Evaluation of Countermeasures Against Fault Attacks on Smart Cards

Ahmadou A. SERE
SSD - XLIM Labs, University of Limoges JIDE, 83 rue Isle, Limoges, France ahmadou-al-khary.sere@xlim.fr

Julien Iguchi-Cartigny
SSD - XLIM Labs University of Limoges JIDE, 83 rue Isle, Limoges, France Julien.cartigny@unilim.fr

Jean-Louis Lanet
SSD - XLIM Labs, University of Limoges JIDE, 83 rue Isle, Limoges, France Jean-louis.lanet@unilim.fr

Abstract

Java Card are devices subject to either hardware and software attacks. Thus several countermeasures need to be embedded to avoid the effects of these attacks. Recently, the idea to combine logical attacks with a physical attack to bypass bytecode verification has emerged. For instance, correct and legitimate Java Card applications can be dynamically modified on-card using laser beam. Such applications become mutant applications, with a different behavior. This internal change could lead to bypass control and protection and thus should offer illegal access to secret data and operation inside the chip. In this paper, we propose a set of countermeasures that can be activated by the developer using the annotation mechanism. These countermeasures are efficient but also affordable for the smart card domain, as shown by the evaluation of the coverage and memory usage.

Keywords: *Smart Card, Java Card, Fault Attack, Control Flow Graph*

1. Introduction

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, crypto processor...) 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 second security barrier is the software. The embedded programs are usually designed neither for returning nor modifying sensitive information without guaranty that the operation is authorized.

Java Card is a kind of smart card that implements the standard Java Card 3.0 [10] in one of the two editions “Classic Edition” or “Connected Edition”. Such smart cards embed a virtual machine (called Java Card Virtual Machine or JCVM), which interpret application bytecodes 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 [7]. This protocol ensures that the owner of the code has the necessary credentials to perform the action.

Java Cards have shown improved robustness compared to native applications regarding many attacks. They are designed to resist to numerous attacks using both physical and logical techniques. In the category of hardware attacks, the most powerful attacks are hardware based and particularly fault attacks. A fault attack modifies part of memory content or signal on internal bus and lead to deviant behavior exploitable by an attacker (a comprehensive consequence of such attacks can be found in [9]). Although fault attacks have been mainly used in the literature from a cryptanalytic point of view [2, 8, 11], they can also be applied to the executable code embedded in the device. For instance, while choosing the exact byte of a program the attacker can bypass countermeasures or logical tests. We called *mutants* such modified application.

Designing efficient countermeasures against fault attacks is important for smart card manufacturers but also for application developers. For the manufacturers, countermeasures must have the lowest cost in term of memory and processor usage. These metrics can be obtained with an evaluation on a target [12]. For the application developers, they have to understand the ability of their applets to become mutants and potentially hostiles in case of fault attacks. Thus, the coverage (the number of mutants which can be potentially generated by fault attacks) and the detection latency (the 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 fault hypothesis and the different implemented countermeasures.

The rest of this paper is organized as follows: first, we introduce a brief state of the art of fault injection attacks (also called fault attacks) and existing countermeasures, then we define what is a mutant application and its impact on the card security. The section 4 is dedicated to the different countermeasures we have designed. Finally we present an evaluation of our solutions and conclude with the future works for our researches.

2. Fault Attacks

Faults can be induced into a chip using physical perturbations like: a power spike, the heat, a laser, a clock glitch, etc. These errors can generate different versions of a program (a mutant version) by changing some instructions, interpreting operands as instructions, branching to other (or invalid) labels and so on.

To prevent a fault attack to happen, we need to know what are its effects on smart cards.

References [5, 15] present taxonomy of fault models in detail. In our case, we choose a subset of these models. We choose the precise byte error model as the most realistic attack model. When an attacker physically injects energy in a memory cell to change its state and depending of the underlying technology, the memory physically takes the value 0x00 or 0xFF. If memory cells are encrypted the physical value becomes random (more precisely a value which depends of the data, its address, and the encryption key). Thus, we assume that an attacker can:

- Make a fault injection at a precise clock cycle (she can target any operation she wants),
- Only set a byte to 0x00 or to 0xFF (bsr for bit set or reset fault type), or change this byte to a random value which cannot been predicted (random fault type),
- Target any memory cells (precise memory cell of a variable or register).

3. Defining a Mutant Application

The mutant generation and detection is a new research field introduced simultaneously by [3, 14] using the concepts of combined attacks (we have already discussed mutant detection in [13]). To define a mutant application, we use an example on the following debit method that belongs to a wallet Java Card applet. Here, the user pin must be validated prior to the debit operation.

Table 1 presents the corresponding bytecode representation. An attacker wants to bypass the pin test. A fault on the cell containing the conditional test bytecode changes the ifeq instruction (byte 0x60) to a nop instruction (byte 0x00). The resulting Java code and bytecode are showed in Table 2.

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

Table 1 - Bytecode Representation Before Attack

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

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

Table 2 - Bytecode Representation After Attack

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

If the attack is successful the pin code check is bypassed, the debit operation is made. Then the exception is thrown but the attacker had already achieved his goal. This attack is an example of a dangerous mutant application: *"an application that has been modified by an attack in such a way that the result is correct for the virtual machine interpreter but that doesn't have the same behavior than the original application"*. In this paper, we present several countermeasures to detect when such modifications happen.

4. Software Countermeasures on Smart Card

The software countermeasures are introduced at different stages during the development process and aimed to strengthen the application code against fault injection attacks. Software countermeasures can be classified by their means:

– *Cryptographic countermeasures* aim to get better implementation of cryptographic algorithms like RSA (which is the most frequently used public key algorithm in smart cards), DES, and hash functions (MD5, SHA-1, etc).

– *Applicative countermeasures* target the application code and generally produce application with bigger size (because the code and the data structures for enforcing the security mechanism are embedded with the functional code of the application). Java is an interpreted language (using a virtual machine); therefore, it is slower to execute than a native language (like C or assembler). So this kind of countermeasures suffers of low execution time.

– *System countermeasures* aim to harden the system to offer a safe environment to the applications (using dual computation of sensitive data, execution counter, etc). 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 detect modification of data stored in the ROM.

– *Hybrid countermeasures* are at the crossroad between applicative and system countermeasures. The application programmer inserts data in the applications that are exploited later by the system to protect the application code against fault attacks. They provide a better balance between application size and the CPU overheads.

All previous categories (with the exception of the cryptographic countermeasures) use a generalist approach to detect the fault because they don't focus on a particular algorithm. The mechanisms proposed in this paper are hybrid countermeasures.

4.1. General Idea Behind the Proposed Countermeasures

The proposed solution uses Java annotations, which are used by the virtual machine interpreter. Then, the JVCM switches to a "secure mode" when it enters a function which has been annotated as sensitive. The following code fragment shows the use of an annotation on the debit method. The @SensitiveType annotation denotes that this method must be checked for integrity with the FoB mechanism.

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

We have developed a tool that processes an annotated classfile. Thus, the annotations become a custom component containing security information. This is possible in JavaCard because the Java Card specification [20] allows adding custom components to a classfile. In this case, a virtual machine processes custom components if it knows how to use them or else, silently ignores them. To process the information in these custom components the virtual machine must be modified but annotations used in Java code keeps its portability property (a JCVM which doesn't have support for countermeasure annotations can still execute the applet, albeit without the protection offers by our mechanisms).

This approach requires from the attacker to succeed to simultaneously inject two faults at the right time, one on the application code and the other on the system during its interpretation of the code, which is something hard to realize, and outside the scope of the chosen fault model (an attacker can only make one fault at a time).

4.2. First Detection Mechanism: The Field of Bit (FoB)

This countermeasure considers that if an attack modifies an opcode by another one, it is possible that operands of the previous opcode are inconsistent with the new one. More precisely we can obtain the following situations:

1. An increase of the number of operands for the instruction: when add (no operand) is replaced by icmpeq (one operand) for instance;
2. A reduction of the number of operands for the instruction: when aload (one operand) is replaced by athrow (no operand) for instance;
3. The same number of operands: when iload (one operand) is replaced by return (one operand) for instance.

4.2.1. Off-card: After compilation of Java file, a tool generates a field of bit representing the type of each element in the method's byte array. In this field, opcode is marked with an X (execute) and operands representing the data manipulated by this instruction are marked with an R (read). Table 3 shows the byte representation and the corresponding field of bit for the debit method.

Table 3 - Field of Bit

Byte	FoB
00 : 18	X
01 : 83 00 04	XRR
04 : 8B 00 23	XRR
07 : 60 00 3B	XRR
10 : … …	… …
59 : 13 63 01	XRR
63 : 8D 00 0D	XRR
66 : 7A	X

The resulting table is saved as a component of the classfile (as the Java Card specification allows the creation of custom component).

4.2.2. On-card: During the method's byte array interpretation, the JVCM interpreter has first to check the kind of security enforced. It checks the presence of the "FoB" annotation parameter. If it is present then it verifies the concordance between the instruction currently interpreted and the values stored in the table. For example, if we assume that we are in the same configuration as in Table 2. We can note that the field of bit has changed its value from Table 4 (before the attack) to Table 5 (after the attack).

Table 4 - Field of Bit Before Attack

FoB	X	X	R	R	X	R	R	X	R	R	…	X	R	R	X	R	R	X
PC	0	1	2	3	4	5	6	7	8	9	…	59	60	61	62	63	64	65

Table 5 - Field of Bit After Attack

FoB	X	X	R	R	X	R	R	X	X	X	…	X	R	R	X	R	R	X
PC	0	1	2	3	4	5	6	7	8	9	…	59	60	61	62	63	64	65

Thus, when the interpreter tries to execute the ifeq at the PC 7, it detects that at PC 8 the table is filled with an X instead of R. In other term, the byte's semantic has changed from readable to executable. Thus, the interpreter tests for each instruction if MethodByteArray[vmpc] is equal to FieldofBit[vmpc]. If it is different, the interpreter stops the execution because a fault has probably occurred.

This method has a drawback: if an instruction replaces another instruction that has the same number of operands then the replacement is not detected; this case is called an indistinguishable replacement. Fortunately, with the chosen fault model, we have estimated that the probability for this to happen in Java Card application is 10% (this percentage has been obtained by statistics over 10000 instructions in 15 different applets).

4.3. Second Detection Mechanism: The Basic Block (BB)

This protection mechanism uses the basic block notion and the application control flow notion (from graph theory) to check the code integrity. A basic block is a sequence of instructions with a single entry point and a single exit point. Thus execution of a basic block can start only at its entry point, and can only leave at its exit point. These points are called leaders. To compute the basic blocks, the main task is to find the set of leaders:

– The first instruction of the method and each first instruction of every handler of method is a leader.
– Each instruction that is the target of an unconditional branch (goto, jsr, ret) is a leader.
– Each instruction that is the target of a conditional branch (ifeq, iflt, ifne, ifgt, ifge, ifnull, ifnotnull, if_icmpeq, if_icmpne, if_icomplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq) is a leader.
– Each instruction target of a composed conditional branch (tableswitch or lookupswitch) is a leader.

Each instruction that immediately follows a conditional or unconditional branch, or a return instruction (ireturn, areturn and return), or a composed conditional branch instruction is a leader.

Each individual leader starts a basic block, consisting of all instructions up to the next leader or to the end of the method's byte array. For simplification of this paper we enclose each method invocation (invokevirtual, invokespecial, invokestatic, and invokeinterface) in a basic block of its own and we do not consider the instruction that can implicitly or explicitly throws an exception.

4.3.1. Off-card: The next step is to compute the checksum of each block using the XOR operation on all the bytes composing a basic block. This computation leads to a table containing for each basic block the following data:

– The PC of its beginning,

– The PC of its ending,
– The value of its checksum.

Then this table is stored in the classfile as a custom component and sent to the card. The interpreter has to be modified to use the previous data. For instance, if we refer to the Table 1, the debit method shows two basic blocks: from 00 to 09 with a checksum value of 0x61, and from 59 to 66 with a check sum value of 0x98.

4.3.2. On-card: During runtime, the virtual machine interpreter first checks the presence of the "BB" annotation parameter that corresponds to the current detection mechanism. If this tag is present, it computes the basic blocks on the fly and compares them with the ones stored in the custom components of the classfile, using the following conditions:

1. The program counter (PC) of the current instruction exists in the table as a beginning or an ending for a basic block.
2. If the first condition is verified then the JVCM checks that the current instruction is really a leader.
3. During the basic block processing, it computes step by step the checksum value until the end of the current basic block.
4. At the end of the current basic block, before resuming to the next instruction it checks that the checksum computed for the current block is equal to the value stored for this block. If they aren't, it's probably because of an attacks and the interpreter can take appropriate measures for instance stops the execution.

If a fault occurs (according to the chosen fault model) then the mechanism will detect it whatever the modification is because the XOR result will also change. If two bytes are modified by a fault the resulting XOR can be different whence the importance of the fault model: an attacker can only modify one byte at a time.

4.4 Third Detection Mechanism: The Path Checking (PC)

This mechanism works on the byte code where all transformations and computations are done on a server (off-card). It is a generalist approach that is not dependent of the type of application. But it cannot be applied to native code such as cryptographic algorithm.

4.4.1 Off-card: The first step is to create the control flow graph of the annotated method, by **splitting** its code into basic blocks (as defined in Section 4.3) and by linking them. Thus after basic block computation our tool computes its control flow graph where the basic blocks represent the vertices of the graph and directed edges denote a jump in the code between two basic blocks (Fig. 2).

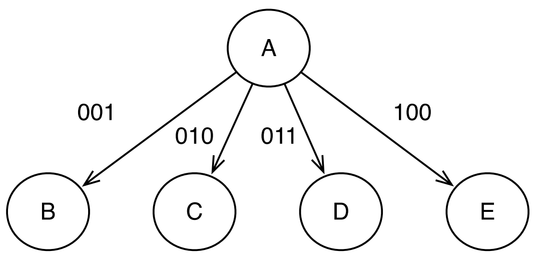

Fig. 1. Coding a Switch Instruction (a Tableswitch or a Lookupswitch) with 3 Branches

The third step is to compute for each vertex a list of paths from the beginning vertex. The computed paths are encoded using the following convention:

- Each path begins with the tag "01". This is 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, when jumping to the target of this instruction (represented by a low edge in Fig. 2), then the tag "0" is used.
- If the execution continues at the instruction that immediately follows the final instruction of the current block (represented by a top edge in Fig. 2), then the tag "1" is used.

If the final instruction of the current basic block is a switch instruction, a particular tag is used, composed by the number of bits necessary to encode all the targets. For example, if we have four targets, we use three bits to code each branch (like in Fig. 1). Switch instructions are not so frequent in Java Card applications. And to avoid a great increase of the application size that uses this countermeasure, they should be avoided.

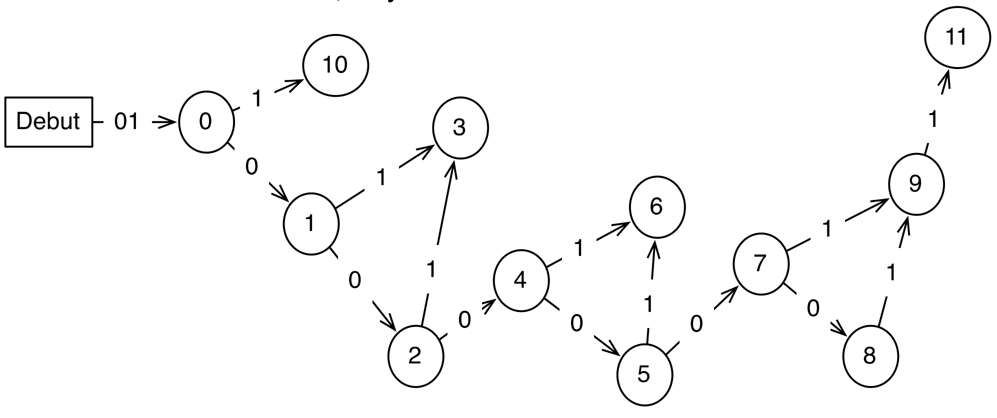

Fig. 2. Control Flow Graph of the Debit Method

Thus a path from the beginning to a basic block is X0...Xn (where X corresponds to a 0 or to a 1 and n is the maximum number of bits 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.

4.4.2. On-card: When interpreting the byte code of the method to protect, the virtual machine detects the annotation and analyzes the type of required security mechanism. In our case, it is the path check security mechanism. Then 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 that belongs to the list of path computed for this basic block. For the basic block 9, it is necessary one of the two previous paths, if not it is probably because to reach this point the interpreter has followed a wrong path; therefore, the card can lock itself.

In the case a loop is detected (backward jump) during the code interpretation, the interpreter checks the path for the loop, the number of references and the number of values on the operand stack before and after the loop, to be sure that for each round the path remains the same.

5. Evaluations

This section aims to evaluate two aspects of the detection mechanisms presented in this paper. The first aspect is their efficiency in detecting the fault attacks according to the chosen fault model. The second aspect is their efficiency in term of resources consumption, because smart cards have small memory footprints and computation power with a high challenge/response latency constraint. Thus, these two categories of metrics are needed to evaluate the adequacy of the countermeasures in a production environment.

5.1. Runtime Overhead

The first category of metrics is the memory footprint and the CPU overhead of our countermeasures. They have been obtained using the SimpleRTJ [1] a Java virtual machine that targets highly restricted constraints device like smart card. The hardware platform used for the evaluation is an AT91EB40A evaluation board with an ARM7TDMI microprocessor. The internal clock speed of this board if by default 32,768 MHZ, which is the same as in a high-end smart card. For the memory, it has 2Mb of flash and 256 Kb of RAM and can use a 16 or 32 bits memory addressing. These characteristics make this hardware as close as possible of common smart cards.

SimpleRTJ is a Java virtual machine designed for small memory footprints embedded devices based on 8, 16, or 32 bits microprocessor. It uses off-card library linking process like java card 2.x. It embeds the entire API to optimize execution and memory overhead. The source code for SimpleRTJ already exist for the evaluation board, so we have modified the code relating to the interpreter to integrate our countermeasures (data type for handling custom components and code fragments to detect modification caused by fault attacks).

Table 6 shows the metrics for resources consumption obtained by using the detection mechanisms on all the methods of our test applications. The increase of the application size is variable for the basic block detection mechanism and the path checking detection mechanism. This is due to the number of paths and to the number of basic blocks generated off card. Even if the path checking mechanism is close to 10 % and the basic block to 5 % of application size and respectively of 8 % and 5 % of CPU overhead, the developer can choose when to activate only for sensitive methods to preserve resources thanks to the annotation mechanism. The field of bits mechanism is very light in comparison of the two other mechanisms, and the memory overhead is constant because the data generated by this mechanism depend only on the number of bytes of an application. These countermeasures need small changes on the virtual machine interpreter. So we can conclude that they are affordable countermeasure for the aimed target.

Table 6 - Runtime Overhead

Countermeasures	EEPROM	ROM	CPU
FoB	+3 %	+1 %	+3 %
BB	+5 %	+1 %	+5 %
PC	+10 %	+1 %	+8 %

5.2. Detection Efficiency

To evaluate the detection mechanisms, 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 reproducing faults on demand. On top of 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 do that, the mutant generator changes each opcode value from 0x00 to 0xFF, and for each of these values an abstract interpretation is made. If the abstract interpretation does not detect a modification then a mutant is successfully created. We have also developed a program to reverse engineering mutant application and thus understand the corresponding Java source.

The mutant generator has different modes of execution:

- The basic mode: the interpreter executes the instruction pushing and popping element on the operands stack and uses local variables without check. In this configuration instructions can use elements of other methods frame like using their operand stacks or using their locals. When running this mode, it has no countermeasures activated.
- The simple mode: that correspond to a partial BCV (ByteCode Verifier) implementation. 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's done inside the method. They consist in some of the verifications done by the Java BCV.
- The advanced mode: it is the simple mode with the ability to activate or to deactivate a given countermeasures like the developed ones: path checking mechanism (PC), field of bits mechanism (FoB), or the basic block mechanism (BB).

5.2.1. Detection coverage: The Table 7 shows the reduction of generated mutants in each mode of the mutant generator for an application. The second line shows mutants generated in each mode of the mutant generator.

Table 7 – Non Detected Mutants by Countermeasures

Applications	Basic mode	Partial BCV	FoB	BB	PC
Wallet	440	434	18	0	37
Utilities	2740	2226	157	0	230
TeaApplet	1944	1916	95	0	163

The most efficient countermeasure is the basic block, which detects all the mutants that have been generated according to the fault model. Because this technique use a checksum value that is calculated with all the byte that compose a basic block.

The field of bit detection mechanism fails to detect mutants that have been generated by indistinguishable instruction replacement. For the evaluation purposes we have generated all the potential mutants, but otherwise we have seen that the indistinguishable replacements have a low probability to appear (see section 4.2.2).

The path checking fails to detect mutant when the fault that generates the mutant don't influence the control flow of the code. Otherwise, when a fault occurs that alter 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.

5.2.2. Latency: 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 if too high maybe instructions that modify persistent memory like: putfield, putstatic or an invoke instruction (invokestatic, invokevirtual, invokespecial, invokeinterface) can be executed. This can cause a modification of sensitive data stored in the smart card (like cryptogrtaphic keys, pin code, etc). So this value has to be as small as possible to lower the chances to have instructions that can modify persistent memory. The Table 8 shows the average latency obtained on the previous applications.

Table 8 - Average Latency by Countermeasures in Number of Instructions

Applications	Basic mode	Partial BCV	FoB	BB	PC
Wallet	-	2	2,42	2,72	3
Utilities	-	106,77	8,61	10,53	13,06
TeaApplet	-	56,86	5,82	7,66	11,72

We can see that the average latency of the developed countermeasures is lower than the Partial BCV countermeasures. Thus, they are more efficient than the common protection that we can find in a smart card.

6. Future Works: Compression Detection Mechanism

We are now working on a detection mechanism that is based on bytecode compression. The idea behind this technique comes from authors of [4,6]. The goal of this mechanism is to reduce the Java Card instruction set because the less instruction we have, the less are the probability for an attack to make a replacement that makes sense for the virtual machine interpreter (according to our fault model). In fact, the fault model induces that when an attack is a success, then the probability of a change to happen is 1/255. In the Java Card specification, there are 185 instructions, all of them are useful to represent a Java Card application, so the probability that a change makes sense for a Java Card interpreter is given by Equation 1. With this equation we can understand that the lower the size of the instruction set is, the lower the probability will be.

$$P = \frac{1}{255} \quad SizeOfInstructionSet = \frac{185}{255} \quad 0{,}72$$

Equation 1

To achieve this goal, we have noticed that in Java Card's application some opcode instructions have a high probability to come along. These instructions form what we call "patterns" A pattern is a group of n instructions that always come together in an application. From a pattern we generate a new instruction, for example, the pattern "sipush 26368, invokestatic 13" becomes the new instruction " sipushandinvokevirtual 26368 13". When all the patterns have been found we obtain a new instruction set that is smaller than the original instruction set.

The principle is theoretically to apply a compression algorithm to the Java Card binary file and to modify the virtual machine to integrate the new instruction set. This mechanism has some advantages first the binary will be smaller than regular one and according to [4] the execution time of the application can be reduced.

7. Conclusions

We had presented in this paper some new approach that are affordable for smart card and that are fully compliant with the Java Card 2.x and 3.x specification. Moreover, it does not have high computation constraint and the produced binary files are under a reasonable limit in term of size. It also does not disturb the applet conception workflow. It saves time to the developer who wants to produce secured applications thanks to the use of the sensitive annotations. Finally, it needs small modifications of the java virtual machine. It also has a good mutant detection capacity.

We have implemented all these countermeasures inside a smart card to have metrics concerning memory footprint and processor overhead, which are all affordable for smart card. In this paper, we presented the second part of this characterization to evaluate the efficiency of countermeasures in smart card operating system. We provide a framework to detect mutant applications according to a fault model and a memory model. This framework can provide to a security evaluator officer all the source code of the potential mutants of the application. She can decide if there is a threat with some mutants and then to implement a specific countermeasure. Within this tool, either the developer, or security evaluator officer can take adequate decision 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] RTJ Computing, 2010. url: http://www.rtjcom.com/main.php?p=home.

[2] C. Aumuller et al. “Fault attacks on RSA with CRT: Concrete results and practical countermeasures”. In: Lecture Notes in Computer Science (2003), pp. 260–275.

[3] G. Barbu, H. Thiebeauld, and V. Guerin. “Attacks on Java Card 3.0 Combining Fault and Logical Attacks”. In: Smart Card Research and Advanced Application, Cardis 2010 LNCS 6035 (2010), pp. 148–163.

[4] G. Bizzotto and G. Grimaud. “Practical Java Card bytecode compression”. In: Proceedings of RENPAR14/ASF/SYMPA. Citeseer. 2002.

[5] 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. ACM New York, NY, USA. 2003, pp. 311–320.

[6] L.R. Clausen et al. “Java bytecode compression for low-end embedded systems”. In: ACM Transactions on Programming Languages and Systems (TOPLAS) 22.3 (2000), p. 489.

[7] Global platform group. Global platform official site. 2010. url:http://www.globalplatform.org.

[8] L. Hemme. “A differential fault attack against early rounds of (triple) DES”. In: Cryptographic Hardware and Embedded Systems-CHES 2004 (2004), pp. 170–217.

[9] J. Iguchi-Cartigny and J.L. Lanet. “Developing a Trojan applets in a smart card”. In: Journal in Computer Virology (2009), pp. 1–9.

[10] Sun Mycrosystems. Java CardTM 3.0.1 Specification. Sun Microsystems. 2009.

[11] G. Piret and J.J. Quisquater. “A differential fault attack technique against SPN structures, with application to the AES and Khazad”. In: Cryptographic Hardware and Embedded Systems-CHES 2003 (2003), pp. 77–88.

[12] 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. ACM. 2009, pp. 1–7.

13] A.A. Sere, J. Iguchi-Cartigny, and J.L. Lanet. “Mutant applications in smart card”. In: Proceedings of CIS 2010 (2010).

[14] E. Vetillard and A. Ferrari. “Combined Attacks and Countermeasures”. In: Smart Card Research and Advanced Application, Cardis 2010 LNCS 6035 (2010), pp. 133–147.

[15] D. Wagner. “Cryptanalysis of a provably secure CRT-RSA algorithm”. In: Proceedings of the 11th ACM conference on Computer and communications security. ACM New York, NY, USA. 2004, pp. 92–97.