# Type classification against *Fault Enabled Virus* in Java based Smart Card

Anonymous submission
Institute : XXX
Email: XXX

*Abstract*—**Smart card are often the target of software or hardware attacks. The most recent attack is based on fault injection which modifies the behavior of the application. We propose a new counter measure implemented in the Java Virtual Machine and a framework to deploy applications. This countermeasure implies a modification of the original program while accepting legacy application; Of course applications that use this mechanism will be less prone to mutant generation. In a second part we evaluate the ability of the platform to resist to *Fault Enabled Viruses*.**

*Keywords*-**smart card, viruses, fault, counter measures**

## I. 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, *etc.*) 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.

Smart cards are devices prone to attacks in order to gain access to services or assets stored by the card. Several means have been used to retrieve these valuable information and recently fault injection appears to be the most efficient. Thus smart card manufacturers try to design countermeasures to embed in their operating system to prevent such attacks. Often solutions are based on dedicated code at the applicative level. We propose in this paper a new counter measure that allows a form of dynamic type checking without to pay the cost of the type inference.

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 (VM) of the type of the Java element and a framework to adapt the java byte code to this 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.

## II. *FAULT ENABLED VIRUSES*

Faults can be induced into the chip by using perturbations in its execution environment [1]. The faults are induced by some physical attack, *i.e.*, some physical setup, which exposes the device to some sort of physical stress. As a reaction, the device malfunctions, *i.e.*, memory cells change their current, bus lines transmit different signals or structural elements are damaged. 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. 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.

### *A. Fault Attacks*

In this subsection, we will give an overview over actual physical methods to induce faults. This will show that there are numerous ways to induce faults into physical devices. Fault attack is an old research field. Research in avionics or space travel brought to the fore that cosmic rays can flip single bits in the memory of an electronic device. Such faults are still an issue until now for such devices. In the smart card field, researches focused on power spikes, clock glitches and optical attacks. A smart card is a portable device without any own power supply neither clock and thus requires a smart card reader providing power and clock in order to work. The reader can be replaced by an adversary with laboratory equipment, able of tampering with the power supply. With short variations of the power supply, which are called spikes, one can use it to induce errors into the computation of the smart card. Spikes allow to induce both memory faults but also faults in the execution of a program. This latter which aim at confusing the program counter, can cause conditionals to work improperly, loop counters to be decreased and arbitrary instructions to be executed. The reader may provide the card with a clock signal, which incorporates short deviations from the standard signal, which are beyond the required tolerance bounds. Such signals are called glitches. Glitches can be defined by a range of different parameters and they can be used to both induce memory faults as well to cause a faulty execution behavior. Hence, the possible effects are the same as for spike attacks. If the chip is unpacked, such that the silicon layer is visible, it is possible to use a laser to induce perturbation in the memory cells. These memory cells, *i.e.*, EEPROM memory and semiconductor transistors, have been found to be sensitive to light. This happens if the photon energy of the applied light is transformed in electron in the semiconductor. Modern green or red lasers can be focused on relatively small regions of a chip, such that faults can be targeted fairly well. The last method use changes in the external electrical field and has been considered as a possible method for inducing faults into smart cards. Here, faults are sought to be induced by placing the device in an electromagnetic field, which may influence the transistors and memory cells.

### B. 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 [2], [5]. We describe, in the table I, 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 [4] 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[1] 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).

We have defined the hypothesis concerning the attacker, then we present the countermeasures embedded in most modern smart cards in order to detect the induced fault.

[1]bit set or reset

Table I: 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 | – |

### C. Known Countermeasures

Smart card manufacturers have been aware of the danger of faults for long time now, hence, they have developed a large variety of hardware countermeasures [3]. Major hardware countermeasures are sensors and filters, which aim to detect attacks, *e.g.*, using anomalous frequency detectors, anomalous voltage detectors, or light detectors. Other countermeasures are to use redundancy, *i.e.*, dual-rail logic, where memory is doubled, doubled hardware, capable of computing a result twice in parallel. If two results are computed, they are considered to be error-free if both values match. This is a very expensive countermeasure, and hence, it is not often implemented in smart cards. Using only hardware countermeasures has two drawbacks. Highly reliable countermeasures are very expensive and low cost countermeasures only detect specific attacks. Since new fault attacks are being developed frequently these days, detecting only currently known forms of physical tampering is not sufficient and for long term applications (an e-passport must be valid for 10 years) it is definitely not sufficient.

Software countermeasures are introduced at different stages of the development process; their purpose is to strengthen the application code against fault injection attacks. Current approaches for software countermeasures include checksums, randomization, masking, variable redundancy, and counters. 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. We have presented in **[XXX]** some countermeasures to avoid the effect of faults and we evaluated their costs, in term of memory and CPU usage, but also their latency and their coverage.

## D. Impact of the fault: the mutant code

The following code is extracted from a dump of a Java Card memory. The method ends by throwing the Java Card exception to PIN verification (code `0x6301`) and the jump at line 7404 throws this exception. If a fault is injected at this line, the transformed code will probably never throw the exception.

Listing 1: Disassembling dumped memory

```
73F6  :  18         aload_0
73F7  :  7B 20 14   getstatic_a     0x2014
73FA  :  8B 02 08   invokevirtual   0x0208
73FD  :  32         sstore_3
73FE  :  1A         aload_2
73FF  :  03         sconst_0
7400  :  1F         sload_3
7401  :  8D 09 75   invokestatic    0x0975
7404  :  60 2B      ifeq            0x2B
7406  :  04         sconst_1
...
742F  :  11 63 01   sspush          25345
7432  :  8D 54 0D   invokestatic    0x540D
7435  :  7A         return
```

One can remark that after the execution of the instruction `ifeq`, the operand stack is empty. Now consider that a laser hits the memory cells that contains the `0x60` code, *i.e.*, `ifeq` the resulting mutant is the following:

Listing 2: Mutant code

```
...
7401  :  8D 09 75   invokestatic    0x0975
7404  :  00
7405  :  2B         astore_0
7406  :  04         sconst_1
...
742F  :  11 63 01   sspush          25345
7432  :  8D 54 0D   invokestatic    0x540D
7435  :  7A         return
```

After executing the `astore_0` instruction, the stack is empty and the mutant program is synchronized with the original program. A countermeasure based on the stack under or overflow will never detect the mutant. If a dynamic type verification had occur this mutant code should have been detected. In the original code the type system should evolve as describe in table II. After executing the first instruction a reference is pushed on top of the stack. The second instruction pushes a value while the third consumes a reference and a value and pushes a value after execution.

Table II: Type evolution

| Address | code | Mnemo | Stack after |
|---|---|---|---|
| 73F6 : | 18 | aload_0 | [ref] |
| 73F7 : | 7B 20 14 | getstatic_a | [ref, val] |
| 73FA : | 8B 02 08 | invokevirtual | [val] |
| 73FD : | 32 | sstore_3 | [] |
| 73FE : | 1A | aload_2 | [ref] |
| 73FF : | 03 | sconst_0 | [ref, val] |
| 7400 : | 1F | sload 3 | [ref, val, val] |
| 7401 : | 8D 09 75 | invokestatic | [val] |
| 7404 : | 60 2B | ifeq 0x2B | [] |
| 7406 : | 04 | sconst_1 | [val] |
| ... | ... | ... | ... |
| 742F : | 11 63 01 | sspush | |
| 7432 : | 8D 54 0D | invokestatic | |
| 7435 : | 7A | return | |

Table III: Type evolution of the mutant code

| Address | code | Mnemo | Stack after |
|---|---|---|---|
| ... | ... | ... | ... |
| 73FF : | 03 | sconst_0 | [ref, val] |
| 7400 : | 1F | sload_3 | [ref, val, val] |
| 7401 : | 8D 09 75 | invokestatic | [val] |
| 7404 : | 00 | nop | [val] |
| 7405 : | 2B | astore_0 | [] |
| 7406 : | 04 | sconst_1 | [val] |
| ... | ... | ... | ... |
| 742F : | 11 63 01 | sspush | |
| 7432 : | 8D 54 0D | invokestatic | |
| 7435 : | 7A | return | |

Now examine the state of the stack with the mutant code. The instruction `ifeq` of the original code consumes a value and the `sconst_1` pushes a value. In the mutant code, the `ifeq` is replaced by a `nop` which does not modify the state of the stack. The `astore_0` pop a reference from the stack, but cannot be executed because a value is on top of the stack. It becomes obvious to see how dynamic type verification should increase the possibility to detect mutants.

## III. THE TYPE CLASSIFICATION

As we have seen, the most obvious countermeasures are related with under or overflow of the stack but their coverage is low: a lot of mutant can bypass these controls. The dynamic type verification is probably one of the most efficient countermeasure against mutant. It has to verify that the content on top of the stack is of the exact type expected by the next instruction. To obtain a dynamic type verification, the virtual machine needs to infer dynamically the type of locals and type of each element on top of the stack. This is known to be costly in term of computation and memory space. In fact the virtual machine must keep the stack evolution in term of type, which means to have a second stack where the type of the content of the stack are stored. After executing an instruction, the virtual machine must evaluate the type stack with regard to the executed instruction. Such a mechanism is not embeddable into a resource constrained device like a smart card. Hereafter we propose a simpler mechanism with no run time cost and only the cost of one pointer in memory.

### A. *Principle*

The principle is to implement a mechanism to process in a different way references and values. If the operand stack is separated into two zones, one reserved for values and one for references, then we immediately obtain a dynamic type checking. These two areas fill the same memory space reserved for a normal stack. What changes with the dual stack is just the place where you will find items.

Here is an example showing you how the dual stack works compared to a normal stack. Just imagine a program which push on the stack one value then two references. To begin we push a value :

Table IV: Dual stack 1

| **Normal stack** | **Dual stack** |
|---|---|
| | |
| | |
| | |
| value | value |

Then we push the first reference :

Table V: Dual stack 2

| **Normal stack** | **Dual stack** |
|---|---|
| | reference 1 |
| | |
| reference 1 | |
| value | value |

And the last reference is pushed :

Table VI: Dual stack 3

| **Normal stack** | **Dual stack** |
|---|---|
| | reference 1 |
| reference 2 | reference 2 |
| reference 1 | |
| value | value |

You can see with the dual stack that there are two zones, one at the bottom for the values and the other one at the top for the references. The normal stack has one pointer called top of stack, but for the dual stack we need of two pointers, one to indicate the top of the values and one for the references.

For such a system works requires that implementations of instructions in the VM know get elements in the right part of the stack operands. Most of Java instructions are typed, so you can easily implement these instructions, knowing the types of elements one instruction will push or pop on the stack. However, there are some instructions untyped and these instructions will cause problems for the implementation of the VM because it is unclear whether they will treat references or values. These instructions are:

- `pop`, `pop2`,
- `dup`, `dup2`, `dup_x`,
- `swap_x`.

It will be understood why these instructions are not typed. For example with a simple stack, for a dup instruction which duplicates the last element stacked, the VM does not need to know if it duplicates a value or reference, it must just go for the element pointed by the top of stack. However, with a dual-stack, the VM must know the type of the last stacked item to see if it will get the element to duplicate from the top of stack of the values or the top of stack of the references. So, with the dual stack, the VM can not process these untyped instructions.

### B. *Modifying the Virtual Machine*

This idea requires a new implementation of the Virtual Machine to split the Java Card stack in two parts during the application execution. As we said before, there is also a problem with untyped instructions. Currently, We have not written this new implementation of the Virtual Machine but we found a solution for these untyped instructions, by removing such instructions from the initial program code.

*1) Program Transformation:* Untyped instructions are an issue. Although these instructions are rare in a Java Card program, we must be able to process these instructions properly. This requires transforming the original program code so that the VM can run the program without errors. One solution would be to transform the program to execute by replacing untyped instructions by one or more other instructions which lead to the same result. These replacement instructions would use temporary variables to properly perform the treatment.

To do this transformation we just need to analyse an modify each methods, one after the other, because the method stack is local. Before we can replace untyped instructions we need of the stack history. With this information, we will be able to substitute untyped instructions. For instance if you want to replace a `pop`, you have to know the type of the last element pushed on the stack; so if it is a reference you just replace the `pop` by a `astore` into a local variable and if it is a value, you replace by a sstore instruction. To have this historic, we analyse the byte code, instruction after instruction, and as we know exactly for each instructions what changes are made on the stack, we just make a simulation of the stack.

The analysis of byte code is completely linear, just read the instructions one after the other. However jumps complicate the analysis. Indeed, the first approach is to say when We find a jump, We go to the location pointed by the jump and We continue the analysis. However it is unnecessary to go twice the same instruction, and with the jumps we can even enter into an infinite loop. That's why the analyzer performs checks to verify if an instruction has been executed and if it has, the analyser stops. There is another issue: conditional jumps. If the condition is true, then the analysis must continue to the instruction pointed to by the jump, and if it is false, the analysis must continue ignoring the jump. So the analysis must explore two branches: the analyzer must launch two sub-analyzes by making a recursive call. Each of these analyzes must be run with an identical stack, one obtained just before the conditional jump.

While the analysis, if the program transformer encounter an untyped instruction, it knows the stack state and it can insert the right instructions (`astore`, `aload`, `sstore`, `sload`) in the appliction code to simulate a pop, dup or swap instruction.

*2) CapMap Integration:* The CapMap is a Java-framework which provides an easy way to parse and modify the CAP file. The CAP file is the file sent to the Java Card as a lightweight Java Class file.

This Java-library helps us to analyse the execution flow of the current Java Card applet. For each instruction, you can measure its impact on the stack (with the knowledge of the previoulsy pushed type and value) in order to dynamically modify the CAP file, and may update each CAP file component to create an well-formed file. Indeed, this tool is used to test card against logical attack.

In your case, the CapMap parses each CAP file to protect and, for each applet method, verify if there is untyped operations on the stack. If there are some sensitive operation, the CapMap modify this instructions like explain in the previously part. This step is explained in the figure **??**.

Figure 1: CapMap integration

With this integration, we provide an all-in-one tool to protect each Java Card applet.

## IV. EXPERIMENTATION AND RESULTS

A countermeasure is affordable if its latency (the number of instructions executed between the fault and the detection) is low, its coverage is high (the ability to detect mutant code) and its footprint is low. These three points are the most important when designing a countermeasure for a smart card. This later can be split into RAM and ROM usage knowing that the scarcest resource is the RAM. This metrics needs to implement it into our own prototype while the others metrics can be obtained thought a fault simulator.

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 (registered and identified in the cardlet). The second is more specialized 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. The SfrOtp application has 4568 instructions and the AgentLocalisation 3504 instructions.

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 replace an untyped instruction, the program transformer creates local variables which allow to push or pop elements on the stack, and it inserts new instructions to simulate the same effect than the untyped instruction. The metrics give us the occurrences of these instructions: `pop` (2%), `dup` (3%), `dup2` (<1%), and the others are extremely rare. As occurrences of these instructions are low in an Java Card application, there is not many changes to do. If we want remove one of these three instructions it does not cost much, to replace a `pop`, we just need a new local variable, to replace a `dup`, we need to insert an instruction and a new local variable and for a `dup2` instruction we insert three instructions and two variables. Moreover we could optimize local variables, taking those that are not used.

The metric related to detection coverage and latency on the applications show that 95% of the mutants have been detected on the SfrOtp application while on AgentLocalisation the detection rate is 99%. On the first one the latency is around 3.5 instruction while in the second the latency reaches 12 instructions. Previous studies have shown that Basic Block countermeasure had a latency between 12 and 13 which is very closed.

## V. CONCLUSIONS

In this paper, we presented a new approach affordable for the card and that fully backward compatible with the available platforms. This could provide a competitive advantage to a platform that implements this countermeasure. An application executed on a regular platform it will be more prone to fault attack than if the platform embeds this countermeasure. We have seen that the cost in term of memory footprint was negligible while its detection capacity was important. The approach do not have any impact on the applicative development. The application needs a prepossessing which does not increase the size of the application.

## REFERENCES

[1] 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.

[2] 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.

[3] Ko Gadella. *Fault Attacks on Java Card (Masters Thesis)*. Master thesis, Universidade de Eindhoven, 2005.

[4] S.P. Skorobogatov and R.J. Anderson. Optical fault induction attacks. *Lecture notes in computer science*, pages 2–12, 2003.

[5] 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.