Evaluation of the Ability to Transform SIM Applications into Hostile Applications

Guillaume Bouffard[1], Jean-Louis Lanet[1], Jean-Baptiste Machemie[1], Jean-Yves Poichotte[2], and Jean-Philippe Wary[2]

[1] SSD - XLIM Labs, University of Limoges,
83 rue d'Isle, 87000 Limoges, France
{guillaume.bouffard, jean-louis.lanet, jean-baptiste.machemie}@xlim.fr

[2] SFR, Direction Fraud and Information Security,
1 Pl. Carpeaux, 92915 Paris la Defense, France
{jean-philippe.wary,jean-yves.poichotte}@sfr.com

Abstract. The ability of Java Cards to withstand attacks is based on software and hardware countermeasures, and on the ability of the Java platform to check the correct behavior of Java code (by using byte code verification). Recently, the idea of combining logical attacks with a physical attack in order to bypass byte code verification has emerged. For instance, correct and legitimate Java Card applications can be dynamically modified on-card using a laser beam. Such applications become *mutant* applications, with a different control flow from the original expected behaviour. This internal change could lead to bypass controls and protections and thus offer illegal access to secret data and operations inside the chip. This paper presents an evaluation of the application ability to become *mutant* and a new countermeasure based on the runtime checks of the application control flow to detect the deviant mutations.

1 INTRODUCTION

A smart card usually contains a microprocessor and various types of memories: RAM (for runtime data and OS stacks), ROM (in which the operating system and the *romized* applications are stored), and EEPROM (to store the persistent data). Due to significant size constraints of the chip, the amount of memory is small. Most smart cards on the market today have at most 5 KB of RAM, 256 KB of ROM, and 256 KB of EEPROM. A smart card can be viewed as a secure data container, since it securely stores data and it is securely used during short transactions. Its safety relies first on the underlying hardware. To resist probing an internal bus, all components (memory, CPU, cryptoprocessor...) are on the same chip which is embedded with sensors covered by a resin. Such sensors (light sensors, heat sensors, voltage sensors, etc.) are used to disable the card when it is physically attacked. The software is the second security barrier. The embedded programs are usually designed neither for returning nor modifying sensitive information without guaranty that the operation is authorized.

Today, Mobile Network Operators (MNO) are looking to open their native secure element : the Universal Subscriber Identity Module (USIM) Card [10] to third party services providers, in order to allow them to develop value added services, like m-payment or m-transport for instance (on an NFC based technology). To be able to warrant the security of third parties applications hosted in USIM Card, MNO have choose to certify their Secure Element under the Common Criteria (CC) scheme, and two reference protection profiles have been developped [3], [4] (the CC targeted assurance level is EAL4+ for the USIM platform, it allows any applications to achieve EAL4+ CC level through the CC composition model [12]). To summarize the model USIM card will be certified under CC scheme at an EAL4+ level, and will allow post issuance third parties applications downloading without existing CC certification loss. It means that the industrial process to be put in place will have to warrant the inocuity of every candidate application to a download. The technology described in this paper may allow MNOs to evaluate the ability of every application candidate to download to become a mutant.

Today most of the SIM cards are based on a Java Virtual Machine. Java Card is a kind of smart card that implements the standard Java Card 3.0 [22] in one of the two editions “Classic Edition” or “Connected Edition”. Such a smart card embeds a virtual machine, which interprets application byte codes already romized with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [12]. This protocol ensures that the owner of the code has the necessary credentials to perform the action. Java Cards have shown an improved robustness compared to native applications regarding many attacks. They are designed to resist numerous attacks using both physical and logical techniques. Currently, the most powerful attacks are hardware based attacks and particularly fault attacks. A fault attack modifies part of the memory content or signal on internal bus and leads to deviant behaviour exploitable or not by an attacker. A comprehensive consequence of such attacks can be found in [14]. Although fault attacks have generally been used in the literature from a cryptanalytic point of view [5,13,16], they can be applied to every code layer embedded in a device. For instance, while choosing the exact byte of a program, the attacker can bypass countermeasures or logical tests. We call such modified applications "mutant".

Designing efficient countermeasures against fault attacks is important for smart card manufacturers but also for application developers. Manufacturers need countermeasures with the lowest cost in term of memory and processor usage. These metrics can be obtained with an evaluation of a target. As regards application developers, they have to understand the ability of their applets to become mutants and potentially hostile in case of fault attack. Thus the coverage (reduction of the number of mutant generated) and the detection latency (number of instructions executed between an attack and its detection) are the most important metrics. In this paper, we present a workbench to evaluate the ability of a given application to become a hostile applet with respect to the different implemented countermeasures, and the fault hypothesis.

In order to minimize the impact of fault attacks, developers need to implement countermeasures in applicative code. Examples of such applicative countermeasures are: redundant choices, counters, redefining the value of true and false constants, etc. But in this case the developer must have knowledge of the underlying platform architecture, which can differ from a smart card supplier to another one. This low interoperability of the security aspects between different platforms is a huge problem for smart card application development. The detection mechanism discussed in this paper is integrated to the system thus it ensures portability of the application code.

The contribution of this paper with respect to our prior work is twofold. We define a novel system countermeasure based on a verification by the virtual machine of the control flow and a framework to evaluate the efficiency of countermeasures. This paper is organized as follows: first we introduce a brief state of the art of fault injection attacks and existing countermeasures, then we discuss about the new countermeasure we have developed. In the fourth paragraph we present the evaluation framework and the collected metrics, and finally we conclude with the perspectives.

2 FAULT ATTACKS AND COUNTERMEASURES

Faults can be induced into the chip by using physical perturbations in its execution environment. These errors can generate different versions of a program by changing some instructions, interpreting operands as instructions, branching to other (or invalid) labels and so on. A fault attack has the ability to physically disturb the smart card chip. These perturbations can have various effects on the chip registers (like the program counter, the stack pointer), or on the memories (variables and code changes). Mainly, it can permit an attacker to execute a treatment beyond his rights, or to access secret data in the smart card. Different manners to produce fault attacks [6] exist, most of them differ from the model of the attacker.

2.1 Fault Model

To prevent a fault attack from happening, we need to know its effects on the smart card. Fault models have already been discussed in details [8,24]. We describe in the table 1 the fault models in descending order in terms of attacker power. We consider that an attacker can change one byte at a time. Sergei Skorobatov and Ross Anderson discuss in [21] an attack using the precise bit error model. But it is not realistic on current smart cards, because modern components implement hardware security on memories like error correction and detection code or memory encryption.

In real life, an attacker physically injects energy in a memory cell to change its state. Thus and up to the underlying technology, the memory physically takes the value 0x00 or 0xFF. If memories are encrypted, the physical value becomes a random value (more precisely a value which depends on the data, the address, and an encryption key). To be as close as possible to the reality, we choose the precise byte error that is the most realistic fault model. Thus, we have assumed that an attacker can:

- make a fault injection at a precise clock cycle (he can target any operation he wants),
- only set or reset a byte to 0x00 or to 0xFF up to the underlying technology (bsr[3] fault type), or he can change this byte to a random value beyond his control (random fault type),
- target any memory cell he wishes (he can target a specific variable or register).

[3] bit set or reset

Table 1: Existing Fault Model

Fault Model	Precision	Location	Timing	Fault Type	Difficulty
Precise bit error	bit	total control	total control	bsr, random	++
Precise bit error	byte	total control	total control	bsr, random	+
Precise bit error	byte	loose control	total control	bsr, random	-
Precise bit error	variable	no control	no control	random	–

We have defined the hypotesis concerning the attacker, then we present the effects of such an attack on a Java Card if the attack targets the permanent memory generating a mutant application.

2.2 Defining a Mutant Application

The mutant generation and detection is a new research field introduced simultaneously by [7,23] using the concepts of combined attacks, and we have already discussed mutant detection in [20]. To define a mutant application, we use an example on the following debit method that belongs to a wallet Java Card applet. In this method, the user PIN (Personal Identification Number) must be validated prior to the debit operation.

Listing 1.1: Original Java code

```
private void debit(APDU apdu) {
  if ( pin.isValidated() ) {
     // make the debit operation
  } else {
     ISOException.throwIt (SW_PIN_VERIFICATION_REQUIRED);
  }
}
```

Table 2: Byte Code representation before attack

Byte	Byte Code
00 : 18	00 : aload_0
01 : 83 00 04	01 : getfield #4
04 : 8B 00 23	04 : invokevirtual #18
07 : 60 00 3B	07: ifeq 59
10 : ...	10 : ...
...	...
59 : 13 63 01	59 : sipush 25345
63 : 8D 00 0D	63 : invokestatic #13
66 : 7A	66 : return

In table 2 resides the corresponding byte code representation (the full byte code representation can be found in section B). An attacker wants to bypass the PIN test. He injects a fault on the cell containing the conditional test byte code. Thus the `ifeq` instruction (byte 0x60) changes to a `nop` instruction (byte 0x00). The obtained Java code follows with its byte code representation in table 3:

Listing 1.2: Mutant Java code

```
private void debit(APDU apdu) {
    // make the debit operation
    ISOException.throwIt (SW_PIN_VERIFICATION_REQUIRED);
}
```

Table 3: Byte code representation after attack

Byte	Byte code
`00 : 18`	`00 : aload_0`
`01 : 83 00 04`	`01 : getfield #4`
`04 : 8B 00 23`	`04 : invokevirtual #18`
`07 : 00`	`07 : nop`
`08 : 00`	`08 : nop`
`09 : 3B`	`09 : pop`
`10 : ...`	`10 : ...`
`...`	`...`
`59 : 13 63 01`	`59 : sipush 25345`
`63 : 8D 00 0D`	`63 : invokestatic #13`
`66 : 7A`	`66 : return`

The verification of the PIN code is bypassed, the debit operation is made and an exception is thrown but too late because the attacker will have already achieved his goal. This is a good example of dangerous mutant application: "*an application that passes undetected through the virtual machine interpreter but that does not have the same behavior than the original application*". This attack has modified the control flow of the application and the goal of the countermeasure described in this paper is to detect when such modifications happen.

2.3 Hardware Countermeasures

Fault attacks are powerful and can threaten the card security. So smart card manufacturers have redoubled their efforts by integrating hardware protections that can prevent the fault attacks or make them more difficult to implement. This section is about hardware measures that can be found in the card. We do not claim to enumerate all the possible hardware measures that exist but to give a glimpse of the type of technology that resides in the card. A smart card contains two categories of hardware protection (see [11]):

- Passive protections increase the difficulty of a successful attack. These protections can use bus and memory encryption, random dummy cycles, unstable internal frequency, etc.
- Active protections include mechanisms that check whether tampering occurs and take countermeasures (possibly by locking the card or by generating hardware exceptions on the platform). Some active protection mechanisms are sensors (light, supply voltage, etc), hardware redundancy, etc.

Hardware countermeasures are a good way to protect the card, but are specialized, and they increase the smart card production cost.

2.4 Software Countermeasures

Software countermeasures are introduced at different stages of the development process; their purpose is to strengthen the application code against fault injection attacks. Software countermeasures can be classified by their end purpose:

- *Cryptographic countermeasures*: better implementation of the cryptographic algorithm like RSA (which is the most frequently used public key algorithm in smart cards), DES, and hash functions (MD5, SHA-1, etc.).
- *Applicative countermeasures*: only modify the application with the objective to provide resistance to fault injection. Generally, this class produces application with a greater size. Because beside the functional code (the code that process data), we have the security code and the data structure for enforcing the security mechanism embedded in the application. Java is an interpreted language therefore it is slower to execute than a native language (like C or assembler), so this category of countermeasures suffers of bad execution time and add complexity for the developer.
- *System countermeasures*: harden the system by checking that applications are executing in a safe environment. The main advantage is that the system and the protections are stored in the ROM, which is a less critical resource than the EEPROM and cannot be attacked thanks to checksum mechanisms that allow to identify modification of data that are stored in the ROM. Thus, it is easier to deal with integration of the security data structures and code in the system. Another thing that must be considered is the CPU overhead, if we add some treatments to the functional code.
- *Hybrid countermeasures*: are at the crossroads between applicative and system countermeasures. They consist in inserting data in the application that are used later by the system to protect the application code against fault attacks. They have a good balance between the increasing of the application size and the CPU overhead.

All previous categories with the exception of cryptography, use a generalist approach to detect the fault because they do not focus on a particular algorithm. The developed path check detection mechanism proposed in this paper is a hybrid countermeasure.

2.5 Control Flow Integrity

We have already proposed several solutions to check code integrity during execution using basic blocs check in a previous publication [19]. This paper is about the control flow integrity. In the previous description of a mutant we have shown how to bypass some checks but in [9] we describe how to modify the control flow of an application either by return oriented programming or using a laser beam. In both cases we succeed in executing arbitrary byte code.

The control flow integrity has been already studied for fault tolerance [17,15,18]. These methods attempt to discern program execution deviation from a registered static control flow graph; but they are not adapted to the smart card because of the memory and of the computation power required.

The authors in [2] proposed a generic framework which consists of macros to include in C code. The piece of program can help the programmer to have different structures and mechanisms to save and check various forms of execution history. For instance, a programmer can have a counter to check the number of iterations. One of the most complete methods verifies that execution follows predetermined paths computed during development and stored with the application. If an execution does not follow one of these paths then an error is raised and the platform can take appropriate steps (stop the program, lock the card, etc.). More precisely, the application maintains an execution history composed of a list of program points already passed. These program points have been set during development by the programmer, using macro to easily include code for execution history update. Then the programmer adds "checkpoints" at specific program points. During execution, the history is checked when the application reaches one of the checkpoints. The principle of this mechanism is simple and all valid paths are computed off-card. Significant inputs are required from the developer because he has to explicitly set the program points in the code; this leads to a high level of complexity because the developer needs to determine where program points are to be put, which invariants are to be checked and where these verifications are to be put. Moreover, because the detection mechanism is included in the application, a fault attack against the checking mechanism can bypass the detection phase. Another problem is the increase of the application size, because the list of valid histories and the detection mechanisms are contained inside the application. Thus, an application takes more space in the EEPROM, which is a rare commodity in the card.

Another technique, exposed in [1] is based on the same idea. It uses dynamic runtime checks to allow flow controls to remain within a given control flow graph. They propose to dynamically rewrite machine code of some byte code instructions to ensure that jumps go to a valid code location. To achieve this, an off-card application tags the targets of jump instructions with a unique label and saves the label of targeted instructions for each jump instruction. Then the jump instructions are modified to check during execution that jump instructions continue to one of the valid targeted instructions. But this application suffers from the same problems as the previous solution: it is an applicative countermeasure.

3 CHECKING PATHS DURING RUNTIME EXECUTION

Some points are important when designing a new countermeasure for smart card. This countermeasure must not disturb the application development workflow. It should keep the application size close to the original size. And it should not use up processor resources. To comply with these requirements, we have designed a lightweight software countermeasure that uses static byte code analysis to guarantee that at each step, the virtual machine interpreter follows an authorized path to access some resources.

3.1 Using Java Annotation

The proposed solution uses a security feature found in the Java Card 3 platform: annotations. But it is also fully applicable to Java Card 2 platforms using a preprocessor. When the virtual machine interprets the application code and enters a method or class tagged with a security annotation, it switches to a "secure mode". The code fragment that follows shows the use of annotations provided by Java Card 3 on the debit method. The `@SensitiveType` annotation denotes that this method must be checked for integrity with the `PATHCHECK` mechanism.

Listing 1.3: Java Annotation

```
@SensitiveType{
        sensitivity= SensitiveValue.INTEGRITY,
        proprietaryValue="PATHCHECK"
}
private void debit(APDU apdu) {
   if ( pin.isValidated() ) {
      // make the debit operation
   } else {
     ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
   }
}
```

With this approach, we provide a tool that processes an annotated class file. The annotations become a custom component containing security information. This is possible because the Java Card specification [22] allows adding custom components to a classfile: a virtual machine processes custom components if it knows how to use them or else, silently ignores them. But to process the information contained in these custom components, the virtual machine must be modified.

This approach allows to achieve a successful attack. With this mechanism, for an attacker to succeed, he must simultaneously inject two faults at the right time, one on the application code, the other on the system during its interpretation of the code (difficult to achieve) and outside the scope of the chosen fault model. Now we will detail the principle of the detection mechanism.

3.2 Principle of the "PATHCHECK" (PC) Method

The principle of the mechanism is divided in two parts: one part off-card and one part on-card. Our module works on the byte code, and has sufficient client computation power available because all the following transformations and computations are done on a server (off-card). It is a generic approach that is not dependent on the type of application. But it cannot be applied to native code such as cryptographic algorithm.

Off-card The first step is to create the control flow graph of the annotated method (in the case that it is an annotated class the operation is repeated for all the methods belonging to the class), by separating its code into basic blocks and by linking them. A basic block is a set of uninterrupted instructions. It is ended by any byte code instruction that can break the control flow of the program (i.e. conditional or unconditional branch instructions, multiple branch instructions, or instructions that raise an exception). If this operation is applied to the debit method, we obtained the basic blocks represented in figure 1. Once the method is divided into basic blocks, the second step is to compute its control flow graph; the basic blocks represent the vertices of the graph and directed edges in the graph denote a jump in the code between two basic blocks (c.f. figure 3). The third step is about computing for each vertex that compounds the control flow graph a list of paths from the beginning vertex. The computed path is encoded using the following convention:

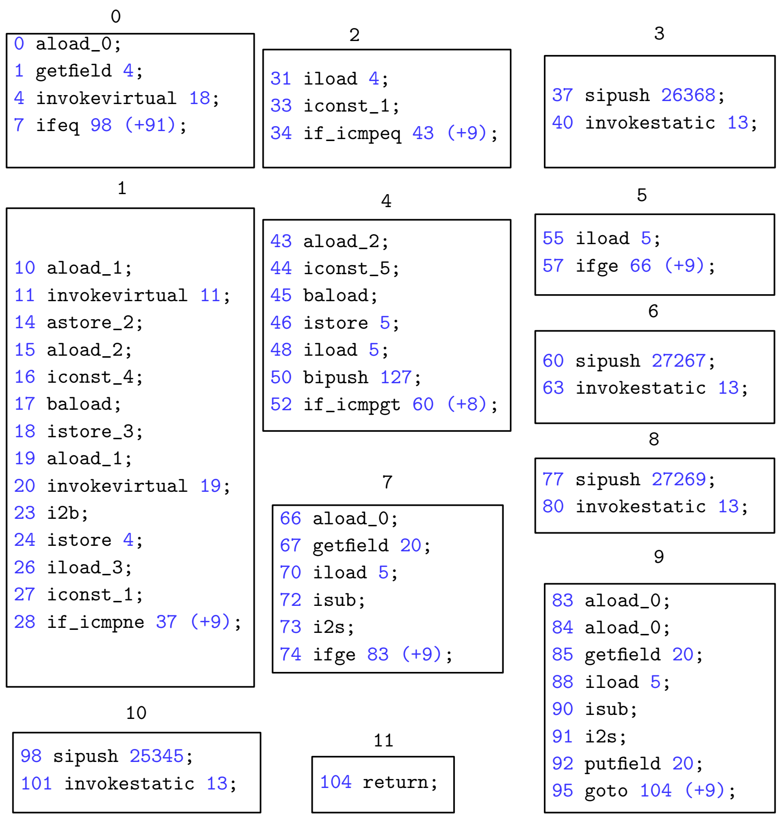
- Each path begins with the tag 01. This to avoid an attack that changes the first element of a path to 0x00 or to 0xFF.
- If the instruction that ends the current block is an unconditional or conditional branch instruction, a jump to the target of this instruction (represented by a low edge in 3) will use the tag 0 .
- If the execution continues at the instruction immediately following the final instruction of the current block (represented by a top edge in figure 3), then the tag 1 is used.

If the final instruction of the current basic block is a switch instruction, a particular tag is used, formed by any number of bits that are necessary to encode all the targets. For example, if we have four targets, we use three bits to code each branch (like in figure 2). Switch instructions are not so frequent in Java Card applications, but at least present in the ProcessApdu method. And to avoid a great increase of the application size that uses this countermeasure, they must be avoided.

Thus a path from the beginning to a given basic block is $X_0 ... X_n$ (where X corresponds to a 0 or to a 1 and n is the maximum number of bit necessary to code the path). In our example, to reach the basic block 9, which contains the update of the balance amount, the paths are : `01 0 0 0 0 0 0 1` and `01 0 0 0 0 0 1`.

On-card When interpreting the byte code of the method to protect, the virtual machine looks for the annotation and analyzes the type of security it has to use. In our case, it is the path check security mechanism. So during the code interpretation, it computes the execution path; for example, when it encounters a branch instruction, when jumping to the target of this instruction then it saves the tag “0”, and when jumping to the instruction that follows it saves the tag “1”. Then prior to the execution of a basic block, it checks that the followed path is an authorized path, i.e a path belonging to the list of paths computed for this basic block.

For the basic block 9 this should be one of the two previous paths; if this is not the case, then it is probably because the interpreter followed a wrong path to arrive there. If a loop is detected (backward jump) during the code interpretation, then the interpreter checks the path for the loop, the number of references and the number of value on the



```
0
0 aload_0;
1 getfield 4;
4 invokevirtual 18;
7 ifeq 98 (+91);
```

```
2
31 iload 4;
33 iconst_1;
34 if_icmpeq 43 (+9);
```

```
3
37 sipush 26368;
40 invokestatic 13;
```

```
1
10 aload_1;
11 invokevirtual 11;
14 astore_2;
15 aload_2;
16 iconst_4;
17 baload;
18 istore_3;
19 aload_1;
20 invokevirtual 19;
23 i2b;
24 istore 4;
26 iload_3;
27 iconst_1;
28 if_icmpne 37 (+9);
```

```
4
43 aload_2;
44 iconst_5;
45 baload;
46 istore 5;
48 iload 5;
50 bipush 127;
52 if_icmpgt 60 (+8);
```

```
5
55 iload 5;
57 ifge 66 (+9);
```

```
6
60 sipush 27267;
63 invokestatic 13;
```

```
7
66 aload_0;
67 getfield 20;
70 iload 5;
72 isub;
73 i2s;
74 ifge 83 (+9);
```

```
8
77 sipush 27269;
80 invokestatic 13;
```

```
9
83 aload_0;
84 aload_0;
85 getfield 20;
88 iload 5;
90 isub;
91 i2s;
92 putfield 20;
95 goto 104 (+9);
```

```
10
98 sipush 25345;
101 invokestatic 13;
```

```
11
104 return;
```

Fig. 1: Basic blocks of the debit method

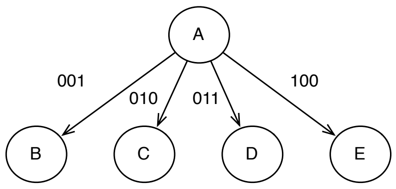


Fig. 2: Coding a switch instruction

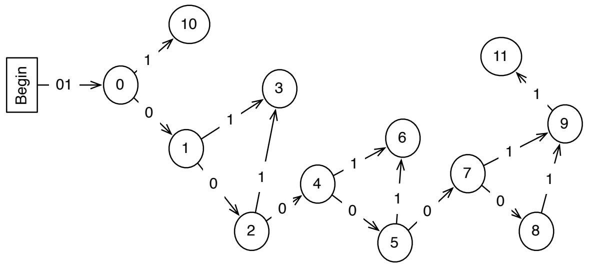


Fig. 3: Control flow graph of the debit method

operand stack before and after the loop, to be sure that for each round the path remains the same.

4 EXPERIMENTATION AND RESULTS

4.1 Evaluation Context

Two Java Card applets have been used for the evaluation. Those two cardlets are representative of the type of code that a MNO may want to add to their USIM Card. The first (AgentLocalisation) is oriented geolocalization services, this cardlet is able to detect when the handset (the device in which the USIM card is inserted) is entering or leaving a dedicated or a list of geographical dedicated cells (each cell is identified through a CellID value, which is stored on the USIM interface) and then sends a notification to a dedicated service (registred and identified in the cardlet). The second is more specialised to authentication services, the cardlet is able to provide a One Time Password (OTP) to the customer and/or an application in the handset. This OTP value is already shared and synchronized by the cardlet with a central server, which is able to check every collected OTP value by dedicated web services.

Evaluating Resources Consumption The first category of metrics is the memory footprint and the CPU overhead. They have been obtained using the SimpleRTJ Java virtual machine that targets highly restricted constraints device like smart cards. The hardware platform for the evaluation is a board which has similar hardware as the standard smart cards.

These metrics are very important for the industry because the size of the used memories directly impacts the production cost of the card. In fact, applications are stored in the EEPROM that is the most expensive component of the card.

The CPU overhead is also important because most of the time, when challenging the card for some computation a quick answer is needed. So when designing a countermeasure for smart cards, it is important to have these properties in mind. To obtain the metrics in table 4, the PATHCHECK countermeasure has been implemented on an embedded system that has similar properties as common smart card.

Evaluating Mutants Detection To evaluate the path check detection mechanism, we have developed an abstract Java Card virtual machine interpreter. This abstract interpreter is designed to follow a method call graph, and for each method of a given Java Card applet, it simulates a Java Card method's frame. A frame is a memory area allocated for the operand stack and the local variables of a given method.

The interpreter can also simulate an attack by modifying the method's byte array. This is important because it allows us to reproduce faults on demand. In addition to the abstract interpreter, we have developed a mutant generator. This tool can generate all the mutants corresponding to a given application according to the chosen fault model. To achieve this, for a given opcode, the mutant generator changes its value from 0x00 to 0xFF, and an abstract interpretation is performed for each of these values. If the abstract interpretation does not detect a modification then a mutant is created enabling us to regenerate the corresponding Java source file and to color the path that lead to this mutant.

The mutant generator has a number of modes of execution:

- *The Basic Mode (BM)*: the interpreter executes, without running any checks, the instruction push and pop elements on the operands stack and using local variables. With this configuration, instructions can use elements of other methods frame like using their operands stack or using their locals. When running this mode, no countermeasures are activated.
- *The Byte Code Verifier mode (BCV mode)*: the interpreter checks that no overflow or no underflow occurs, that the used locals are inside the current table of locals, and that when a jump occurs it takes place inside the method. They consist in some verifications done by the BCV.
- *The advanced mode*: the simple mode plus the ability to activate or to deactivate a given countermeasure like the developed ones: path checking mechanism (PC), field of bits mechanism (FB) see [19], or PS mechanism. PS is a detection mechanism that is not described in this paper and for which a patent is pending.

Graphical User Interface (GUI) We have created a GUI for the mutant generator, that allows a user to choose a specific application, to parameterize the vulnerability analysis, and to navigate through the generated mutant applications showing the mutation that has led to it. This allows a security officer to have all the useful information in a user friendly environment.

We also provide a web application that allows a registered user to upload an application binary file, and to choose between the different analysis mode seen previously.

4.2 Results

Resources Consumption Table 4 shows the metrics for resources consumption obtained by applying the detection mechanism to all the methods of our test applications. The increasing of the application size is variable, this is due to the number of paths that exist on a method. Even if the mechanism is close to 10 % increasing of application size and 8 % of CPU overhead, the developer can choose when to activate only for sensitive methods to preserve resources. This countermeasure needs small changes on the virtual machine interpreter if we refer to the 1 % increment. So, we can conclude that it is an affordable countermeasure.

Table 4: Ressources consumption

Countermeasures	CPU	EEPROM	RAM	ROM
Path check	+ 8 %	+10 %	<1 %	1%
Field of bits	+ 3 %	+ 3 %	<1 %	1%
Basic block	+ 5 %	+15%	<1%	1%

Mutant Detection and Latency Tables 5, 6 and 7 show the reduction of generated mutants in each mode of the mutant generator for three applications. The second line shows the number of mutants (1) generated in each mode of the mutant generator. The third line of those tables shows the latency (2). The obtained results show the efficiency of the developed countermeasures.

The latency is the number of instructions executed between the attack and the detection. In the basic mode, no latency is recorded because no detection is made. This value is also really important because if a latency is too high maybe instructions that modify persistent memory like: `putfield`, `putstatic` or an invoke instruction (`invokestatic`, `invokevirtual`, `invokespecial`, `invokeinterface`) can be executed. If a persistent object is modified then it is manipulated during all future sessions between the smart card and a server. So this value has to be as small as possible to reduce the chances of having instructions that can modify persistent memory.

Table 5: Wallet (simple class) - 470 Instructions

	BM	BCV	PS	FB	BB	PC
(1)	440	54	30	10	0	37
(2)	-	2,91	2,92	2,43	2,72	2,42

Path check fails to detect mutants whenever the fault that generates the mutant does not influence the control flow of the code. Otherwise, when a fault occurs that alters the control flow of the application then this countermeasure detects it. With this countermeasure it becomes impossible to bypass systems calls like cryptographic keys verification. And if some mutants remain, applicative countermeasures can be applied on demand to detect them.

Table 6: SfrOtp (simple class) - 4568 of instructions - 9136 attacks

	BM	BCV	PS	PC	FoB	BB
(1)	7960	94%	95%	86%	99%	100%
(2)	-	3,64	3,56	17.18	8.61	12

Table 7: AgentLocalisation (simple class) - 3504 instructions - 7008 attacks

	BM	BCV	PS	PC	FoB	BB
(1)	7960	94%	99%	88%	99%	100%
(2)	-	11.8	12.1	2.43	10.20	13

5 CONCLUSIONS

In this paper, we presented a new approach that is affordable for the card and that is fully compliant with the Java Card 2.x and 3.x specifications. Moreover it does not consume too much computation power and the produced binary files increases in an affordable way. It does not disturb the applet conception workflow, because we just add a module that will make a lightweight modification of the byte code. It saves time to the developer looking to produce secured applications thanks to the use of the sensitive annotation. Finally, it only requires that a slight modification be made to the Java Virtual Machine. It also has a good mutant applications detection capacity.

We have implemented all these countermeasures inside a smart card in order to have metrics concerning memory footprint and processor overhead, which are all affordable for smart cards. In this paper, we presented the second part of this characterization to evaluate the efficiency of countermeasures in a smart card operating system. We provide a framework to detect mutant applications according to a fault model and a memory model. This framework is able to provide to a security evaluator all the source code of the potential mutant of the application; he can decide if there is a threat with some mutants and then to implement a specific countermeasure. We applied this approach on SIM card applications but it could be applied to every Java Card application.

With this tool, both the developer and security evaluator can take informed decisions concerning the security of its smart card application. For the developer company, reducing the size of the embedded code minimizes the cost of the application. For the security evaluator it provides a semi automatic tool to perform vulnerability analysis.

References

1. M. Abadi, M.B.U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS'05: proceedings of the 12th ACM Conference on Computer and Communications Security: November 7-11, 2005, Alexandria, Virginia, USA*, page 340. Citeseer, 2005.
2. M.L. Akkar, L. Goubin, and O. Ly. Automatic integration of counter-measures against fault injection attacks. *Pre-print found at http://www. labri. fr/Perso/ly/index. htm*, 2003.

3. ANSSI. *Protection Profile (U)SIM Java Card Platform Protection Profile - Basic Configuration, ANSSI-CC-PP-2010/04 12/07/2010*. ANSSI, 2010.
4. ANSSI. *Protection Profile (U)SIM Java Card Platform Protection Profile - SCWS Configuration, ANSSI-CC-PP-2010/05, 12/07/2010*. ANSSI, 2010.
5. C. Aumuller, P. Bier, W. Fischer, P. Hofreiter, and J.P. Seifert. Fault attacks on rsa with crt: Concrete results and practical countermeasures. *Lecture Notes in Computer Science*, pages 260–275, 2003.
6. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, D.T. Ltd, and I. Rehovot. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.
7. G. Barbu, H. Thiebeauld, and V. Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. *Smart Card Research and Advanced Application, Cardis 2010*, LNCS 6035:148–163, April 2010.
8. J. Blomer, M. Otto, and J.P. Seifert. A new CRT-RSA algorithm secure against Bellcore attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 311–320. ACM New York, NY, USA, 2003.
9. G. Bouffard, J. Cartigny, and J.-L.. Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In *Cardis 2011*. Springer.
10. ETSI. *3GPP TS 31.102, Technical Specification Group Core Network and Terminals*. ETSI, 2005.
11. Ko Gadella. *Fault Attacks on Java Card (Masters Thesis)*. Master thesis, Universidade de Eindhoven, 2005.
12. GP. Global platform official site, 2010.
13. L. Hemme. A differential fault attack against early rounds of (triple-) DES. *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 170–217, 2004.
14. J. Iguchi-Cartigny and JL. Lanet. Developing a trojan applet in a smart card. *Journal in Computer Virology*, 6 Issue 4:343–351, 2010.
15. N. Oh, P.P. Shirvani, E.J. McCluskey, et al. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002.
16. G. Piret and J.J. Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. *Cryptographic Hardware and Embedded Systems-CHES 2003*, pages 77–88, 2003.
17. G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. Swift: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society Washington, DC, USA, 2005.
18. K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference*, page 209. Citeseer, 2002.
19. A.A. Sere, J. Iguchi-Cartigny, and J-L. Lanet. Automatic detection of fault attack and countermeasures. In *Proceedings of the 4th Workshop on Embedded Systems Security*, pages 1–7. ACM, 2009.
20. A.A. Sere, J. Iguchi-Cartigny, and J-L. Lanet. A path check detection mechanism for embedded systems. *Proceedings of SecTech 2010*, 6485:459–469, 2010.
21. S.P. Skorobogatov and R.J. Anderson. Optical fault induction attacks. *Lecture notes in computer science*, pages 2–12, 2003.
22. SunMicrosystems. *Java Card 3.0.1 Specification*. Sun Microsystems, 2009.
23. E. Vetillard and A. Ferrari. Combined Attacks and Countermeasures. *Smart Card Research and Advanced Application, Cardis 2010*, LNCS 6035:133–147, April 2010.
24. D. Wagner. Cryptanalysis of a provably secure crt-rsa algorithm. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 92–97. ACM New York, NY, USA, 2004.

A Full Java Code of the Debit method

```
Private void debit(APDU apdu) {
  // access authentication
  if ( pin.isValidated() ) {
    byte[] buffer = apdu.getBuffer();
    byte numBytes = (byte) ( buffer[ISO7816.OFFSET_LC]);
    byte byteRead = (byte) (apdu.setIncomingAndReceive());
    if ((numBytes != 1 ) || (byteRead != 1))
      ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    // get debit amount
    byte debitAmount = buffer[ISO7816.OFFSET_CDATA];
    // check debit amount
    if ( ( debitAmount > MAX_TRANSACTION_AMOUNT) || ( debitAmount < 0 ) )
      ISOException.throwIt(SW_INVALID_TRANSACTION_AMOUNT);
    // check the new balance
    if ( (short)( balance - debitAmount ) < (short)0 )
      ISOException.throwIt(SW_NEGATIVE_BALANCE);
    balance = (short) (balance - debitAmount);
    } else {
      ISOException.throwIt( SW_PIN_VERIFICATION_REQUIRED);
    }
}
```

B Full Byte Code Representation of the Debit Method

```
private void debit(APDU buffer) {
        0 aload_0;
        1 getfield 4;
        4 invokevirtual 18;
        7 ifeq 91;
        10 aload_1;
        11 invokevirtual 11;
        14 astore_2;
        15 aload_2;
        16 iconst_4;
        17 baload;
        18 istore_3;
        19 aload_1;
        20 invokevirtual 19;
        23 i2b;
        24 istore 4;
        26 iload_3;
        27 iconst_1;
        28 if_icmpne 9;
        31 iload 4;
        33 iconst_1;
        34 if_icmpeq 9;
        37 sipush 26368;
        40 invokestatic 13;
        43 aload_2;
        44 iconst_5;
        45 baload;
        46 istore 5;
        48 iload 5;
        50 bipush 127;
        52 if_icmpgt 8;
        55 iload 5;
        57 ifge 9;
        60 sipush 27267;
        63 invokestatic 13;
        66 aload_0;
        67 getfield 20;
        70 iload 5;
        72 isub;
        73 i2s;
        74 ifge 9;
        77 sipush 27269;
        80 invokestatic 13;
        83 aload_0;
        84 aload_0;
        85 getfield 20;
        88 iload 5;
        90 isub;
        91 i2s;
        92 putfield 20;
        95 goto 9;
        98 sipush 25345;
        101 invokestatic 13;
        104 return;
}
```