Dynamic Fault Injection Countermeasure
A New Conception of Java Card Security

Guillaume Barbu, Philippe Andouard, and Christophe Giraud

Oberthur Technologies
Security Group
4, allée du Doyen Georges Brus, 33 600 Pessac, France
{g.barbu,p.andouard,c.giraud}@oberthur.com

Abstract. Nowadays Fault Injection is the main threat for any sensitive applications being executed on embedded devices. Indeed, such an attack allows one to efficiently recover any secret or to gain unauthorized privileges if no appropriate countermeasure is implemented. In the context of Java Card applications, the main method to counteract Fault Injection consists in adding redundancy for sensitive operations and integrity verification for sensitive variables. While being efficient from a security point of view, such a method substantially impacts the performance of the application. In this article we introduce a new pragmatic approach to counteract Fault Injection by dynamically increasing the security level of the application. This methodology, based on upgrading the Java Card Virtual Machine, allows us to optimize the performance of sensitive applications in every day life while providing a strong security level as soon as an attacker tries to disturb their executions.

Keywords: Java Card, Fault Injection, Countermeasures.

1 Introduction

1996 was an amazing year for attacks in the embedded environment. Indeed, the concepts of *Side-Channel Analysis* (SCA) and *Fault Injection* (FI) were published that year and they allow an attacker to recover secrets stored in embedded devices even if they are protected by very strong cryptography.

The first kind of attacks have been published by Kocher in [1] where he noticed that the difference of time when executing an application could depend on the secrets manipulated by the device. This attack was then extended by using the power consumption or the electromagnetic radiation of the device as side-channel leakage instead of the execution timing [2,3]. SCA is now reinforced by numerous new attacks and countermeasures every year.

The second kind of attacks revealed in 1996 were published through a press release by the company Bellcore [4]. Three researchers of this company noticed that if the execution of a cryptographic implementation can be disturbed, then the analysis of the corresponding faulty output can lead to the disclosure of the secret key. The announcement of this new way of attacking embedded devices

aroused enthusiasm amongst the cryptographic community and a dozen articles dealing with this subject were published in the few weeks after the Bellcore's announcement. As for SCA, a very large amount of new attacks and countermeasures are published every year to extend the domain of FI which now applies not only to cryptographic algorithms but to each and every kind of application being executed on embedded devices.

These new attacks have been a major breakthrough for the smart card industry. Indeed, at the beginning of 1996, the security of embedded applications relied on the theoretical security of the cryptography which was implemented and on their resistance to *Logical Attacks* (LA) which were known for years and for which the countermeasures were well-known. One year later, basing the security of an application on these two factors only was hopeless. The developers had then had to invent and to implement ingenious side-channel and fault countermeasures which must have the lowest impact on the performances of the application.

At that time, another breakthrough occurred in the smart card environment: the first Java enabled smart card was produced [5]. Java Cards allow the developer to implement an application independently from the device on which it is going to be executed. Such an abstraction layer is provided by the Java Card Virtual Machine (JCVM) which interprets the Java basic instructions, called *bytecodes*, and executes the corresponding instructions for a specific device. Therefore, executing a brand new Java Card application on each and every Java Card on the market costs only one development, leading to the very fast deployment of such an application which cannot be achieved when using native products. Originally used in the mobile environment, Java Cards are now widely used in banking and identity environments where the constraints in terms of security are very strong. Therefore Java Card applications, called *applets*, possibly together with the JCVM, must implement logical, side-channel and fault countermeasures to prevent them from being tampered with.

In this work we present a new way of counteracting FI attacks in the context of Java Cards. Up to now, securing an applet against fault attacks means mainly adding redundancy on sensitive operations and verifying the integrity of the sensitive objects at the applet level. Although being efficient against FI, this leads to a very important overhead in terms of performance and memory consumption. This approach faces its limit when the application has strong constraints in terms of performances which is always the case for contactless applications for instance. Moreover, our new solution also aims at another problematic: why would honest customers (which are the vast majority) have to pay for the dishonest ones? For instance, why would they have to wait 400μs to perform a transaction whereas the same application could run twice as fast if fault countermeasures were removed? Our concept is based on upgrading the JCVM in order to dynamically enforce the security level of an application when detecting an attack. This allows the device to execute by default an application with critical countermeasures only, being thus very fast, and to activate the maximum security level as soon as an attack is detected.

The rest of this paper is organised as follows. In Section 2 we recall some generalities about fault attacks on Java Cards as well as a brief description of the main corresponding countermeasures. In Section 3 we present our new protection concept and we describe two different ways of applying it in practice. We also discuss the benefits and drawbacks of our proposals versus the traditional way of securing an application by adding redundancy and integrity verifications. Finally, Section 4 concludes the paper.

2 Fault Injection Attacks and Common Countermeasures

FI attacks and the analysis of their consequences are well-known in the context of embedded cryptography [6]. However, as stated in [7] and [8], such attacks are absolutely not restricted to arithmetic computations or cryptographic algorithms and are then likely to target any part of an embedded system.

In this section, we briefly present the most common FI attacks against Java Card platform and how they have been combined with LA to bypass specific Java Card security mechanisms. Secondly, we present the usual countermeasures to prevent such attacks.

2.1 Attacks against Java Card Platforms

The first attacks that have been mounted against Java Card platforms were LA. These are usually 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 [9]. Such modifications aim at circumventing certain controls enforced by the JCVM. However 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 allowing it.

On the other hand, FI has been mainly used on Java Card applications to disrupt conditional branching instructions to force a jump in a given branch, favorable to the attacker [10]. For instance, let us observe the piece of code depicted in Listings 1 and 2 which update the balance of an electronic wallet

depending on whether there is a purchase or a refund.

Listing 1. Standard *if-then-else* statement

```
// assume b is a boolean.
if ( b ) {
   // e-wallet credit
}
else {
   // e-wallet debit
}
```

Listing 2. Standard *if-then-else* statement (bytecode sequence)

```
     iload_1
     ifeq L1
     // e-wallet credit
     goto L2
L1:  // e-wallet debit
L2:  ...
```

Although the test `if` is performed on a boolean variable, one may note that there is no boolean type at the bytecode level. The Java compiler produces only bytecodes manipulating values of type `int` when processing operations on boolean variables. Therefore the Java specifications mandate that any value different from 0 will be considered as `true`. From this remark, an attacker disturbing b when performing a purchase will obtain a credit of her e-wallet instead of a debit.
FI have also been used to disturb values returned by methods of the APIs. For instance, the `arrayCompare()` method returns 0 if the two buffers provided as input are identical. If such a method is used to compare a PIN (Personal Identification Number) or a MAC (Message Authentication Code), an attacker presenting a wrong value can force its acceptance by sticking at 0 the corresponding return value.
Moreover, it is now common to assume that attackers with high attack potential are able to perform two faults during the execution of the same command [11]. This statement has a strong impact on the cost of the countermeasures as it will be shown in Section 2.2.

To conclude this brief overview of attacks on Java Card platforms, we recall that recently the use of FI to provoke incorrect behaviours within a malicious but well-formed application appeared as a possible solution to attack Java Card platforms where the bytecode verification is mandatory [12, 13]. In these publications, FI is used to bypass certain security mechanisms in order to allow a LA. The so-called *Combined Attacks* allow then to take benefits of both FI and LA. Indeed, they are more realistic than LA since they do not rely on an unverified application loading and potentially more powerful than FI attacks since the malicious application can make permanent changes and act like a *Trojan* inside the card [14].

2.2 Common Countermeasures and Main Drawbacks

As presented above, FI is a real threat for sensitive applications being executed on a Java Card platform. To counteract such attacks, applet developers must implement specific countermeasures to detect any incoherence during the applet execution. As FI mainly focuses on disturbing conditional branchings and

Listing 3. 2^{nd}-order-secured *if-then-else* statement

```
// assume b is a boolean.
if ( !b ) {
  // e-wallet debit
}
else if ( b ) {
  if ( !b ) {
    ISOException.throwIt(
       ATT_DET_SW);
  }
  else if ( b ) {
    // e-wallet credit
  }
  else {
    ISOException.throwIt(
       ATT_DET_SW);
  }
}
else {
  ISOException.throwIt(
     ATT_DET_SW);
}
```

Listing 4. 2^{nd}-order-secured *if-then-else* statement (bytecode sequence)

```
     iload_1
     ifne L1
     // e-wallet debit
     goto L2
L1:  iload_1
     ifeq L3
     iload_1
     ifne L4
     bipush 18 <ATT_DET_SW>
     invokestatic #6 <throwIt>
     goto L2
L4:  iload_1
     ifeq L5
     // e-wallet credit
     goto L2
L5:  bipush 18 <ATT_DET_SW>
     invokestatic #6 <throwIt>
     goto L2
L3:  bipush 18 <ATT_DET_SW>
     invokestatic #6 <throwIt>
L2:  ...
```

methods execution, one of the most efficient way to detect a disturbance is to add redundancy for each and every sensitive conditional branching and call to a sensitive method. For instance, if we want to protect the `if` used in the code of Listing 1 against double fault attacks, one has to add redundancy testing in the branch favorable to the attacker such as depicted in Listings 3 and 4.

As one can easily observe by comparing Listings 1 and 3, the cost of such a countermeasure is very important. Table 1 illustrates the size and the execution time of both the unsecured and secured versions of the previous sample code.

In order to overcome any specific platform implementation, and therefore specific optimizations, the extra execution time is expressed in terms of number of executed instructions multiplied by t_{ins} where t_{ins} is the average execution time for an instruction.

Table 1. Compared size and execution time overhead of unsecured and 2^{nd}-order-secured *if-then-else* statement.

Mnemonic	Listing	Size (byte)	Timing
Unsecured	2	0	0
2^{nd}-order-secured	4	30	$6 \cdot t_{ins}$

Such an approach is also valid to protect disturbance of methods execution, for instance by executing three times the method `arrayCompare()` and checking that the three different outputs are coherent. This will therefore multiply by a factor 3 the time required to compare two buffers.

To ensure the security of an application, it is of common sense to implement protections against the most efficient practical attacks which are currently double fault attacks. However, even for the best attackers, it is impossible to achieve such attacks the first time round. Indeed, the attacker will have a few failures before succeeding in bypassing a double conditional testing for instance. From this observation, an obvious solution would be to deactivate all security sensitive applications if the card is under attack. However, such an option cannot be applied in some contexts where the provider wants to keep the functionality alive as long as possible (e.g. in the context of Secure Elements or SIM cards). In the next section, we present a new security concept for Java Card platforms. This new methodology is based on modifying the JCVM in such a way that it can dynamically increase the security level of the application when an attack attempt is detected.

3 Dynamic Fault Injection Countermeasure: A Generic Concept Available in Different Flavours

As stated in Section 2, common countermeasures against FI strongly penalize the performance of Java Card applications on a permanent basis. However, applications deployed on the field do not always have to face a real attacker. This section describes how the security of Java Card applications can be modulated by the JCVM without loss of security insurance and details two particular methods to implement this dynamic security concept.

3.1 The Dynamic Security Concept

The execution of an application within the JCVM relies on the interpretation of the bytecode instructions it is made of. Typically, we can consider that the JCVM interpreter associates a given bytecode value to a given function, implementing the Java instruction, according to the JCVM specification [15]. The interpretation of an application is then operated according to a **fetch-decode-execute** sequence, similarly to most *real* machines (by opposition to *virtual* machines), where:

fetch corresponds to the reading of the instruction value in the bytecode array of the current method;

decode corresponds to the translation from the read integer value to the function implementing the corresponding bytecode instruction in the underlying machine's language;

execute corresponds to the execution of the selected function.

Having hands on this sequence, the JCVM can dynamically alter these different steps in order to modify the behaviour of an application on the fly. The generic concept we describe here consists in using this capacity to adapt the security level of a given application to the threats it actually faces. That is to say that the JCVM initially enforces a given security level which will be raised if an attack is detected. As a result, the performance of the application are preserved for honest users whereas attackers will have to deal with the augmented security level.

The security insurance of our concept relies on the state of fact that a few attempts at least are necessary before achieving a successful FI attack. After the detection of a first attack, potentially all countermeasures are activated and the security of the application is ensured in a traditional way.

A prerequisite to the implementation of our concept is then to be able to categorize the different countermeasures ensuring the security of the application into different groups, so that the first group of countermeasures would be always activated, whereas the other group(s) would be activated only after an attack has been detected, according to our concept.

Several options can be adopted to achieve such a discrimination, either based on the attacker's fault injection capabilities or on the data to protect.

Fault attack order First, we can categorize the countermeasures according to the order of the fault attack they are meant to counteract. That is to say that countermeasures against 1^{st}-order attacks (*i.e.* single fault attacks) would be always activated, 2^{nd}-order attack would only be activated if an attack is detected, etc.

Standardized asset hierarchy Second, we can categorize the countermeasures according to the sensitivity of the assets they protect. For instance in the scope of a banking application, countermeasures protecting primary assets, such as the PIN and the DES and RSA secret keys, would be always activated, whereas countermeasures protecting secondary assets, such as the PTC (PIN Try Counter) and the CRM (Card Risk Management), would only be activated if an attack is detected.

Custom asset hierarchy Similarly, we can imagine to let application developers define and organize the assets by using dedicated annotations for instance.

In the following sections, we propose different solutions to implement the dynamic security concept and we discuss their relative advantages.

3.2 *Vanilla*: Inhibiting Security Bytecode Instructions

To save memory in resource constrained devices like smart cards, Java Card bytecode run by the JCVM uses an encoding optimized for size. As a design

tradeoff, Java Card bytecodes are coded on one byte. Nevertheless, only 186 bytecodes are standardized in the context of the JCVM [15], thus leaving free 70 possible proprietary bytecodes.

***Bourbon Vanilla*: Inhibiting Instructions.** Our first proposition is to take advantage of the unused bytecodes by completing the standard instruction set with some security-specific instructions. These new security-specific bytecodes will be recognized by the JCVM and not executed (i.e. inhibited) while no attack has been detected.

On the other hand, once an attack is detected, the execution of these bytecodes will be disinhibited and the extra security will be enabled for the next executions. Those bytecodes (the `inhib_*` below) would implement classical countermeasures, such as:

- executing a software desynchronization function (*e.g.* `inhib_desynchro`).
- verifying the integrity of the currently executing application (*e.g.* `inhib_appcrc`).
- verifying the types of the current local variables (*e.g.* `inhib_typesafe`).
- redundant check of a previous `if*` instruction (*e.g.* `inhib_if*red`).

Based on unused opcodes and classical countermeasures previously described, one is now able to fill up the Java Card instruction set. Table 2 gives one example of this concept.

Table 2. Filling up the instruction set.

Bytecodes	0x00	0x01	...	0xB8	0xB9
Instructions	`nop`	`aconst_null`	...	`putfield_i_this`	`inhib_desynchro`
Bytecodes	...	0xBC	...	0xFE	0xFF
Instructions	...	`inhib_typesafe`	...	`impdep1`	`impdep2`

The next step consists in adding those new bytecodes in the code of an applet. One way to achieve this is to perform a post processing on the .CLASS file from custom rules that add security bytecodes when needed. For instance, each time the function `OwnerPIN.check` is invoked, the `inhib_desynchro` bytecode could be added just before the invocation. Listings 5 and 6 show how this can be applied to the if statement used so far as example.

Such insertion, as well as the numerous modifications on either the .CLASS or .CAP file it implies can be easily achieved by using public tools such as BCEL [16], or CAPMAP [17].

Listing 5. Initial source code

```
// assume b is a boolean.
if ( !b ) {
  // e-wallet debit
}
else if ( b ) {
  // e-wallet credit
}
else {
  ISOException.throwIt(
     ATT_DET_SW);
}
```

Listing 6. *Bourbon Vanilla*-secured bytecode

```
     iload_1
     ifne L1
     // e-wallet debit
     goto L2
L1:  inhib_desynchro
     iload_1
     ifeq L3
     inhib_ifeq_red L3
     // e-wallet credit
     goto L2
L3:  bipush 18 <ATT_DET_SW>
     invokestatic #6 <throwIt>
L2:  ...
```

From now on, by using a dedicated flag of a state machine that indicates whether an attack has been detected or not, the JCVM can enable or disable the security-specific instructions freshly added in the instruction set. Listing 7 describes a way to achieve this goal by modifying the interpreter routine.

This proposition is based on a tradeoff between reaching a high-level of security without impacting the performances when it is not necessary. The solution presented above can be fully automated which implies that no human intervention is required in the process of applet protection.

Still exploiting the concept of inhibiting instructions, we propose another approach which involves the developer in the process of applet protection. The next section introduces such a concept and develops how it could be deployed.

***Tahitian Vanilla*: Inhibiting Sequences of Instructions.** The idea developed above is based on dedicated bytecodes that, when they will be interpreted by the JCVM, will trigger specific countermeasures implemented by the platform. Another embodiment of the previous concept is to leave the choice to the developer to inhibit certain portions of his code, in order to increase the applet security when needed. So, one solution is that the JCVM conditionally triggers the execution of bytecode instructions. This mechanism can be implemented in two stages.

First, specific methods in the applet code delimit the part corresponding to the extra security added by the developer (e.g. `Protection.begin()` and `Protection.end()`) as depicted in Listing 9. These two methods are static and are defined *via* a proprietary API which implementation allow to activate/inhibit the instructions living between the static methods markups. In order to perform the activation/inhibition, again the interpreter routine must be modified as exposed in Listing 8 for instance.

Finally, two scenarii are possible:

Listing 7. Inhibition of security-specific bytecodes

```
// The instruction is read from memory
instruction = fetch();

// Test Inhibiting or not the instruction
if ( isSecuritySpecific( instruction ) ) {
    if ( !flagAttack ) {
        if ( flagAttack ) {
            // Fault detected
        }
        // No attack, no execution of security instruction
    }
    else {
        if ( !flagAttack ) {
            // Fault detected
        }
        // An attack has been detected
        execute( decode(instruction) );
    }
}
else {
    // Protection against fault attacks
    if ( !isSecuritySpecific( instruction ) ) {
        // No attack has been detected
        execute( decode(instruction) );
    }
}
```

- no attack has been detected and the JCVM does not execute the instructions comprised between the `invokestatic #X` and `invokestatic #Y` bytecodes where `#X` (resp. `#Y`) corresponds to the method that enable (resp. disable) the security (see Listing 10).
- an attack has been detected and the JCVM executes all the bytecodes corresponding to the extra security added between the `invokestatic #X` and `invokestatic #Y` bytecodes.

Discussion Ensuring that an applet can trigger tuned levels of security countermeasures only when specific threats are detected can be done via an enrichment of the language recognized by the JCVM. Our first proposition consists in defining new bytecodes in the JCVM. Thus, those security bytecodes can be added anywhere in the .CLASS file and are not executed while a specific flag is not raised. Our second proposition makes use of static methods as markups, to inhibit the non-crucial countermeasures until an attack is detected.

Listing 8. Inhibition of blocks of bytecodes

```
// The instruction is read from memory
instruction = fetch();

// Test Inhibiting or not the instruction
if ( openMarkup && !isCloseMarkup(instruction) ) {
    if ( !flagAttack ) {
        if ( flagAttack ) {
            // Fault Attack detected
        }
        // No attack, no execution of security instruction
        // except to close markup
    }
    else {
        if ( !flagAttack ) {
            // Fault detected
        }
        // An attack has been detected
        execute( decode(instruction) );
    }
}
else {
    // Protection against fault attack
    if ( !openMarkup || isCloseMarkup(instruction) ) {
        // No attack has been detected
        execute( decode(instruction) );
    }
}
```

Table 3 presents the overhead for both propositions in terms of memory footprint and execution time when no attack has been detected compared to the traditional way of securing an applet. As exposed in Listings 7 and 8, an additional `if` is executed within the interpreter routine in order to determine whether an instruction should be first decoded and then executed or not.

Such instructions, at the native level, do not take more than a couple of cycles to be executed. Therefore their impact is limited, although it should not be completely omitted. Consequently, the execution time of an instruction, expressed as $t_{ins} = t_{fetch} + t_{decode} + t_{execute}$ becomes $t'_{ins} = t_{fetch} + 2 \cdot t_{if} + t_{decode} + t_{execute}$. Subsequently, we denote by n_{cred} the number of instructions for the credit operation.

Both these solutions afford a better time/security tradeoff than the common countermeasure. However, we observe that the *Bourbon Vanilla* implementation also allows to reduce the size of the applet. This is due to the fact that the security mechanisms are deported within the JCVM in this case. On the other hand, *Tahitian Vanilla* is equivalent to the traditional method in terms of

Listing 9. Source code with security markups

```
// assume b is a boolean.
if (!b) {
  // e-wallet debit
}
else {
  if (b) {
    Protection.begin();
    if ( !b ) {
      ISOException.throwIt(
          ATT_DET_SW);
    }
    else if (b) {
      Protection.end();
      //e-wallet credit
    }
    else {
      ISOException.throwIt(
          ATT_DET_SW);
    }
  }
  else {
    ISOException.throwIt(
        ATT_DET_SW);
  }
}
```

Listing 10. Bytecode with security markups

```
    iload_1
    ifne L1
    // e-wallet debit
    goto L2
L1: iload_1
    ifeq L3
    invokestatic #7 <begin>
    iload_1
    ifne L4
    bipush 18 <ATT_DET_SW>
    invokestatic #6 <throwIt>
    goto L2
L4: iload_1
    ifeq L6
    invokestatic #8 <end>
    // e-wallet credit
    goto L2
L6: bipush 18 <ATT_DET_SW>
    invokestatic #6 <throwIt>
    goto L2
L3: bipush 18 <ATT_DET_SW>
    invokestatic #6 <throwIt>
L2: ...
```

memory footprint, only the markups being added.

However, these two solutions require modifications of both the applet and the JCVM. The main impact of this approach is that the applet loses its portability.

In the following we propose another approach where the applet is not modified to keep its portability feature.

3.3 *Strawberry*: Inhibiting Secured Bytecode Implementations

The main design goals of the Java Card technology are portability and security. Nevertheless, the protection mechanism presented in Section 3.2 is at the cost of the portability: the "write once, run everywhere" principle does not hold anymore.

On the other hand, one does not want a security mechanism that impacts performances (in term of execution time or code size) when the applet is not under attack.

Table 3. Compared size and execution time overhead of the 2^{nd}-order-secured and *vanilla*-secured `if`.

Mnemonic	Listing	Size (byte)	Timing
2^{nd}-order-secured	4	30	$6 \cdot t_{ins}$
Bourbon Vanilla	6	13	$2 \cdot t_{if} \cdot (5 + n_{cred})$ $+2 \cdot (t_{fetch} + 2 \cdot t_{if})$
Tahitian Vanilla	10	36	$2 \cdot (3 + n_{cred}) \cdot t_{if}$ $+4 \cdot t_{ins'} + 7 \cdot (t_{fetch} + 2 \cdot t_{if})$

To circumvent this issue and keep the portability feature of an applet, an approach consists in different interpretations of a bytecode depending on the execution context (e.g. nominal or under attack). For instance one can implement two different versions of bytecodes: a secure and a non-secure. Thus, by default the JCVM executes the non-secure versions of the bytecodes while when an attack is detected, the secure implementation is executed (see Listing 11).

Listing 11. Inhibited secure bytecodes implementation

```
// The instruction is read from memory
instruction = fetch();

// Switch between non-secure or secure implementation
if ( flagAttack ) {
    // Execution of the secure implementation
    execute( decodeSecure(instruction) );
}
else {
    // Protection against fault attacks
    if ( !flagAttack ) {
        execute( decode(instruction) );
    }
    else {
        // Fault Attack detected
    }
}
```

As FI on Java Card mainly focuses on disturbing conditional branchings and methods execution, it follows that not each and every bytecode in the applet needs to be protected. So, it is sufficient to apply this principle only on bytecodes that are sensitive to fault attacks (e.g. `ifeq`, `ifne`, `sipush`, *etc.*) and on sensitive API methods (e.g. `OwnerPIN.check`, `JCSystem.arrayCompare`, *etc.*).

A secured `ifeq` implementation should, for instance, ensure the integrity of the value read from the operand stack as well as that of the branch taken. The integrity of the value pushed onto the operand stack is liable to a secured implementation of the `sipush` instruction or of the `arrayCompare` API for instance.

The if-then-else statement could then be written straightforwardly, as in Listing 1, the security being dynamically ensured by the JCVM.

Discussion Although coding two versions of bytecodes is a costly process that increases the JCVM's size, it nevertheless has several advantages.

Firstly, unlike the methods detailed in Section 3.2, this approach does not need a modification of the applet. It follows that the applet does not break off the portability paradigm and can be deployed on any standard JCVM.

Secondly, coding an applet while adding security requires a lot of experience to obtain a good tradeoff between execution time, code size and security. So, using that method, even a developer who is not familiar with the concept of FI can develop an applet on a product that could be evaluated and certified.

Moreover, with that approach, there is no need to involve a third party to perform a security proofreading of the applet.

Table 4 presents the additional cost for the *Strawberry* proposition in terms of memory footprint and execution time when no attack has been detected. As for the Vanilla methods, the execution time estimations take into account the additional `if` added within the interpreter routine which is the only overhead for this method, as exposed in Listing 11.

Table 4. Compared size and execution time overhead of the 2^{nd}-order-secured and *Strawberry*-secured if.

Mnemonic	Listing	Size (byte)	Timing
2^{nd}-order-secured	4	30	$6 \cdot t_{ins}$
Strawberry	2	0	$(2 \cdot t_{if}) \cdot (3 + n_{cred})$

Anyway, all the solutions described in Sections 3.2 and 3.3 are not mutually exclusive and can be used in a very flexible way. Indeed, depending on the context, one can adopt a hybrid approach by combining the previous solutions leading to a greater security for the product.

4 Conclusion

In this paper we presented a new approach to protect Java Card platform against Fault Injection. This new methodology allows sensitive applications to run much faster in every day life while providing a very high security level against fault

attacks. By modifying the JCVM, we showed how such a concept can be implemented in practice in two different ways. The first one consists in adding dedicated bytecodes which provides specific security features such as redundancy or desynchronization. These bytecodes will be inhibited until a threat is detected. The second solution consists in implementing some bytecodes twice: one implementation being the standard one and the second implementation including advanced fault countermeasures. The JCVM switches from the first implementation to the second one as soon as an attack is detected. We also compared the advantages and drawbacks of each solution in order to provide all useful information to Java Card developers allowing them to choose the best possible solution depending on their context.

This new methodology is definitely more pragmatic than the traditional approach as it allows Java Card applications to fulfill performances requirements more easily while successfully counteracting advanced fault injection attacks.

Acknowledgments

The authors would like to thank Emmanuelle Dottax for her helpful comments on the preliminary version of this paper.

References

1. Kocher, P.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Koblitz, N., ed.: Advances in Cryptology – CRYPTO '96. Volume 1109 of LNCS., Springer (1996) 104–113
2. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In Wiener, M., ed.: Advances in Cryptology – CRYPTO '99. Volume 1666 of LNCS., Springer (1999) 388–397
3. Quisquater, J.J., Samyde, D.: A New Tool for Non-intrusive Analysis of Smart Cards Based on Electro-magnetic Emissions, the SEMA and DEMA Methods. Presented during EUROCRYPT'00 Rump Session (2000)
4. Bellcore: New Threat Model Breaks Crypto Codes. Press Release (1996)
5. du Castel, B.: Personal History of the Java Card (2012) French version originally published in MISC magazine, HS-2, Nov. 2008.
6. Joye, M., Tunstall, M.: Fault Analysis in Cryptography. Information Security and Cryptography. Springer (2012)
7. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. IEEE **94** (2006) 370–382
8. Giraud, C., Thiebeauld, H.: A Survey on Fault Attacks. In Quisquater, J.J., Paradinas, P., Deswarte, Y., Kalam, A.E., eds.: Smart Card Research and Advanced Applications VI – CARDIS 2004, Kluwer Academic Publishers (2004) 159–176
9. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In Grimaud, G., Standaert, F.X., eds.: Smart Card Research and Advanced Applications, 8th International Conference – CARDIS 2008. Volume 5189 of LNCS., Springer (2008) 1–16

10. Barbu, G., Duc, G., Hoogvorst, P.: Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In Prouff, E., ed.: Smart Card Research and Advanced Applications, 10th International Conference – CARDIS 2011. Volume 7079 of LNCS., Springer (2011) 297–313
11. Common Criteria: Application of Attack Potential to Smartcards. (2009)
12. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In Gollmann, D., Lanet, J.L., eds.: Smart Card Research and Advanced Applications, 9th International Conference – CARDIS 2010. Volume 6035 of LNCS., Springer (2010) 148–163
13. Vétillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In Gollmann, D., Lanet, J.L., eds.: Smart Card Research and Advanced Applications, 9th International Conference – CARDIS 2010. Volume 6035 of LNCS., Springer (2010) 133–147
14. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In Prouff, E., ed.: Smart Card Research and Advanced Applications, 10th International Conference – CARDIS 2011. Volume 7079 of LNCS., Springer (2011)
15. Sun Microsystems Inc.: Virtual Machine Specification – Java Card Plateform, Version 3.0.1 (2009)
16. The Apache Software Foundation: (Apache Commons BCEL, The Byte Code Engineering Library) `http://commons.apache.org/bcel/`.
17. Smart Secure Devices (SSD) Team – XLIM, Université de Limoges: CapMap – The CAP file manipulator. (`http://secinfo.msi.unilim.fr`)