Checking the Paths to Identify Mutant Application on Embedded Systems

Ahmadou Al Khary Séré[1], Julien Iguchi-Cartigny[2], and Jean-Louis Lanet[2]

[1] XLIM Labs, Université de Limoges, JIDE, 83 rue Isle, Limoges, France
`ahmadou-al-khary.sere@xlim.fr`
[2] Université de Limoges, JIDE, 83 rue Isle, Limoges, France
{`julien.cartigny`,`jean-louis.lanet`}@unilim.fr

Abstract. The resistance of Java Card against attack is based on software and hardware countermeasures, and the ability of the Java platform to check the correct behaviour of Java code (by using bytecode verification for instance). Recently, the idea to combine logical attacks with a physical attack in order 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 control flow from the original expected behaviour. This internal change could lead to bypass control and protection and thus offer illegal access to secret data and operation inside the chip. This paper presents an evaluation of the ability of an application to become mutant and a new countermeasure based on the runtime check of the application control flow to detect the deviant mutations. . . .

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, 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.

Java Card is a kind of smart card that implements the standard Java Card 3.0 [7] in one of the two editions "Classic Edition" or "Connected Edition". Such smart card embeds a virtual machine, which interprets 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 [4]. 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 to 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 memory content or signal on internal bus and lead to deviant behaviour exploitable by an attacker. A comprehensive consequence of such attacks can be found in [6]. Although fault attacks have been mainly used in the literature from a cryptanalytic point of view see [1, 5, 8], they can be applied to every code layers embedded in a device. For instance, while choosing the exact byte of a program the attacker can bypass countermeasures or logical tests. We called such modified application mutant.

Designing efficient countermeasures against fault attacks is important for smart card manufacturers but also for application developers. For the manufacturers, they need countermeasures with the lowest cost in term of memory and processor usage. These metrics can be obtained with an evaluation on a target [9]. For the 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.

The rest of this paper is organized as follow: 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.Then, we present the experimentation and the results, and finally we conclude with the perspectives.

2 Fault Attacks

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.

To prevent a fault attack to happen, we need to know what its effects on the smart card are. References [3, 13] has already discussed about fault model in detail.

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 have decided to choose the *precise byte error* that is the most realistic fault model. Thus, we assume that attacker can:

– make a fault injection at a precise clock cycle (she can target any operation she wants),

– only set or reset a byte to 0x00 or to 0xFF up to the underlying technology (bsr fault type), or she can change this byte to a random value out of his control (random fault type),
– target any memory cell she desires (she can target a precise variable or register).

3 Defining a Mutant Application

The mutant generation and detection is a new research field introduced simultaneously by [2, 12] using the concepts of combined attacks, and we have already discussed about mutant detection in [11]. 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 must be validated prior to the debit operation.

```
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
`00 : 18`	`00 : aload_0`
`01 : 83 00 04`	`01 : getfield #4`
`04 : 8B 00 23`	`04 : invokevirtual #18`
`07 : 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 1 resides the corresponding bytecode representation. An attacker wants to bypass the pin test. She injects a fault on the cell containing the conditional test bytecode. Thus the `ifeq` instruction (byte 0x60) changes to a `nop` instruction (byte 0x00). The obtained Java code follows with its bytecode representation in Table 2.

```
private void debit(APDU apdu) {
     // make the debit operation
```

```
      ISOException.throwIt (SW_PIN_VERIFICATION_REQUIRED);
}
```

Table 2. Bytecode representation after attack

Byte	Bytecode
`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 had already achieved his goal. This is a well example of dangerous mutant application: "*an application that has been modified by an attack that is correct for the virtual machine interpreter but that doesn't have the same behavior than the original application*". This attack has modified the control flow of the application and the goal of the countermeasure developed in this paper is to detect when such modification happen.

4 A Novel Approach to Path Check During Application Runtime

We have already proposed several solutions to check code integrity during execution in our previous publications [9, 10]. This paper is about the control flow integrity. Thus this section discusses existing countermeasures which protect the control flow integrity.

4.1 Using Java Annotation

The proposed solution uses Java annotations, when the virtual machine interpreter encounters an annotation it switches to a "secure mode". The fragment of code that follows, shows the use of an annotation on the debit method. The `@SensitiveType` annotation denotes that this method must be checked for integrity with the `PATHCHECK` mechanism.

```
@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 process an annotated classfile. The annotations become a custom component containing security information. This is possible because the Java Card specification [20] 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 that to achieve a successful attack, an attacker needs to simultaneously inject two faults at the right time, one on the application code, 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. Now we expose the principle of the detection mechanism.

4.2 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 it has at its disposal sufficient computation power because all the following 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.

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 method 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.

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. Fig. 2).

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, 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, 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 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 must be avoided. Thus a path from the beginning to a

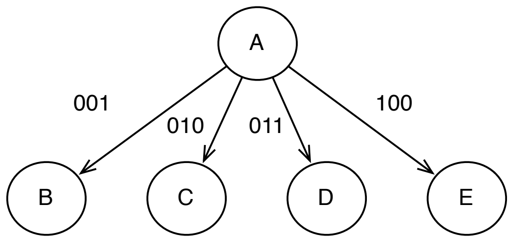

Fig. 1. Coding a switch instruction

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 that belong 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 arrive here the interpreter has followed a wrong path; therefore, the card can lock itself.

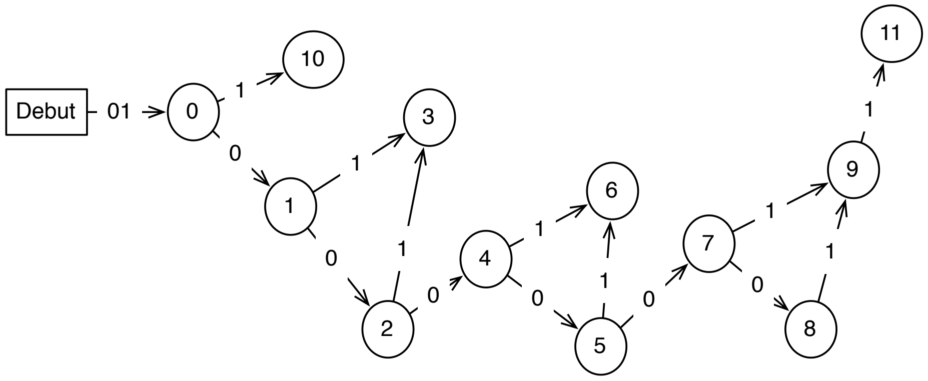

Fig. 2. Control flow graph of the debit method

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

5 Experimentation and Results

5.1 Resources Consumption

Table 3 shows the metrics for resources consumption obtained by activating the detection mechanism on all the method of our test applications. The increasing of the application size is variable, this is due to the number of paths presents 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 % of increasing. So we can conclude that it is an affordable countermeasure.

Table 3. Ressources consumption

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

5.2 Mutant Detection and Latency

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 to reproduce 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 realize this, for a given opcode, the mutant generator changes its 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 created enabling us to regenerate the corresponding Java source file and to color the path that lead to this mutant.

The mutant generator has different mode of execution:

– *The basic mode*: the interpreter executes the instruction pushing and popping element on the operands stack and using local variables without check. In this configuration instructions can use elements of other methods frame like using their operands stack or using their locals. When running this mode, it has no countermeasures activated.
– *The simple 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's done inside the method. They consist in some verifications done by the Java verifier.
– *The advanced mode*: 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 (FB) see [9], or PS mechanism. PS is a detection mechanism that is not described in this paper and for which a patent is pending.

The Table 4 shows the reduction of generated mutants in each mode of the mutant generator for an application. The second line shows the number of mutant generated in each mode of the mutant generator. The third line of those tables shows the latency.

The latency is the number of instruction 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. If a persistent object is modified then it is manipulated during all the future session between the smart card and a server. So this value has to be as small as possible to lower the chances to have instructions that can modify persistent memory.

Table 4. Wallet (simple class) - 470 Instructions

	Basic mode	Simple mode	PC	FB	PS
Number of mutants	440	54	23	10	30
Latency	-	2,91	3,33	2,43	2,92

Path check fails to detect mutant when the fault that generate 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. And if it remains some mutant, applicative countermeasures can be applied on demand to detect them.

6 Conclusion and Future Works

We had presented in this paper, a new approach that is affordable for the card and that is fully compliant with the Java Card 2.x and 3.x specification. Moreover it does not consume too much computation power and the produced binary files are under a reasonable limit in term of size. It does not disturb the applet conception workflow, because we just add a module that will makes lightweight modification of the byte code. It saves time to the developer who wants to produce secured applications thanks to the use of the sensitive annotation. Finally, it needs a tiny modification of 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 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 is able to provide to a security evaluator officer all the source code of the potential mutant 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 is able to 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] 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.

[2] 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.

[3] 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.

[4] Global platform group. *Global platform official site.* 2010. URL: `http://www.globalplatform.org`.

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

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

[7] Sun Mycrosystems. *Java CardTM 3.0.1 Specification.* Sun Microsystems. 2009.

[8] 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.

[9] 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.

[10] A.A. Sere, J. Iguchi-Cartigny, and J.L. Lanet. "Checking the Path to Identify Control Flow Modification". In: *PAca Security Trends In embedded Systems* (2010).

[11] A.A. Sere, J. Iguchi-Cartigny, and J.L. Lanet. "Mutant applications in smart card". In: *Proceedings of CIS 2010* (2010).

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

[13] 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.