Automatic detection of fault attack and countermeasures

Ahmadou Al Khary Sere Julien Iguchi-Cartigny Jean-Louis Lanet

Université de Limoges
Laboratoire XLIM
Departement informatique
83 rue d'Isle
87000 Limoges, France
ahmadou.sere@xlim.fr
julien.cartigny@unilim.fr
jean-louis.lanet@unilim.fr

Abstract

Security of smart cards is continuously threatened by many software and hardware attacks. Those attacks focus on giving secret information that are stored in the card like PIN code, secret cryptographic keys, or on granting access to some restricted operations. The main line of this paper is to integrate countermeasure against fault attacks into a Java Card 3.x smart card. So the solutions proposed, allow to transform some annotations that are put by the developer, in information which will guide the code interpreter to detect faults that can occur during programs execution. the proposals are generic in sense that they don't focus on a particular algorithm.

General Terms Fault Attack, java Card, bytecode, opcode, basic bloc

1. Introduction

Smart cards are tools that today are commonly used in our daily live because they supply some computing capabilities and security in a very small device. Examples of application that use smart cards are credit card, electronic passport, health insurance card, pay tv, telephony SIM card, etc. Therefore, they contain some sensitive information that has to be protect against intrusion. Since, apparition of smart cards many hardware or software attacks have been created to endanger their security.

Boneh, DeMillo and Lipton have proposed in [7] 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 [5] by Bihan and Shamir that DES is also vulnerable to these attacks.

The effect of these types of attacks surpasses the implementation of cryptographic algorithms. They actually concern the whole software embedded on the card and can be focused on any point on which the security of the card relies. They can for example be used to bypass the verification of the PIN code, or to bypass the runtime software checking system deployed on the chip. Thus, they constitute a serious threat against smart card security.

The field of our actions is Java Card enabled smart cards. This technology comes with a virtual machine that allows multiple applications (applets) to run in a secured environment. Java Card language is a subset of Java, therefore it works the same way: a virtual machine, an execution environment, and a programming API. The last release of the Java Card specifications is the 3.x branch.

After the compilation of a Java Card 3.x applet, we obtain a cap file (classic edition) or a class file (connected edition). These files contain bytecodes that are Java instructions. When interpreting these instructions, the virtual machine creates an operand stack where opcodes put intermediary data they need or results they produce.

We describe in this paper some methods that allow to protect the code execution against unwanted changes that can occur in results of a fault and such in an automatic manner. The methods apply to any Java Card software, because, a tool integrates data, that contain information about the code useful to check his integrity during runtime. These data are add to the application code after his compilation and before his charging in the card.

The protection proposed in this paper must satisfy some strong constraints regarding dynamic space, size of code and execution time overhead. Because depending of the chip, they have indeed a few quantities of bytes for the RAM and for the EEPROM.

The rest of the paper is organized as follows: first, we speak about different category of fault attacks, second we expose our problematic, third we discuss of the fault model, after we talk about existing countermeasures, finally we talk about our proposals.

2. fault attack

Faults are errors that can be induced into the chip by the perturbation of his execution environment. These errors can make some mess in the computation, or in the good running of a program. Based on this, a fault attack is an attack which has the ability to disturb the smart card chip by disturbing in physical way his operation. These perturbations can have various effects, principally it can allow an attacker to make some treatment that he hasn't the right to do or to have an access to some secret data that are on the smart card. Consequences of faults attacks can be perturbation of the chip registers (like the program counter, the stack pointer), of the memory (variables and code changes). In the literature, we can find different manner to produce fault attacks. To get an idea on how fault can be introduced, this section gives an overview of various types of attacks, these attacks are all hardware attacks.

2.1 Spike Attacks

By varying the power sent to the chip by his VCC, it is possible to disrupt a computation. This work, because between some predefine limits, the chip comportment is known, over these bounds its behavior is unpredictable. In some cases, this can be enough to introduce a fault [3]. A power spike can vary in some 9 different variables, including height, shape, build up and power down, duration, etc. As a spike works by simply connecting to the power led to the chip, they do not need direct access to the chip, and the equipment requirements are totally dependent on the type of the spike to be generated, but are not necessarily expensive.

2.2 Glitch Attacks

Similar to spikes, it is sometimes possible to disturb the clock speed. For example, by doing an update cycle at double speed some instructions will be effected where others are not, so it is possible that old data are used as the new data haven't arrived yet [9]. The introduced "glitch" can be used to influence conditional jumps (by not performing them, or performing them when it is unwanted, etc [2]), a shift register shifts twice (instead of once), or not at all, etc. In general, it results in changes off the program, and depending on those changes an attack can be done. In the same way of spike, the hardware needed to produce "glitch" is not necessarily expensive.

2.3 Optical Attacks

By using focused light with specific wavelengths, it is possible to inverse the containing of a memory cell. By doing this, it is possible to change or to modify the memory by using photoelectric effect. these attacks require light to be able to reach the chip, and thus that any protective layer needs to be removed. As for equipment, in [11] it was shown that these attacks can be done relatively cheaply with simple equipment, and also that they can be very precise. Example of equipment to produce this attack can be a flashlight, a photo flash, a laser, etc.

2.4 Electromagnetic Perturbation Attacks

It is possible to fulfill a fault attack by creating a strong electromagnetic field near memory. This will cause that the ions representing the states in the memory will move around and by that will be disturb. It is claimed in [10] that the so called "eddy current" can be created to use an active coil with sufficient strength. This can then give a fine control of exactly what bit needs to be touched by the attack. With that claim, also comes the one that it can be done relatively cheap.

Today, attacks in sections 2.1, and 2.2 are attacks that are difficult to achieve with modern cards because they are protected to the hardware level. Those in sections 2.3 and 2.4 are the ones that really can cause trouble.

3. Problematic

A developer creates his application with the Java Card platform; this application is sent to the card in secure way by a Global Platform protocol. When the application arrives into the card, it transits by the ram and is save into the static heap area (EEPROM), after that the card can use it for execution purposes. A fault attack can happen from the sending of the application, to its execution into the card. We are interesting by the fault that arrives from the saving of the application to its execution. So we assume that no faults occur until the program is saved. We also focus on faults which touch the area where are stored the data containing information about the method code: the byte array. So what we want to do is to be sure that the code that was earlier stored in the static heap is not modified by a fault attack when come the time to execute it.

We also have to compose with the facts that smart card is embedded device with limited capacity, so the answer to the previous interrogation, must fit to the device capacity and mustn't increase greatly the memory and the time need for the computation. With that in mind we put a bound of 5% for the increase of the previous two values.

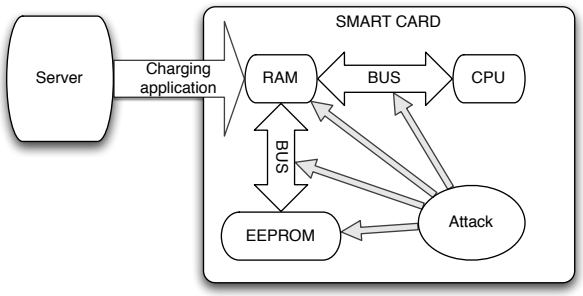

Figure 1. Problematic

4. Fault model

In the literature [12, 6], it exists principally 4 fault models. These models are classified by the control over the amount of memory change and the timing. A fault model is useful in sense that it allow to know which effects, attacks can have on smart card. the 4 fault models are:

1. **Precise bit error** : it's a fault model where attacker have a total control over the timing i.e the location of the fault, and a precise control over the bit he wants to flip.
2. **Precise byte error** : it's a fault model where attacker have a total control over the timing, and a precise control over the byte he wants to change.
3. **Unknown byte error** : here the attacker have a lose control over the timing and he doesn't know what byte he are attacking.
4. **Unknown error** : The attacker makes a fault but he doesn't know where or when it happens.

The fault model have been enumerated by descending order in term of attack power, so if we are protected against the fault attack following model 1, we are also protected against fault following model 2, 3 and 4. However, precise bit error doesn't reflect the real world because today high end smart card integrates hardware counter-measures that hardened the location of one bit. So we have to lower the rank of fault model we choose to take the fault model 2 (precise byte error). Our choice can be considered to be realistic, in facts, the attacker can modify a byte of his choice, but except for the value 0x00 and the value 0xFF, he can't control the value that the byte will take.

So with all those previous properties, we had to take the precise byte error model and we also had other criteria to it :

- The byte ***can be set to 0x00 or 0xFF*** or it ***can take a random value*** between this two bounds.
- In the real world, when an attacker make his fault, he make one fault following the chosen model then he waits for the result. so for our model, ***he can only make one fault at a time***.

Effects of fault following this model can be: register modification (like PC, or stack pointer), program modification (variables, code, etc). We are interesting by the effect of fault attack on code.

5. Existing countermeasures

It exists different type of countermeasures against fault attack: the hardware countermeasures [4, 8], which harden the ability to modify on-card programs and program flows, and 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 includes those which can be implemented by the industry to provide tamper resistant chips. In this category we can cite passive protection that contains protection that increase the difficulty of a successful attack (like random dummy cycles, bus and memory encryption, unstable internal frequency generators, etc) and active protection that contains mechanisms that check whether tampering occurs and take countermeasures (generally the family of detectors like light detectors, supply voltage detectors, frequency detectors, etc).

Those countermeasures exist, and they work fairly well but they have some drawbacks. First, they are made to protect against predefine attacks, so if some new attacks are found they probably won't work for them. Second, for each type of attacks you have to integrate on board a new countermeasure, then it increases the cost of the smart card. We know that money is an important factor for industry; it isn't acceptable for certain market to build some cards that will be expensive to sell. Hardware countermeasures are out of the topic for this paper.

Software countermeasures can be classified by their type. So we can distinguish:

- Cryptographic algorithm countermeasures focus on the implementation of cryptographic algorithm like RSA. It does exist many research paper which treat about it [3, 8, 6]. This branch is out of our study and we decided to rely on what have already be done.
- Applicative countermeasures [1] where protection system are add only into programs code. This kind of measures have side effects like the great increasing of the program file size. And the program files are stored in EEPROM but this kind of memory is really expensive and for that constitute direct payload for the smart card vendors.
- System countermeasures where protection are integrate directly in the system program.

6. Contributions

Our proposals take place beetwen the last two categories. it's an hybrid solution because we use the applicative part to integrate some information into the compiled file of the program. Those information are used by the code interpreter which is previously modified to integrate the detection system. This method insure that the source files size and the interpreter files size will not greatly increase. For allowing this method to work and to not protect all of the program code, we have create a Java annotation which is used by our system to add custom components. Those components are add to the method level. To illustrate our discourse, we will take an example: the method in figure 2 and his bytecode representation in figure 3 are extract from a wallet applet.

6.1 The programmer point of view

The programmer is the best person to know on what portion of the code rely the security of his application. So it can be a good idea to allow him to mark the method he wants to protect. First, this will permit to improve the execution time of the overall mechanism because the detection system will be activated only when needed. Second, it will allow to reduce the amount of data added to the

```
private void debit(APDU apdu) {

  // access authentication
  if ( ! pin.isValidated() )
    ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

  byte[] buffer = apdu.getBuffer();

  byte numBytes = (byte)(buffer[ISO7816.OFFSET_LC]);

  byte byteRead = (byte)(apdu.setIncomingAndReceive());

  if ( ( numBytes != 1 ) || (byteRead != 1) )
    ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

  // get debit amount
  byte debitAmount = buffer[ISO7816.OFFSET_CDATA];

  // check debit amount
  if ( ( debitAmount > MAX_TRANSACTION_AMOUNT) || (
       debitAmount < 0 ) )
    ISOException.throwIt(SW_INVALID_TRANSACTION_AMOUNT);

  // check the new balance
  if ( (short)( balance - debitAmount ) < (short)0 )
    ISOException.throwIt(SW_NEGATIVE_BALANCE);

  balance = (short) (balance - debitAmount);
} // end of debit method
```

Figure 2. A debit method from a Java Card applet

application source file. To achieve this, he can use the following Java 5 annotation:

```
@Retention{RetentionPolicy.CLASS}
public @interface secure {
        type security();
        public static enum type (FIELDOFBIT, BASICBLOC,
             ...)
}
```

With this at his disposal, he only have to mark the method he want to protect like in the next example:

```
@Secure(security=FIELDOFBYTE)
private void debit(APDU apdu) {
        ...
}
```

Here the programmer decided to mark the method he want to protect with the type of security FIELDOFBIT (section 6.3). So the modified interpreter will be aware of that and will execute this method with the correct type of protection. Annotations are proceeded by an Java API which can manipulate bytecode like BCEL, or ASM Assembly.

6.2 Renaming the nop operation

In the section 3, it was established that the attacker had the capacity to force a byte to 0x00 or 0xFF. In Java, the instruction 0xFF is reserved, so it will be useless for an attacker to replace an instruction by this value. This let the value 0x00, which correspond to the **nop** instruction (do nothing), as a critical value. The solution to this problem is to choose another byte to code the instruction 0x00 thus when the interpreter will meet the 0x00 instruction he will know that a problem occur. In practice, java opcodes are encoding using one byte which goes from 0 to 254. The Java Card instructions set contains 185 opcodes so all the number beetwen 0 and 254 aren't used, we can then take a number going from 185 to 254 for encoding the will be call the **new_nop** operation. We took the byte 0xC8 for this purpose. So beetween the verification phase of the code and before storing the application on board, a small program have to change all 0x00 to 0xC8.

```
0   aload_0
1   getfield #4 <com/sun/javacard/samples/wallet/Wallet.
      pin>
4   invokevirtual #18 <javacard/framework/OwnerPIN.
      isValidated>
7   ifne 16 (+9)
10  sipush 25345
13  invokestatic #13 <javacard/framework/ISOException.
      throwIt>
16  aload_1
17  invokevirtual #11 <javacard/framework/APDU.getBuffer>
20  astore_2
21  aload_2
22  iconst_4
23  baload
24  istore_3
25  aload_1
26  invokevirtual #19 <javacard/framework/APDU.
      setIncomingAndReceive>
29  i2b
30  istore 4
32  iload_3
33  iconst_1
34  if_icmpne 43 (+9)
37  iload 4
39  iconst_1
40  if_icmpeq 49 (+9)
43  sipush 26368
46  invokestatic #13 <javacard/framework/ISOException.
      throwIt>
49  aload_2
50  iconst_5
51  baload
52  istore 5
54  iload 5
56  bipush 127
58  if_icmpgt 66 (+8)
61  iload 5
63  ifge 72 (+9)
66  sipush 27267
69  invokestatic #13 <javacard/framework/ISOException.
      throwIt>
72  aload_0
73  getfield #20 <com/sun/javacard/samples/wallet/Wallet.
      balance>
76  iload 5
78  isub
79  i2s
80  ifge 89 (+9)
83  sipush 27269
86  invokestatic #13 <javacard/framework/ISOException.
      throwIt>
89  aload_0
90  aload_0
91  getfield #20 <com/sun/javacard/samples/wallet/Wallet.
      balance>
94  iload 5
96  isub
97  i2s
98  putfield #20 <com/sun/javacard/samples/wallet/Wallet.
      balance>
101 return
```

Figure 3. The bytecode representation of the method in figure 2

```
int16 BC_ifeq(void) {
  vm_sp--;//Variable that contains the operand
       stack
  if (vm_sp[0].i == 0)//if the last value on the
       stack is equal to zero
    return BC_goto();//call the goto operation
  vm_pc += 2;
  return ACTION_NONE;
}
```

Figure 4. Ifeq operation in simplertj JVM interpreter

```
int16 BC_ifeq(void) {
  vm_sp--;//Variable that contains the operand
       stack
  if ( vm_sp[0].i == NUMBER)// the new comparison
       to do
    return BC_goto();//call the goto operation
  vm_pc += 2;
  return ACTION_NONE;
}
```

Figure 5. Ifeq operation in simplertj JVM interpreter

Another problem that can be raised is some of the *if instructions*: **ifeq, ifne, ifgt, iflt, ifge, ifle**. Because when these instruction are met during runtime, the interpreter compares the last value that is on the operand stack with 0x00 (see figure 4). The attacker can decide to change the value that is on the operand stack to put a value of

If an attacker change this value to put other byte than 0x00 then it can access to some restricted data. So we also have to modify the code which is in charge of these instructions to make a comparison with another value: the value that is on the operand stack with a value big enough to have a length of 2 bytes minimum. The reason of this choice is that the attacker can only modify one value at a time thus he will only modify a part of the new value which will give a number that is not equal to 0 and is also not equal to the expected value. The result of this changes can be found in the figure 5.With this modification made to the concerned instructions they will be protect against the change of the 0x00 value on the operand stack.

6.3 The field of bit detection mechanism

This countermeasure comes with the fact that if an attack modify an opcode by another one, it is possible that operands number will be inconsistent with the new one. More precisely we can obtain the following situations:

1. An augmentation of operands number for the instruction, it's the case when **add** (no operand) is replace by **icmpeq** (one operand).
2. A diminution of operands number for the instruction, it's the case when **aload** (one operand) is replace by **athrow** (no operand).
3. An equal number of operand it's the case when an **iload** (one operand) is replaced by a **return** (one operand).

This method can detect when the change 1 and 2 happen. During the compilation time, a field of bit is generated representing the type of each element contain in the method's byte array. In this field, opcode that represent the execute part of the instruction is marked with an X (execute) and operands that represent the data manipulate by this instruction is marked with an R (read). Figure

0:	15	X
1:	83 00 04	X R R
4:	8B 00 12	X R R
7:	97 00 10	X R R
10:	19 63 01	X R R
13:	8D 00 0D	X R R

Figure 6. Byte representation of lines 0 to 13 of figure3

0:	15	X
1:	83 00 04	X R R
4:	8B 00 12	X R R
7:	00	X
8:	00	X
9:	10	X
10:	19 63 01	X R R
13:	8D 00 0D	X R R

Figure 7. Byte representation of lines 0 to 13 of figure 3 after modifying the **ifne** operation

6 shows the byte representation and the corresponding field of bit that are at lines 0 to 13 of figure 3.

The resulting table is saved as a component of the classfile (the JavaCard specification allow the creation of custom component). When comes the time to interpret the byte array, the interpreter had to verify the concordance of the instructions it is executing with values that are stored in the table. For example, if we supposed that in the figure 6, the **ifne** operation that correspond to 97 00 10 is replaced by a **nop** operation then the field of bit for this code portion is in figure 7.

We can note that the field of bit has change: his value is X X R R X R R X X X X R R X R R instead of X X R R X R R X R R X R R X R R. Thus, when the interpreter try to execute the **ifne** at the PC 7, it will see that at 9th position in the table it has an X instead of an R, in other term it has to test if $MethodByteArray[vmpc] \neq FieldofBit[vmpc]$. If the answer to the previous test if true then, it will stop the execution because a fault has probably occurred.

This method has a drawback in facts if an instruction is replaced by another one that has the same operand number then it can't detect the modification. Fortunately, with the chosen fault model we have approximately a rate of 10% for this to happen in Java Card application (this percentage have been obtained by statistic with 10000 instructions in 15 different applets).

6.4 The XOR detection mechanism

This mechanism borrows its principle to the graph theory precisely it takes the notion of basic block. This protection allows two things: first to check that the code integrity is safe second it permits to verify the control flow of the application. To understand the principle, we have to specify the notion of basic block.

A basic block is a sequence of instructions with a single entry point and a single exit point: execution of a basic block can start only at its entry point, and can leave a basic block only at this point. Thus, if control enters a basic block, each instruction in that block will be executed. The rules used to determine the basic block by finding the set of leaders (a leader is an instruction that can begin or finish a basic block). These rules are:

- The first instruction of the method and each first instruction of every handler of method is a leader.
- Each instruction that is the target of an unconditional branch (**goto, jsr, ret**) is a leader.
- Each instruction that is the target of a conditional branch (**ifeq, iflt, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icomplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq**) is a leader.
- Each instruction that is one of the target of a compound conditional branch (**tableswitch or lookupswitch**) is a leader.
- Each instruction that immediately follows a conditional or unconditional branch, or a **<T>return (ireturn, areturn and return)**, or a compound conditional branch instruction is a leader.

Each individual leader gives a rise to a basic block, consisting of all instructions up to the next leader or the end of the bytecode. Furthermore, we enclose each method invocation (**invokevirtual, invokespecial, invokestatic, and invokeinterface**) in a basic block of its own. In figure 3 example, the basic blocks are: {0,7}, {10,13}, {16,34}, {37,40}, {43,46}, {49,58}, {61,63}, {66,80}, {83,86}, {89,101}.

Now that we have our basic blocks, we can go to the next step that is to compute what we will call a checksum for all the blocks. These checksums are computed by making a XOR operation between all the bytes composing a basic block. This computation led us to a table compose for each basic block of:

1. The PC of its beginning
2. The PC of its ending
3. The value of its checksum

Then this table is store in the classfile as a custom component like in the section 6.3. When done the classfile is send to the card where it waits for its use. The interpreter has to be modified to make good use of the previous data. During runtime, it has to compute again the checksum and to compare it with the earlier store values. If they Are not identical it's probable that it's because of a fault. For each instruction, it will check some properties:

1. The PC of the current instruction is a PC that is in the table for beginning or for ending a block.
2. if the 1st property is verified, then it checks that current instruction is really an instruction that can begin or finish a block.
3. If the end of the current block is reach then before continuing to the next instruction it has to check that the checksum compute for the current block is equal to the one stored for this block. if they aren't, it's because that a problem had occurred and the interpreter can stop here its execution.

If a fault occur following the chosen fault model then this mechanism will detect it whatever the modification is. Because if one byte is modified, the XOR result will change too. If two bytes are modified by a fault the resulting xor can be different whence the importance of the fault model: an attacker can only modify one byte at a time.

The methods in section 6.3 and 6.4 have been implemented on an abstract Java virtual machine (JVM) interpreter on which we can also simulate the behavior of a fault attack. Thus, the runtime benchmark doesn't reflect the reality but we are implementing the previous countermeasure on a real JVM: simplertj. This Java virtual machine is designed to work on highly restricted embedded device like smart card. We are modifying the JVM to integrate our detection mechanisms; we planned to make our test on an AT91 board with hardware capability close to a real smart card. For the moment, on the simulator we have good results in term of detection (more than 80% for the field of bit, and 100% for the XOR mechanism with the chosen fault model) and the classfile incrementation is under the 5% limit (3% for the field of bit and

1 to 5% for the xor mechanism and this with all the methods mark to be protected).

7. Future works

As this is an ongoing work, so we first have to produce the benchmarks for the previous countermeasures to be sure that we are under the 5% limit and to confirm the detection rate that was given in section 6.4. And in answer to the problem raise by the method exposed in section6.3, we want to use error-correcting code in smart card to at least detect an error in the code and at best correct it, but the details for this new countermeasure will be in a future publication.

8. Conclusion

The vast majority of literature on the attacks should be dedicated only to the problem of misuse of cryptographic algorithms. But, these attacks can also be used to modify behavior much more basic in the code, whether native, or interpreted. For Java Card, these attacks have a much better feasibility because of the knowledge that the attacker has of the structure of a portion of memory (usually identical to a cap). Have protection mechanisms for the programmer simple and inexpensive in terms of memory and time treatment offers the possibility to protect the code of a program without breaking the development chain of Java Card applets. However, this approach has a limit, because it exploits the known structure of the code in a virtual machine, but cannot think about the meaning of data fields of a card. An attacker has therefore, for example, still be able to attack cryptographic algorithms by changing only the data.

References

[1] M.L. Akkar, L. Goubin, and O. Ly. Automatic Integration of Counter-Measures Against Fault Injection Attacks. *Pre-print found at http://www. labri. fr/Perso/ly/index. htm*, 2003.

[2] R.J. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. *Lecture notes in computer science*, pages 125–136, 1998.

[3] C. Aumuller, P. Bier, W. Fischer, P. Hofreiter, and J.P. Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. *Lecture Notes in Computer Science*, pages 260–275, 2003.

[4] 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.

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

[6] 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.

[7] 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.

[8] KO GADELLAA. Fault Attacks on Java Card (Masters Thesis). *Universidade de Eindhoven*, 2005.

[9] O. Kommerling, M. Kuhn, and P. Street. Design principles for tamper-resistant smartcard processors.

[10] J.J. Quisquater and D. Samyde. Eddy current for magnetic analysis with active sensor. In *Proceedings of Esmart*, volume 2002, 2002.

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

[12] 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.