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

Jean Dubreuil Guillaume Bouffard Jean-Louis Lanet Julien Cartigny
Secure Smart Devices (SSD) Team - XLIM Labs, Université de Limoges,
123 Avenue Albert Thomas - 87060 Limoges, France
Email: jean.dubreuil@etu.unilim.fr, guillaume.bouffard@xlim.fr, jean-louis.lanet@xlim.fr, julien.cartigny@xlim.fr

***Abstract*—Smart card are often the target of software or hardware attacks. For instance the most recent attacks are based on fault injection which can modify the behavior of applications loaded in the card, changing them as mutant application. In this paper, we propose a new protection mechanism which makes application to be less prone to mutant generation. This countermeasure requires a transformation of the original program byte codes which remains semantically equivalent. It requires a modification of the Java Virtual Machine which remains backward compatible and a dedicated framework to deploy the applications. Hence, our proposition improves the ability of the platform to resist to *Fault Enabled Mutant*.**

***Keywords*-smart card; Java Card; viruses; fault; countermeasures**

## 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 of the 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 which can store data in a secured manner and ensure security during data transactions. Its safety relies first on the underlying hardware. To resist probing of 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 to transfer nor to modify sensitive information without guaranty that the operation is authorized.

Smart cards are devices prone to attacks in order to gain access to services or datas stored into the card. Several methods 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 their operating system against such attacks. Usually most of the solutions are based on dedicated code at the applicative level. We propose in this paper a new countermeasure that allows a form of dynamic type checking without the high cost of type inference.

The contribution of this paper with respect to our prior work is based on two mechanisms: 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, the first section provides a brief state of the art on fault injection attacks and the existing countermeasures. In the second section outlines the impact of a fault and the mutant generation. Third section introduces the developed countermeasure. The evaluation framework and the collected metrics are highlighted in section four and finally, section five concludes the work with further perspectives.

## II. *FAULT ENABLED MUTANT*

Faults can be induced into the chip by using perturbations in its execution environment [1]. The faults are induced by some physical attacks which expose the device to some sort of physical stress. As a result the device has eratic functionioning, *i.e.*, current in memory cells change , bus lines transmit different signals or structural elements are damaged. Thus, 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. Fault attack is an old research field mainly in avionics or space travel. Researchers 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 three kind of fault attacks: power spikes, clock glitches and optical attacks.

### A. Fault Attacks

A smart card is a portable device without embedded power supply or clock and thus requires a smart card reader (which provides external power and clock sources) in order to work. The reader can also be replaced by an attacker with specific

equipments in the laboratory. With short variations of the power supply, which are called spikes, one can induce errors into the smart card internal operations. Spikes allow to induce memory faults but also faults in the execution of a program. Latter aims at confusing the program counter, can cause conditional checks to work improperly, loop counters to be decreased and arbitrary instructions to be executed.

The reader provides the card with a clock signal, which may incorporate short deviations beyond the required tolerance from the standard signal bounds. Such signals are called glitches. They can be defined by a range of different parameters and can be used to induce memory faults as well as 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 occurs due to photoelectric effect.Modern green or red lasers can be focused on relatively small regions of a chip, such that faults can be targeted fairly well. Another method is to make changes in the external electrical field of the smart card and has been considered as a possible method for inducing faults.

### 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], [3]. 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 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).

[1]bit set or reset

### C. Known Countermeasures

Smart card manufacturers have been aware of the danger of faults attacks for long time now. Hence, they have developed a large variety of hardware countermeasures [5]. 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 use redundancy, *i.e.*, dual-rail logic (keeping data in two redundant memories), and doubled hardware (computing a result twice in parallel). A data is considered to be error-free if both values (computed or memorized) match. But these are very expensive countermeasures, and hence, redundancy is not often implemented in smart cards.

We can notice that 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, especially for long term applications (an e-passport must be valid for 10 years).

An alternative or additional countermeasures are the use of software countermeasures. They 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, temporal 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, besides 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, 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. But we need to evaluate 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 of inserting data in the application that are used later by the system to protect the application code against fault

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 | – |

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 [6] 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 an attacked 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 occurred, 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 mutants 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 the type of each element on the top of the stack. But this is known to be costly in term of computation and memory space because 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 embedded into a resource constrained device like a smart card. Hereafter we propose a simpler mechanism for type classification with no run time cost and only the cost of one pointer in memory.

### A. Principle

The cornerstone of our mechanism is to process references and values in a different way. 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 used by the regular stack. The changes with our typed stack is just the place where you will find items.

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

Table IV: Typed stack 1

| | Normal stack | Typed stack | |
|---|---|---|---|
| | | | ⇓ |
| | | | |
| | | | |
| ⇑ | value | value | ⇑ |

Then we push the first reference :

Table V: Typed stack 2

| | Normal stack | Typed stack | |
|---|---|---|---|
| | | reference 1 | ⇓ |
| | | | |
| | reference 1 | | |
| ⇑ | value | value | ⇑ |

And the last reference is pushed :

Table VI: Typed stack 3

| | Normal stack | Typed stack | |
|---|---|---|---|
| | | reference 1 | ⇓ |
| | reference 2 | reference 2 | |
| | reference 1 | | |
| ⇑ | value | value | ⇑ |

With the typed stack, 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 typed stack we need two pointers, one pointing the top of the values and one for the references.

To reuse the example of a mutant application previously explained in the table III, at the `NOP` instruction, address `0x7404`, the stack has only a value reference. The next instruction, `astore_0`, stores the last pushed value in the local 0. You are two parts. The first one, you have a single-stack JCVM implementation and, the last pushed value is the return of the `invokestatic` instruction (address `0x7401`). This return type is a value.

On a typed stack JCVM implementation, when the `astore_0` is executed, the high part of the Java Card stack – which contains the pushed reference value – is empty. On the contrary, the low part contains the return of the `invokestatic` instruction. Thus, the JCVM detects an unexpected behavior. Thus, this countermeasure prevents type confusion to be exploited as described in [7]. Used with dual-linked stack, our countermeasure prevents the Java Card stack from the overflow and underflow, as explained in [8].

But our typed stack mechanism requires that implementations of instructions in the VM to know which stack operand to use to get elements. 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 untyped instructions = and these instructions cause problems for the implementation of the VM. They cannot differentiate the references or values. These instructions are:

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

The question is why these instructions are not typed ? For instance, for a dup instruction which duplicates the last element stacked, the VM does not need to know if it duplicates a value or reference with a simple stack, it must just go for the element pointed by the top of stack. However, with a typed 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 the stack of the values or the top of the stack of the references. So, with the typed 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. Currently, in this new implementation of the Virtual Machine we added a pointer to manage the values pushed on the stack. With two pointers, we can push and pop the references in the top of the stack and the value of the bottom. In order to respect the backward compatibility of the JCVM, a custom component is added to indicate if the VM runs the applet in typed stack or simple-stack mode.

Moreover, to correctly run an application, each untyped instruction must be removed. Indeed, `pop` instruction, with dual-stack JCVM implementation, might be non executable. So, to correctly run your applet on a dual-stack JCVM or if the applet may run in the JCVM without this countermeasure, we provide a way to protect your application against external modification (with a laser beam for example). For that, we remove each untyped instructions put in the Java Card applet.

*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. It requires transformation of the original program code so that the VM can run the program without errors. One solution is to replace 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.

This transformation requires the analysis and the modification of each methods, one after the other, because the method stack is local. Before we can replace untyped instructions we need 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 information, we analyze the byte code, instruction after instruction, and as we know exactly for each instruction what changes are made on the stack, we just perform a stack simulation.

This byte code analysis is completely linear, i.e it just reads the instructions one by one. However "jumps" complicate the analysis. Indeed, the first approach is to go to the location pointed by a jump instruction and to continue the analysis. But it is not necessary to analyze twice the same instruction, and furthermore the analysis can even enter into an infinite loop. This is why the analyzer stops when it found that an instruction has already been parsed.

Conditional jumps are another issue. 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 ignore the jump and continue. So the analysis must explore two branches and launch two sub-analyzes. Each of these analyzes must be run with an identical stack, one obtained just before the conditional jump.

*2) CapMap Integration:* The CapMap [9] is a Java-framework which provides an easy way to parse and modify a 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 analyze 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 previously pushed type and value) in order to dynamically modify the CAP file. Then you can also update each CAP file component to create a well-formed file. Indeed, this tool is used to test card against logical attack.

In our 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 instruction blocks with untyped byte code, the CapMap modify these instructions as described in the section III-B1. This step is explained in the figure 1.

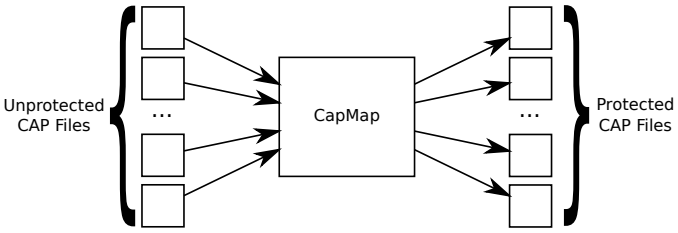


Figure 1: CapMap integration

## 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 mutant detection success ratio is high
- its memory footprint is low.

The above three points are the most important when designing a countermeasure for a smart card. The last point can be split into RAM and ROM usage knowing that the limited resource is the RAM. These metrics requires the implementation of our methods in our own prototype while the others metrics can be obtained thought a fault simulator [10].

Two Java Card applets have been used for the evaluation. Those two cardlets are representative of the type of code that a mobile network operator (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 and then sends a notification to a dedicated service (registered and identified in the cardlet). The second (SfrOtp) 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 by implementing our proposition 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.

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 a Java Card application, there are not many changes to do. In order to 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 close.

## V. CONCLUSIONS

In this paper, we presented a new approach to improve resistance of Java Card virtual machine. It is affordable for the card and is 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 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. Furthermore, the approach do not have any impact on the applicative development and the application transformer does not significantly increase the size of the application.

## REFERENCES


[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, D. Ltd, and I. Rehovot, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[2] J. Blomer, M. Otto, and J. 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.

[3] 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.

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

[5] K. Gadella, "Fault Attacks on Java Card (Masters Thesis)," Master Thesis, Universidade de Eindhoven, 2005.

[6] 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.

[7] J. Iguchi-Cartigny and J. Lanet, "Developing a trojan applet in a smart card," *Journal in Computer Virology*, vol. 6 Issue 4, pp. 343–351, 2010.

[8] G. Bouffard, J. Cartigny, and J.-L. Lanet, "Combined Software and Hardware Attacks on the Java Card Control Flow," in *Cardis 2011*. Springer, 2011, pp. 309–318.

[9] Smart Secure Devices (SSD) Team – XLIM, Université de Limoges, "The CAP file manipulator," http://secinfo.msi.unilim.fr/.

[10] J.-B. Machemie, C. Mazin, J.-L. Lanet, and J. Iguchi-Cartigny, "SmartCM A Smart Card Fault Injection Simulator," *IEEE International Workshop on Information Forensics and Security (WIFS 2011)*, November, 29th to December, 2nd 2011.