level and the Java compiler produces only bytecodes manipulating values of type `int` when processing operations on boolean variables. Finally, and most importantly with regards to the remainder of this section, the conditional branching instructions produced by the compilation of a simple if statement: `ifeq` and `ifne`, only compare the top of stack value (*i.e.* the previously pushed operand) with 0 and branch or not depending on the result of this comparison (branch if the comparison succeeds in the case of an `ifeq`, branch if the comparison fails in the case of an `ifne`). That is to say, the specification imposes that any other value than 0 will be interpreted as `true` by the JCVM.
One may note that this statement is true for any Java-based system.

FA against conditional branching instructions. Several choices are offered to an attacker in order to corrupt a conditional branching instruction on a Java Card. In this section we give the details of a FA against an `ifeq` instruction evaluating a positive (true) condition by setting the previously pushed operand to 0. We consider the following code (application Java source code on the left and the corresponding bytecode on the right):

```
1.  boolean b = dummyTrue();          |        1. aload_0
                                      |        2. invokevirtual #96
                                      |        5. istore 6
2.  if (b) {                          |        7. iload 6
```

```
                                       |       9. ifeq 12
 3.    Util.setShort(buffer,           |      12. aload_2
         (short)0,(short)0x1111);      |      13. iconst_0
                                       |      14. sipush 0x1111
                                       |      17. invokestatic #84
                                       |      20. pop
 4.  }                                 |
 5.  else {                            |
 6.    Util.setShort(buffer,           |      21. aload_2
         (short)0,(short)0x2222);      |      22. iconst_0
                                       |      23. sipush 0x2222
                                       |      26. invokestatic #84
                                       |      29. pop
 7.  }                                 |      ...
 8.  Util.setShort(buffer,             |
         (short)2, proof);             |
```

The `dummyTrue()` method initializes the instance field `proof` (used at line 8) proving the method has been executed and return a boolean value (`true`).

The target of our attack is this value pushing, before the `dummyTrue()` returns. The goal of the attack is then to force this value to 0. In case of success, the `ifeq` instruction will result in a jump at line 21 in the bytecode sequence and the returned value will be 0x2222 instead of 0x1111.

Experimental results. We put this attack into practice on a recent smartcard embedding a Java Card 2.2.2 VM. The fault injection is achieved with a laser beam applied on the rear-side of the component.

After empirically searching the fault injection parameters (timing, impact location, intensity) that "maximize" the number of successful FA, we reach a success rate of 78.25%, out of 10,000 disturbed executions of the application. We then adapt the test application to attack an `ifne` instruction by changing line 2 of the Java source code into `if (!b)`. Once the fault injection parameters adjusted, we reach this time a success rate of 70.92%, also out of 10,000 disturbed executions.

The results of similar attacks on a `false` condition evaluation are expected to be at least as good as the results obtained above. Indeed, since any value other than 0 is interpreted as `true`, any alteration of the pushed operand would lead to a successful attack.

These FA definitely raise the security issue caused by the specifications of the `ifeq` and `ifne` instructions, and to a certain extent by the lack of a real boolean type at the Java bytecode level.

4 Combined Attack through Faulty Object References

In this section we consider the combination of a fault injection in the operand stack and a malicious application. This implicitly assumes that the attacker has the opportunity to load and execute her own application on the platform. This privilege is far from obvious on released products. However such attacks must be considered in the context of platforms allowing post-issuance application loading like Java Cards. In the following, we describe two CA taking advantage of a faulty object reference on the operand stack in slightly different ways: type confusion and instance confusion.

4.1 Yet Another Way to Type Confusion

Type safety is a fundamental element of Java-based systems in general and of Java Cards in particular. Consequently this property has been largely studied and is used in many of the published attacks against Java Cards.

We do not provide another particular type-confusion-based attack in this section, but we describe how a fault injection in the operand stack can lead to break the type safety property with a good probability.

As previously stated, we consider that the attacker can load her own application on-card. Nevertheless, we assume that the application has to pass a bytecode verifier to be loaded. This bytecode verifier can be either on-card, as specified in the latest Java Card specifications or off-card. Provided the bytecode verifier is sound, the malicious application has then to be well-formed. As a consequence, the well-known *.cap* file or *.class* file manipulation to cause a type confusion is not an option.

Our strategy to break type safety is basically the same as the one proposed in [20]. That is to say, the attacker creates in her application several instances of a given class C and counts on an error to modify the Java reference of a given instance of another class $C*$ into that of one of the several instances of C. The main difference with the work presented in [20] is that our fault model allows us to predetermine the error. Therefore the success rate of the attack should not depend on the number of instances of class C that have been created although a sufficient number of instances of class C can be necessary in practice.

4.2 Instance Confusion: The Case Study of Security Role Impersonation

In this section, we introduce the concept of instance confusion and present the case study of an attack using this concept.

Instance confusion. By analogy to the concept of type confusion, where an instance of a class C is used as if it were an instance of another class $C*$, we introduce the concept of *instance confusion*. An instance confusion consists in using an instance i of a given class (or of a class implementing a given interface) as if it were another instance $i*$ of the same class (or of a class implementing the same interface).

Obviously, instance confusions within the bounds of the attacker's application may not represent a threat. Furthermore, to take advantage of an instance confusion outside the bounds of her application, the attacker should have to circumvent the Java Card application firewall. In the remainder of this section, we show that an appropriate instance confusion can allow the attacker to unduely gain privileges in another application on a Java Card 3.0 through the use of an authentication service.

JC3.0 user authentication. The Java Card 3.0 specifications provide an authentication facility through a dedicated set of service interfaces. These interfaces are organized as illustrated in Fig. 2.

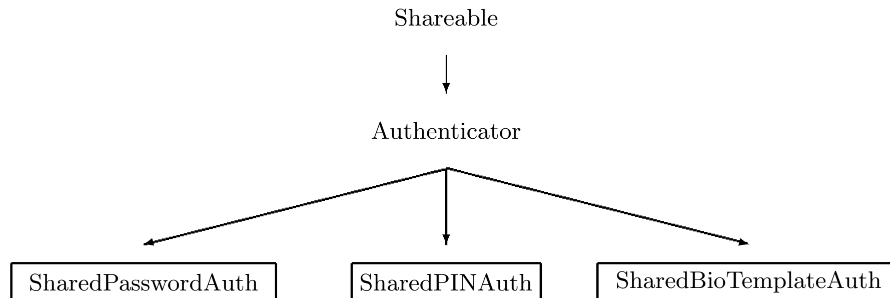

Fig. 2. Java Card 3 authenticator classes and interfaces hierarchy.

To allow user authentication, these shared services are mapped to specific Unified Resource Identifiers (URI) as any other Shareable Interface Object (SIO). These SIOs are first registered into the service registry by the application providing the service through the `register` method of the `ServiceRegistry` class of the Java Card API. They can then be retrieved in the service registry using their URI by calling the `lookup` method of the same class.
Each of these authentication service interfaces expose methods allowing to:

- authenticate a user with provided credentials (`check`),
- check wether or not a user is authenticated (`isValidated`),
- reset the authentication status of an authenticated user (`reset`).

These methods are typically called either by the application, or by the web container to restrict access to specific services or content, as detailed in the JCRE specifications (§6.4.4 and 6.4.5 of [4]). The important point to notice is that these methods are exposed in shareable interfaces. That is to say they are accessible across the application firewall. They are then likely to be abused by an attacker through an instance confusion.

Setting up and exploiting instance confusion. Let us assume the attacked application uses the service offered by the `SharedPasswordAuth` interface. The first step for the attacker is then to create and load an application with several instances of a class implementing this interface. This class would typically have `check` and `isValidated` methods always returning `true` and `reset` method doing nothing, to bypass access control.

When the targeted application is about to get the authenticator instance (*i.e.* when the `lookup` method pushes its Java reference onto the operand stack), the attacker can then try to corrupt it. If she manage to provoke an instance confusion between the legitimate authenticator and one of her own, any call to the `check` or `isValidated` method would return `true` and the targeted client application would have all the reasons to consider her as an authenticated user.

The attacker may then access critical services within the attacked application. It is important to notice that even if redundant checks are performed to verify the authentication, one successful fault on the authenticator's reference is sufficient. This attack appears then more powerful than the FA on conditional branching of Sect. 3.2, since in this case each additional check would require an additional perturbation of the component.

Experimental results. Our experiments are done on a Java Card 2.2.2. Therefore, we cannot use the API's authenticator interfaces. However, we experiment our attack on an applet holding one instance a of a class A implementing an interface I and 256 instances of a class B, also implementing I. We intend then to prove that an attacker should be able to provoke an instance confusion on specific objects. To match the previously described attack, the application obtains object a through a virtual method `getA()` and stores it in a local variable of type I, the interface implemented by A and B. The attack consists then in injecting a fault while pushing a onto the stack and check if the resulting operand is an instance of B. The test application is then the following:

```
byte[] buffer = apdu.getBuffer();
buffer[0] = 0x7F;
I a = getA(); // attacked method

if (a instanceof Object) {
   if (a instanceof B)
      buffer[0] = 0x01;  // SUCCESS
   else if (a instanceof A)
```

```
        buffer[0] = 0x02;
}
```

Out of 10,000 attacked executions of our application, we obtain the following results:[3]

- 8.74% of success: the operand popped from the stack is an instance of B.
- 25.42% of attack failure : the operand popped from the stack is an instance of A, i.e. a itself, the fault injection had no effect.
- 65.86% of unknown error : the execution of the application did not complete, *i.e.* an exception was thrown or the fault injection caused a card failure.

With regards to a theoretic security of about 2^{40} if we consider an 8-character password, an attack on a password-based authenticator with a success rate of about 10% is quite outstanding.

This section has shown that a CA taking advantage of an erroneous value on the operand stack can be even more dangerous than FA. Finally, the two previous sections have proven the need to ensure the integrity of the operand stack.

5 Countermeasures

Within this section we present different approaches to design a software countermeasure against the attacks previously described. The aim of these countermeasures is then restrained to protecting the system against a faulty value on the operand stack. Also, we focus here on dynamic checks in the context of a defensive VM and do not consider static defenses such as detecting potential dangerous mutation within an application [21, 22].

5.1 When to Check for Faults?

A foundation of countermeasure designing lies in the definition of the assets to protect. A good comprehension of the threats is necessary to achieve this work. This section aims at analysing the attacks previously described in order to determine the operations that are sensitive and thus worth protecting. As the attacks target the JCRE, we will consider operations at the Java bytecode level.

In the context of Java Cards, we want to prevent:

- Data from being unduely sent out of the card,
- Applications from ill-behaving.

[3] As expected with regards to our fault model, increasing the number of instances of class B up to 1024 did not really enhance the success rate of the attack (almost 10%).

Indeed, these identified assets summarize the assets defined in the Java Card Protection Profile (§3.2 of [23]).

We can then restrict fault detection to the bytecode instructions related to:

- Field manipulation (get/putfield, get/putstatic).
- Control-flow breaks (invokes, returns, conditions, exceptions).

Other instructions are arithmetic or logic operations, or operate on local variables. Apart from operation on static class fields, those are the operations protected by the Java Card application firewall.

5.2 Software Fault Detection

In this section we detail different approaches to detect faults within the scope of the JCVM.

The basic approach: redundant checks. The most straightforward implementation of a fault detection mechanism on the stack would be to check the coherency between the value pushed onto the stack and the top of stack value after the push operation. Likewise, with regards to the pop operation, we will check the coherency between the value that has been popped and the former top of stack value. That is to say:

```
push(expected);
if (get_tos() != expected)
   handle_fault();
```

for the push operation.

```
expected = pop();
if (get_prev_tos() != expected)
   handle_fault();
```

for the pop operation.

We implemented this countermeasure on a Java Card Virtual Machine. The additional costs on various bytecode instructions are presented in Table 1 of Sect. 5.3.

1st refined approach: propagating errors to ensure fault detection. Our approach to reduce the cost of redundant check is to propagate a potential error to another component of the JCVM. This is then only valid if the standard JCVM behaviour is to check this other component.
The Java Card application firewall aims at ensuring a strict isolation between the different applications and the JCRE. A typical implementation of this mechanism we found on numerous cards and simulation tools is to assign a context identifier to each application. This identifier is also assigned to each object instance created within the scope of an application. The context isolation is then enforced by comparing an object context identifier and the current application identifier, according to the JCRE specification [4]. We choose to propagate the operand stack errors to this value. The implementation of the countermeasure

is then:

```
push(expected);
fw_context_id |=
   (get_tos() ^ expected);
```

for the push operation.

```
expected = pop();
fw_context_id |=
   (get_prev_tos() ^ expected);
```

for the pop operation.

Consequently if an error occurs on the pushed value, the current context of ownership is modified. Therefore an attacker is no longer able to retrieve data from the attacked application since she would have to either call virtual or interface methods to send data out of the card or eventually use instance class fields. In both ways, she would have to pass through the application firewall and the firewall will not allow it. Similarly, if the fault aims at corrupting a conditional branch, the subsequent execution will be interrupted as soon as a firewall check occurs. An additional check is only necessary on access to static fields that are not protected by the application firewall.

The fact that few additional checks need to be inserted (only for access to static fields) is clearly an advantage regarding the computational cost of this method. The major drawback of this method is that corrupting the `fw_context_id` value, it is possible (although we consider the chances as low) that we fix it to the value identifying another application installed on the card. In such a case, our countermeasure would eventually open a breach in the application firewall. Table 1 of Sect. 5.3 presents the experimental cost of this refined countermeasure.

2nd refined approach: introduction of a stack invariant. A second approach to detect faults in the operand stack consist in adding in the Java frame structure a variable σ that allows to exhibit an invariant property.

Definition 1. *σ is the sum, considering the XOR operation, of all the values pushed on and popped from the operand stack.*

We can then exhibit the following invariant property:

Property 1. Let $\mathcal{S}_T$ be the set of all the values contained by the operand stack at a given time T. Then at any given time T,

$$\sigma \oplus \Sigma \mathcal{S}_T = 0$$

Proving this property and its invariance is straightforward. Indeed since σ is by definition the sum of the values pushed onto and popped of the stack, all the values that have been popped have been eliminated from the XOR sum. Therefore, only the values that are still on the stack at a given time T are components of σ.

The implementation of the countermeasure is then:

```
push(expected);
sigma ^= expected;
```

for the push operation.

```
expected = pop();
sigma ^= expected;
```

for the pop operation.

As previously stated, we can check the invariant property on firewall checking and access to static fields and methods by XORing all the values on the stack to σ.
This approach requires then to add a routine in charge of checking the invariant property. Also it requires to add one word in each Java frame created. Table 1 in Sect. 5.3 presents the experimental cost of this countermeasure.

5.3 Costs Comparison

In this section, we present in Table 1 the cost (in time) of the different countermeasures introduced in the previous sections. Then we discuss the result of this comparison and the different benefits and drawbacks of the different approaches.

Instructions	Basic	Propagation	Invariant
`aload+astore`	39.09%	21.98%	12.29%
`aload+getfield+astore`	19.83%	12.39%	11.75%
`aload+aload+putfield`	27.93%	18.77%	17.59%
`aload+invokevirtual+return`	7.53%	1.69%	1.77%
`aload+invokevirtual+areturn+astore`	8.82%	3.26%	2.38%
`aload+putstatic`	18.60%	11.58%	8.89%
`getstatic+astore`	19.18%	10.76%	10.21%

Table 1. Countermeasures impact on bytecode instructions execution time. (% referenced to an initial implementation with no countermeasures.)

As the costs for the different countermeasures are given in percentage, it is important to bare in mind that the different instructions have very different complexity (which explains the large difference between the results for the sequences `aload+astore` and `aload+invokevirtual+return` for instance).

The redundant approach. The performance degradation caused by this straightforward countermeasure turns out to be not acceptable.

The propagation approach. This countermeasure is definitely more efficient than the basic one. To fix the potential issue of context identifier manipulation, an option could be to force legitimate identifiers to even values and propagated errors to odd values. The detection of an invalid identifier would then be straightforward.

The invariant approach. The invariant method is also more efficient than the basic one. Its performance is even a little better than that of the propagation countermeasure. Another advantage of this approach is that it does not present the drawback of a potential breach opening.

As expected, the invariant and propagation approaches turn out to be more efficient than the basic one and are relatively close in terms of performance. Nevertheless, the propagation method requires no additional data and only few additional checks on access to static fields. However, as we implemented it, the propagation method potentially opens a security breach in the application firewall. The use of another variable that would be frequently checked may be recommended. Such variables are typically implementation-dependent and we could not exhibit another quasi-standard one. If such variable should not be found in a particular implementation, the invariant approach has proven to be slightly better than the propagation one in terms of execution time. It should easily be implemented at the cost of an additional data-word per Java frame.

6 Conclusion

As stated in the introduction of this work, Java Cards are safer than native devices with regards to Logical Attacks by nature. However, we have raised in this article the issue of the possible alteration of an operand stack and demonstrated how such attacks can eventually compromise both the Java Card platform and the applications loaded on-card. Indeed, we have described and put into practice both Fault and Combined Attacks against a Java Card by disturbing the operand stack.

In particular, this work has permitted to highlight the weakness that represents the lack of a boolean type at the VM-instruction level. Furthermore, we have exhibited new means to combine fault injection with malicious applications to cause a type confusion despite the bytecode verification, and to abuse authentication services through instance confusion.

Finally, we have detailed and compared different countermeasures. Amongst them, both the propagation and invariant approaches bring a good security level without impacting too much the performances of on-card applications. However, future works intending to refine the identification of the moment when an integrity check should be performed may allow to reduce the cost of these countermeasure.

Acknowledgement

The authors would like to thank Nicolas Morin, for his helping hand during the fault injection campaign, and Christophe Giraud for his fruitful review(s).

References

1. Witteman, M.: Java Card Security. In: Information Security Bulletin. Volume 8. (2003) 291–298
2. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In: Smart Card Research and Advanced Applications. CARDIS '08, Springer-Verlag (2008) 1–16
3. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan Applet in a Smart Card. Journale on Computers and Virology **6** (2010) 343–351
4. Sun Microsystems Inc.: Runtime Environment Specification, Java Card Platform Version 3.0.1 Connected Edition (2009)
5. Sun Microsystems Inc.: Virtual Machine Specification, Java Card Platform Version 3.0.1 Connected Edition (2009)
6. Sun Microsystems Inc.: Application Programming Interface, Java Card Platform Version 3.0.1 Connected Edition (2009)
7. Sun Microsystems Inc.: Java Servlet Specification, Java Card Platform Version 3.0.1 Connected Edition (2009)
8. Giraud, C., Thiebeauld, H.: A Survey on Fault Attacks. In: Smart Card Research and Advanced Applications. (CARDIS'04)
9. Anderson, R., Kuhn, M.: Tamper Resistance: a Cautionary Note. In: Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2. (1996) 1–1
10. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults. In: Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques. EUROCRYPT'97, Springer-Verlag (1997) 37–51
11. Bihame, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystem. In: Advances in Cryptology. CRYPTO'97, Springer (1997) 513–525
12. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of The IEEE **94** (2006) 370–382
13. Barbu, G.: Fault Attacks on Java Card 3 Virtual Machine. In: e-Smart'09, http://www.strategiestm-net.com/proceedings/e-smart/OBERTHUR Guillaume Barby - Fault Attacks on Java Card 3 Virtual Machine.pdf (2009)
14. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In: Smart Card Research and Advanced Application. CARDIS'10, Springer-Verlag (2010) 148–163
15. Vétillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In: Smart Card Research and Advanced Application. CARDIS'10 (2010) 133–147
16. Lindholm, T., Yellin, F.: Java Virtual Machine Specification. 2nd edn. Addison-Wesley, Inc. (1999)
17. Hyppönen, K.: Use of Cryptographic Codes for Bytecode Verification in Smartcard Environment. Master's thesis, University of Kuopio (2003)
18. Hogenboom, J., Mostowski, W.: Full Memory Attack on a Java Card. In: 4th Benelux Workshop on Information and System Security, Proceedings, Louvain-la-Neuve, Belgium (2009) Available at http://www.dice.ucl.ac.be/crypto/wissec2009/static/13.pdf.
19. ISO/IEC: 7816-3: Identification Cards – Integrated Circuit Cards – Part 3: Cards with Contacts – Electrical Interface and Transmission Protocols (2006)
20. Govindavajhala, S., Appel, A.W.: Using Memory Errors to Attack a Virtual Machine. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy. SP '03, IEEE Computer Society (2003)

21. Sere, A.A., Iguchi-Cartigny, J., Lanet, J.L.: Automatic Detection of Fault Attack and Countermeasures. In: Proceedings of the 4th Workshop on Embedded Systems Security. WESS '09 (2009) 1–7
22. Séré, A.A.K., Iguchi-Cartigny, J., Lanet, J.L.: Checking the Paths to Identify Mutant Application on Embedded Systems. In: FGIT. (2010) 459–468
23. Sun Microsystems Inc.: Java Card Protection Profile Collection, version 1.1. Technical report (2006)

A Practical Validation of the Fault Model

The experimental results we present here have been obtained on a recent ARM-based smartcard device. A Java Card 2.2.2 Virtual Machine is running on the device. The fault injection was achieved with a laser equipment.

The test application. We intend to provide an experimental validation of the previously introduced fault model. We then develop a Java Card applet, the `process` method of which is exposed below.

```
1. public void process(APDU apdu) {
2.   [...]
3.   ref = Util.getShort(buffer, OFFSET_CDATA);
4.   target = Util.getShort(buffer, (short) (OFFSET_CDATA+2));
5.   ref = ref;     // push and pop ref : sload #ref
                                         sstore #ref
6.   res = target;  // push and pop target : sload #target
                                            sstore #res
7.   Util.setShort(buffer, OFFSET_CDATA, res);
8.   [...]
9. }
```

The first step of our applet consists then in pushing a reference value onto the operand stack and popping it. Therefore we know which value was written in the operand stack before we proceed.
Then we push a second value onto the stack. This second push is the target of our fault injection. The subsequent popped value which is stored in variable `res` is then expected to be an erroneous value. The last step of the applet process is then to send this value out of the card.
Fig. 3 illustrates the evolution of the operand stack along the execution of lines 5 and 6.

Experimental results. To evaluate the validity of our fault model, we perform several fault attacks with different parameters (namely, the time and space parameters of the attack as well as the width and intensity of the laser beam) and different input parameters.
Table 2 sums up the different results obtained. In this table we only present the results regarding a given couple of input : (`ref`, `target`) = (`0xAABB`, `0xCCEE`).

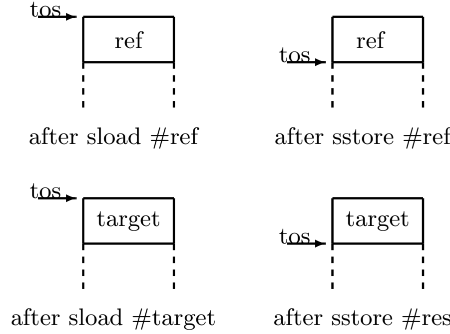

Fig. 3. Evolution of the operand stack content and of the top-of-stack (tos) along execution of lines 5 and 6 of the test applet.

The cells highlighted in grey within Table 2 denote the results that were dependent on the inputs. We also highlighted the results 0x0000 and 0xFFFF which correspond to the two *stuck-at* fault models. Note that we only present in this table the results that were reproducible.

0x00F1	0x0000	0x00F2	0x00CC	0x149C	0x0121	0x0D88	0xFF19
0x0006	0x0129	0x149E	0xEA00	0x00BB	0x27FF	0x168F	0x000D
0x1490	0x4778	0x0011	0xD203	0xC14A	0x00D9	0xAABC	0x1200
0x2B2B	0x0012	0x5576	0xBB00	0x6000	0x7600	0xFFFF	0xAA0B
0xAA8B	0x2AAF	0xAAEC	0xAAEB	0xABB0	0x2AAE	0xAB6B	0xEEA0

Table 2. Results (`res` values) of the fault attacks with various fault injection parameters and fixed `ref` and `target` values.

Conclusions. It is difficult to deduce the exact perturbation caused by the laser beam in all cases, especially when the results are not correlated with the inputs. Such results may be correlated with internal values contained in the registers of the processor at the time the laser is activated.
On the other hand, some results can be easily interpreted since they are compound of different chunks of either the reference or the target value and 0s (for instance, `0x0000`, `0x00CC`, `0xBB00`).

To conclude, the results we obtained only partially validate our fault model since `res` is either all-0, all-1, truncated `ref`, truncated `target` or other unknown values.
However this proves that an adversary can manage to disturb the push operation and may have a certain control on the erroneous operand eventually pushed.