SmartCM A Smart Card Fault Injection Simulator

Jean-Baptiste Machemie, Clement Mazin, Jean-Louis Lanet, Julien Cartigny

SSD - XLIM Labs, University of Limoges,
83 rue d'Isle, 87000 Limoges, France,
{jean-baptiste.machemie, clement.mazin, jean-louis.lanet, julien.cartigny}@xlim.fr

***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 an evaluation of the effect of the propagation and the generation of hostile application inside the card. We designed several countermeasures and models of smart cards. Then we evaluate the ability of these countermeasures to detect the faults, and the latency of the detection. In a second step we evaluate the mutant with respect to security properties in order to focus only on the dangerous mutants.**

I. INTRODUCTION

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 try here to evaluate the effect of a fault on smart card program in order to design efficient countermeasures. For that purpose we have developed our software fault injection simulator *SmartCM*.

Software fault injection is the process of evaluating software under anomalous circumstances involving external inputs or internal system state. It is often used to assess the correctness of a system design. Software fault injection tries to measure the degree of confidence that one can have in a given system by evaluating what could happen when faults are activated. Traditionally, the software-based fault injection involves the modification of the software execution on the system under analysis in order to provide the capability to modify the system state according to the programmer model view of the system. All sorts of faults may be injected, from the register, flags, and memory faults.

SmartCM modifies the EEPROM memory where the application is stored according to a fault model, examines the effect on the program and if the detection mechanisms embedded in the card are not able to discover the modification it saves the mutant application. Later, all the mutant applications are checked to decide if the mutation is dangerous or not.

The reminder of the paper is organized as follow. Section 2 presents an overview of the smart card attacks and defenses. Section 3 discusses *SmartCM*, an automated tool that we have developed to evaluate the fault propagation. Section 4 presents the experimental evaluation of *SmartCM* on industrial cases studies. Section 5 introduces our future developments and then we conclude.

II. BACKGROUND

A. Smart Card Attacks

Smart cards are objects commonly used in our daily life providing some computing capabilities and security features in a very small device. Examples of applications using smart cards are banking applications, electronic passport, health insurance card, pay TV, SIM card, etc. Therefore, they contain some sensitive information which must be protected against fraud. Since the beginnings, smart cards have suffered many hardware and software attacks in order to gain access to their assets.

Boneh, DeMillo and Lipton have proposed in [5] a new attack model against smart card which they called cryptanalysis in presence of hardware fault. This attack model initially focused on several public-key cryptographic algorithms like the RSA signature scheme and the Fiat-Shamir and Shnorr authentication schemes. It has been shown in [3] by Bihan and Shamir that DES is also vulnerable to these attacks. This has led to numerous forms of hardware attacks against smart cards using fault injection [14], [2].

Faults can be induced into the chip by the perturbation of its execution environment. Consequences of fault attacks can be perturbation of the chip registers (*e.g.*, the program counter, the stack pointer,...), or the writable memories (variables and code modifications). These perturbations can have various effects, and in particular, they can allow an attacker to gain illegally access to data or services if not detected. In the literature, we can find different manners to produce fault attacks but currently laser beam attack is the most difficult to tackle with.

B. Fault Model

To prevent a fault attack, we need to know its impact on the smart card. Fault models have already been discussed in details [4], [15]. We describe in the table I the different fault models in descending order in terms of attacker power. In this paper, we consider that an attacker can change one byte at a time *i.e.*: the precise byte error fault model. Sergei Skorobatov and Ross Anderson discussed in [12] an attack using the precise bit error model. But it is not realistic on current smart cards, because modern components implement hardware security on

Table I: Existing Fault Model

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

memories like error correction and detection code or memory encryption.

The process is the following, 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). Then the attacker observes the effect of the fault characterized in terms of temporal and spatial parameters. If the result did not provide him any valuable information he restarts the process. This scenario validated with our industrial partner helps us to select the fault model: we choose the precise byte error. Thus, we assume that an attacker can:

- inject a fault at a precise clock cycle (he can target any operation he wants),
- only set or reset a byte to `0x00` or `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).

After defining the fault hypotheses we focus on the effect of the fault on a specific memory: the EEPROM. The system code is stored in ROM, it can suffer from a fault attack while the code is executed by the processor but this remains a transient fault which is more difficult to exploit for the attacker. At the opposite, the permanent error is the most valuable. This can occur when attacking the applicative code stored in EEPROM. But some recent smart cards have also system code stored in FLASH memory which may be subject to permanent error too. In both situations, modifying the code stored can change the behavior of the application leading to a potential aggressive application. Such a modified application, not detected by the embedded countermeasures is defined as a mutant.

C. Detection Mechanisms

There exists two different types of countermeasures against fault attack. The hardware countermeasures [1] which harden the ability to modify on-card programs and program flows. The software countermeasures in which software may check for faults or to ensure that no valuable information can be learned from injecting faults.

Hardware countermeasures include those which can be implemented by the industry to provide tamper resistant chips. In this category we can cite passive protections to increase the difficulty to succeed an attack (like random dummy cycles, bus and memory encryption, unstable internal frequency generators, etc.) and active protections that contain mechanisms checking whether tampering occurs and take countermeasures (generally the family of detectors like light detectors, supply voltage detectors, frequency detectors, etc). But those countermeasures are not dedicated to fault attacks and their detection ability is low. Currently software countermeasures are the most efficient solution.

Software countermeasures can be classified by their type.

- Cryptographic algorithm countermeasures which focus on the implementation of specific cryptographic algorithm and often provide better implementations of the 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; It is possible to implement at the application level several checks to ensure that the program always executes a valid sequence of code on valid data. It includes double condition checks, redundant execution, counter etc. and are well known by the developers. Unfortunately this kind of countermeasures increase drastically the program size. Because beside the functional code, it needs security code and the data structure for enforcing the security mechanism embedded in the application. Furthermore Java is an interpreted language therefore its execution is slower than with a native language, so this category of countermeasures suffers from bad execution time and add complexity for the developer.
- System countermeasures where protections are integrated directly at the system level. The main advantage is that the system and the protections are stored in the ROM memory, which is a less critical resource than the EEPROM and cannot be attacked. Thus, it is easier to deal with integration of the security data structures and code in the system. But often the design of an embedded Java Virtual Machine (JVM) relies on an offensive interpretation with a few system countermeasures and a robust Byte Code Verifier (BCV) that checks during load time the application.

Nevertheless all these countermeasures need to be evaluated in terms of efficiency, ability to detect a fault and cost: size of the memory footprint and execution time overhead.

D. Driving the Execution Mode on the JVM by the Application

In a previous paper [10], we have proposed a solution using a security feature available in Java Card 3.0 platform: the

[1] bit set or reset

annotations. But this approach is also fully applicable to Java Card 2.x platform using the custom component facility [13]. The idea is to drive the execution mode of the JVM by the application. The developer knows the semantics of its application and then can choose the best embedded countermeasure for each fragment of its code. Some of them are not sensitive while others need to be executed in the most secure mode. It is then possible to adjust the memories and CPU overhead to the optimum and reduce code in the application dedicated to countermeasure. The process is executed in two phases: outside the card we generate the annotations and in the card we change the execution mode according to the annotations. For example, the `@SensitiveType` annotation denotes that the whole method must be checked for integrity with the *check paths* 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);
   }
}
```

For the off card part, we provide a tool that processes an annotated class file. It allows the use of several countermeasures like those defined in [11] (*i.e.*, Check paths, Basic Blocks, etc.) on methods or on byte code fragments. Then the file is securely loaded inside the card using the Global Platform protocol [8]. Java based smart cards can process custom components if it knows how to use them or else, silently ignores them. Of course to process the information contained in these custom components, the virtual machine must be adapted, for that purpose we have designed our own JVM. The virtual machine interprets the application code and while entering a method or class tagged with a security annotation, it switches to the required secure mode. This approach is compatible with non modified virtual machine.

With this mechanism, an attacker must simultaneously inject two faults at the same time on two different memory areas, one on the application code and the other on the system during the interpretation of the code. A dual fault is outside the scope of the chosen fault model and is not realistic according to the literature.

III. TOOL ARCHITECTURE

The simulation tool *SmartCM* aims at analyzing the effect of a fault on a Java Card program. Three different programs are used in this analysis. The first one is the mutation engine which takes as input a model of the card and the applicative program at the byte code level. It emulates the effect of the fault on the program according to a fault model and generates the mutant code. The mutant code is symbolically interpreted by the card and if an embedded countermeasure detects the deviant behavior the mutant is rejected, else it is stored as a mutant. The second tool is a risk analysis tool. If a mutant is generated we need to evaluate the impact of its behavior to decide if it is a hostile behavior or not. Then a last tool which is under development aims to integrate code into the application in order to reduce the number of mutants. We expect also to add (see future works) a module to recognize sensitive patterns during the development in order to provide a complete framework.

A. Smart Card Model

The smart card model integrates well known software countermeasures and specific ones developed in a previous thesis. It models also the nature of the memories used by the smart card. If the given smart card used an encrypted memory then the effect of a fault on the memory will be a random byte according to encryption algorithm. The second entry is a set of class files of the program. The user can choose a profile corresponding to registered smart cards.

B. Mutation Engine

The mutation engine is a brute force process which modifies the memory where the byte code is stored. The effect of the fault on the program is evaluated with an abstract interpreter that includes the management of the Java annotations. If the byte that has been impacted by the fault is an opcode, then according to the kind of memory (encrypted or not) the value of the new opcode is either `0x00` or `0xFF` or any value in this range. Then the mutation engine uses the smart card model to evaluate the execution of this new code for each values of the opcode and propagates the error until a countermeasure detects it (stack underflow, overflow, wrong local variable, wrong expected type,...). If a return of the method `ProcessAPDU()` (which can be considered as a main for a Java Card applet) of the applet is reached or an exception is never caught, then we consider that the mutant cannot be detected. We generate a class file that corresponds to the mutant code and the corresponding class file is stored for further analysis. Less secure is the card, more we must interpret the code and longer is the simulation. Parallelization is one of our current improvement in order to be able to analyze low-end smart card; each new code is independent from the others and then can be analyzed concurrently on different computers.

C. Risk Analyzer

The mutation of an application can generate several mutants according to the security of the platform. To help the programmer to understand the effect of the error, it outputs the original Java Code and the Java perspective (if possible) of the mutant code, it highlights the area where the code has been modified. Often the mutants are harmless but a security officer must check all of them. In order to facilitate this task we developed a risk analysis module that verifies a set of security properties on the mutant and decide to tag the mutants as

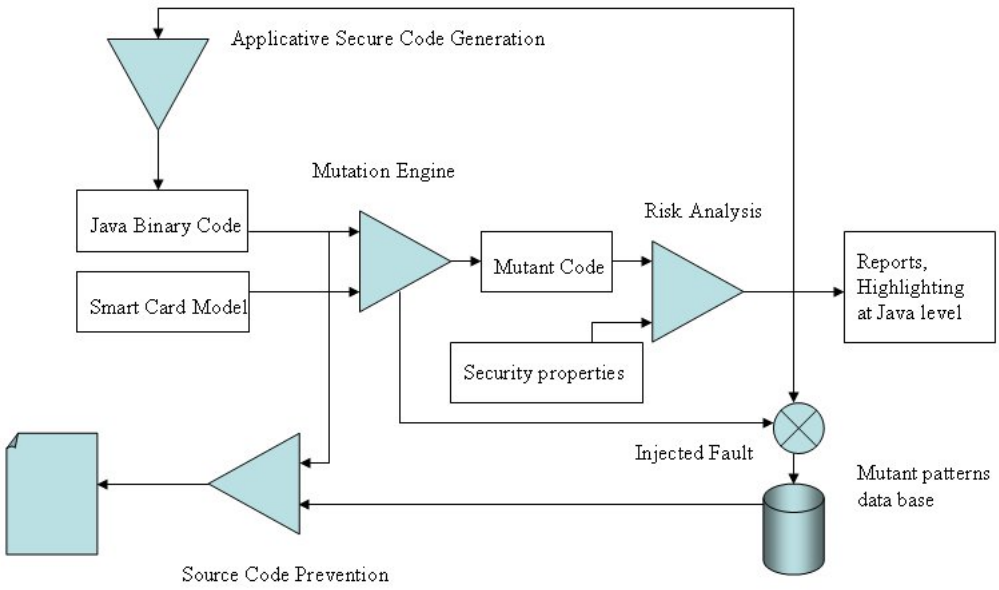

Figure 1: General architecture

dangerous or not. This tool can also include the possibility to add other security properties, to include an internal map of the the method addresses.

A mutant can be dangerous in several cases: it accesses objects and methods that are normally non authorized (potentially call to unwanted methods : `getKey()`), it changes its own internal behavior (*e.g.*: remove test), it performs action on its own data in an unexpected way.

1) Method calls: The issue is when another method is called or if one of the parameters has mutated. Methods must be invoked with the appropriate arguments (number and type), for that purpose we need to model the effect of the instruction on the operand stack and local variable array and verifies that it corresponds to the signature. If the number of arguments are not correct in terms of types on top of the stack a type confusion can occur inside the called method which is forbidden. For example, an address is given instead of a short and the short value is modified in the called method: it is the way to perform arithmetic on addresses which leads to a dangerous mutant. In terms of number of arguments: too few arguments is detected thanks to a stack underflow detection (if present in the card), too much arguments cannot be detected locally but will be detected by the caller. Normally at the end of a method the stack contains either zero element (return void) or one element that has the correct type.

The address of the called method is valid but different from the original. In fact, the linking process is done in the card. So the code to analyze do not have the real addresses (it contains only offset to the constant pool). According to industrial partners, it is possible to obtain the internal mapping of the methods (in the ROM or EEPROM area). In such a case, it is possible to generate the mutants where the address of a call is valid. We have to consider two cases, the arguments are valid (number and type) and it is a dangerous mutant or not and then it can be considered as harmless. If the mutation concerns an address that does not correspond to the structure of method, it will be detected by the JVM and thus it must not be considered as dangerous.

In case of `invokeInterface`, `invokeSpecial` and `invokeVirtual`, methods are invoked on another object. If the address does not correspond to a data structure of an object, the mutant will be detected during run-time, it is not dangerous. If it is a concrete address of an object that do not belong to the current security context, the mutant will be detected during run-time, thus it is not dangerous. If it is a concrete address of an object that belongs to the current security context, the method will be applied on the wrong object : the static analysis can not detect it, there is no possibility to have a dynamic map of the objects. We apply the same rule when accessing a field.

2) Changes in the control flow or data: The mutant can change the control flow of the method. The issue is that some methods are not called or the result is not evaluated. We must verify that all calls and all evaluations are performed because if one is skipped it is a dangerous mutant. The last point concerns the static fields or the local variables. If a local is used in an assignment *e.g.* `boolean testVariable = pin.isValid()`, and later this variable is used into a test, any assignment between definition and use can lead to a dangerous mutant.

IV. INDUSTRIAL CASE STUDY

A. Applications

Several Java Card applets have been used for the evaluation. Two SIM applets are representative of the type of code that a MNO (Mobile Network Operator) may want to add to their USIM Card. The first (*AgentLocalisation*) is oriented geolocalization services, this applet 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 applet). The second (*OTP*) is more specialized to authentication services, the applet 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 applet with a central server, which is able to check every collected OTP value using dedicated web services. We also evaluate a protocol payment applet designed by a major smart card manufacturer [7] and a hostile applet specifically designed to execute shell code when mutated [6].

B. Metrics

The collected metrics cover two different aspects: how efficient are the countermeasures and are they affordable. One can design very efficient code but if it does not fit the industrial requirements it remains useless. To fulfill the first one we need to verify their detection coverage and the latency of the detection. The second point is related with run time execution and memory footprint. It is widely accepted that an overhead over 5 % is considered as a maximum. So we need in a one hand to simulate the code (the objective of *SmartCM*) on the other hand to implement it in a JVM and evaluate its run time impact.

1) Evaluating Resources Consumption: The first category of metrics is the memory footprint and the CPU overhead. They have been obtained using the SimpleRTJ [9] Java virtual machine modified to accept multiple execution modes driven by annotations. This JVM targets highly restricted constraints devices like smart cards. The hardware platform for the evaluation is a board which has similar hardware as high end smart cards.

These metrics are very important for the industry because memories size directly impacts the production cost of a card. In fact, applications are stored in the EEPROM which is the most expensive component of the card. The CPU overhead is also important because most of the time, when challenging the card for some computation a quick answer is needed. So when designing a countermeasure for smart cards, it is important to have these properties in mind. To obtain the metrics in table II, all the countermeasures have been implemented on an embedded JVM that has similar properties as common smart cards.

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

The mutant generator has different smart card profiles:

- *The Basic Profile (BP)*: the interpreter executes, without running any checks, the instruction set. It corresponds to low end smart cards which rely entirely on the execution of BCV outside the card. Most smart cards corresponding to the 2.1 standard and some of the 2.2 standard fall into this category.
- *The Defensive Profile (DP)*: the interpreter checks that no overflow or no underflow occurs, that the used locals are inside the current table of locals, and that when a jump occurs it takes place inside the method. They consist in some verifications done by the BCV.
- *The Customized Profile*: this profile corresponds to the dynamically adjustable security of the JVM. The user can activate different countermeasures see [10] like the developed ones: path checking mechanism (PC), basic bloc integrity (BB), field of bits mechanism (FB) , or TCM mechanism. TCM is a detection mechanism that is not described in this paper and for which a patent is pending. For the purpose of this evaluation, we compare independently each countermeasure for the whole program. It is a pessimistic approximation because it is applied to the whole program, while it needs to be applied only on the sensitive methods or fragments of them.

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

Table II: Resources consumption for Customized Profile

Countermeasures	EEPROM	ROM	CPU
Field of bits (FB)	+ 3 %	+ 1 %	+ 3 %
Basic block (BB)	+ 5 %	+1 %	+ 5 %
Path check (PC)	+ 10 %	+1 %	+ 8 %
Typed CM (TCM)	0 %	<1 %	+<1 %

4) Mutant Detection and Latency: The obtained results show the efficiency of the developed countermeasures. The table III shows the generated mutants in each mode of the mutant generator for five applications. The table IV shows the latency which can be defined as the number of instructions executed between the attack and the detection. With the basic profile, no latency is recorded because no detection is made. This value is really important because if a latency is too high maybe instructions that modify persistent memory like: `putfield`, `putstatic` or an invoke instruction (`invokestatic`, `invokevirtual`, `invokespecial`, `invokeinterface`) can be executed. If a persistent object is modified then it is manipulated during all future sessions between the smart card and a server. So this value has to be as small as possible to reduce the chances of having instructions that can modify persistent memory or send data to the reader.

Path check (PC) fails to detect mutants whenever the fault that generates the mutant does not influence the control flow of the code. Otherwise, when a fault occurs that alters the control flow of the application then this countermeasure detects

Table III: Results - Mutation

	BP	DP	CP			
			TCM	FB	BB	PC
Wallet (470 inst.)	440	54	30	10	0	37
Otp (4568 inst.)	7960	464	378	40	0	1032
AgentLoc. (3504 inst.)	6486	356	343	10	0	784
Payment (1100 inst.)	2140	304	250	1266	0	2140
Hostile (825 inst.)	1622	0	50	14	0	76

Table IV: Results - Latency

	BP	DP	CP			
			TCM	FB	BB	PC
Wallet	-	2,91	2,92	2,43	2,72	2,42
Otp	-	3,64	3,56	8,61	12	17,18
AgentLoc.	-	11,8	12,1	2,43	10,20	13,06
Payment	-	5,54	6,55	0,5	-	-
Hostile	-	-	0,77	0,71	-	0,79

it. With this countermeasure it becomes impossible to bypass systems calls like cryptographic keys verification. Basic bloc is the most efficient countermeasure.

5) Risk Analysis: The risk analysis tool is able to cut for the configuration the AgentLocalisation applet using the TCM, the 343 mutants to 8 dangerous mutants which reduces drastically the effort for manual inspection of the code. It needs only a few seconds to execute this analysis, in the case where the internal mapping is provided. The tool is able to classify between 90% to 97% of the mutants into the non dangerous category.

V. FUTURE WORKS

In the next steps, we will try to improve the development phase. We want to analyze the mutant codes, to deduce at the Java level which structures are sensible for a given Java Card platform. Then we will propose to generate the code that could (or not) eliminate the mutant. For example, if the mutant is generated thanks to a modification of the control flow, then a double conditional can solve this issue or a sequence counter. In some cases it is possible to generate automatically applicative counter measures. The objective is to pinpoint the sensitive structures, suggest a countermeasure or a warning. The second improvement concern the development phase, close to the previous one but using learning networks. While coding its application the developer could we warned that a given pattern (at the byte code level) is sensitive. Each time an application is analyzed, we enrich the data base of sensisitive patterns. Each sensitive pattern is associated to Java source code, if possible.

VI. CONCLUSIONS

In this paper, we presented a simulation approach to evaluate the impact of a fault on the program memory of a Java based smart card. We define a platform model through a profile then we generate all the possibilities of an error. The collected metrics during the simulation are used to adopt a given counter-measure: ability to detect a fault, latency of the detection, cost in term of memory (ROM, RAM and EEPROM), cost in term of CPU overhead. We have developed our own countermeasures and evaluate them in this context. The designed tool offers also the possibility to eradicate non relevant mutants and view only those that endanger the security of the platform. it provides the ability to read it either at the byte code level for experts but also (if possible) at the Java language level.

Designing counter measures takes into account the compliance with previous file formats according to either the Java Card 2.x or 3.x specifications. Our countermeasures are affordable due to a low memory footprint. For the applet designer, he can adjust dynamically the security level of the JVM according to the semantics of its program. The developer knows exactly which program fragment needs to be protected and which can be executed without specific treatment.

With this framework, both the developer and security evaluator can take decisions concerning the security of the 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] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, and I. Rehovot. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[2] G. Barbu, H. Thiebeauld, and V. Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. *Smart Card Research and Advanced Application, Cardis 2010*, LNCS 6035:148–163, April 2010.

[3] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. *Lecture Notes in Computer Science*, 1294:513–525, 1997.

[4] 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.

[5] D. Boneh, R.A. DeMillo, and R.J. Lipton. On the importance of checking cryptographic protocols for faults. *Lecture Notes in Computer Science*, 1233:37–51, 1997.

[6] G. Bouffard, J-L. Lanet, and J. Cartigny. Combined software and hardware attacks on the java card control flow. In *Proceedings of Cardis Tenth Smart Card Research and Advanced Application Conference*, 2011.

[7] P. Girard, J-L. Lanet, A. Plateaux, and K. Villegas. A new payment protocol over the internet. In *CRISIS, International Conference on Risks and Security of Internet and Systems*, pages 51–56, 2010.

[8] Global Platform. Official web site, http://www.globalplatform.org, 2010.

[9] Simple RTJ. Official web site, http://www.rtjcom.com, 2010.

[10] A. Sere, J. Cartigny, and J-L. Lanet. Automatic detection of fault attack and countermeasures. In *Proceedings of the 4th Workshop on Embedded Systems Security*, pages 1–7. ACM, 2009.

[11] J. Sere, A. Cartigny and J-L. Lanet. A path check detection mechanism for embedded systems. *Proceedings of SecTech 2010*, 6485:459–469, 2010.

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

[13] SunMicrosystems. *Java Card 3.0.1 Specification*. Sun Microsystems, 2009.

[14] E. Vetillard and A. Ferrari. Combined attacks and countermeasures. *Smart Card Research and Advanced Application, Cardis 2010*, LNCS 6035:133–147, April 2010.

[15] 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.