Emmanuel Prouff (Ed.)

LNCS 7079

Smart Card Research and Advanced Applications

10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011
Leuven, Belgium, September 2011
Revised Selected Papers

ifip

Springer

Emmanuel Prouff (Ed.)

Smart Card Research and Advanced Applications

10th IFIP WG 8.8/11.2 International Conference
CARDIS 2011
Leuven, Belgium, September 14-16, 2011
Revised Selected Papers

Springer

Volume Editor

Emmanuel Prouff
Oberthur Technologies
71-73 rue des Hautes Pâtures
92726 Nanterre Cedex, France
E-mail: e.prouff@oberthur.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-27256-1 e-ISBN 978-3-642-27257-8
DOI 10.1007/978-3-642-27257-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011943215

CR Subject Classification (1998): E.3, C.2, K.6.5, D.4.6, C.3, D.2

LNCS Sublibrary: SL 4 – Security and Cryptology

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

These proceedings contain the papers selected for CARDIS 2011, the 10th IFIP Conference on Smart Card Research and Advanced Applications hosted by the Katholieke Universiteit of Leuven, Belgium. Since 1994, CARDIS has been the foremost international conference dedicated to the security of smart cards and embedded systems. Initially biennial, the conference became annual in 2010 to take into account the very fast evolution of the smart card technology.

Security of smart cards is today an established and dynamic research area. Since 1994, CARDIS offers a privileged environment where the scientific community can congregate, present new ideas and discuss recent developments with both an academic and industrial focus. It covers a wide range of topics including hardware design, operating systems, systems modelling, cryptography and systems security. This year, the Program Committee of CARDIS refereed 45 submitted papers. Each paper was reviewed by at least 3 referees and the committee selected 20 papers to be presented at the conference. Two invited talks completed the technical program. The first one was given by Helena Handschuh, Chief Technology Officer at Intrinsic-ID, and the second one, by Olivier Ly, Associate Professor at LaBRI, University of Bordeaux.

There are many volunteers who offered time and energy to put together the symposium and who deserve our acknowledgement. I first would like to thank all the members of the Program Committee and the external reviewers for their hard work in evaluating and discussing the submissions. I am also very grateful to Vincent Rijmen, the General Chair of CARDIS 2011, and his team for the local conference management. I am particularly grateful to the CARDIS Steering Committee for allowing me to serve at such a recognized conference. Especially, I would like to say a big thank you to Jean-Jacques Quisquater for all the energy and hard work he put into the organization of this event.

Last, but certainly not least, my thanks go to all the authors who submitted papers and all the attendees. I hope you will find the proceedings stimulating.

September 2011 Emmanuel Prouff

Organization

CARDIS 2011 was organized by the Katholieke Universiteit of Leuven, Belgium.

Executive Commitee

Conference General Chair

Vincent Rijmen	Katholieke Universiteit Leuven, Belgium

Program Committee Chair

Emmanuel Prouff	Oberthur Technologies

Publicity Chair

Jean-Jacques Quisquater	Université Catholique de Louvain, Belgium

Program Committee

Gildas Avoine	UCL Crypto Group, Belgium
Guido Bertoni	STMicroelectronics, Italy
Sébastien Canard	Orange Labs, France
Odile Derouet	Samsung, Korea
Josep Domingo Ferrer	Rovari i Virgili University, Catalonia
Hermann Drexler	Giesecke And Devrient, Germany
Benoit Feix	Inside Secure, France
Benedikt Gierlichs	K.U. Leuven, ESAT-COSIC, Belgium
Sylvain Guilley	GET/ENST, CNRS/LTCI, France
Kimmo Järvinen	Helsinki University of Technology, Finland
Jean-Louis Lanet	University of Limoges, France
Thanh-Ha Le	Morpho, France
Stefan Mangard	Infineon Technologies, Germany
Konstantinos Markantonakis	University of London, UK
Amir Moradi	Ruhr University Bochum, Germany
Ventzi Nikov	NXP, Belgium
Elisabeth Oswald	University of Bristol, UK
Axel Poschman	Nanyang Technological University, Singapore
Emmanuel Prouff	Oberthur Technologies, France
Matthieu Rivain	CryptoExperts, France
Jörn-Marc Schmidt	IAIK TU Graz, Austria

Sergei Skorobogatov	University of Cambridge, UK
François-Xavier Standaert	UCL Crypto Group, Belgium
Pim Tuyls	Intrinsic-ID, The Netherlands
David Vigilant	Gemalto, France

Referees

J. Balasch	J. Daemen	L. Hoffmann
S. Riou	R. Trujillo-Rasua	M. Hutter
A. Barenghi	E. De Mulder	M. Kasper
M. Roussellet	G. Van Assche	C.H. Kim
E. Beck	F. De Santis	S.-K. Kim
S. Salgado	V. Van Der Leest	F. Koeune
A. Berzati	E. Dottax	S. Kutzner
G. Schrijen	N. Veyrat-Charvillon	M. Le Guin
S. Bhasin	I. Eichhorn	M. Medwed
M. Seysen	K. Villegas	F. Melzani
B. Brumley	J. Fan	O. Mischke
J. Soria-Comas	O. Farràs	D. Oswald
C. Capel	W. Fischer	P. Paillier
A. Séré	G. Gagnerot	T. Plos
C. Clavier	B.M. Gammel	B. Qin
P. Teuwen	J. Großschädl	M. Quisquater
I. Coisel	V. Guérin	F. Regazzoni
C. Troncoso	S. Hajian	

Sponsoring Institution

Oberthur Technologies

Table of Contents

Evaluation of the Ability to Transform SIM Applications into Hostile Applications

Guillaume Bouffard[1], Jean-Louis Lanet[1], Jean-Baptiste Machemie[1], Jean-Yves Poichotte[2], and Jean-Philippe Wary[2]

[1] SSD - XLIM Labs, University of Limoges,
83 rue d'Isle, 87000 Limoges, France
{guillaume.bouffard,jean-louis.lanet,jean-baptiste.machemie}@xlim.fr
[2] SFR, Direction Fraud and Information Security,
1 Pl. Carpeaux, 92915 Paris la Defense, France
{jean-philippe.wary,jean-yves.poichotte}@sfr.com

Abstract. The ability of Java Cards to withstand attacks is based on software and hardware countermeasures, and on the ability of the Java platform to check the correct behavior of Java code (by using byte code verification). Recently, the idea of combining logical attacks with a physical attack in order to bypass byte code verification has emerged. For instance, correct and legitimate Java Card applications can be dynamically modified on-card using a laser beam. Such applications become *mutant* applications, with a different control flow from the original expected behaviour. This internal change could lead to bypass controls and protections and thus offer illegal access to secret data and operations inside the chip. This paper presents an evaluation of the application ability to become *mutant* and a new countermeasure based on the runtime checks of the application control flow to detect the deviant mutations.

1 Introduction

A smart card usually contains a microprocessor and various types of memories: RAM (for runtime data and OS stacks), ROM (in which the operating system and the *romized* applications are stored), and EEPROM (to store the persistent data). Due to significant size constraints of the chip, the amount of memory is small. Most smart cards on the market today have at most 5 KB of RAM, 256 KB of ROM, and 256 KB of EEPROM. A smart card can be viewed as a secure data container, since it securely stores data and it is securely used during short transactions. Its safety relies first on the underlying hardware. To resist probing an internal bus, all components (memory, CPU, cryptoprocessor...) are on the same chip which is embedded with sensors covered by a resin. Such sensors (light sensors, heat sensors, voltage sensors, etc.) are used to disable the card when it is physically attacked. The software is the second security barrier. The embedded programs are usually designed neither for returning nor modifying sensitive information without guaranty that the operation is authorized.

Today, Mobile Network Operators (MNO) are looking to open their native secure element : the Universal Subscriber Identity Module (USIM) Card [10] to third party services providers, in order to allow them to develop value added services, like m-payment or m-transport for instance (on an NFC based technology). To be able to warrant the security of third parties applications hosted in USIM Card, MNO have choose to certify their Secure Element under the Common Criteria (CC) scheme, and two reference protection profiles have been developped [3], [4] (the CC targeted assurance level is EAL4+ for the USIM platform, it allows any applications to achieve EAL4+ CC level through the CC composition model [12]). To summarize the model USIM card will be certified under CC scheme at an EAL4+ level, and will allow post issuance third parties applications downloading without existing CC certification loss. It means that the industrial process to be put in place will have to warrant the inocuity of every candidate application to a download. The technology described in this paper may allow MNOs to evaluate the ability of every application candidate to download to become a mutant.

Today most of the SIM cards are based on a Java Virtual Machine. Java Card is a kind of smart card that implements the standard Java Card 3.0 [22] in one of the two editions “Classic Edition” or “Connected Edition”. Such a smart card embeds a virtual machine, which interprets application byte codes already romized with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [12]. This protocol ensures that the owner of the code has the necessary credentials to perform the action. Java Cards have shown an improved robustness compared to native applications regarding many attacks. They are designed to resist numerous attacks using both physical and logical techniques. Currently, the most powerful attacks are hardware based attacks and particularly fault attacks. A fault attack modifies part of the memory content or signal on internal bus and leads to deviant behaviour exploitable or not by an attacker. A comprehensive consequence of such attacks can be found in [14]. Although fault attacks have generally been used in the literature from a cryptanalytic point of view [5,13,16], they can be applied to every code layer embedded in a device. For instance, while choosing the exact byte of a program, the attacker can bypass countermeasures or logical tests. We call such modified applications "mutant".

Designing efficient countermeasures against fault attacks is important for smart card manufacturers but also for application developers. Manufacturers need countermeasures with the lowest cost in term of memory and processor usage. These metrics can be obtained with an evaluation of a target. As regards application developers, they have to understand the ability of their applets to become mutants and potentially hostile in case of fault attack. Thus the coverage (reduction of the number of mutant generated) and the detection latency (number of instructions executed between an attack and its detection) are the most important metrics. In this paper, we present a workbench to evaluate the ability of a given application to become a hostile applet with respect to the different implemented countermeasures, and the fault hypothesis.

In order to minimize the impact of fault attacks, developers need to implement countermeasures in applicative code. Examples of such applicative countermeasures are: redundant choices, counters, redefining the value of true and false constants, etc. But in this case the developer must have knowledge of the underlying platform architecture, which can differ from a smart card supplier to another one. This low interoperability of the security aspects between different platforms is a huge problem for smart card application development. The detection mechanism discussed in this paper is integrated to the system thus it ensures portability of the application code.

The contribution of this paper with respect to our prior work is twofold. We define a novel system countermeasure based on a verification by the virtual machine of the control flow and a framework to evaluate the efficiency of countermeasures. This paper is organized as follows: first we introduce a brief state of the art of fault injection attacks and existing countermeasures, then we discuss about the new countermeasure we have developed. In the fourth paragraph we present the evaluation framework and the collected metrics, and finally we conclude with the perspectives.

2 Fault Attacks and Countermeasures

Faults can be induced into the chip by using physical perturbations in its execution environment. These errors can generate different versions of a program by changing some instructions, interpreting operands as instructions, branching to other (or invalid) labels and so on. A fault attack has the ability to physically disturb the smart card chip. 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. Different manners to produce fault attacks [6] exist, most of them differ from the model of the attacker.

2.1 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 [8,24]. We describe in the table 1 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 [21] an attack using the precise bit error model. But it is not realistic on current smart cards, because modern components implement hardware security on memories like error correction and detection code or memory encryption.

In real life, an attacker physically injects energy in a memory cell to change its state. Thus and up to the underlying technology, the memory physically takes the value 0x00 or 0xFF. If memories are encrypted, the physical value becomes a random value (more precisely a value which depends on the data, the address, and an encryption key). To be as close as possible to the reality, we choose the precise byte error that is the most realistic fault model. Thus, we have assumed that an attacker can:

– make a fault injection at a precise clock cycle (he can target any operation he wants),
– only set or reset a byte to 0x00 or to 0xFF up to the underlying technology (bsr[1] fault type), or he can change this byte to a random value beyond his control (random fault type),
– target any memory cell he wishes (he can target a specific variable or register).

[1] Bit set or reset.

Table 1. 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	–

We have defined the hypotesis concerning the attacker, then we present the effects of such an attack on a Java Card if the attack targets the permanent memory generating a mutant application.

2.2 Defining a Mutant Application

The mutant generation and detection is a new research field introduced simultaneously by [7,23] using the concepts of combined attacks, and we have already discussed mutant detection in [20]. To define a mutant application, we use an example on the following debit method that belongs to a wallet Java Card applet. In this method, the user PIN (Personal Identification Number) must be validated prior to the debit operation.

Listing 1.1. Original Java code

```java
private void debit(APDU apdu) {
  if ( pin.isValidated() ) {
     // make the debit operation
  } else {
     ISOException.throwIt (SW_PIN_VERIFICATION_REQUIRED);
  }
}
```

Table 2. Byte Code representation before attack

Byte	Byte Code
00 : 18	00 : aload_0
01 : 83 00 04	01 : getfield #4
04 : 8B 00 23	04 : invokevirtual #18
07 : 60 00 3B	07: ifeq 59
10 : ...	10 : ...
...	...
59 : 13 63 01	59 : sipush 25345
63 : 8D 00 0D	63 : invokestatic #13
66 : 7A	66 : return

In table 2 resides the corresponding byte code representation (the full byte code representation can be found in section B). An attacker wants to bypass the PIN test. He injects a fault on the cell containing the conditional test byte code. Thus the `ifeq` instruction (byte 0x60) changes to a `nop` instruction (byte 0x00). The obtained Java code follows with its byte code representation in table 3:

Listing 1.2. Mutant Java code

```
private void debit(APDU apdu) {
      // make the debit operation
      ISOException.throwIt (SW_PIN_VERIFICATION_REQUIRED);
}
```

Table 3. Byte code representation after attack

Byte	Byte code
00 : 18	00 : aload_0
01 : 83 00 04	01 : getfield #4
04 : 8B 00 23	04 : invokevirtual #18
07 : 00	07 : nop
08 : 00	08 : nop
09 : 3B	09 : pop
10 : ...	10 : ...
...	...
59 : 13 63 01	59 : sipush 25345
63 : 8D 00 0D	63 : invokestatic #13
66 : 7A	66 : return

The verification of the PIN code is bypassed, the debit operation is made and an exception is thrown but too late because the attacker will have already achieved his goal. This is a good example of dangerous mutant application: "*an application that passes undetected through the virtual machine interpreter but that does not have the same behavior than the original application*". This attack has modified the control flow of the application and the goal of the countermeasure described in this paper is to detect when such modifications happen.

2.3 Hardware Countermeasures

Fault attacks are powerful and can threaten the card security. So smart card manufacturers have redoubled their efforts by integrating hardware protections that can prevent the fault attacks or make them more difficult to implement. This section is about hardware measures that can be found in the card. We do not claim to enumerate all the possible hardware measures that exist but to give a glimpse of the type of technology that resides in the card. A smart card contains two categories of hardware protection (see [11]):

– Passive protections increase the difficulty of a successful attack. These protections can use bus and memory encryption, random dummy cycles, unstable internal frequency, etc.
– Active protections include mechanisms that check whether tampering occurs and take countermeasures (possibly by locking the card or by generating hardware exceptions on the platform). Some active protection mechanisms are sensors (light, supply voltage, etc), hardware redundancy, etc.

Hardware countermeasures are a good way to protect the card, but are specialized, and they increase the smart card production cost.

2.4 Software Countermeasures

Software countermeasures are introduced at different stages of the development process; their purpose is to strengthen the application code against fault injection attacks. Software countermeasures can be classified by their end purpose:

- *Cryptographic countermeasures*: better implementation of the cryptographic algorithm like RSA (which is the most frequently used public key algorithm in smart cards), DES, and hash functions (MD5, SHA-1, etc.).
- *Applicative countermeasures*: only modify the application with the objective to provide resistance to fault injection. Generally, this class produces application with a greater size. Because beside the functional code (the code that process data), we have the security code and the data structure for enforcing the security mechanism embedded in the application. Java is an interpreted language therefore it is slower to execute than a native language (like C or assembler), so this category of countermeasures suffers of bad execution time and add complexity for the developer.
- *System countermeasures*: harden the system by checking that applications are executing in a safe environment. The main advantage is that the system and the protections are stored in the ROM, which is a less critical resource than the EEPROM and cannot be attacked thanks to checksum mechanisms that allow to identify modification of data that are stored in the ROM. Thus, it is easier to deal with integration of the security data structures and code in the system. Another thing that must be considered is the CPU overhead, if we add some treatments to the functional code.
- *Hybrid countermeasures*: are at the crossroads between applicative and system countermeasures. They consist in inserting data in the application that are used later by the system to protect the application code against fault attacks. They have a good balance between the increasing of the application size and the CPU overhead.

All previous categories with the exception of cryptography, use a generalist approach to detect the fault because they do not focus on a particular algorithm. The developed path check detection mechanism proposed in this paper is a hybrid countermeasure.

2.5 Control Flow Integrity

We have already proposed several solutions to check code integrity during execution using basic blocs check in a previous publication [19]. This paper is about the control flow integrity. In the previous description of a mutant we have shown how to bypass some checks but in [9] we describe how to modify the control flow of an application either by return oriented programming or using a laser beam. In both cases we succeed in executing arbitrary byte code.

The control flow integrity has been already studied for fault tolerance [17,15,18]. These methods attempt to discern program execution deviation from a registered static control flow graph; but they are not adapted to the smart card because of the memory and of the computation power required.

The authors in [2] proposed a generic framework which consists of macros to include in C code. The piece of program can help the programmer to have different structures and mechanisms to save and check various forms of execution history. For instance, a programmer can have a counter to check the number of iterations. One of the most complete methods verifies that execution follows predetermined paths computed during development and stored with the application. If an execution does not follow one of these paths then an error is raised and the platform can take appropriate steps (stop the program, lock the card, etc.). More precisely, the application maintains an execution history composed of a list of program points already passed. These program points have been set during development by the programmer, using macro to easily include code for execution history update. Then the programmer adds "checkpoints" at specific program points. During execution, the history is checked when the application reaches one of the checkpoints. The principle of this mechanism is simple and all valid paths are computed off-card. Significant inputs are required from the developer because he has to explicitly set the program points in the code; this leads to a high level of complexity because the developer needs to determine where program points are to be put, which invariants are to be checked and where these verifications are to be put. Moreover, because the detection mechanism is included in the application, a fault attack against the checking mechanism can bypass the detection phase. Another problem is the increase of the application size, because the list of valid histories and the detection mechanisms are contained inside the application. Thus, an application takes more space in the EEPROM, which is a rare commodity in the card.

Another technique, exposed in [1] is based on the same idea. It uses dynamic runtime checks to allow flow controls to remain within a given control flow graph. They propose to dynamically rewrite machine code of some byte code instructions to ensure that jumps go to a valid code location. To achieve this, an off-card application tags the targets of jump instructions with a unique label and saves the label of targeted instructions for each jump instruction. Then the jump instructions are modified to check during execution that jump instructions continue to one of the valid targeted instructions. But this application suffers from the same problems as the previous solution: it is an applicative countermeasure.

3 Checking Paths during Runtime Execution

Some points are important when designing a new countermeasure for smart card. This countermeasure must not disturb the application development workflow. It should keep the application size close to the original size. And it should not use up processor resources. To comply with these requirements, we have designed a lightweight software countermeasure that uses static byte code analysis to guarantee that at each step, the virtual machine interpreter follows an authorized path to access some resources.

3.1 Using Java Annotation

The proposed solution uses a security feature found in the Java Card 3 platform: annotations. But it is also fully applicable to Java Card 2 platforms using a preprocessor. When the virtual machine interprets the application code and enters a method or class tagged with a security annotation, it switches to a "secure mode". The code fragment that follows shows the use of annotations provided by Java Card 3 on the debit method. The `@SensitiveType` annotation denotes that this method must be checked for integrity with the `PATHCHECK` mechanism.

Listing 1.3. Java Annotation

```
@SensitiveType{
     sensitivity= SensitiveValue.INTEGRITY,
     proprietaryValue="PATHCHECK"
}
private void debit(APDU apdu) {
    if ( pin.isValidated() ) {
       // make the debit operation
    } else {
      ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
    }
}
```

With this approach, we provide a tool that processes an annotated class file. The annotations become a custom component containing security information. This is possible because the Java Card specification [22] allows adding custom components to a classfile: a virtual machine processes custom components if it knows how to use them or else, silently ignores them. But to process the information contained in these custom components, the virtual machine must be modified.

This approach allows to achieve a successful attack. With this mechanism, for an attacker to succeed, he must simultaneously inject two faults at the right time, one on the application code, the other on the system during its interpretation of the code (difficult to achieve) and outside the scope of the chosen fault model. Now we will detail the principle of the detection mechanism.

3.2 Principle of the "PATHCHECK" (PC) Method

The principle of the mechanism is divided in two parts: one part off-card and one part on-card. Our module works on the byte code, and has sufficient client computation power available because all the following transformations and computations are done on a server (off-card). It is a generic approach that is not dependent on the type of application. But it cannot be applied to native code such as cryptographic algorithm.

Off-card. The first step is to create the control flow graph of the annotated method (in the case that it is an annotated class the operation is repeated for all the methods belonging to the class), by separating its code into basic blocks and by linking them. A basic block is a set of uninterrupted instructions. It is ended by any byte code instruction that can break the control flow of the program (i.e. conditional or unconditional branch instructions, multiple branch instructions, or instructions that raise an exception). If this operation is applied to the debit method, we obtained the basic blocks represented in figure 1. Once the method is divided into basic blocks, the second step is to compute its control flow graph; the basic blocks represent the vertices of the graph and directed edges in the graph denote a jump in the code between two basic blocks (c.f. figure 3). The third step is about computing for each vertex that compounds the control flow graph a list of paths from the beginning vertex. The computed path is encoded using the following convention:

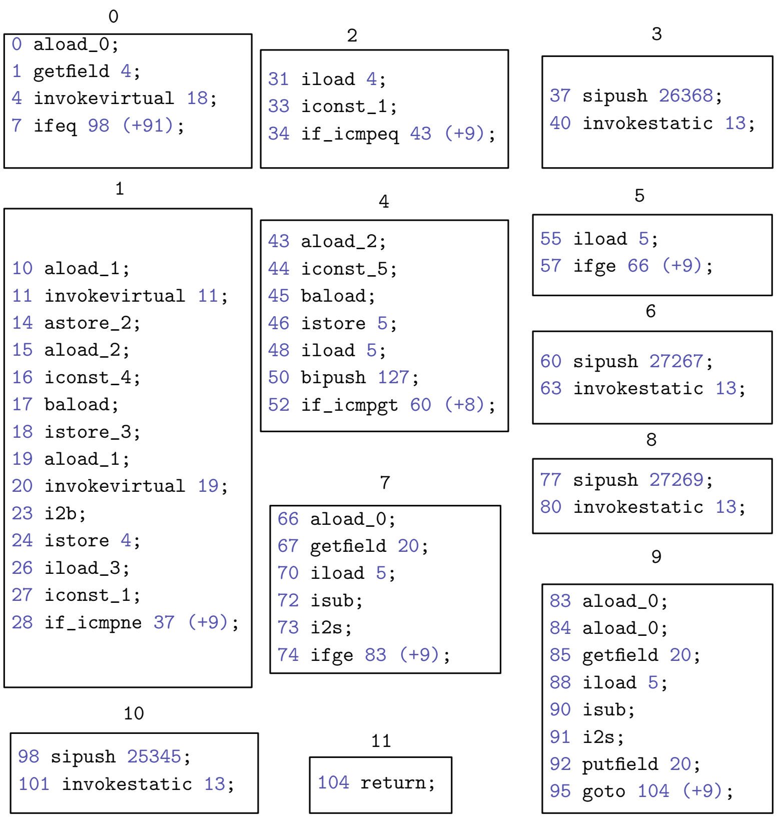
- Each path begins with the tag 01. This to avoid an attack that changes the first element of a path to 0x00 or to 0xFF.
- If the instruction that ends the current block is an unconditional or conditional branch instruction, a jump to the target of this instruction (represented by a low edge in 3) will use the tag 0 .
- If the execution continues at the instruction immediately following the final instruction of the current block (represented by a top edge in figure 3), then the tag 1 is used.

If the final instruction of the current basic block is a switch instruction, a particular tag is used, formed by any number of bits that are necessary to encode all the targets. For example, if we have four targets, we use three bits to code each branch (like in figure 2). Switch instructions are not so frequent in Java Card applications, but at least present in the ProcessApdu method. And to avoid a great increase of the application size that uses this countermeasure, they must be avoided.

Thus a path from the beginning to a given basic block is $X_0...X_n$ (where X corresponds to a 0 or to a 1 and n is the maximum number of bit necessary to code the path). In our example, to reach the basic block 9, which contains the update of the balance amount, the paths are : `01 0 0 0 0 0 0 1` and `01 0 0 0 0 0 1`.

On-card. When interpreting the byte code of the method to protect, the virtual machine looks for the annotation and analyzes the type of security it has to use. In our case, it is the path check security mechanism. So during the code interpretation, it computes the execution path; for example, when it encounters a branch instruction, when jumping to the target of this instruction then it saves the tag “0”, and when jumping to the instruction that follows it saves the tag “1”. Then prior to the execution of a basic block, it checks that the followed path is an authorized path, i.e a path belonging to the list of paths computed for this basic block.

For the basic block 9 this should be one of the two previous paths; if this is not the case, then it is probably because the interpreter followed a wrong path to arrive there. If a loop is detected (backward jump) during the code interpretation, then the interpreter checks the path for the loop, the number of references and the number of value on the operand stack before and after the loop, to be sure that for each round the path remains the same.



```
0
0 aload_0;
1 getfield 4;
4 invokevirtual 18;
7 ifeq 98 (+91);
```

```
1
10 aload_1;
11 invokevirtual 11;
14 astore_2;
15 aload_2;
16 iconst_4;
17 baload;
18 istore_3;
19 aload_1;
20 invokevirtual 19;
23 i2b;
24 istore 4;
26 iload_3;
27 iconst_1;
28 if_icmpne 37 (+9);
```

```
2
31 iload 4;
33 iconst_1;
34 if_icmpeq 43 (+9);
```

```
3
37 sipush 26368;
40 invokestatic 13;
```

```
4
43 aload_2;
44 iconst_5;
45 baload;
46 istore 5;
48 iload 5;
50 bipush 127;
52 if_icmpgt 60 (+8);
```

```
5
55 iload 5;
57 ifge 66 (+9);
```

```
6
60 sipush 27267;
63 invokestatic 13;
```

```
7
66 aload_0;
67 getfield 20;
70 iload 5;
72 isub;
73 i2s;
74 ifge 83 (+9);
```

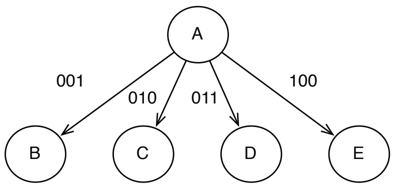
```
8
77 sipush 27269;
80 invokestatic 13;
```

```
9
83 aload_0;
84 aload_0;
85 getfield 20;
88 iload 5;
90 isub;
91 i2s;
92 putfield 20;
95 goto 104 (+9);
```

```
10
98 sipush 25345;
101 invokestatic 13;
```

```
11
104 return;
```

Fig. 1. Basic blocks of the debit method



$$\begin{tikzcd}
 & & A \arrow[dll, "001"'] \arrow[dl, "010"'] \arrow[dr, "011"] \arrow[drr, "100"] & & \\
B & C & & D & E
\end{tikzcd}$$

Fig. 2. Coding a switch instruction

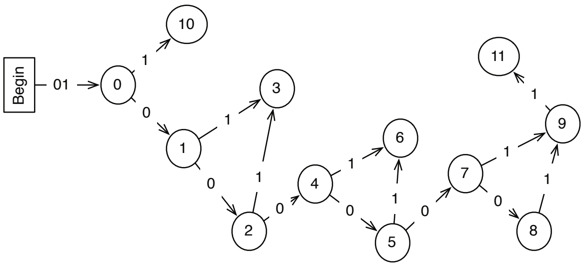


Fig. 3. Control flow graph of the debit method

4 Experimentation and Results

4.1 Evaluation Context

Two Java Card applets have been used for the evaluation. Those two cardlets are representative of the type of code that a MNO may want to add to their USIM Card. The first (AgentLocalisation) is oriented geolocalization services, this cardlet is able to detect when the handset (the device in which the USIM card is inserted) is entering or leaving a dedicated or a list of geographical dedicated cells (each cell is identified through a CellID value, which is stored on the USIM interface) and then sends a notification to a dedicated service (registred and identified in the cardlet). The second is more specialised 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.

Evaluating Resources Consumption. The first category of metrics is the memory footprint and the CPU overhead. They have been obtained using the SimpleRTJ Java virtual machine that targets highly restricted constraints device like smart cards. The hardware platform for the evaluation is a board which has similar hardware as the standard smart cards.

These metrics are very important for the industry because the size of the used memories directly impacts the production cost of the card. In fact, applications are stored in the EEPROM that is the most expensive component of the card.

The CPU overhead is also important because most of the time, when challenging the card for some computation a quick answer is needed. So when designing a countermeasure for smart cards, it is important to have these properties in mind. To obtain the metrics in table 4, the PATHCHECK countermeasure has been implemented on an embedded system that has similar properties as common smart card.

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 interpreter can also simulate an attack by modifying the method's byte array. This is important because it allows us to reproduce faults on demand. In addition to the abstract interpreter, we have developed a mutant generator. This tool can generate all the mutants corresponding to a given application according to the chosen fault model. To achieve this, for a given opcode, the mutant generator changes its value from 0x00 to 0xFF, and an abstract interpretation is performed for each of these values. If the abstract interpretation does not detect a modification then a mutant is created enabling us to regenerate the corresponding Java source file and to color the path that lead to this mutant.

The mutant generator has a number of modes of execution:

- *The Basic Mode (BM)*: the interpreter executes, without running any checks, the instruction push and pop elements on the operands stack and using local variables. With this configuration, instructions can use elements of other methods frame like using their operands stack or using their locals. When running this mode, no countermeasures are activated.
- *The Byte Code Verifier mode (BCV mode)*: the interpreter checks that no overflow or no underflow occurs, that the used locals are inside the current table of locals, and that when a jump occurs it takes place inside the method. They consist in some verifications done by the BCV.
- *The advanced mode*: the simple mode plus the ability to activate or to deactivate a given countermeasure like the developed ones: path checking mechanism (PC), field of bits mechanism (FB) see [19], or PS mechanism. PS is a detection mechanism that is not described in this paper and for which a patent is pending.

Graphical User Interface (GUI). We have created a GUI for the mutant generator, that allows a user to choose a specific application, to parameterize the vulnerability analysis, and to navigate through the generated mutant applications showing the mutation that has led to it. This allows a security officer to have all the useful information in a user friendly environment.

We also provide a web application that allows a registered user to upload an application binary file, and to choose between the different analysis mode seen previously.

4.2 Results

Resources Consumption. Table 4 shows the metrics for resources consumption obtained by applying the detection mechanism to all the methods of our test applications. The increasing 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 % increasing of application size and 8 % of CPU overhead, the developer can choose when to activate only for sensitive methods to preserve resources. This countermeasure needs small changes on the virtual machine interpreter if we refer to the 1 % increment. So, we can conclude that it is an affordable countermeasure.

Mutant Detection and Latency. Tables 5, 6 and 7 show the reduction of generated mutants in each mode of the mutant generator for three applications. The second line

Table 4. Ressources consumption

Countermeasures	CPU	EEPROM	RAM	ROM
Path check	+ 8 %	+10 %	<1 %	1%
Field of bits	+ 3 %	+ 3 %	<1 %	1%
Basic block	+ 5 %	+15%	<1%	1%

shows the number of mutants (1) generated in each mode of the mutant generator. The third line of those tables shows the latency (2). The obtained results show the efficiency of the developed countermeasures.

The latency is the number of instructions executed between the attack and the detection. In the basic mode, no latency is recorded because no detection is made. This value is also really important because if a latency 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.

Table 5. Wallet (simple class) - 470 Instructions

	BM	BCV	PS	FB	BB	PC
(1)	440	54	30	10	0	37
(2)	-	2,91	2,92	2,43	2,72	2,42

Path check 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 it. With this countermeasure it becomes impossible to bypass systems calls like cryptographic keys verification. And if some mutants remain, applicative countermeasures can be applied on demand to detect them.

Table 6. SfrOtp (simple class) - 4568 of instructions - 9136 attacks

	BM	BCV	PS	PC	FoB	BB
(1)	7960	94%	95%	86%	99%	100%
(2)	-	3,64	3,56	17.18	8.61	12

Table 7. AgentLocalisation (simple class) - 3504 instructions - 7008 attacks

	BM	BCV	PS	PC	FoB	BB
(1)	7960	94%	99%	88%	99%	100%
(2)	-	11.8	12.1	2.43	10.20	13

5 Conclusions

In this paper, we presented a new approach that is affordable for the card and that is fully compliant with the Java Card 2.x and 3.x specifications. Moreover it does not consume too much computation power and the produced binary files increases in an affordable way. It does not disturb the applet conception workflow, because we just add a module that will make a lightweight modification of the byte code. It saves time to the developer looking to produce secured applications thanks to the use of the sensitive annotation. Finally, it only requires that a slight modification be made to the Java Virtual Machine. It also has a good mutant applications detection capacity.

We have implemented all these countermeasures inside a smart card in order to have metrics concerning memory footprint and processor overhead, which are all affordable for smart cards. In this paper, we presented the second part of this characterization to evaluate the efficiency of countermeasures in a smart card operating system. We provide a framework to detect mutant applications according to a fault model and a memory model. This framework is able to provide to a security evaluator all the source code of the potential mutant of the application; he can decide if there is a threat with some mutants and then to implement a specific countermeasure. We applied this approach on SIM card applications but it could be applied to every Java Card application.

With this tool, both the developer and security evaluator can take informed decisions concerning the security of its smart card application. For the developer company, reducing the size of the embedded code minimizes the cost of the application. For the security evaluator it provides a semi automatic tool to perform vulnerability analysis.

References

1. Abadi, M., Erlingsson, M.B.U., Ligatti, J.: Control-flow integrity. In: CCS 2005: Proceedings of the 12th ACM Conference on Computer and Communications Security, Alexandria, Virginia, USA, November 7-11, p. 340. Citeseer (2005)
2. Akkar, M.L., Goubin, L., Ly, O.: Automatic integration of counter-measures against fault injection attacks. Pre-print found (2003), http://www.labri.fr/Perso/ly/index.htm
3. ANSSI. Protection Profile (U)SIM Java Card Platform Protection Profile - Basic Configuration, ANSSI-CC-PP-2010/04 12/07/2010. ANSSI (2010)
4. ANSSI. Protection Profile (U)SIM Java Card Platform Protection Profile - SCWS Configuration, ANSSI-CC-PP-2010/05, 12/07/2010. ANSSI (2010)
5. Aumuller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.P.: Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 260–275. Springer, Heidelberg (2003)
6. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE 94(2), 370–382 (2006)
7. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg (2010)
8. Blomer, J., Otto, M., Seifert, J.P.: A new CRT-RSA algorithm secure against Bellcore attacks. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, pp. 311–320. ACM, New York (2003)

9. Bouffard, G., Cartigny, J., Lanet, J.-L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011)
10. ETSI. 3GPP TS 31.102, Technical Specification Group Core Network and Terminals. ETSI (2005)
11. Gadella, K.: Fault Attacks on Java Card (Masters Thesis). Master thesis, Universidade de Eindhoven (2005)
12. GP. Global platform official site (2010)
13. Hemme, L.: A Differential Fault Attack Against Early Rounds of (Triple-)DES. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 254–267. Springer, Heidelberg (2004)
14. Iguchi-Cartigny, J., Lanet, J.L.: Developing a trojan applet in a smart card. Journal in Computer Virology 6(4), 343–351 (2010)
15. Oh, N., Shirvani, P.P., McCluskey, E.J., et al.: Control-flow checking by software signatures. IEEE Transactions on Reliability 51(1), 111–122 (2002)
16. Piret, G., Quisquater, J.-J.: A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 77–88. Springer, Heidelberg (2003)
17. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: Swift: Software implemented fault tolerance. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 243–254. IEEE Computer Society, Washington, DC, USA (2005)
18. Scott, K., Davidson, J.: Safe virtual execution using software dynamic translation. In: Proceedings of the 18th Annual Computer Security Applications Conference, p. 209. Citeseer (2002)
19. Sere, A.A., Iguchi-Cartigny, J., Lanet, J.-L.: Automatic detection of fault attack and countermeasures. In: Proceedings of the 4th Workshop on Embedded Systems Security, pp. 1–7. ACM (2009)
20. Sere, A.A., Iguchi-Cartigny, J., Lanet, J.-L.: A path check detection mechanism for embedded systems. In: Proceedings of SecTech 2010, vol. 6485, pp. 459–469 (2010)
21. Skorobogatov, S.P., Anderson, R.J.: Optical Fault Induction Attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 2–12. Springer, Heidelberg (2003)
22. SunMicrosystems. Java Card 3.0.1 Specification. Sun Microsystems (2009)
23. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 133–147. Springer, Heidelberg (2010)
24. Wagner, D.: Cryptanalysis of a provably secure crt-rsa algorithm. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, pp. 92–97. ACM, New York (2004)

A Full Java Code of the Debit Method

```
Private void debit(APDU apdu) {
  // access authentication
  if ( pin.isValidated() ) {
    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);
    } else {
      ISOException.throwIt( SW_PIN_VERIFICATION_REQUIRED);
    }
}
```

B Full Byte Code Representation of the Debit Method

```
private void debit(APDU buffer) {
    0 aload_0;
    1 getfield 4;
    4 invokevirtual 18;
    7 ifeq 91;
    10 aload_1;
    11 invokevirtual 11;
    14 astore_2;
    15 aload_2;
    16 iconst_4;
    17 baload;
    18 istore_3;
    19 aload_1;
    20 invokevirtual 19;
    23 i2b;
    24 istore 4;
    26 iload_3;
    27 iconst_1;
    28 if_icmpne 9;
    31 iload 4;
    33 iconst_1;
    34 if_icmpeq 9;
    37 sipush 26368;
    40 invokestatic 13;
    43 aload_2;
    44 iconst_5;
    45 baload;
    46 istore 5;
    48 iload 5;
    50 bipush 127;
    52 if_icmpgt 8;
    55 iload 5;
    57 ifge 9;
    60 sipush 27267;
    63 invokestatic 13;
    66 aload_0;
    67 getfield 20;
    70 iload 5;
    72 isub;
    73 i2s;
    74 ifge 9;
    77 sipush 27269;
    80 invokestatic 13;
    83 aload_0;
    84 aload_0;
    85 getfield 20;
    88 iload 5;
    90 isub;
    91 i2s;
    92 putfield 20;
    95 goto 9;
    98 sipush 25345;
    101 invokestatic 13;
    104 return;
}
```

Synchronized Attacks on Multithreaded Systems - Application to Java Card 3.0 -

Guillaume Barbu[1,2] and Hugues Thiebeauld[3,⋆]

[1] Oberthur Technologies, Innovation Group,
Parc Scientifique Unitec 1 - Porte 2,
4 allée du Doyen George Brus, 33600 Pessac, France
[2] Institut Télécom / Télécom ParisTech, CNRS LTCI,
Département COMELEC,
46 rue Barrault, 75634 Paris Cedex 13, France
[3] RFI Global Services Ltd,
Pavilion A, Ashwood Park, Ashwood Way,
Basingstoke, Hampshire, RG23 8BG, United Kingdom

Abstract. Up to now devices in charge of performing secure transactions mainly remained limited regarding their functionalities. However the trend has recently gone towards an increasing integration of features and technologies, which could potentially represent a source of additional threats. This article introduces an innovative attack exploiting advanced functionalities and offering unrivalled opportunities. This attack targets specifically the multithreaded systems featuring network capabilities. By the way of a network flooding we show how a process can be interrupted at the precise time a sensitive operation is being executed. This interruption aims at subsequently modifying the execution context and consequently breaking the sensitive operation. The practical feasibility of this attack is illustrated on a Java Card 3.0 Connected Edition platform. This description reveals that going through with the full attack scenario is not obvious. However this apparent complexity must not conceal the potential breach, which may significantly alter any application running on the system. Finally the goal of this work is to emphasize that the increasing products complexity may generate new security issues rather than to highlight a specific weakness on released products.

Keywords: Fault Injection, Logical Attack, Multithreading, Network Flooding, Java Card 3, Technological Convergence.

1 Introduction

Every software developer knows the usefulness of debug sessions in an application development process. To be in a position to debugging, the developer needs to use some specific tools enabling him to set breakpoints in the middle of a code execution. He has then the ability to control the internal process flow by

⋆ Part of this work done while at Oberthur Technologies.

modifying the execution context. Such a capability on released device would represent an outstanding breach. Indeed, it would allow to alter any application running on the defeated system.

This article introduces a new attack revealing a malicious mean to set a breakpoint. This attack targets multithreaded systems with network capabilities. By exploiting a network flooding we demonstrate how a multithreaded system can be abused to artificially freeze an application during a sensitive operation process. Combined with a temporary illegal access to the execution context in memory, we describe how the context modification may alter the security of the application when its process is resumed.

As a proof of concept we detail a practical attack on a Java Card. The recently released Java Card 3.0 Connected Edition specifications [1,2,18,3] associate the multithreading and network connectivity features with the Java Card technology. To fit with the particular context of secured Java Cards, our description reveals that some inherent system protections must be circumvented requiring a fault injection [4,5,6,7]. In spite of its apparent complexity we have successfully implemented the attack to alter the security of a Java Card web application. We also present different ways to withstand such attacks.

Beyond the practicability of the attack this work seeks to highlight that any feature is a potential source of attack, even though the links with security are not obvious. The article is subsequently organized as follows : In Section 2, we briefly present the involved mechanisms. Then we expose the generic attack concept in Section 3 and detail in Section 4 a complete attack path on a JC3.0. Finally, in Section 5 we discuss the protections preventing this attack and we use this example to emphasize the importance of the implementation to fit a high level of security.

2 Involved Mechanisms

As stated above, a prerequisite for our attack concept is the support of multithreading and network communications over standard protocols. This section intends to outline these mechanisms to set the basement of our work.

2.1 Multithreading

Our concept is grounded on the multithreading capacity of the targeted system, which allows the concurrent execution of different processes. In this work, we only consider multithreading on single-core devices. To process several threads simultaneously, the system assigns resources to a thread for a given time slice before switching to another. The entity in charge of distributing resources to the threads is the scheduler. Numerous rules can be used to decide when the scheduler will order a thread switching, *i.e.* to set the size of the so-called time slice. For instance thread switching can be triggered by a timer, an instruction counter, control flow breaks, access to certain resources, *etc.*

When a thread has consumed its allocated time slice, the scheduler orders a switch. This switch should be processed as follows :

– The current thread's execution context is saved.
– The next thread is elected and its execution context is loaded.[1]

The different threads are then successively given access to the system resources.
The attack concept does not directly target the multithreading but rather lies on an abuse of this feature. Next section introduces the feature we take advantage of to achieve this abuse.

2.2 I/O Network Interfaces

We consider in the scope of this work a device providing a logical network interface supporting standard network protocols (TCP, HTTP(S), ...) over physical I/O interfaces. The aim of this section is to set the system architecture assumed in the remainder of this work.

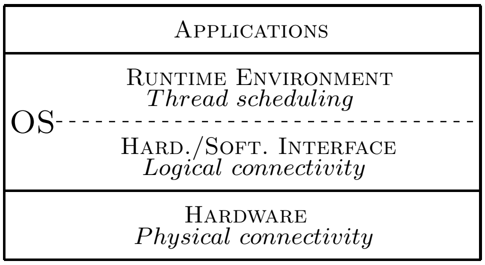


Fig. 1. Alleged architecture of the system

As depicted in Fig. 1, these interfaces are most likely not part of the so-called runtime environment (RE), but belong to lower layers. According to this statement, we can expect that some incoming requests are handled in these lower layers only and do not reach the RE. Therefore they do not enter the system's multithreading mechanism.

This last statement is a key element rendering our attack concept practicable, regardless of the targeted system. For the sake of clarity, Fig. 2 illustrates it with different requests. As depicted on the figure, requests $\mathcal{B}$ and $\mathcal{D}$ handling leads to a thread creation within the RE, whereas requests $\mathcal{A}$ and $\mathcal{C}$ are handled by the system before entering the RE.

Now we have set the basement of our attack concept, introducing the required mechanisms and properties of the targeted system, the attack concept itself can be exposed.

[1] The next thread election can possibly take into account the concept of thread priority. This concept will not be further considered in our context since an attacker able to start new threads should also have the ability to modify their priority.

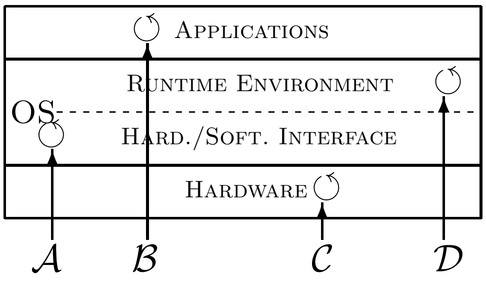


Fig. 2. Different requests handling

3 The Attack Concept

This section introduces the attack principle in a generic way. The goal is to emphasize that this threat may potentially affect a wide range of systems. On the other hand we will see that the attack success is closely related to implementation choices in the platform, providing some leads to find adequate protections later on.

The attack aims at altering a sensitive execution flow at a precise time. To achieve this, two steps must be performed as follows :

– Freezing the application execution at time T_0. It comes to setting a breakpoint on a specific operation.
– Altering the execution context available in memory to change the application behaviour when it is resumed.

In the scope of this article, we assume that the thread scheduler is based upon a timer. A certain amount of time is then allocated to each thread. When an execution exceeds the time slice T, the scheduler stops the process and switches to the next thread in the queue. This hypothesis is obviously not the single way to implement multithreading. However for the sake of clarity, we intentionally focus our description on one kind of scheduler. The adaptation of this attack to other schedulers may be subject to future works.

Our attack relies on the corruption of the multithreading system to force the interruption of an application. The objective is to cheat the scheduler, so that the thread switch occurs at T_0 rather than T. This is obtained by sending a sequence of requests to the device when the targeted thread is being processed. As the process in charge of the network requests is unlikely to be handled as a thread, the time of processing initially devoted to the current thread is lost. As a consequence the thread execution is curtailed, as depicted in Fig. 3.

T_0 is then adjustable depending on the number of network requests sent. To appropriately determine T_0, it is better to have an idea of the thread execution flow. Nevertheless, the code knowledge may not be necessary as side channel analysis may provide sufficient information, depending on the attacked system.

Once the targeted thread is frozen, the scheduler switches to the next ones in the queue. During this time the context of the sensitive code is available in

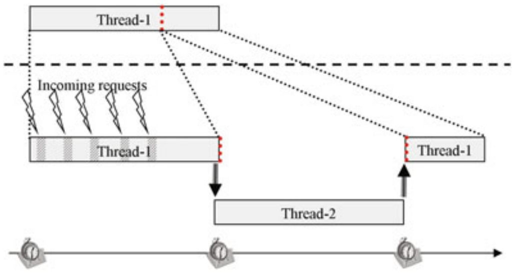


Fig. 3. Normal (up) and curtailed (down) execution of a thread

memory. An attacker in position of executing a malicious application would have then the opportunity to get to the context in memory and alter it. Depending on the attacked system, the right to load and run a application can be more or less restrictive. However in the context of a multi applicative platform these rights necessarily exist, indeed can be forced.

The remaining issue to achieve this attack is memory access. In some open systems the volatile memory remains fully available. But some systems isolate the memory access to respective areas. Therefore this restriction does not allow a thread to get to the execution contexts. To successfully perform the attack, the isolation mechanism must be overcome. The next section illustrates how such a protection has been circumvented on a Java Card by the mean of a physical perturbation.

Once memory access is obtained, a full range of possibility is offered to the attacker. The control of the execution context of the thread gives access to its program counter, local variables and execution stack. Although access to these data obviously stands as a compromission of the system, an attacker has no guarantee that she will be able to take benefit of it. On the other hand, provided she has properly adjusted T_0 , well-chosen alterations would break almost any security operation. The complete attack scenario is depicted on Fig. 4.

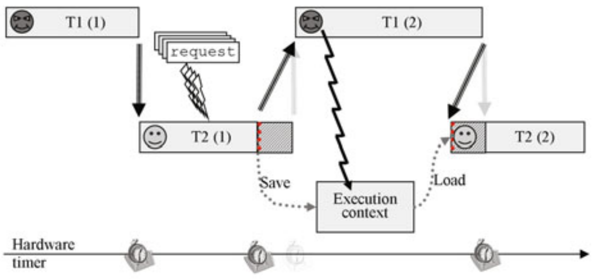


Fig. 4. The complete attack scenario

The attack potentially concentrates several issues which strongly depends on the kind of attacked system. But its consequences may be tragic for an application, even if the code has been proficiently secured. Furthermore this attack also underlines that the security of an application has a value only if the platform underneath is secured enough.

4 Practical Implementation on a Java Card

This section details the full attack scenario we have put into practice on a recent device to illustrate the feasibility of this concept and outline its consequences.

4.1 Context of the Attack

The Attacked Platform. With regards to the different features involved in the attack, a device implementing the JC3.0 specifications appears to be a potential target. Indeed, it is a security device offering both multithreading and network communication support. Furthermore, such platform may allow post-issuance application loading, as long as the application is well-formed.

The Target Application. We consider in the remainder of this section an application $\mathcal{T}$ offering sensitive services. Access to those services requires an authentication, achieved through a signature. $\mathcal{T}$ then contains the following lines of code (or equivalent) :

```
if (sig.verify(inBuf,inOf,inLen,sigBuf,sigOf,sigLen) != true)
  accessDenied();
else
  accessGranted();
```

The bytecode sequence that is actually executed on-card is then :

```
0E. invokevirtual #4    <javacard/security/Signature.verify>
11. iconst_1
12. if_icmpne 0x1C (+10)
15. aload_0             <app/Target this>
16. invokespecial #5    <app/Target.accessDenied>
19. goto 0x20 (+7)
1C. aload_0             <app/Target this>
1D. invokespecial #6    <app/Target.accessGranted>
```

Attack Goal. The attack aims at gaining access to the sensitive services without producing a valid signature.

4.2 The Attack Concept Key Assumptions

We have stated in the previous section that the success of our attack concept relies on a couple assumptions validity. This section details the validation of these assumptions on the attacked platform.

Loading the Attack Application. Loading application on the platform is not an obvious right for Java Card users. This capacity is generally limited by the knowledge of authentication keys through GlobalPlatform [8]. However, we consider that an attacker may be able to load an application. This ability can have various origins :

– The load keys are known (this knowledge being either legitimate or not).
– One or several fault injection(s) can lead to a breach in the GP implementation on the card.

Loading and executing a malicious application $\mathcal{A}$ have two rationales. First it should permit the modification of the targeted thread's execution context. But it is also in charge of ensuring the expected thread scheduling scenario. Details of its implementation are given along the attack path.

How to Access and Corrupt the Java Frame. The first challenge is to access the memory and to identify the execution context. Considering a JC3.0, we are interested in the following elements (refer to [9] for a detailed description of Java *Threads* and *Runtime Areas*) :

– the Java program counter : the address of the currently executing instruction in the current method of the current frame.
– the stack of frames : a frame is pushed onto the stack when a method is invoked and is popped out when this method completes.
– in the frames : the local variables and the operand stack.

We intend to reach these values by forging a fake byte array. For that matter, we consider a type confusion provoked by means of a fault injection.

The fault attack. At CARDIS'10, Barbu et al. proposed a combined attack provoking a type confusion and permitting to forge an object's reference and content [10]. In our context, this attack turns out to have a couple of advantages :

– A single physical attack of the device is required, a perturbation during the execution of a `checkcast` instruction for instance.
– Since forged references are persistent :
  - The fault injection can be the first step of our attack scenario.
  - Once one perturbation has been successful, a failure in the following steps will not require to start again from scratch.

We successfully applied this technique to the attacked platform. The physical perturbation was achieved using a laser. Its success depends on a couple of parameters (time, location and wavelength of the beam) found experimentally.

Accessing the Java frame. We assume that execution contexts are saved in volatile memory on thread switching. The type confusion is used to forge a byte array in memory in order to access the execution context of $\mathcal{T}$. We also assume the internal representation of a Java array contains a pointer (say a 32-bit word) to its content in memory, as exposed in [11] and depicted within Fig. 5.

To build a fake array, we only have then to set our "confused object" fields to appropriate values. Then, we expect to be able to access the memory as if it was the content of the forged array. This process is illustrated in Fig. 5.[2]

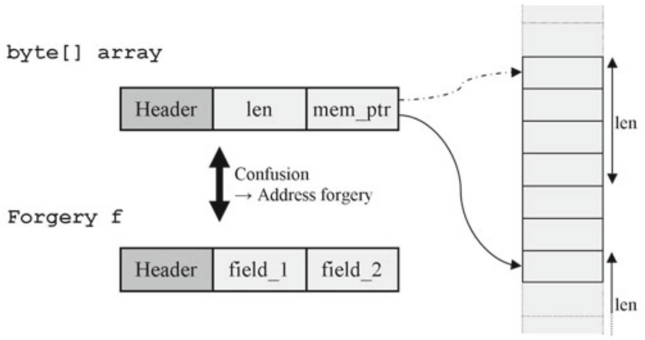


Fig. 5. Confusion between instance of two classes in order to forge an array's address

Once we have gained access to memory, we need to identify the frame within the forged array and to figure out its internal structure.

Finding and learning the structure of the Java frame. To locate the frame, we can take advantage of a straightforward linear memory allocation mechanism. According to the scheduling scenario of our attack concept, initializing a new array with obvious values when $\mathcal{A}$ is resumed permits to delimit the memory used by the targeted application. Furthermore, we have run a training session of our attack in order to learn the structure of frames on the platform. For that matter, we have built a target application that interrupts itself with easy-to-detect `short` values in local variables (`0x1903` for instance) and on the operand stack (`0x1902` for instance). The dump array obtained when $\mathcal{A}$ is resumed is depicted in Fig. 6.

We can then detect the frame and gain sufficient information on its structure :

- <number of local variables : nb_loc> <nb_loc * local variables>
- <maximum stack size : max_stk> <max_stk * operand values>
- <current top of stack>
- <jpc>

How Ping Flooding Affects Application Execution. We have stated in Section 2.2 that some incoming requests do not require the attention of the JCRE. An Internet Control Message Protocol (ICMP) *echo* request (a ping) is a typical example of such a request. Our claim is that when a ping request is incoming, the processor handles it whereas in the meantime the scheduler's timer is still running. We can then manage to shorten a thread's execution as

[2] See Appendix A.1 for implementation details.

```
0x0000 : 55 55 55 55  55 55 XX XX  XX XX XX XX  XX XX XX XX
                           <proprietary data>

0x0030 : XX 14 00 01  00 BA E2 01  00 E2 04 01  00 F2 45 00
0x0040 : 00 00 70 00  00 03 19 00  00 03 19 00  00 03 19 00
0x0050 : 00 03 19 00  00 03 19 00  00 03 19 00  00 03 19 00
0x0060 : 00 03 19 12  E1 53 12 00  12 E1 08 00  00 00 02 19
0x0070 : 00 00 02 19  00 00 02 19  00 00 02 19  00 00 02 19
0x0080 : 00 00 02 19  00 00 02 19  00 00 02 19  00 00 20 00
0x0090 : 00 00 49 00  XX XX XX XX  XX XX XX XX  XX XX XX XX
                           <proprietary data>
0x00D0 : XX XX XX XX  XX XX XX XX  XX 55 55 55  55 55 55 55
0x00E0 : 55 55 55 55  55 55 55 55  55 55 55
```

Fig. 6. Memory dump from the forged array

we please, the number of instructions actually executed within a time slice being reduced. To validate this claim, we have run a thread incrementing a counter on the attacked platform. Fig. 7 presents the value reached by the counter after a given amount of time against the number of pings sent in the same time. This proves that the number of instruction executed within the thread is reduced when the system is flooded with pings, since the value reached by the counter is representative of the number of instructions executed.[3]

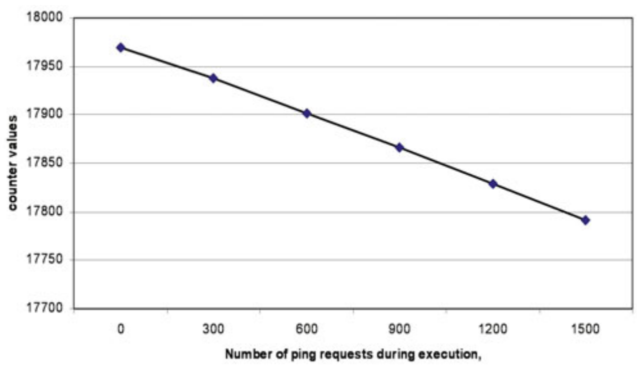


Fig. 7. Influence of communication on instructions execution

This technique could be assimilated to a well-known attack in the network security field : *ping flooding* [12,13]. Ping flooding usually aims at consuming the bandwidth of the targeted system in order to provoke a *Denial of Service* (DoS). Our approach is different as the aim is here to consume the time allocated to the targeted thread in order to curtail it.

[3] Implementation details are given in Appendix A.2.

4.3 The Practical Attack

The attack is divided into three steps detailed within this section.

– In the first step, we achieve the preliminary work to ensure both the access to $\mathcal{T}$'s frame and the scheduling scenario;
– In the second step, we force the "breakpoint hitting" with I/O flooding;
– In the third step, we use the fake array to corrupt $\mathcal{T}$'s frame.

Preliminaries. With regards to the global illustration of the attack concept, this step corresponds to the first segment of the "evil" thread's execution (T1(1) in Fig. 4). The aim of this step is to procure a way to access the memory where the execution context of $\mathcal{T}$ will be stored. This is achieved as presented in Section 4.2.

To ensure the predicted thread scheduling scenario, the application only has to start a new thread, and force its interruption for a certain amount of time (via the `Thread.sleep()` method). Within that time, $\mathcal{T}$ is launched in a new thread. On the next thread switching, $\mathcal{A}$'s thread will then become active again. This last statement implicitly assumes that no other thread is concurrently running on the platform, or at least that the attacker's thread will be the next one to be executed. We can then focus on $\mathcal{T}$'s execution and when to force its interruption.

Setting the Breakpoint. The aim of this step is to force a thread switching at a precise point during $\mathcal{T}$'s execution. It corresponds to the first segment of the targeted thread's execution of the attack concept illustration (T2(1) in Fig. 4). The challenge at this step is to "synchronize" the pings and the thread's execution. Working on a smartcard, the power consumption analysis can again reveal a strong ally at this step. Actually, we can monitor bytecode instruction execution through the power consumption of the card (as stated in [14]). Therefore, the exact knowledge of the code does not appear necessary to achieve the attack.

The attacked platform comes within a USB smart card connector and communicates as an Ethernet Emulation Model (EEM) device according to the specification [15]. The first task is then to adapt our power consumption acquisition module to monitor power consumption behind the USB smart card connector where the Java Card is plugged (*cf.* Fig. 8).

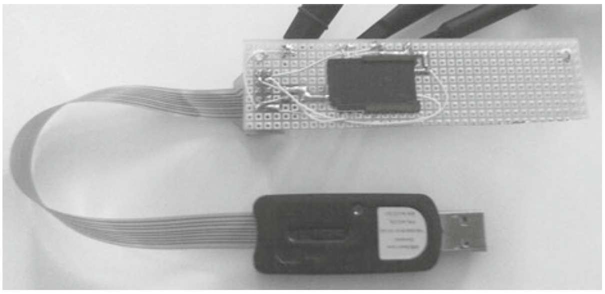


Fig. 8. USB smartcard acquisition module

We can then achieve and monitor the ping flooding of $\mathcal{T}$.

Fig. 9 shows the power traces of $\mathcal{T}$'s execution. On the first power trace, the signature verification is easily identified. The following traces depicts the same execution with an increasing number of ping requests (the numerous peaks on the traces). As we can see, the cryptographic operation is executed more or less shifted depending on the number of pings received during the thread's execution.

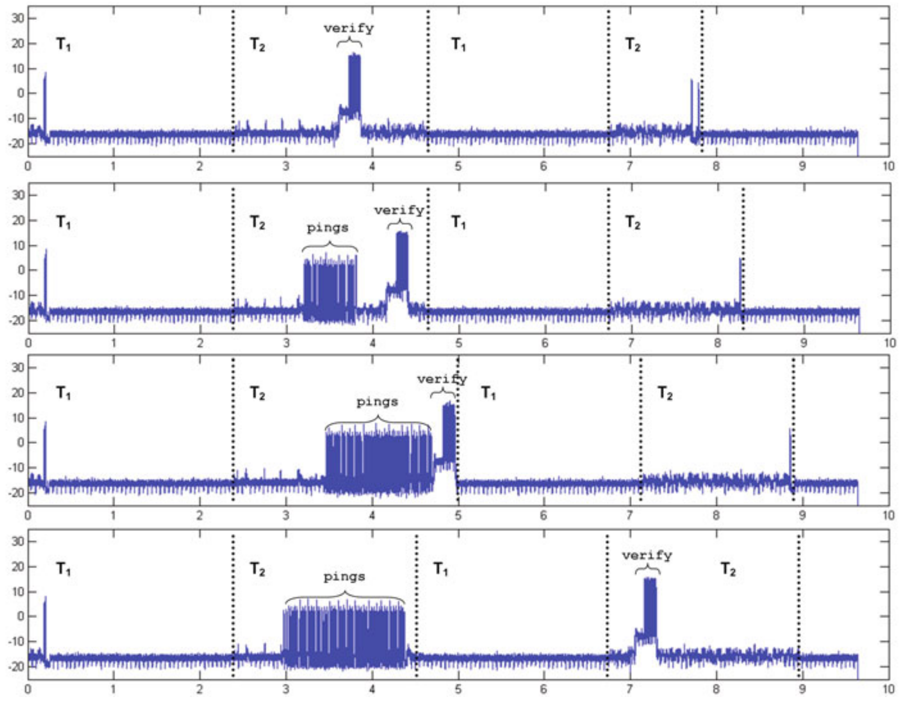


Fig. 9. Execution of the two threads with various number of ping requests (respectively 0, 10, 30 and 40). T_1 and T_2 refer respectively to the attacker's and the target thread.

Based on experimentations, an average sequence of 37 ping requests during the execution of $\mathcal{T}$ causes its interruption after the `verify` method returns but before the execution of the conditional branching. This corresponds to the third power trace in the figure. Actually other "breakpoints" may also allow an attack.

The previous section has given us a mean to read/write the volatile memory. In this section we have exposed how we manage to set the so-called breakpoint within the attacked application $\mathcal{T}$. To complete the attack, we will now try to modify $\mathcal{T}$'s frame in order to bypass its security.

Corruption of the Java Frame. From application $\mathcal{A}$, we can now corrupt $\mathcal{T}$'s frame. We are then literally spoilt for choice in order to bypass the application's security :

– Set the jpc to a given value in order to modify the execution flow,
– Assign given values to references or integral values in the operand stack to have a method executed on the wrong object or with wrong parameters, or return a wrong value,
– Assign given values to references or integral values in the set of local variables, with the same consequences.

With regards to the current state of the art of fault injection, these possibilities are quite outstanding. In a manner of speaking, completing the three previous steps enhances tremendously the initial fault model. We present hereafter one of the numerous way to render the security check of $\mathcal{T}$ useless by modifying the value of the Java Program Counter.

Modification of the jpc. As expected, the signature verification has failed and the conditional branching at line 0x12 leads the execution flow in the "`accessDenied` branch". Because of the flooding, the so-called breakpoint is "hit" at line `0x1D` (`invokespecial #6 <accessDenied>`).

$\mathcal{A}$ is then resumed and we can read $\mathcal{T}$'s frame and identify the jpc value (Fig. 10).

```
0x0040 : XX XX XX XX  XX XX XX XX  XX XX 08 00  01 00 38 FC
0x0050 : 01 00 78 F2  01 00 D2 FE  00 00 05 00  00 00 00 02
0x0060 : 01 00 46 F2  00 00 00 00  00 00 01 00  E7 D2 66 00
0x0070 : 0B D4 07 00  01 00 38 FC  00 00 01 00  00 00 05 00
0x0080 : 00 00 00 02  01 00 46 F2  00 00 00 00  00 00 01 00
0x0090 : 00 00 18 00  00 00 1D 00  XX XX XX XX  XX XX XX XX
```

Fig. 10. Memory dump from the forged array

An easy way to overcome the signature invalidity is then to modify the jpc in order to jump "manually" in the desired branch. That is to say to say to modify the jpc value from `0x1D` to `0x16`. This is done by a mere affectation in the forged array : `ac.array [jpc_offset] = 0x16;`

When the scheduler switches back to $\mathcal{T}$, its execution continues according to the frame. That is to say at the offset we have just set. The next executed instruction will then be the invoke of the `accessGranted` method. As a consequence, we gain access to the privileged method, although we do not have the private key to produce a valid signature.

Depending on the moment when the ping flooding force the interruption, similar results have been obtained by modifying operands on the stack and local variables. What emerges from these different options is that a certain inaccuracy in the ping flooding phase is tolerable. Indeed, depending on the "breakpoint" location, an attacker with a good knowledge of the targeted application will often (not to say always) find a path to meet her objective.

This proof of concept demonstrates that such a threat should be taken in consideration when addressing the security of an embedded platform. Hereafter, we discuss this particular topic and tackle the issue of countermeasures designing.

5 Discussion on Protections

Making a product secure against any known attacks is not straightforward, as it requires a significant expertise in the field. This is particularly true in embedded technology when the security must coexist with restrictions of cost, performances and resources. Therefore the best way to suit all these requirements remains an optimisation of the security at the right protection level. This be can achieved thanks to a thorough vulnerability analysis. Regarding the attack introduced in this article, the developer should wonder if this attack is worth of being considered as a reasonable threat.

This attack is undoubtedly not easy to set up. However its apparent complexity should not conceal the potential consequences for an application. This statement is particularly true for the following reasons:

– The fault model turns out to be extremely powerful. Therefore most of the sensitive functions of an application may be defeated, even if they have been secured with care.
– A weak system may lead to an alteration of any hosted applications.
– The adequate protection is unlikely to be found in the application. As a result, an application with a thorough concern of security may be broken. It is then of the utmost importance that a system shows the evidence it is reliable and trustworthy.

Many ways can be explored to find efficient protections against this threat. Firstly it is worth of strengthening the scheduler to make sure it cannot be abused. The protection must be adapted to the rule enforced by the multithreaded system. Based on a time slice the scheduler of our Java Card makes use of a timer. By managing this timer appropriately in the handler in charge of the network requests, we have experimented that the Java Card withstands the attack.

The identification of the targeted instruction on the power consumption trace has also been an elementary step of our attack. Therefore, the difficulty to set up the attack increases with the difficulty to locate the instruction to attack. Techniques to harden the power consumption analysis such as described in [16, 17] would then stand as an additional barrier to circumvent for the attacker.

Another way consists of a strong isolation between the memory areas of different contexts. This includes the runtime environment area where the thread contexts are stored. Such an isolation may prevent the attacker to have access to the sensitive context. It is more or less difficult to achieve according to the systems. On a smartcard it may be interesting to take advantage of specific hardware features, such as a memory protection unit (MPU). This kind a protections enforces a strong isolation by the mean of hardware controls, which remain very difficult to overcome.

Lastly it may be worth of implementing some integrity controls on the contexts during the thread switch. As the control value must be prevented from being modified by an adversary, this may be achieved through a MAC verification using an internal symmetric key for instance. Before restoring a context, the scheduler would be in charge of checking that nothing has been tampered with and could send an alarm if an inconsistency is found. This implementation has shown a great efficiency on the Java Card platform we used.

To meet with today's best practices it is assumed that the security should not rely on one single countermeasure. Therefore it is strongly recommended to combine at least two of these protections. As a conclusion, everyone has to bear in mind that finding the right compromise between security and performance is not obvious. This can be achieve by combining a high expertise in existing attacks with a strong experience in secure implementations.

6 Conclusion

This article introduces a novel attack exploiting a potential weakness of a multithreaded platform. An established breach would severely damage the security of any application running on this system. The principle lies on attempting to fool the scheduler. By this mean the attacker gets the ability to interrupt a sensitive code execution. By analogy he sets a breakpoint with the aim to subsequently modify the execution context and change the application behaviour.

The feasibility of this attack has been demonstrated on a smartcard implementing the JC3.0 specifications. Indeed this technology turned out to be a perfect target. With regards to the inherent constraints of embedded systems, it implements a relatively straightforward multithreading feature and offers network capabilities in a context of high security. As a result, we have shown how a strong authentication based on a signature may be bypassed.

Several ways of protecting an implementation have been introduced. All these techniques have shown a good level of efficiency on the Java Card. Now it is the developer's responsibility to figure out if this threat is worth of being considered.

With the growing complexity of some devices, several technologies are increasingly integrated together. This attack interestingly reveals that none of them must be neglected during the vulnerability analysis. Therefore any feature or functionality should be deemed as a potential door for an attack, even though they are not obviously related to the product security. The illustration on the Java Card with an exploitation of the multithreading and the network capability is meaningful.

Acknowledgement. The authors would like to thank Nicolas Morin, for his *practical* contribution to this work, and Christophe Giraud, for his most valuable comments on the different versions of this article.

References

1. Sun Microsystems Inc.: Runtime Environment Specification, Java Card Platform Version 3.0.1 Connected Edition (2009)
2. Sun Microsystems Inc.: Virtual Machine Specification, Java Card Platfokrm Version 3.0.1 Connected Edition (2009)

3. Sun Microsystems Inc.: Java Servlet Specification, Java Card Platform Version 3.0.1 Connected Edition (2009)
4. Anderson, R., Kuhn, M.: Tamper Resistance – a Cautionary Note. In: 2nd USENIX Workwhop on Electronic Commerce
5. Boneh, D., DeMillo, R., Lipton, R.: On the Importance of Checking Cryptographic Protocols for Faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)
6. Giraud, C., Thiebeauld, H.: A Survey on Fault Attacks. In: Smart Card Research and Advanced Application Conference (CARDIS 2004). LNCS, pp. 159–176. Springer, Heidelberg (2004)
7. Govindavajhala, S., Appel, A.W.: Using Memory Errors to Attack a Virtual Machine. In: SP 2003: Proceedings of the 2003 IEEE Symposium on Security and Privacy, Washington, DC, p. 154 (2003)
8. GlobalPlatform Inc.: GlobalPlatform Card Specification 2.2 (2006)
9. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley (1999)
10. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg (2010)
11. Hogenboom, J., Mostowski, W.: Full memory read attack on a java card. In: 4th Benelux Workshop on Information and System Security Proceedings, WISSEC 2009 (2009)
12. Handley, M., Rescorla, E.: RFC 4732: Internet Denial-of-Service Considerations (2006)
13. CERT Coordination Center: Denial of Service Attacks. Technical report (1997), http://www.cert.org/tech_tips/denial_of_service.html
14. Vermoen, D., Witteman, M., Gaydadjiev, G.N.: Reverse Engineering Java Card Applets Using Power Analysis. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 138–149. Springer, Heidelberg (2007)
15. USB Implementers Forum, Inc.: Universal Serial Bus Communication Class Subclass Specification for Ethernet Emulation Model Devices (2005)
16. Shamir, A.: Protecting Smart Cards from Passive Power Analysis with Detached Power Supplies. In: Koç, Ç.K., Paar, C. (eds.) CHES 2000. LNCS, vol. 1965, pp. 71–77. Springer, Heidelberg (2000)
17. Muresan, R., Gregori, S.: Protection Circuit against Differential Power Analysis Attacks for Smart Cards. IEEE Transactions on Computers 57, 1540–1549 (2008)
18. Sun Microsystems Inc.: Application Programming Interface, Java Card Platforms Version 3.0.1 Connected Edition (2009)

A Implementation

A.1 Array Forgery

The classes loaded wit the attacker's application :

```
public class ArrayContainer {
    byte[] array;
}
```

```
public class ForgeryContainer {
    Forgery f;
}

public class Forgery {
    int field_1, field_2;
}
```

The code within the attacker's application to forge the array in the volatile memory :

```
ArrayContainer ac = new ArrayContainer();
ac.array = new byte[1];
ForgeryContainer fc = (ForgeryContainer) (Object) ac;
fc.f.field_1 = 0x100; // set the length of ac.array to 256
fc.f.field_2++;       // increment the memory pointer of ac.array
// Access to memory through ac.array[i]
```

A.2 Request Flooding Validation Thread

The run method of the thread used to validate the influence of communication :

```
public void run() {
    i = 0;
    startTime = System.currentTimeMillis();
    while ((System.currentTimeMillis() - startTime) < TIME_BOUND){
        i++;
    }
}
```

The Python ping flooder *:*

```
def flood(host, url, delay, socket, ID, count, ping_delay)
    # Send request to target application
    conn = httplib.HTTPConnection(host)
    conn.request("GET", url)

    # Wait
    time.sleep(delay)

    # Send ping flood
    for i in xrange(count):
        send_ping(ID, socket, host)
        time.sleep(ping_delay)
```

A Formal Security Model of a Smart Card Web Server

Pierre Neron[1] and Quang-Huy Nguyen[2]

[1] Ecole Polytechnique
92128 Palaiseau, France
[2] Trusted Labs
5 rue du Bailliage, 78000 Versailles, France

Abstract. Smart card Web server provides a modern interface between smart cards and the external world. It is of paramount importance that this new software component does not jeopardize the security of the smart card. This paper presents a formal model of the smart card Web server specification and the proof of its security properties. The formalization enables a thoughtful analysis of the specification that has revealed several ambiguities and potentially dangerous behaviors. Our formal model is built using a modular approach upon a model of Java Card and Global Platform. By proving the security properties, we show that the smart card Web server preserves the security policy of the overall model. In other words, this component introduces no illegal access to the card resources (*i.e.,* file system and applications). Furthermore, the smart card Web server provides a means for securely managing the card contents (*i.e.,* resources update).

1 Introduction

Since the beginning of the smart card era, the I/O interface is defined by the ISO-7816 standard [10] in terms of APDU (Application Protocol Data Unit) commands and responses. For example, to access to a binary file, the card reader sends a `SELECT FILE` command to select the file, then a `GET BINARY` command to retrieve its contents. For the next generation of multimedia SIM cards that may embed up to one gigabyte of data, this interface becomes outdated: because the multimedia data are heterogeneous and stored in complex multi-leveled file systems, the APDU commands do not provide an efficient access method. Furthermore, the ISO-7816 standard requires ad-hoc software on the handset to communicate with the SIM card. On the other hand, most handsets own a Web browser to access to the MNO (Mobile Network Operator) services and it is tempting to use the HTTP protocol for interfacing with the SIM card. In this interface, the SIM card embeds a Web server and the handset accesses to the card resources via a Web browser.

The so-called smart card Web server (SCWS) provides a modern interface and dramatically simplifies the access to multimedia services. Standardized by Open Mobile Alliance (OMA)[12], the SCWS also allows the MNOs to remotely administer their clients via an OTA (over-the-air) infrastructure. Recently, this remote administration capacity has been leveraged by the Global Platform consortium [8] to a content management method for multi-actor multi-application smart cards. For Java Card framework, the European Telecommunications Standards Institute (ETSI) also specifies additional APIs to transform a Java Card applet into a servlet (*i.e.,* a Web application)[6]. In other words, SCWS is a cross-industry specification based on existing infrastructures and it is worth to investigate its overall consistency before any implementation.

On the other hand, the Web server also exposes the SIM card to numerous well-known threats and it is of paramount importance that it does not compromise the security services provided by the card. In particular, it is essential to ensure that no illegal access to card resources can be done through the SCWS.

In this paper, we present a formalization the SCWS specifications [12,8,6,17,16] and a formal proof of the security properties related to the card resources access. The SCWS is formalized as a state machine that receives a HTTP request, interacts with the other card components before sending back the corresponding HTTP response. All formal models and proofs have been developed in Coq [15], a proof assistant based on higher-order type theories. This choice was firstly motivated by the safety of Coq that has well-studied mathematical foundation and a robust implementation: all proofs are re-checked by a (tiny) kernel that is the only Coq trusted reasoning base. Secondly, the expressive power of the logic underlying Coq deals more efficiently with the (universally) quantified security properties. Furthermore, this work benefits from the Coq libraries built for the certification of the Java Card platform in a previous work [5] and hence, smoothly handles the interaction between different card components (*e.g.,* SCWS, Java Card and Global Platform). The reuse of an existing formal infrastructure also reduces the modeling and proving workload.

The rest of this paper is organized as follows. Section 2 summarizes the different specifications related to the SCWS and discussed its expected security properties. The Coq model of the SCWS is described in Section 3. In Section 4, we present the Coq formalization and the proof of the security properties. We discuss the related work and give some concluding remarks in Sections 5 and 6.

2 Smart Card Web Server

This section provides an overview of the SCWS functions as specified in [12,8,6,17], then discusses several ambiguities and dangerous behaviors discovered while formally analyzing these specifications. An analysis of the SCWS security policy is also presented.

Figure 1 shows the integration of the SCWS in a multi-actor and multi-application Java smart card (*i.e.,* a smart card that supports both Java Card and Global Platform). The SCWS includes three components:

- a HTTP server
- a repository containing the SCWS data
- an administration task manager that updates the SCWS repository

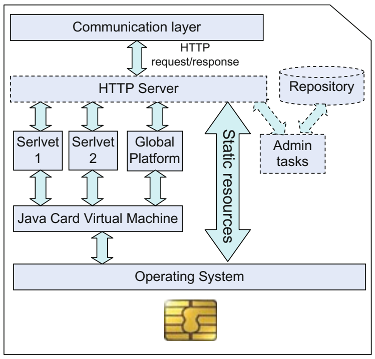


Fig. 1. SCWS inside a Java smart card

The external entity (*e.g.,* a Web browser or an OTA infrastructure) communicates with the card using the HTTP requests and responses. After being relayed by the communication layer, the HTTP request is sent to the HTTP server. Note that the multi-request management is ensured by the communication layer: a HTTP server (instance) is supposed to receive and manage one request per session. A SCWS session starts with the reception of the request and finishes by the emission of the response. The HTTP server dispatches the request to its destination indicated by the URL included in the header of the request:

– If this URL points to a servlet (*i.e.,* an applet registered to the SCWS), then the request is forwarded to this servlet.
– If this URL points to a static resource (*i.e.,* a file in the file system), then this file is accessed following the HTTP method (`GET`, `PUT`, etc) of the request. The access to the file system is done through the OS low-level services.
– If the URL points to the administration task manager, then this is an administration command for updating the SCWS repository.
– If the URL points to the Global Platform component (also known as the Card Manager), then the request contains a command that updates the card contents (*i.e.,* update the servlets).

Interface between SCWS *and servlets.* On a Java Card platform, a servlet is an applet that was registered to the SCWS. A servlet needs to implement a specific Java interface for each HTTP method. For example, the `doPost()` interface must be implemented to handle the POST requests. The servlet registration and Java interfaces are defined by ETSI in a dedicated API [6].

The interface between the SCWS and the servlet is done by method invocation. For example, for a `POST` request, the SCWS identifies the servlet, and invokes the `doPost()` method of this servlet. The HTTP request is passed to this method as a global variable. The method constructs the HTTP response that is also a global variable. Finally, the SCWS envelops the HTTP response and sends it back to the communication layer.

This interface corresponds to the "normal" servlets that receive a HTTP request and returns a HTTP response. The SCWS also manage the "interception" servlets that only collect information from the header of the HTTP requests. The interception servlets do not return any HTTP response.

Access control. According to [12], the HTTP access control relies on the *protection sets.* A protection set contains a list of users. If a URL subtree is mapped to a protection set, then access to the resources inside this subtree is reserved to the users defined in that protection set.

The user authentication may use different protocols:

– HTTP Basic or Digest authentication as defined in [16], which is a simple authentication with username and password contained in the request's header
– HTTPS protocol which uses cryptographic procedure to authenticate users
– ADMIN protocol *i.e.,* the current user is the remote administration server

Building the HTTP response. In successful cases, the result is put in the body of the response and the status `0x200` is set in the header of this response. Otherwise, an error status is sent back in the header. Several categories of status are defined by the HTTP protocol *e.g.,* `0x2xx` for successful cases, `0x4xx` for client errors, `0x5xx` for internal errors while processing the request.

2.1 Specification Analysis and Recommendations

The formal analysis of the SCWS specification has revealed several ambiguities and dangerous behaviors that we discuss here. Furthermore, several recommendations are also provided to overcome these issues.

1. **URL sharing:** Sharing a URL between servlets and static resources is not forbidden in the specification. In some cases, URL sharing does not generate any conflict, *e.g.,* between a normal servlet and an interception servlet. In other cases, *e.g.,* between a servlet and a file, the data returned by the SCWS is not precisely defined. In order to ensure the consistency, URL sharing is not allowed in the formal model.
2. **Multi-role applet:** Can an applet be registered as both interception servlet and normal servlet? Multi-role applet is not forbidden by the specification but an interception servlet can only access to the header while a normal servlet has access to all the HTTP request. In other words, there is an inconsistency on the access rights of the applet. The formal model clearly separates interception servlets from normal ones: multi-role applet is not allowed.
3. SCWS/Java Card **consistency:** Can a servlet be uninstalled or updated during a servlet invocation (that is handled by Java Card)? In principle, during the execution of a method, the list of servlets may be updated. However, a servlet cannot be uninstalled or updated during its invocation.

4. SCWS/Global Platform **consistency:** The list of registered servlets and the Global Platform registry table that manages the list of (on-card) applets shall be consistent. Because a servlet is an applet that was registered to the SCWS, any modification on the list of applets (kept in Global Platform registry table) shall be synchronized with the SCWS. Furthermore, the modification rights of these lists shall be reserved to the same group of users.
5. **Servlet collaboration:** If several servlets are invokable on a request (for example, one URL is covered by two different servlets), a priority order is defined between these servlets. Usually, the servlet that is mapped to the closer URL has greater priority. The SCWS invokes the servlets following this order and if a servlet is invoked but refuse to handle the request, then the next servlet will be invoked. However, the refusing servlet is still able to access to the HTTP request. In other words, if a refusing servlet is mapped to the directory `/a/b/c`, then it can learn about the activities of the other servlets that are mapped to the directories `/a` and `/a/b`. Hence, the URL distribution to servlets (done by the SCWS administrator), shall be carefully done to avoid data leaking between them. Another solution would be to restrict the servlet collaboration by only allowing the servlet mapping to the exact URL to access to the HTTP request.
6. **Unsafe default configuration:** If no protection set is defined, then any resource is accessible. This mechanism is advocated in the HTTP protocol to avoid unnecessary authentications to the Web servers. However, on a smart card, this behavior seems to be dangerous. It is recommended to map an empty protection set to the root URL and hence, force an authentication on any access.
7. **Unsafe Fail:** In an unsuccessful operation, the HTTP response is only required to contain an error code that is different from `0x200`. This is necessary to inform the handset about the failed operation. However, a confidential information may still be leaked through the other components of the response. Hence, it is recommended that the HTTP response does not contain other data than the error code.

While the three first points correspond to the imprecision of the specification, the other issues are directly related to the security of the SCWS. The formal model described in Section 3 takes into account the above security recommendations to deal with these issues.

2.2 Security Policy

The principal SCWS security policy consists in preventing illegal access to the card resources. This security policy can be decomposed into several sub-policies as follows:

(1) URL separation: Static resources, interception servlets and normal servlets shall be separated in terms of mapped URL.
(2) No illegal access to static resources: an access cannot target

  - the un-mapped static resources,
  - the static resources not mapped to the request's URL, and
  - the unauthorized static resources.

(3) No illegal invocation of smart card applications: an invocation cannot target
  - the unregistered servlets,
  - the unmapped servlet,
  - the servlets not mapped to the request's URL, and
  - the unauthorized servlets.

(4) The smart card data outside the scope of the SCWS cannot be accessed *i.e.,* any data in a HTTP response shall belong to a SCWS-managed resource.

(5) Safe Fail: in an unsuccessful operation, the HTTP response does not contain other data than the error code.

(6) Secure card content management: the card contents (*i.e.,* static resources, applets and repository) can only be updated by an"Admin" user.

These sub-policies are then decomposed into simpler security properties in order to ease the Coq formalization.

Property 1. (URL separation) *All normal servlet, interception servlet and static resources are mapped to separate URLs.*

Property 2. (Invalid static resource) *If a static resource is not mapped to any URL, then its data cannot be sent out by a HTTP response.*

Property 3. (Unauthorized access to static resource) *If a static resource is protected by a protection set, then any HTTP request to a static resource will fail if no username/password were provided or the provided username/password are not correct.*

Property 4. (Unregistered application) *An unregistered servlet cannot be invoked by any HTTP request.*

Property 5. (Unmapped application) *A registered servlet that is not mapped to any URL cannot be invoked by any HTTP request.*

Property 6. (Unrelated application*) A servlet that is not mapped to the URL of the HTTP request and any of its ancestors cannot be invoked by this request.*

Property 7. (Unauthorized access to application) *A servlet that is mapped to the URL of the HTTP request cannot be invoked if this request is not authorized (no username/password were provided or the provided username/password are not correct).*

Property 8. (Access to administrative tasks) *Only an "Admin" user can perform the administrative tasks on the* SCWS.

Property 9. (Access to GlobalPlatform operations) *Only an "Admin" user can perform the* Global Platform *tasks.*

Property 10. (Safe fail) *If the status code of a HTTP response is not 0x200, then this response contains no data.*

It is straightforward to see that, the policy (1) is fulfilled by Property 1; the policy (2) is fulfilled by Properties 1, 2, and 3; the policy (3) is fulfilled by Properties 1, 4, 5, 6 and 7; the policy (4) is fulfilled by Properties 1, 2; the policy (5) is fulfilled by Property 10; the policy (6) is fulfilled by Properties 8 and 9.

3 A Coq Model of the Smart Card Web Server

The SCWS is formalized as a state machine. The SCWS state formalizes the repository while a SCWS transition formalizes the modification caused by a HTTP request on a state.

3.1 SCWS State

The state defines the global components of the SCWS as the following record:

$\mathsf{SCWS}\_State \triangleq \{$
$\quad registered\_servlets : registered\_servlet\_table;$
$\quad servlet\_mapping : url\_servlet\_table;$
$\quad interception\_servlets : url\_servlet\_intercept\_table;$
$\quad users\_id : user\_table;$
$\quad ps\_defined : ps\_list;$
$\quad ps\_table : path\_ps\_table;$
$\quad listen\_http : http\_state;$
$\quad auth\_status : scws\_status;$
$\quad curr\_request : option\ http\_request;$
$\quad curr\_response : http\_response$
$\quad \}$

where the components are formalized as record fields associated to their types:

– *registered_servlets* is the table of all servlets (*i.e.,* applets registered to the SCWS).
– *servlet_mapping* and *interception_servlets* are respectively two tables mapping the URLs to the normal servlets and the interception servlets.
– *users_id* is the table of all registered users in the SCWS. This table maps each user's identifier to its password.
– *ps_defined* is the table of all registered protection sets. This table maps each protection set to its parameters (protocol, list of authorized users, etc).
– *ps_table* is the table mapping each protected URL to its protection set.
– *listen_http* indicates if the SCWS is currently ready for receiving a HTTP request.
– *auth_status* indicates whether the current user is an "Admin" user (in this case *auth_status* is "ADMIN").

- *curr_request* is the HTTP request being processed by the SCWS: the request is defined as an optional type to handle "no request" error.
- *curr_response* is the HTTP response being processed by the SCWS: the SCWS always returns a (error or success) response.

All these elements define the current state of the SCWS and determine if a static resource is accessible or if a servlet is invokable by the current request.

URL tree. The resources managed by the SCWS are addressed by their URLs. In the model, the URL path name includes the directory names stored in the reverse order to speed up the recursive search in the ancestors of a name. For example, the path name `/scws/appl/epurse` is stored by the list $epurse \rightarrow appl \rightarrow scws$. There are three constructors to generate a set of URLs from a path name as follows:

$$\begin{aligned} path\_url \triangleq\ & exact\_url :\ path\_name \rightarrow path\_url \\ & \mid directory\_url :\ path\_name \rightarrow path\_url \\ & \mid *\_url\_from :\ path\_name \rightarrow path\_url. \end{aligned}$$

For example, `/scws/appl/epurse` is an exact URL, `/scws/appl/` is a URL directory and `/scws/appl/*` is a URL subtree.

SCWS *file system.* The static resources are effectively stored in the smart card file system. However, the SCWS only manages part of the smart card file system that is in its scope. Without loss of generality, a SCWS file system is represented by a table mapping the URLs to the associated files.

3.2 SCWS Transition

The transition formalizes the modification of the state caused by the process of a HTTP request. Each transition is defined by the relations between the input and the output states. The card components that can be modified by a transition are:

- the SCWS state *e.g.,* due to an administrative tasks
- the SCWS file system *e.g.,* due to `PUT` and `DELETE` requests
- the JCVM (Java Card virtual machine) state[1] *e.g.,* due to the invocation of a servlet by the SCWS

The process of a HTTP request is done in three steps. First, the authentication checks if the user has the sufficient right to process the request. Then, a servlet is resolved and invoked if necessary. Finally, the request is processed *w.r.t.* the contents of the request (see Section 2).

Authentication. The authentication process is defined according to [12]:

- `PUT` and `DELETE` requests require the current user to be an "Admin" user
- `TRACE` request that returns the routing information towards the server (*e.g.,* the list of proxies), does not require any authentication

[1] A record that contains the global components of the JCVM.

- for any other request, the authentication process relies on the defined protection sets. We need to find if the requested URL or its ancestors are mapped to any protection set:
  - If no protection set is found, then the URL is in free access.
  - If the found protection set requires the ADMIN protocol, then the current user must be an "Admin" user.
  - If the found protection set requires the HTTP protocol with no realm, then the URL is in free access.
  - If the found protection set requires the HTTP protocol with the realm "Basic authentication", then the user and password must be present in the request and must correspond to a valid user defined in the protection set (and *user_id* table).

Servlet resolution and invocation. The SCWS first attempts to find a servlet that is exactly mapped to the requested URL using the `servlet_mapping` table. If such a servlet is not found, then the search is done recursively in the ancestors of the URL.

Once a servlet is resolved, the SCWS invokes the servlet's method that corresponds to the request (*e.g.,* `doPost()`, `doGet()`). This invocation is done by the method invocation mechanism provided by the Java Card model. The execution of the servlet's method may modify the JCVM state but this modification is managed by the Java Card model. The SCWS only manages the modification caused by this method on the current HTTP response (formalized by *curr_response*).

If the resolved servlet's method produces no effect on the current HTTP response, then this servlet refuses the current request and the search is continued in the URL tree to locate the next candidate.

Request processing. The actions to be done by the SCWS depend on the method of the request (*i.e.,* `GET, POST, PUT, DELETE, HEAD, OPTION, TRACE` or `CONNECT`) and the requested URL (see Section 2):

- if the URL points to the administration task manager, then the SCWS state is modified according to the definitions in [12].
- if the URL points to the Global Platform component (*i.e.,* the Card Manager), the appropriate method of the Global Platform model is invoked.
- if this is a POST request that accesses to a servlet, then this servlet is resolved and invoked using the requested URL as described above.
- otherwise, the SCWS checks if the requested URL is mapped to a valid static resource, and returns it (or an eventual error).

The SCWS transition is defined by the following relation:

$$transition(A, scws\_st_{in}, jcvm\_st_{in}, fs_{in}, scws\_st_{out}, jcvm\_st_{out}, fs_{out})$$

where A is the action that causes the transition, $scws\_st_{in}$, $jcvm\_st_{in}$, fs_{in} respectively represent the input states of the SCWS, the JCVM and the file system while $scws\_st_{out}$, $jcvm\_st_{out}$, fs_{out} respectively represent the output states of the SCWS, the JCVM and the file system.

Example 1. The model of the `GET` request in the case where no servlet provides a response is as follows:

$$\begin{aligned}
&transition(GET\_request, scws\_st_{in}, jcvm\_st_{in}, fs_{in},\\
&\qquad scws\_st_{out}, jcvm\_st_{out}, fs_{out}) \triangleq \exists\ request\ \exists scws\_st_{aux}.\\
&\ (0)\ fs_{in} \ = \ fs_{out} \wedge\\
&\ (1)\ curr\_request(scws\_st_{in}) = \ Some(request) \wedge\\
&\ (2)\ servlet\_resolution\_and\_invocation(doGet(), request, scws\_st_{in},\\
&\qquad\quad jcvm\_st_{in}, fs_{in}, scws\_st_{aux}, jcvm\_st_{out}, fs_{out}) = NO\_ERROR \wedge\\
&\ (3)\ curr\_response(scws\_st_{aux}) = \ curr\_response(scws\_st\_in) \wedge\\
&\ (4)\ get\_file(fs_{in}, scws\_st_{aux}, scws\_st_{out})
\end{aligned}$$

(0) means that the file system is not modified by this transition.
(1) means the current SCWS state ($scws\_st_{in}$) contains some request to be processed.
(2) means the servlet invocation produces the output state of the JCVM ($jcvm\_st_{out}$), an temporary SCWS state ($scws\_st_{aux}$) and no error.
(3) means the servlet invocation has no effect on the HTTP response (because the response component of $scws\_st_{aux}$ is that of $scws\_st_{in}$). In other words, all servlets refuse the HTTP request. In this case, the request is forwarded to the file system (static resource).
(4) returns a file (pointed by the requested URL) or an error included in the output SCWS state ($scws\_st_{out}$).

4 Proof of the Security Properties

This section describes the formal statement and the proof of the security properties presented in Section 2.2. The security properties are formally expressed as Coq theorems in the following form:

$$\begin{aligned}
&\forall\ scws\_st_{in}, scws\_st_{out}, jcvm\_st_{in}, jcvm\_st_{out}, fs_{in}, fs_{out}.\\
&\ transition(A, scws\_st_{in}, jcvm\_st_{in}, fs_{in}, scws\_st_{out}, jcvm\_st_{out}, fs_{out}) \Rightarrow\\
&\ Pre(A, scws\_st_{in}, jcvm\_st_{in}, fs_{in}) \Rightarrow \ Post(A, scws\_st_{out}, jcvm\_st_{out}, fs_{out})
\end{aligned}$$

where *Pre* states the conditions on the input states of the transition and *Post* states the property of the output states.

Theorem 1. *(No map on static resource) Mapping a servlet to a URL already mapped to a static resource, returns error status code and this mapping will not be registered.*

Theorem 2. *(Admin access required) Access to a URL protected by a protection that requires ADMIN authentication is only allowed if the authentication status of the input state is ADMIN.*

Theorem 3. *(Invalid Static Resource) If the* SCWS *provides a response containing some data (i.e., some file), then this file was already in the file system before the request processing and the status code of the response is* $0x200$ *(success).*

Theorem 4. *(Invalid Authentication) If the requested URL is protected by a protection set (or inherits a protection set from its ancestors) and if the authentication failed i.e., (i) no authentication parameter in the request or, (ii) wrong password or, (iii) the user is not defined in the protection set, then the response is an error status code.*

4.1 External Observation of Servlet Invocation

In order to formalize the properties requiring that a servlet is not invoked by the SCWS, we use the external observation approach. A servlet is not invoked by the SCWS if the final state of any SCWS transition is independent of the code of this servlet. In other words, the servlet's behavior has no effect on the SCWS transitions. This approach simplifies the formalization of these properties and keeps it independent of the Java Card model.

Theorem 5. *(Unregistered servlet) Unregistered servlets have no effect on the final state of a* SCWS *transition.*

To this end we suppose the determinism of the JCVM transitions and of the file system transitions (*i.e.,* those transitions are functions). The JCVM transition describes the modification of the JCVM due to the invocation of a servlet by the SCWS while the file system transition describes the modifications of the file system due to static resource access requests (*e.g.,* PUT or GET). Intuitively, all JCVM and file system operations (bytecode execution and file access) are deterministic. Those are two properties of the JCVM and file system model that are not in the scope of the SCWS.

The formalization is based on two actions $call\_servlet_1$ and $call\_servlet_2$ that represent the effects of any two different codes of the unregistered servlet (these two actions having an identical effect on other servlets).

$$\begin{array}{l}
\forall\ S, scws\_st_{in}, scws\_st_{out1}, scws\_st_{out2}, jcvm\_st_{in}, jcvm\_st_{out1}, jcvm\_st_{out2} \\
\hfill fs_{in}, fs_{out1}, fs_{out2}. \\
\quad transition(call\_servlet_1,\ scws\_st_{in}, jcvm\_st_{in}, fs_{in}, \\
\hfill scws\_st_{out1}, jcvm\_st_{out1}, fs_{out1}) \Rightarrow \\
\quad transition(call\_servlet_2, scws\_st_{in}, jcvm\_st_{in}, fs_{in}, \\
\hfill scws\_st_{out2}, jcvm\_st_{out2}, fs_{out2}) \Rightarrow \\
\quad \forall\ serv.(S \neq serv \Rightarrow call\_servlet_1(serv) = call\_servlet_2(serv)) \Rightarrow \\
\quad unregistered(scws\_st_{in}, S) \Rightarrow scws\_st_{out1} = scws\_st_{out2}
\end{array}$$

Theorem 6. *(Unmapped servlet) Unmapped servlets have no effect on the final state of a* SCWS *transition.*

This theorem is formalized similarly as Theorem 5 using the hypothesis that the unmapped servlet is not mapped to the requested URL and any of its ancestors.

4.2 Proof

Notice first that these Coq theorems are sufficient to verify the security properties presented in Section 2.2. Indeed:

- Property 1 is verified by Theorem 1; Property 2 is verified by Theorem 3; Property 3 is verified by Theorem 4; Property 4 is verified by Theorem 5.
- Property 5 is verified by Theorem 6; Property 6 is verified by Theorem 6; Property 7 is verified by Theorem 4.
- Properties 8 and 9 is verified by Theorem 2 under the hypothesis that the URLs used in the administration commands and the Global Platform commands are only accessible to an "Admin" user. This hypothesis corresponds to the security policy required by the SCWS administration and the Global Platform card content management [8].
- Property 10 is the corollary of Theorem 3.

The proof of the Coq theorems is interactive. According to the data structures used in the model, a relevant proof scheme (*e.g.,* case analysis, induction) is used. Most of the proofs of the different properties relies on a case analysis of the different transitions provided by the SCWS and modeled through the *transition* predicate. Therefore, for each of this case there are two possibilities :

- The "nominal" case corresponds to the proof hypotheses (*e.g.,* wrong password), thus the *transition* relation provides us the corresponding conclusion (*e.g.,* sending an error code) is in general simple.
- The specific cases contradict the proof hypotheses, and we have to construct the right premise (from the current context)that allows us to prove the contradiction. This construction can be costly due to the size and the complexity of the context. If a recursive data structure or action is involved in the property (*e.g.,* the URLs or the servlet resolution), then the proof requires an induction on this structure or action.

Proof of the external observation properties (Theorem 5 and 6) also relies on case analysis of the two separate transitions (that respectively use *call_servlet_*1 and *call_servlet_*2):

- if the transitions are identical and the request does not invoke the (unregistered or unmapped) servlet S, then we can use the determinism hypotheses to conclude that both output SCWS states are also identical
- if the transitions are identical and the request invokes S, then one can prove that because S is either unmapped or unregistered, the SCWS does not invoke it and thus S has no effect on the output state

– otherwise, the transitions being different, the model is proved to be deterministic by showing that the intersection of the premises in two transitions is empty.

Dedicated tactics have been used to factorize the proof and reduce the maintenance workload. However, a lots of user-interactions are still required to allow the Coq's kernel to re-check the whole proof at the end in order to increase the trustworthiness. In other words, user-interactions are somewhat the cost to pay for getting a higher level of trust.

5 Related Work

The SCWS is a pretty new software component and to our knowledge, no formal analysis has been done on it. However, formal techniques have been used by numerous researchers to analyze the security of the Web-related systems. The network protocols are the preferred targets for formal methods (see [11,13,4,3]). The SCWS uses the well-known HTTP and HTTPS protocols that have been intensively investigated and hence, are not in the scope of our model.

In [1], the authors formalize part of a SMTP mail server in Coq. This model only covers the mail receiving process. The authors build the model following a Java implementation rather than the SMTP specification. In contrast, we aim at preserving the generality of the SCWS specifications as much as possible. If a choice is required between several specifications, the generality is considered as the first criterion. Our objective is to ensure that no new smart card backdoor is introduced by the SCWS and the security services provided by the smart card are preserved. We do not focus on a specific implementation as done in the source-code verification approach. Source-code verification is trendy because it may detect bugs on a real implementation. However, in the current state of the art, source-code verification is usually limited by the size and the complexity of the code.

Model checking is also widely used to formalize and verify the properties of the Web applications (or services). The drawback of the model checking approach is that an abstraction is usually required on the properties in order to avoid state explosion. Finding a correct abstraction is not always possible for any property. The paper [9] describes an attempt to apply SPIN model checker to the verification of the Web applications. The authors formalize the properties in a communicating finite automate and use SPIN to verify these properties. Also using SPIN, [7] presents a tool built upon it to formally analyze the Web services. In [14], a recovery framework of the Web services is modeled and checked. The Web applications (or servlets) are not in the scope of our work. Actually, we focus on the interactions between the servlets and the external world through the SCWS. Hence, in some sense, this work complements the verification of Web applications.

The difficulty in designing secure Web systems is partly due to the lack of a formal foundation: [2] is a recent attempt to bridge this gap. That paper presents several attack models on the Web system and evaluates some well-known counter-measures *w.r.t.* these models. This is a promising approach to formally analyze the robustness of the Web systems.

6 Concluding Remarks

In this paper, we presented a formal security model of the SCWS. The formalization identified some ambiguities as well as dangerous behaviors of the specification (see Section 2.1). This feedback is useful for the standard institutions such as `Global Platform` consortium, OMA and ETSI in order to improve their specifications. On the other hand, several security properties of the SCWS were proved on the model. These security properties ensure that the SCWS preserves the overall security of the Java Card/Global Platform model. These security policies are also required by different protection profiles[2] of SCWS-embedded products. Hence, this work can be used to fulfill the ADV_SPM.1 evaluation task[3] that is the only formal method related task of the Common Criteria **EAL6** level.

It is worth to highlight the modular approach advocated by this work. The reuse of the model (and associated proof) bricks provided by the existing Java Card/Global Platform baseline model[4] reduces the overall workload to roughly two man-months for a Formal Methods expert having general knowledge on smart (Java) cards. Three thousand lines of model and ten thousand lines of proof have been developed in this project. The SCWS model also extends the baseline model by new bricks such as "HTTP protocol" and "servlet management". Without the existing bricks, building the Coq model and proof for a new software component from the scratch may not comply with the industrial constraints. Indeed, the formal analysis is best done between the specification/design freeze and the implementation. Otherwise, the formal analysis does not generate sufficient added value on the final product.

Further work consists in refining the security model in more concrete representations (*e.g.,* "Functional Specification" and "Design" in Common Criteria scheme) to get a complete formal chain from the specification to the implementation and hence, fulfills the Common Criteria highest requirements on the ADV assurance class that ensures the correctness of the product. Again the development of the formal chain will be facilitated by the existing bricks provided by [5]. Note that in order to get a full **EAL7** certificate, the other assurance classes are still to be addressed, in particular, the ATE (Application TEsting) assurance class and the AVA (Application Vulnerability Analysis) assurance class. These classes are not really related to formal methods: in the current state of the art, formal models and proofs are mainly used to ensure the correctness

[2] A protection profile defines the set of security requirements on an IT product for a Common Criteria security evaluation.

[3] This task requires to build a formal security model of the product and to prove the evaluation security objectives on this model.

[4] More than 100 000 lines of Coq model have been developed for fulfilling the highest requirements on the ADV (Application DeVelopment) assurance class [5].

(the robustness of the product being ensured by the ATE and AVA assurance classes).

Finally, it is worth to mention that in the latest Java Card specification (version 3.0 - connected edition), the Web server is part of the virtual machine that also includes almost all Java features such as garbage collection and multi-threading. Java Card 3.0 platform requires significantly more computing resources than the previous version and is still yet to be accepted by the market. On the contrary, the SCWS is a pragmatic implementation of a HTTP-based I/O for the currently deployed smart cards. A security model of the Java Card 3.0 Web server can also be built at a reasonable cost using the model bricks provided by this work.

Acknowledgments. We thank the anonymous reviewers for their comments on the previous version of this paper.

References

1. Affeldt, R., Kobayashi, N.: Formalization and Verification of a Mail Server in Coq. In: Okada, M., Pierce, B.C., Scedrov, A., Tokuda, H., Yonezawa, A. (eds.) ISSS 2002. LNCS, vol. 2609, pp. 217–233. Springer, Heidelberg (2003)
2. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J.C., Song, D.: Towards a formal foundation of web security. In: Proc. of the 23rd IEEE Computer Security Foundations Symposium, pp. 290–304. IEEE Computer Society (2010)
3. Barth, A., Jackson, C., Mitchell, J.C.: Securing frame communication in browsers. In: Proc. of 17th USENIX Security Symposium (2008)
4. Bhargavan, K., Fournet, C., Gordon, A.D.: Verified reference implementations of WS-security protocols. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 88–106. Springer, Heidelberg (2006)
5. Chetali, B., Nguyen, Q.-H.: Industrial Use of Formal Methods for a High-Level Security Evaluation. In: Cuéllar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 198–213. Springer, Heidelberg (2008)
6. European Telecommunications Standards Institute. Smart Cards; Application invocation Application Programming Interface (API) by a UICC webserver for Java Card platform, Release 7 (2008), Ref: ETSI TS 102 588, `http://pda.etsi.org/pda/queryform.asp`
7. Fu, X., Bultan, T., Su, J.: WSAT: A Tool for Formal Analysis of Web Services. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 510–514. Springer, Heidelberg (2004), available at `http://www.cs.ucsb.edu/~su/WSAT/`
8. Global Platform. GlobalPlatform Card Specification 2.2 - Remote Application Management over HTTP - Amendment B, version 0.5 (2008), `http://www.globalplatform.org/specificationscard.asp`
9. Haydar, M., Petrenko, A., Sahraoui, H.: Formal Verification of Web Applications Modeled by Communicating Automata. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 115–132. Springer, Heidelberg (2004)
10. International Organization for Standardization. Identification cards – Integrated circuit(s) cards with contacts. Part 1-11 (1987-2007), `http://www.iso.org/`
11. Mitchell, J.C., Shmatikov, V., Stern, U.: Finite-state analysis of ssl 3.0. In: Proc. of 7th USENIX Security Symposium, pp. 201–216 (1998)

12. Open Mobile Alliance. Smartcard Web Server V1.1 (2009), http://www.openmobilealliance.org/technical/release_program/SCWS_v1_1.aspx
13. Roscoe, A.W.: Modelling and verifying key-exchange protocols using csp and fdr. In: Proc. of 8th IEEE Computer Security Foundations Workshop, pp. 98–107. IEEE Computer Soc. Press (1995)
14. Shegalov, G., Weikum, G.: Formal verification of web service interaction contracts. In: Proc. of the 2008 IEEE International Conference on Services Computing, pp. 525–528. IEEE Computer Society (2008)
15. The Coq Development Team. The Coq Proof Assistant, http://coq.inria.fr/
16. The Internet Society. HTTP Authentication: Basic and Digest Access Authentication (1999), http://www.ietf.org/rfc/rfc2617.txt
17. The Internet Society. Hypertext Transfer Protocol – HTTP/1.1 (1999), http://www.ietf.org/rfc/rfc2616.txt

Differential Fault Analysis of AES-128 Key Schedule Using a Single Multi-byte Fault

Sk Subidh Ali and Debdeep Mukhopadhyay

Dept. of Computer Science and Engineering
Indian Institute of Technology Kharagpur, India
{subidh,debdeep}@cse.iitkgp.ernet.in

Abstract. In this paper we propose an improved multi-byte differential fault analysis of AES-128 key schedule using a single pair of fault-free and faulty ciphertexts. We propose a four byte fault model where the fault is induced at ninth round key. The induced fault corrupts all the four bytes of the first column of the ninth round key which subsequently propagates to the entire tenth round key. The elegance of the proposed attack is that it requires only a single faulty ciphertext and reduce the search space of the key to 2^{32} possible choices. Using two faulty ciphertexts the attack uniquely determines the key. The attack improves the existing DFA of AES-128 key schedule, which requires two faulty ciphertexts to reduce the key space of AES-128 to 2^{32}, and four faulty ciphertexts to uniquely retrieve the key. Therefore, the proposed attack is more lethal than the existing attack as it requires lesser number of faulty ciphertexts. The simulated attack takes less than 20 minutes to reveal 128-bit secret key; running on a 8 core Intel Xeon E5606 processor at 2.13 GHz speed.

Keywords: Differential Fault Analysis, Fault Attack, Advanced Encryption Standard, Key Schedule, DFA.

1 Introduction

External noise such electromagnetic radiation, glitch in the input clock line, variation in the supply voltage can create fault in the electronic devices such as smart card. These properties of electronic devices are being exploited by the attackers. An attacker can deliberately inject fault into a device running a cryptographic algorithm. Then by analysing the faulty output he can reveal the secret key. This kind of attack is known as fault attack which was originally introduced by Boneh et. al. [7] in 1996 to break RSA crypto-system running on a smart-card . Subsequently, Biham and Shamir showed a modified form of the attack which is known as Differential Fault Analysis (DFA) based on combination of Differential Cryptanalysis and fault analysis [5]. The DFA was applied on DES crypto-system which successfully retrieved the secret key.

In 2001, NIST standardised Rijndael as the Advanced Encryption Standard (AES) [1]. Subsequently, many DFA were proposed on AES cryptosystem [6,11, 14,15,17] with the aim to reduced the number of faulty ciphertext required by the attack. However the DFA on AES can be divided into two categories. One in which the fault is induced in AES states, another in which the fault is induced in the key schedule. The literature is rich in attacks on the states. The most recent among these attacks is the attack on AES-128 proposed by M. Tunstall et. al. in [21,22]. They proposed an attack where a single byte fault induced at the input of eighth round can reduce the AES-128 key space to 2^8 choices. However, there is not much research on attacks of AES key schedule.

In 2003, Giraud first proposed a DFA against the AES key schedule [10], which combined both kind of fault attack; the fault analysis in AES states as well as in key schedule. The attack was subsequently improved by Chen and Yen in [9]. Chen et. al. attack required to induce fault at the ninth round key. The attack required less than thirty faulty ciphertexts to successfully retrieve the secret key. In 2006 Peacham and Thomas in [16], proposed an improved DFA against the AES key schedule, which reduced the number of faulty ciphertext required by the attack to 12. In Peacham's attack, fault was induced at the ninth round key during execution of the key schedule operation, so that the induced fault subsequently propagates to tenth round key.

Finally, Takahashi et. al. in [20], proposed an attack, which can reduce the search space of the key to 2^{48} choices using two faulty ciphertexts. The attack can reduce the key space to 2^{16} choices using four faulty ciphertexts and can determine the key uniquely by using seven faulty ciphertexts. In 2008, Kim et. al. proposed an improved DFA by inducing 3-byte fault at the first column of ninth round key. Kim's attack required two faulty ciphertexts and a brute-force search of 2^{32}. With four faulty ciphertexts the attack can uniquely determine the secret key.

In this paper we propose an improved attack against the AES-128 key schedule. We propose a fault model where the induced fault corrupts all the four bytes of the first column of ninth round key and the fault subsequently propagates to the entire tenth round key. Our attack requires only one faulty ciphertext to reduces the search space of the AES-128 key to 2^{32} choices. Using two faulty ciphertext our attack can uniquely determine the key. We present extensive simulation results which shows that the attack takes less than 20 minutes to uniquely retrieve the 128-bit secret key.

Organization

The paper is organized as follows: We start with Section 2 where we describe the preliminaries to this paper. In Section 3 we briefly describe the existing attack. We propose our attack in Section 4. In Section 5 we present experimental results. In Section 6 we compare our work presented in this paper with the previous works, and we conclude in Section 7.

2 Preliminaries

2.1 The AES Algorithm

AES [1] is a 128-bit symmetric key block cipher comes in three different versions AES-128, AES-192, and AES-256 with key length 128-bit, 192-bit, and 256-bit respectively. The 128-bit intermediate results are represented as 4×4 matrix, known as state. Each elements of the matrix is a byte. The algorithm is divided into round function. Each round function except the last round consists of four transformations namely: `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`. The three versions of AES has three different number of rounds : AES-128 has 10 rounds, AES-192 has 12 rounds, and AES-256 has 14 rounds. The last round of each of the three versions of AES does not have `MixColumns` operation. The four basic transformations are described as follows:

`SubBytes` : It is the only non-linear byte-wise transformation in AES. Each element of the state matrix is replaced by its inverse and followed by an affine mapping. All the operations are under $\mathbf{F_{2^8}}$.

`ShiftRows` : It is a row-wise transformations where the i^{th} row is cyclically shifted by i bytes towards left where $0 \leq i \leq 3$.

`MixColumns` : It is a column level linear transformation of the state matrix. Each column of the state matrix is considered as a polynomial of degree 3 with coefficient in $\mathbf{F_{2^8}}$ and multiplied with the polynomial $\{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$.

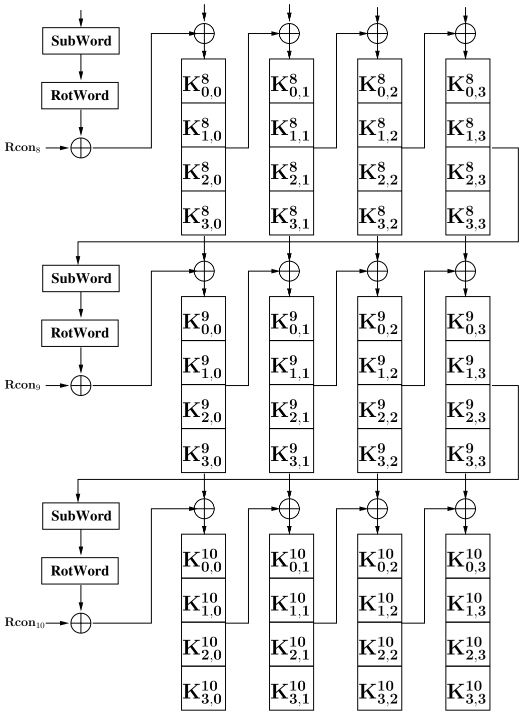

SubWord
RotWord
Rcon$_8$
Rcon$_9$
Rcon$_{10}$
$K^8_{0,0}$ $K^8_{0,1}$ $K^8_{0,2}$ $K^8_{0,3}$
$K^8_{1,0}$ $K^8_{1,1}$ $K^8_{1,2}$ $K^8_{1,3}$
$K^8_{2,0}$ $K^8_{2,1}$ $K^8_{2,2}$ $K^8_{2,3}$
$K^8_{3,0}$ $K^8_{3,1}$ $K^8_{3,2}$ $K^8_{3,3}$
$K^9_{0,0}$ $K^9_{0,1}$ $K^9_{0,2}$ $K^9_{0,3}$
$K^9_{1,0}$ $K^9_{1,1}$ $K^9_{1,2}$ $K^9_{1,3}$
$K^9_{2,0}$ $K^9_{2,1}$ $K^9_{2,2}$ $K^9_{2,3}$
$K^9_{3,0}$ $K^9_{3,1}$ $K^9_{3,2}$ $K^9_{3,3}$
$K^{10}_{0,0}$ $K^{10}_{0,1}$ $K^{10}_{0,2}$ $K^{10}_{0,3}$
$K^{10}_{1,0}$ $K^{10}_{1,1}$ $K^{10}_{1,2}$ $K^{10}_{1,3}$
$K^{10}_{2,0}$ $K^{10}_{2,1}$ $K^{10}_{2,2}$ $K^{10}_{2,3}$
$K^{10}_{3,0}$ $K^{10}_{3,1}$ $K^{10}_{3,2}$ $K^{10}_{3,3}$

Fig. 1. Last three rounds of AES-128 key scheduling algorithm

AddRoundKey: In this transformation the 128-bit round key is bit-wise XOR-ed with the 128-bit state.

An additional AddRoundKey operation is performed at the beginning of first round which is known as key whitening phase. Each round key is generated by the AES key scheduling algorithm. Figure 1 shows the generation of last three rounds according to the AES-128 key schedule. For more detail on the AES key scheduling, one can refer the AES specification [1]

2.2 Notations

In the rest of the paper we use following symbols and notations: SubBytes, ShiftRows,and MixColumns will be referred as SB, SR and MC respectively and the corresponding inverse functions as SB^{-1}, SR^{-1} and MC^{-1}.

$K^r_{i,j}$: Represent $\{i, j\}$ byte of the r^{th} round key K^r.
$C_{i,j}$: Represent $\{i, j\}$ byte of the fault-free ciphertext C.
$C^*_{i,j}$: Represent $\{i, j\}$ byte of the faulty ciphertext C^*.

2.3 Fault Model Used

It is clear from the past research on DFA of AES that the fault model is the key to successful fault analysis. Slightest change in the fault model can drastically increase or decrease the complexity of the fault analysis. This fact was clearly

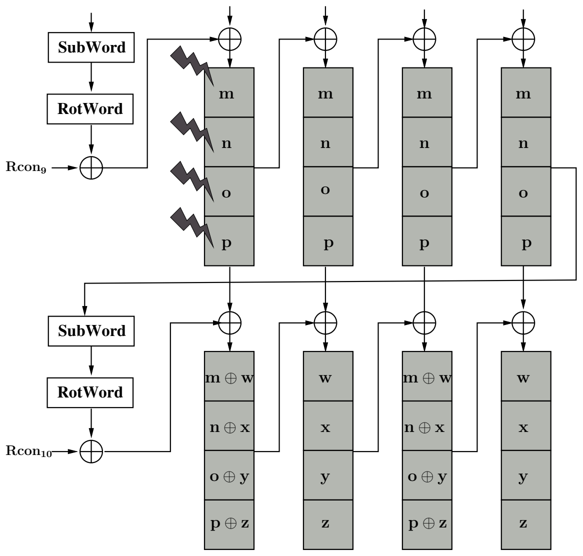


Fig. 2. Fault induced in 9^{th} round key corrupting entire first column

depicted in [3] where it was shown that with the increase of number of byte faults the search space of the key drastically increases. In our proposed attack we follow the fault model of Peachman and Thomas [16] where the fault is expected to induce in the ninth round key while it is being executed and subsequently propagated to the entire tenth round key. We assume that the induced fault corrupts all the four bytes of the first column of the ninth round key. Figure 2 show the flow of fault in the last two round keys.

3 Existing Fault Analysis

The most recent attack against the AES-128 key schedule shows that two faulty ciphertexts are enough to reduce the key space to 2^{32} choices [13]. The attack assumed a fault model where the induced fault corrupts three out of four bytes of the first column of the ninth round key K^9. As Figure 3 depicts, the induced fault subsequently propagates to the tenth round key. The three key bytes : $K^9_{0,0}, K^9_{1,0}$, and $K^9_{2,0}$ are the modified bytes and induced differences due to the faults are a, b, and c respectively. As per the AES-128 key scheduling algorithm the fault is propagated to all the subsequent three bytes of the same row of the ninth round key. As the fault is induced during the generation of key therefore the fault in the ninth round key also propagated to tenth round key bytes.

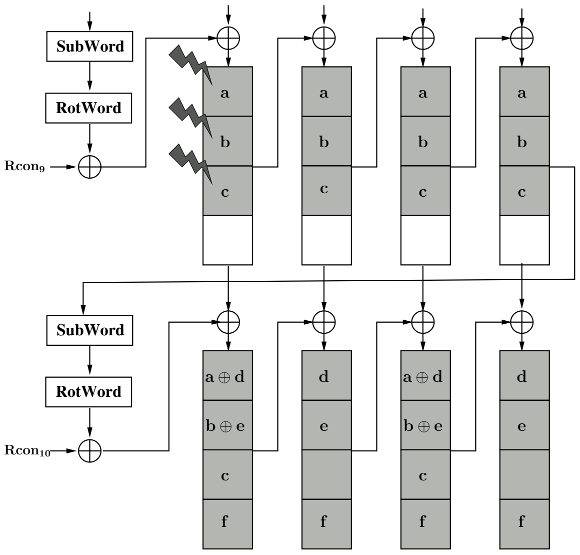


Fig. 3. Fault induced in 9th round key

The fault values d, e, and f can be calculated as follows:

$$\begin{aligned} d &= SB(K^9_{1,3}) \oplus SB(K^9_{1,3} \oplus b) \\ &= SB(K^{10}_{1,3} \oplus K^{10}_{1,2}) \oplus SB(K^{10}_{1,3} \oplus K^{10}_{1,2} \oplus b) \end{aligned} \tag{1}$$

$$\begin{aligned} e &= SB(K^9_{2,3}) \oplus SB(K^9_{2,3} \oplus c) \\ &= SB(K^{10}_{2,3} \oplus K^{10}_{2,2}) \oplus SB(K^{10}_{2,3} \oplus K^{10}_{2,2} \oplus c) \end{aligned} \tag{2}$$

$$\begin{aligned} f &= SB(K^9_{0,3}) \oplus SB(K^9_{0,3} \oplus a) \\ &= SB(K^{10}_{0,3} \oplus K^{10}_{0,2}) \oplus SB(K^{10}_{0,3} \oplus K^{10}_{0,2} \oplus a) \end{aligned} \tag{3}$$

The fault in the ninth round key affects the AES state after the ninth round *AddRoundKey*. Figure 4 shows the flow of fault in the last two rounds.

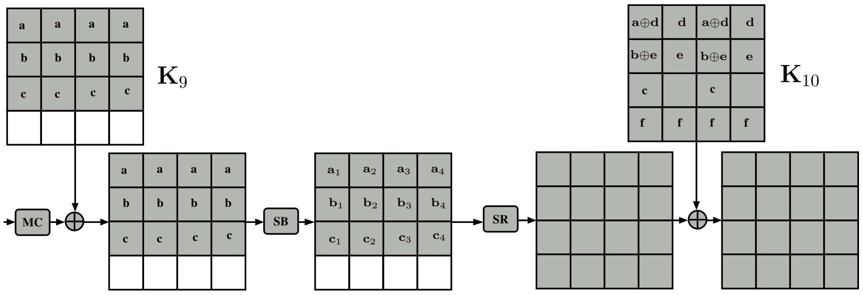


Fig. 4. Propagation of Faults in the AES-datapath due to the 3-byte Fault Induction in the first column of K^9

The fault-free and faulty ciphertexts (C, C^*) are known to the attacker. Therefore, he can represent the fault values $\{c, c, c, c\}$, at the input of ninth round using following equations:

$$\begin{aligned} c &= SB^{-1}(C_{2,2} \oplus K^{10}_{2,2}) \oplus SB^{-1}(C^*_{(2,2)} \oplus K^{10}_{2,2} \oplus c) \\ &= SB^{-1}(C_{2,3} \oplus K^{10}_{2,3}) \oplus SB^{-1}(C^*_{(2,3)} \oplus K^{10}_{2,3}) \\ &= SB^{-1}(C_{2,0} \oplus K^{10}_{2,0}) \oplus SB^{-1}(C^*_{(2,0)} \oplus K^{10}_{2,0} \oplus c) \\ &= SB^{-1}(C_{2,1} \oplus K^{10}_{2,1}) \oplus SB^{-1}(C^*_{(2,1)} \oplus K^{10}_{2,1}) \end{aligned} \tag{4}$$

In the above set of equations the attacker guesses the values of c and gets the corresponding values of key quartet $\langle K^{10}_{2,2}, K^{10}_{2,3}, K^{10}_{2,0}, K^{10}_{2,1}\rangle$. As there are 2^8 possibilities of c therefore the above set of equations will reduce the possible choices of $\langle K^{10}_{2,2}, K^{10}_{2,3}, K^{10}_{2,0}, K^{10}_{2,1}\rangle$ to 2^8 choices from 2^{32} choices. For more details on the analysis, one can refer to [13,21]. Similarly, the attacker uses another faulty ciphertext and again reduces the possible choices of $\langle K^{10}_{2,2}, K^{10}_{2,3}, K^{10}_{2,0}, K^{10}_{2,1}\rangle$ to 2^8. Then he takes the intersection of two sets of values of $\langle K^{10}_{2,2}, K^{10}_{2,3}, K^{10}_{2,0}, K^{10}_{2,1}\rangle$ generated from two different faulty ciphertexts. The intersection uniquely determine the key quartet. Now the attacker uses equation (2) and determines the values of e from the values of c, $K^{10}_{2,2}$ and $K^{10}_{2,3}$. He follows the same technique to uniquely determine the quartets $\langle K^{10}_{0,0}, K^{10}_{0,1}, K^{10}_{0,2}, K^{10}_{0,3}\rangle$ and $\langle K^{10}_{1,0}, K^{10}_{1,1}, K^{10}_{1,2}, K^{10}_{1,3}\rangle$. Therefore, using two faulty ciphertexts the attacker can retrieve 12 bytes of the tenth round key K^{10}. This implies that the attacker retrieve 96-bit and need to perform 32-bit brute-force search to get the master key.

3.1 Limitation of Existing Attack

The existing attack nicely reveals the secret key of AES-128 crypto-system using only two faulty ciphertexts. However, the attack still needs two faulty ciphertexts which is the bottle neck of the attack. First of all the induced faults need to corrupt three out of the four bytes of the first row of the ninth round key. However, in real life the probability of such an event is small. Fault induction methods such as described in [3,4,12,18,19] gives an attacker an extent of control, but cannot precisely realize the number of faults. The probability of success reduces further if more than one fault inductions are necessary. If probability of getting such fault is p then the probability of getting two such fault is p^2. It its quite obvious from the experimental result reported in [3, 4, 12, 18, 19] that the value of p is quite small. Therefore, ideally an attacker would want an attack which require one faulty ciphertext.

In the next section we propose an improved attack which will produce the same results as in the existing attack using only one faulty ciphertext.

4 Proposed Attack Using Single Faulty Ciphertext

In our proposed attack, the attacker is expected to induce a fault which corrupts all four bytes of the first column of the ninth round key as depicted in Figure 2. The proposed fault model increased the fault coverage where eventually all the bytes of the ninth round key and subsequently all the bytes of the tenth round key are corrupted by the induced fault. In the previous attack only 12 out of 16 bytes of ninth round key were corrupted by the induced fault. As in Figure 3 only the first three rows of the ninth round input state matrix is corrupted by the faulty ninth round key K^9. Therefore, only the first three rows of K^{10} participate in forming differential equations. The last row of K^{10} is not related to any fault values. Hence there is no trace of getting the last row's values. But in our fault model there is no such limitations.

4.1 Attack Principle

The proposed attack exploits the relation between the faulty byte at the input of the ninth round. Figure 5 depicts the flow of faults. At the input of ninth round, all the bytes of the state matrix S_0 are corrupted. However, each of the rows have same fault values, which helps in deducing the differential equations. We have the fault-free and faulty ciphertexts (C, C^*).

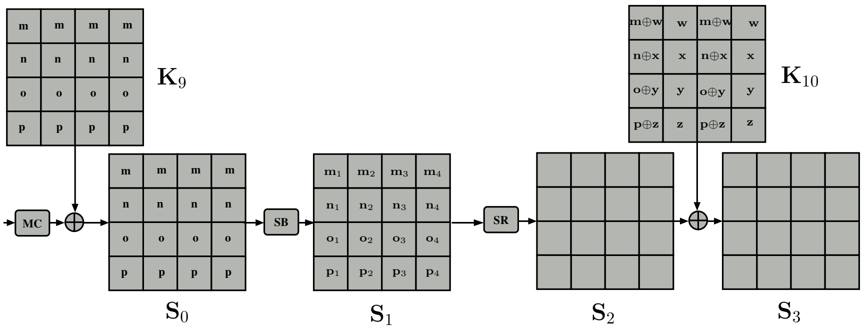


Fig. 5. Flow 4 Bytes Fault Induced At First Column of K^9

Therefore, we can represent the fault values $\{m, m, m, m\}$ at first the row of the state matrix S_0 as follows:

$$\begin{aligned} m &= SB^{-1}(C_{0,0} \oplus K^{10}_{0,0}) \oplus SB^{-1}(C^*_{(0,0)} \oplus K^{10}_{0,0} \oplus m \oplus w) \\ &= SB^{-1}(C_{0,1} \oplus K^{10}_{0,1}) \oplus SB^{-1}(C^*_{(0,1)} \oplus K^{10}_{0,1} \oplus w) \\ &= SB^{-1}(C_{0,2} \oplus K^{10}_{0,2}) \oplus SB^{-1}(C^*_{(0,2)} \oplus K^{10}_{0,2} \oplus m \oplus w) \\ &= SB^{-1}(C_{0,3} \oplus K^{10}_{0,3}) \oplus SB^{-1}(C^*_{(0,3)} \oplus K^{10}_{0,3} \oplus w) \end{aligned} \tag{5}$$

Similarly, the fault values in the rest of the three rows of S_0 can be represented by the following equations:

$$\begin{aligned} n &= SB^{-1}(C_{1,3} \oplus K^{10}_{1,3}) \oplus SB^{-1}(C^*_{(1,3)} \oplus K^{10}_{1,3} \oplus x) \\ &= SB^{-1}(C_{1,0} \oplus K^{10}_{1,0}) \oplus SB^{-1}(C^*_{(1,0)} \oplus K^{10}_{1,0} \oplus x \oplus n) \\ &= SB^{-1}(C_{1,1} \oplus K^{10}_{1,1}) \oplus SB^{-1}(C^*_{(1,1)} \oplus K^{10}_{1,1} \oplus x) \\ &= SB^{-1}(C_{1,2} \oplus K^{10}_{1,2}) \oplus SB^{-1}(C^*_{(1,2)} \oplus K^{10}_{1,2} \oplus x \oplus n) \end{aligned} \tag{6}$$

$$\begin{aligned} o &= SB^{-1}(C_{2,2} \oplus K^{10}_{2,2}) \oplus SB^{-1}(C^*_{(2,2)} \oplus K^{10}_{2,2} \oplus y \oplus o) \\ &= SB^{-1}(C_{2,3} \oplus K^{10}_{2,3}) \oplus SB^{-1}(C^*_{(2,3)} \oplus K^{10}_{2,3} \oplus y) \\ &= SB^{-1}(C_{2,0} \oplus K^{10}_{2,0}) \oplus SB^{-1}(C^*_{(2,0)} \oplus K^{10}_{2,0} \oplus y \oplus o) \\ &= SB^{-1}(C_{2,1} \oplus K^{10}_{2,1}) \oplus SB^{-1}(C^*_{(2,1)} \oplus K^{10}_{2,1} \oplus y) \end{aligned} \tag{7}$$

$$\begin{aligned} p &= SB^{-1}(C_{3,1} \oplus K^{10}_{3,1}) \oplus SB^{-1}(C^*_{(3,1)} \oplus K^{10}_{3,1} \oplus z) \\ &= SB^{-1}(C_{3,2} \oplus K^{10}_{3,2}) \oplus SB^{-1}(C^*_{(3,2)} \oplus K^{10}_{3,2} \oplus z \oplus p) \\ &= SB^{-1}(C_{3,3} \oplus K^{10}_{3,3}) \oplus SB^{-1}(C^*_{(3,3)} \oplus K^{10}_{3,3} \oplus z) \\ &= SB^{-1}(C_{3,0} \oplus K^{10}_{3,0}) \oplus SB^{-1}(C^*_{(3,0)} \oplus K^{10}_{3,0} \oplus z \oplus p) \end{aligned} \tag{8}$$

As per the AES-128 key scheduling algorithm the fault value w in the tenth round key K^{10} can be express in terms of n by the following equations:

$$\begin{aligned} w &= S(K^{9}_{1,3}) \oplus S(K^{9}_{1,3} \oplus n) \\ &= S(K^{10}_{1,3} \oplus K^{10}_{1,2}) \oplus S(K^{10}_{1,3} \oplus K^{10}_{1,2} \oplus n) \end{aligned} \tag{9}$$

Similarly, we can represent x, y, z by n, o, p respectively and K^{10} using the following equations:

$$x = S(K^{10}_{2,3} \oplus K^{10}_{2,2}) \oplus S(K^{10}_{2,3} \oplus K^{10}_{2,2} \oplus o) \tag{10}$$

$$y = S(K^{10}_{3,3} \oplus K^{10}_{3,2}) \oplus S(K^{10}_{3,3} \oplus K^{10}_{3,2} \oplus p) \tag{11}$$

$$z = S(K^{10}_{0,3} \oplus K^{10}_{0,2}) \oplus S(K^{10}_{0,3} \oplus K^{10}_{0,2} \oplus m) \tag{12}$$

Now we have four sets of differential equations for the four quartets of key bytes like the attack in [21]. However, we can not directly apply the solving technique proposed in [21]. The reason is each set of equations contain six unknown variables. For, example the unknown variables in the first set of equations (5) are $\{K^{10}_{0,0}, K^{10}_{0,1}, K^{10}_{0,2}, K^{10}_{0,3}, m, w\}$. Therefore, in this case we have to guess all possible values of m and w. For one choice of (m, w) on an average we get one choice of $\langle K^{10}_{0,0}, K^{10}_{0,1}, K^{10}_{0,2}, K^{10}_{0,3}\rangle$. Therefore, for all possible 2^{16} choices of (m, w) we get 2^{16} choices of the key quartet. If we apply the same technique to the other three sets of equations (6), (7), (8) we get 2^{16} choices for each of the three quartets. Therefore, finally we get $(2^{16})^4 = 2^{64}$ choices of the key K^{10} which is not in practical limits. This means the previous solving technique is not directly applicable.

We follow divide and conquer technique to solve the above four sets of differential equations [2]. We use S-box difference table so that we can directly get the values of the key byte from the given S-box input difference and the output difference. We start with the first set of differential equations (5). In these sets of equation we guess the possible values of (m, w). For one choice of (m, w) we directly get on an average one choice of $\langle K^{10}_{0,0}, K^{10}_{0,1}, K^{10}_{0,2}, K^{10}_{0,3}\rangle$ using S-box difference table. Therefore, for 2^{16} possible choices of (m, w) we get 2^{16} choices of the key quartet with time complexity 2^{16}. For, each value of m and the corresponding value of $K^{10}_{0,2}, K^{10}_{0,3}$ we get the values of z using equation (12). Next we consider the fourth set of equations (8) where the values of z are already known from the previous steps. Therefore, in (8) we only guess the values of p and get the values corresponding to the quartet $\langle K^{10}_{3,0}, K^{10}_{3,1}, K^{10}_{3,2}, K^{10}_{3,3}\rangle$. For each values of (m, w, z) and p we get one value of $\langle K^{10}_{3,0}, K^{10}_{3,1}, K^{10}_{3,2}, K^{10}_{3,3}\rangle$. There are 2^{16} possible choices of (m, w, z) and 2^8 choices of p. Therefore, together we have 2^{24} choices of first and fourth quartets of key bytes from the first and fourth set of equations and the time complexity of the process is 2^{24}.

Now we use the values of $K^{10}_{3,3}, K^{10}_{3,2}$ and p to get the corresponding values of y using equations (11). Similarly, we choose the third set of equations (7) and get the 2^8 values $\langle K^{10}_{2,0}, K^{10}_{2,1}, K^{10}_{2,2}, K^{10}_{2,3}\rangle$. Therefore, from the three sets of equations (5), (7), and (8) we get 2^{32} choices of the first, third and fourth quartet of key bytes. Following the same technique we get the values of x from equation (10) and then get the 2^8 values of the second quartet of key bytes $\langle K^{10}_{1,0}, K^{10}_{1,1}, K^{10}_{1,2}, K^{10}_{1,3}\rangle$ using the second set of equations (6). Therefore, now we have 2^{40} choices of all the four quartets of key bytes which is the tenth round key K^{10}.

But still we have one equation left unchecked i.e. equation (9). Each of the tenth round key and the corresponding values of w is tested by equation (9). Those which satisfy are considered and the rest are discarded. Finally, 2^{32} out of 2^{40} candidates will satisfy equation (9). Therefore, finally we have 2^{32} possible choices of the tenth round key K^{10}. So, we need to do 32-bit brute-force search to get the master key.

However, the attack time complexity is 2^{40} which is in practical limits. But the entire attack takes around 14 hours to generate all the possible 2^{32} keys which is not feasible in terms of side-channel cryptanalysis.

In the next section we show a technique which further reduces the time complexity of the attack.

4.2 Time Complexity Reduction Technique

The proposed attack has got four steps. In the first step we deduce the possible choices $K^{10}_{0,i}$ where $0 \leq i \leq 3$ and z. Similarly, in second, third and fourth steps we get the possible choices of $K^{10}_{1,i}$, $K^{10}_{2,i}$,and $K^{10}_{3,i}$ respectively and the corresponding values of y, x, w.

In the first step, for a given value of w we get 2^8 choices of the quartet $K^{10}_{0,i}$ from which we calculate z. However for getting z from equation (12) we need only two key bytes $(K^{10}_{0,3}, K^{10}_{0,2})$ of quartet $K^{10}_{0,i}$. Therefore, we only consider the unique values of $(K^{10}_{0,3}, K^{10}_{0,2})$ out of 2^8 values of $K^{10}_{0,i}$. The number of unique values of $(K^{10}_{0,3}, K^{10}_{0,2})$ is given by $\frac{2^8}{2^2} = 2^6$ [2]. Similarly, in the rest of the three steps we consider 2^6 choices of $(K^{10}_{3,3}, K^{10}_{3,2})$, $(K^{10}_{2,3}, K^{10}_{2,2})$ and $(K^{10}_{1,3}, K^{10}_{1,2})$ each for getting the values of y, x, and w respectively. This implies, for testing equation (9) indirectly we need only eight key bytes. For a given value of w the possible choices of these eight key bytes is $(2^6)^4 = 2^{24}$. For all possible 2^8 choices of w we have $2^{24} \times 2^8 = 2^{32}$ choice of the required eight bytes. Each choice of these eight key bytes are tested by equation (9) if they satisfy they are accepted, else they are discarded. Thus only 2^{24} out of 2^{32} choices satisfy the test. Those which satisfy are combined with rest of the eight key bytes $(K^{10}_{0,0}, K^{10}_{0,1})$, $(K^{10}_{1,0}, K^{10}_{1,1})$, $(K^{10}_{2,0}, K^{10}_{2,1})$, $(K^{10}_{3,0}, K^{10}_{3,1})$. There are $2^2 \times 2^2 \times 2^2 \times 2^2 = 2^8$ choices for rest of the eight key bytes. Therefore, using this technique the equation (9) is tested for 2^{32} times which correspond to the time complexity of the attack. This implies, that using this technique the time complexity of the attack reduced to 2^{32} from 2^{40} and hence comes in practical limits.

The simulated, attack written in C programing language was run on a 8 core Intel Xeon E5606 processor at 2.13 GHz speed. The attack takes less than 20 minutes to deduce the master key. Algorithm 1 summarizes the attack procedure.

Algorithm 1. DFA on AES-128 Key Scheduling using Single Faulty Ciphertexts

```
Input: $P, C, C^*$
Output: 128-bit AES key
/*P is the plaintext */
for Each candidates of $w$ do
    for Each candidates of $m$ do
        Get $\{K_{0,2}^{10}, K_{0,3}^{10}\}$ from equations (5)
        Get $z$ from equation (12).
        for Each candidates of $p$ do
            Get $\{K_{3,2}^{10}, K_{3,3}^{10}\}$ from the equations (8).
            Get $y$ from equation (11).
            for Each candidates of $o$ do
                Get $\{K_{2,2}^{10}, K_{2,3}^{10}$ from equations (7)
                Get $x$ from equation (10).
                for Each candidates of $n$ do
                    Get $\{K_{1,2}^{10}, K_{1,3}^{10}\}$ from equations (6).
                    Test equation (9).
                    if Satisfied then
                        for Each values of
                        $\{K_{0,0}^{10}, K_{0,1}^{10}, K_{1,0}^{10}, K_{1,1}^{10}, K_{2,0}^{10}, K_{2,1}^{10}, K_{3,0}^{10}, K_{3,1}^{10}\}$ do
                            Get $K^{10}$.
                            Get the AES key $K$ using AES Key Schedule.
                            if $P$=Decrypt($K$,$C$) then
                                Save K;
                            end
                        end
                    end
                end
            end
        end
    end
end
```

4.3 Analysis of the Proposed Attack

There are total 20 differential equations in the proposed attack: 16 in four sets of equations (5), (6), (7), (8), and 4 more for equations (9), (10), (11), and (12). Each of these equations reduces the 2^{16} possible choices of right hand side (corresponding to two S-box output) to 2^8 choices in the left hand side. Therefore, the reduction is given by $(\frac{1}{2^8})$. If there are N differential equations then the reduction is given by $(\frac{1}{2^8})^N$. If N equations contain M unknown variables then the reduced search space is give by $(2^8)^{(M-N)}$. For more details on the analysis, one can refer to the paper [21]. We have 20 equations with 24 unknown variables; namely m, n, o, p, w, x, y, z and 16 unknown key bytes. Therefore, we have $(2^8)^{(24-20)} = 2^{32}$ choices of final round key.

The time complexity of the attack is also 2^{32} as explained in the previous section.

Note:
In some cases the attacker may not have access to the plaintext. So, he can not perform brute-force search on the possible guessed keys. In that case the attacker need to uniquely determine the key. Our attack can be applied in such a situation with two faulty ciphertexts. Say the faulty ciphertexts are (C_1^*, C_2^*). If we apply our proposed attack on these two faulty ciphertexts, we will have two sets of fault values $m_1, n_1, o_1, p_1, w_1, x_1, y_1, z_1$ and $m_2, n_2, o_2, p_2, w_2, x_2, y_2, z_2$. Each of these sets will produce corresponding 20 differential equations.

Now we can apply our attack on each of these sets in step by step fashion. In the first step we guess the values of (w_1, m_1) and get 2^{16} choices of $K_{0,i}^{10}$ where $0 \leq i \leq 3$. For each of these choices we guess one value of w_2 and test equations (5) with the deduced key quartet $K_{0,i}^{10}$. If these two values satisfy the equations we accept them, else we discard them. There are 8 equations (two sets of equations (5) from two faulty ciphertexts) and 8 unknown variables (m_1, m_2, w_1, w_2 and $K_{0,i}^{10}$). This implies only one key candidate will satisfy the test.

In the second step we get the values of z_1, z_2 and guess the possible values of p_1, and p_2 each of which will produce 2^8 choices $K_{3,i}^{10}$. Intersection of these two sets will uniquely determine $K_{3,i}^{10}$. Following the same technique in third and fourth steps we can uniquely identify rest of the two key quartets $K_{1,i}^{10}$ and $K_{2,i}^{10}$. So, finally we will have one choice of K^{10}. The time complexity of these attack is 2^{24} as initially we need to guess w_1, w_2, and m_1 to get the values $K_{0,i}^{10}$.

The attack analysis is quite obvious. One faulty ciphertext reduces the key space to 2^{32} from 2^{128}. Therefore, two faulty ciphertexts will reduce the search space to $(\frac{2^{32}}{2^{128}})^2 \times 2^{128} = \frac{1}{2^{64}}$. This implies only the actual key will left and rest of the guessed keys will be discarded by the attack.

5 Experimental Results

In order validate our attack we performed extensive simulations. Some of the simulation results are presented in this section. We used 8 core Intel Xeon E5606 processor of 2.13 GHz speed running on Linux (Ubuntu 10.4). The attack code was written in C programming language and compiled with gcc-4.4.3 with O3 optimization. The simulation was performed over 100 times on different random keys. Some of the results are shown in Table 1. The attack takes less than 20 minutes to reveal the secret key. The first column of Table 1 represents the random 16-bytes keys which were attacked. The second column represents the number of possible key generated by the attack. The last column represents the total time taken by the attack which corresponds to generating possible keys and then performing brute-force search on them to get the master key.

Table 1. Experimental Results

Random 128-bit Key	Number of Possible Keys	Running Time (Minutes)
aaf3100fdf183ef427464fbf4db85f7a	$3976380416 \approx 2^{31.88}$	17.65
5da4e5407bae5f94cc4a264bf694c0d2	$8744409088 \approx 2^{33.025}$	18.533
19345421476b4e2b72191a845d30942a	$8424587264 \approx 2^{32.971}$	18.791
226156432112475303294a5bc2326a96	$4018579456 \approx 2^{31.90}$	18.883
5da4e5407bae5f94cc4a264bf694c0d2	$4223047680 \approx 2^{31.975}$	19

6 Comparison with the Previous Works

In this section we compare our attack with some of the existing attacks in these area. The first DFA on AES key schedule was proposed by Giraud [10]. Giraud's attack requires 250 faulty ciphertexts and five days execution time to retrieve the secret key. The attack proposed by Chen and Yen in [9], was an improvement over Giraud's attack where around 30 faulty ciphertexts were used. Peacham et. al. in [16] proposed an attack using 12 faulty ciphertext. DFA on AES key schedule using two faulty ciphertexts was first time proposed by Junko Takahashi et. al. [20], which reduced the AES key space to 2^{48} choices. Kim et. al. in [13], further improved the attack and reduced the key space to 2^{32} possible choices using two faulty ciphertexts.

Compared to these attacks our attack requires only one faulty ciphertext. The required brute-force search for our attack is 32-bit which is same as in Kim et. al.'s attack [13]. Therefore, the proposed attack required minimal faulty ciphertexts to mount an attack on AES key schedule. Table 2 shows the comparison.

Table 2. Comparison with existing attack on AES-128 key schedule

Reference	Fault Model	Number of Faults	Exhaustive Search
[9]	Single byte fault	22 to 44	1
[16]	Multi byte fault	12	1
[20]	Multi byte fault	2	2^{48}
[13]	Multi byte fault	2	2^{32}
Our Attack	Multi byte fault	1	2^{32}

7 Conclusions

We proposed an improved attack on AES-128 key schedule. The attack require only one pair of fault-free and faulty ciphertexts. The proposed attack reduces the AES-128 key space to 32-bit. The time complexity of the attack is 2^{32}. In order to validate the attack we have provided extensive simulation results. The simulated attack retrieves the secret key on less than 20 minutes on an 8 core Intel Xeon E5606 processor at 2.13 GHz speed. This shows that the attack is indeed practical.

Acknowledgment. The authors would like to thank the anonymous reviewers for their comments and suggestions. The authors also extend their gratitude to the Department of Information Technology, India for funding this work.

References

1. National Institute of Standards and Technology, Advanced Encryption Standard, NIST FIPS PUB 197 (2001)
2. Ali, S.S., Mukhopadhyay, D.: Acceleration of Differential Fault Analysis of the Advanced Encryption Standard Using Single Fault. Cryptology ePrint Archive, Report 2010/451 (2010), http://eprint.iacr.org/
3. Ali, S.S., Mukhopadhyay, D., Tunstall, M.: Differential Fault Analysis of AES using a Single Multiple-Byte Fault. Cryptology ePrint Archive, Report 2010/636 (2010), http://eprint.iacr.org/
4. Barenghi, A., Bertoni, G., Parrinello, E., Pelosi, G.: Low Voltage Fault Attacks on the RSA Cryptosystem. In: Breveglieri, et al. (eds.) [8], pp. 23–31
5. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997)
6. Blömer, J., Seifert, J.-P.: Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In: Wright, R.N. (ed.) FC 2003. LNCS, vol. 2742, pp. 162–181. Springer, Heidelberg (2003)
7. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)
8. Breveglieri, L., Gueron, S., Koren, I., Naccache, D., Seifert, J.-P. (eds.): Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, Switzerland, September 6. IEEE Computer Society (2009)
9. Chen, C.-N., Yen, S.-M.: Differential Fault Analysis on AES Key Schedule and Some Coutnermeasures. In: Safavi-Naini, R., Seberry, J. (eds.) ACISP 2003. LNCS, vol. 2727, pp. 118–129. Springer, Heidelberg (2003)
10. Giraud, C.: DFA on AES. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) AES 2005. LNCS, vol. 3373, pp. 27–41. Springer, Heidelberg (2005)
11. Dusart, P., Letourneux, G., Vivolo, O.: Differential Fault Analysis on A.E.S. Cryptology ePrint Archive, Report 2003/010 (2003), http://eprint.iacr.org/
12. Fukunaga, T., Takahashi, J.: Practical Fault Attack on a Cryptographic LSI with ISO/IEC 18033-3 Block Ciphers. In: Breveglieri, et al. (eds.) [8], pp. 84–92
13. Kim, C.H., Quisquater, J.-J.: New Differential Fault Analysis on AES Key Schedule: Two Faults Are Enough. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 48–60. Springer, Heidelberg (2008)
14. Moradi, A., Shalmani, M.T.M., Salmasizadeh, M.: A Generalized Method of Differential Fault Attack Against AES Cryptosystem. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 91–100. Springer, Heidelberg (2006)

15. Mukhopadhyay, D.: An Improved Fault Based Attack of the Advanced Encryption Standard. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 421–434. Springer, Heidelberg (2009)
16. Peacham, D., Thomas, B.: A DFA attack against the AES key schedule. SiVenture White Paper 001, October 26 (2006)
17. Piret, G., Quisquater, J.-J.: A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 77–88. Springer, Heidelberg (2003)
18. Selmane, N., Guilley, S., Danger, J.-L.: Practical Setup Time Violation Attacks on AES. In: EDCC, pp. 91–96 (2008)
19. Skorobogatov, S.P., Anderson, R.J.: Optical Fault Induction Attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 2–12. Springer, Heidelberg (2003)
20. Takahashi, J., Fukunaga, T., Yamakoshi, K.: DFA Mechanism on the AES Key Schedule. In: Breveglieri, L., Gueron, S., Koren, I., Naccache, D., Seifert, J.-P. (eds.) FDTC, pp. 62–74. IEEE Computer Society (2007)
21. Tunstall, M., Mukhopadhyay, D.: Differential Fault Analysis of the Advanced Encryption Standard using a Single Fault. Cryptology ePrint Archive, Report 2009/575 (2009), http://eprint.iacr.org/
22. Tunstall, M., Mukhopadhyay, D., Ali, S.: Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In: Ardagna, C.A., Zhou, J. (eds.) WISTP 2011. LNCS, vol. 6633, pp. 224–233. Springer, Heidelberg (2011)

Combined Fault and Side-Channel Attack on Protected Implementations of AES

Thomas Roche, Victor Lomné, and Karim Khalfallah

ANSSI, 51, Bd de la Tour-Maubourg, 75700 Paris 07 SP, France
`firstname.lastname@ssi.gouv.fr`

Abstract. The contribution of this paper is twofold: (1) a novel fault injection attack against AES, based on a new fault model, is proposed. Compared to state-of-the-art attacks, this fault model advantage is to relax constraints on the fault location, and then reduce the a priori knowledge on the implementation. Moreover, the attack algorithm is very simple and leaves room for optimization with respect to specific cases; (2) the fault attack is combined with side-channel analysis in order to defeat fault injection resistant and masked AES implementations. More precisely, our fault injection attack works well even when the attacker has only access to the faulty ciphertexts through a side-channel. Furthermore, the attacks presented in this paper can be extended to any SP-Network.

Keywords: AES, SCA, DPA, DFA, Side-Channel Analysis, Fault Attacks, masking scheme, combined attack.

1 Introduction

In a hostile environment, devices embedding cryptographic algorithms are susceptible to so-called *physical* attacks, namely Side-Channel Analysis (SCA) and Fault Attacks (FA). To recover the key, SCA makes use of physical leakages emanating from a device (power consumption, electromagnetic radiations, ...) while it performs a cryptographic operation. FA exploit logical error(s) induced by the adversary on a device running a cryptographic operation to retrieve the secret.

These two classes of attacks have been widely studied these last years, and countermeasures have been suggested to thwart them. As a first contribution, we propose to study the security of block ciphers, more precisely SP-Networks, against a new type of FA. We will show that in many implementations, it is possible to relax the state-of-the-art fault models of FA. Our second contribution is to attack an implementation where countermeasures against FA and SCA have been embedded such that the faulty outputs are only accessible through side-channel leakage. To that purpose we combine our FA with a classical 1^{st}-order SCA. This second contribution is the true goal of this paper, the FA was indeed designed to this end. We remarked that in order to make use of a statistical analysis such as SCA on faulty computations, one must be able to repeat the same fault several times. Interestingly enough, this property led to a simple and generic FA attack that straightforwardly combines itself with SCA. The attacks proposed in this article will be presented on the Advanced Encryption Standard (AES) [2], because it is the most studied SP-Network, but they can be applied to any SP-based cipher.

The paper is organized as follows: Section 2 describes previous works on SCA, FA, and their countermeasures. A brief summary on combined attacks is given. Section 3 defines some notations used in the rest of the document. Section 4 describes our new fault attack against AES. Section 5 explains our combined fault and side-channel attack against FA- and SCA-resistant implementations of AES. Possible countermeasures against this attack are given in Section 6. Finally, we conclude and give future directions of this study in Section 7.

2 Previous Work

In this section, we briefly present the main SCA and FA against block cipher implementations and the sound countermeasures that have been introduced in the literature. The combination of such countermeasures should thwart both SCA and FA. The idea of combining SCA and FA in order to create a more powerful attack is not new, we will list the previous combined attacks and discuss their strength against secure implementations.

2.1 SCA and Masking

The observation of a block cipher implementation through a *side-channel* (*e.g.* power consumption, electromagnetic radiations, timing, etc...) has been shown to give information about intermediate variables manipulated by the block cipher.

SCA attacks make use of leaked information about intermediate variables that are dependent on the unknown secret key and the known plaintext (we will refer to *sensitive variables* in the sequel). SCA attacks have been first introduced by Kocher *et al.* in their seminal paper describing Simple Power Analysis (SPA for short) and Differential Power Analysis (DPA for short) [22]. Since then, SCA has become a real threat to security devices. DPA is especially powerful because of its robustness to noise, and has been applied to many block ciphers. Moreover, several variants were proposed to enhance the attack success. Let us just cite the main improvements of the statistical distinguisher: CPA [10] and MIA [17]. In the following, we use the term Differential Side-Channel Analysis (DSCA for short) for attacks based on DPA, CPA, MIA or other variants.

Principle of DSCA: let us consider the sensitive variable s, depending on both the unknown secret key k and the known plaintext p. When s is computed by the device, some information on s is leaked and may be captured by the adversary. We denote by $L(s)$ this information leakage. The *leakage function* $L()$ is composed of a deterministic part ϕ and some noise $\mathfrak{B}$ (that includes both algorithmic and electronic noise):

$$L(s) = \phi(s) + \mathfrak{B} \qquad (1)$$

Note that $\mathfrak{B}$ is generally assumed to be close to a Gaussian noise of mean μ and standard deviation σ: $\mathfrak{B} \sim \mathcal{N}(\mu, \sigma)$. In DSCA, the adversary tries to correlate the leaked information $L(s)$ with a prediction $\widehat{L(s)}$ of the behavior of $L(s)$ for various values of p. The predictions are dependent on the choice of ϕ (typically the Hamming weight function, HW for short) and a hypothesis on k. For a given choice of ϕ, the right hypothesis on k is expected to lead to the highest correlation.

The threat of DSCA-like attacks led to the research of sound and efficient countermeasures. Among them, the so-called *masking schemes* constitute the only family of countermeasures for which formal proofs of security have been given [11]. A masking scheme consists in randomizing the manipulated intermediate variables, such that the side channel observations do not yield useful information about sensitive variables.

To illustrate the basic idea, we consider a Boolean masking scheme of order d applied on a function f. Let s be a sensitive variable depending on p and k, let $m_1, \cdots, m_d$ be the d random input masks, and let $m'_1, \cdots, m'_d$ be the d random output masks. The variable s is then randomized in d shares, as follows: $s \oplus m_1 \oplus \cdots \oplus m_d, m_1, \cdots, m_d$. So the function f has to be transformed into a new function f' such that:

$$f'(s \oplus m_1 \oplus \cdots \oplus m_d, m_1, \cdots, m_d) = f(s) \oplus m'_1 \oplus ... \oplus m'_d \qquad (2)$$

At the end of a cipher encryption, one has just to unmask the result with the d masks to get the ciphertext. Several methods have been proposed in the literature to implement such a masking scheme, for different values of d, for instance [3, 20, 22, 26, 28].

Nevertheless, a d^{th}-order masking scheme is susceptible to a $(d+1)^{\text{th}}$-order differential side-channel attack, initially introduced in [23], which consists in combining $(d+1)$ side-channel leakages, *e.g.* the one where the masked sensitive variable is handled and those where the different masks are manipulated. It has to be noted that high-order side-channel attacks become impracticable as the order increases, mainly for two reasons:

– when combining the different side-channel leakages, the resulting noise increases exponentially with d [11];
– the adversary has to guess the instants where the different side-channel leakages occur, which, in the absence of truly efficient detection algorithms, implies an exponential increase of the computation time of the attack with d.

2.2 DFA and Redundancy

The first use of logical faults to break a cipher appeared in a paper of the Bellcore team in 1997 [9]. They showed how a unique fault could be used to break a RSA implementation. The next year, Biham and Shamir introduced the concept of Differential Fault Analysis (DFA) [7] on the Data Encryption Standard (DES) [1]. This concept has been applied later on SP-Networks (mainly on AES), through different attacks [8, 12, 16, 18, 21, 25, 27].

Among them, the attack of Piret *et al.* [27] is probably the most powerful. It allows to retrieve an AES key with only one pair of correct/faulty ciphertexts and a computation time of about 2^{40} (note that improvements have been proposed to reduce this offline phase [19, 32]), when the adversary can induce a fault on a single byte of the state before the penultimate `MixColumn`.

The most popular countermeasure against DFA consists in spatial or temporal redundancy, as explained in [6]. The idea is to compute at least two times the cryptographic operation, and to compare the obtained results. The device provides the result of the cryptographic operation only if the different results are equal. Several variants exist:

– one can (temporally or spatially) duplicate only the last rounds of the block cipher, as most of the DFA focus on inducing faults on the last rounds of block ciphers;
– one can encrypt the plaintext, keep the obtained ciphertext in the device, decrypt the ciphertext and compare the new plaintext with the original one; thus the ciphertext is output only if both are equal.

2.3 Combined Attacks and Combined Countermeasures

Recently, different works proposed to exploit SCA and FA together to develop more powerful attacks. In [30], the author uses a laser beam to increase the power consumption of a micro-controller logic cell and exploits this phenomenon via power analysis. In [29], the authors introduce the Differential Behavioral Analysis (DBA), which combines Safe Error Attack (SEA) and DPA. The DBA requires the adversary to be able to induce a fault during several encryptions. Moreover the fault has to be repeatable, must affect a small number of bits (less than 8) and is also expected to induce a fixed (possibly unknown) *stuck-at* value. The authors show in simulation that the DBA can break an AES hardware implementation, but one has to note that the DBA is ineffective on a masked implementation.

The same year, [4] introduced a Passive and Active Combined Attack (PACA) on Public-Key Cryptography. Later, the same idea was applied on AES in [14]. The fault also assumes a *stuck-at* model. It is shown how a PACA can defeat a masked implementation of AES. Nevertheless, it has to be noticed that this attack can only break a first order masking scheme of AES. In the case of an AES implementation masked at order equal or greater than 2, this PACA just reduces the DSCA of one order.

Thus, to our knowledge, no successful practical side-channel attack has been reported in the literature targeting a masking scheme of AES at order equal or greater than 2. One can then consider that an AES implementation with a masking scheme at order at least 2, with one of the DFA countermeasures cited in Section 2.2, is protected against both state-of-the-art SCA and FA. In Section 5, we show that this is not true if the redundancy check of the fault injection countermeasure manipulates unmasked ciphertexts (or plaintexts).

3 Notations

In the following, we will denote respectively by P, C, K, Z and E the plaintext, the ciphertext, the secret key, the targeted intermediate value and the error. All these values are over 16 bytes. We will frequently use their vector representation in $(\mathrm{GF}(2^8))^{16}$ (*e.g.* $P = (P_0, \cdots, P_{15})$).

We will consider the AES cipher as an example to describe the attacks, but all of them can be straightforwardly extended to any SP-Network. As a consequence, we will use the common AES sub-function names (*e.g.* see [2]) to denote either the non-linear layer (`SubByte`) or the linear layer (`ShiftRow` and `MixColumn`) of the cipher. Moreover the different size parameters will correspond to the AES-128's, involving 10 rounds with 128 bits master key, and 11 round keys (denoted $K^0, \cdots, K^{10}$).

The attacks presented here will mainly involve fault injection in the last round of AES, where only the `SubByte`, `ShiftRow` and the last `AddRoundKey` operation appear. Without loss of generality we omit the last `ShiftRow` operation, as its only effect would be to render indices unreadable.

We will use the so called *correlation* side-channel analysis, based on the Pearson correlation coefficient. Let $\rho(X, Y)$ denote the correlation between the two random variables X and Y, we recall that it is defined by:

$$\rho(X, Y) = \frac{cov(X, Y)}{\sigma(X).\sigma(Y)}$$

4 A New DFA

In this section we present a new perturbation attack on AES. We first consider a FA-unprotected AES implementation.

We will consider several types of AES implementations, masked or not, and propose a DFA attack based on more or less restrictive fault models. Many types of faults can be investigated; in practice, the apparition of one kind of fault or another is strongly dependent on the means of the fault injection (laser beam, glitches, etc...) and to the hardware target (micro-controller, ASIC or FPGA implementations). We believe that the fault model considered in some of these scenario is much less constraining than any other proposed by now. In the next section, those attacks will be combined to SCA in order to defeat a FA resistant and masked implementation.

4.1 Best Case Scenario: Key Schedule Pre-computation

For sake of clarity, let us first consider the target implementation where our DFA attack accepts the widest fault model. The (hardware or software) implementation is as follows:

– the implementation possesses no countermeasure against DFA;
– the secret key is stored in (*e.g.* non-volatile) memory;

– at each reset of the device (that an attacker has control upon, this can correspond to simply powering off the device), the key schedule is run and the round keys are stored in volatile memory and will be used for any later ciphering/deciphering execution.

Remark that the implementation may include SCA countermeasures.

Fault Model. The attack will target the key-scheduling algorithm. The fault shall be transient (*i.e.* future computations shall not be affected) and affect the key schedule in its last but one round. The fault could be of any kind, short to destroy or stop the device. The wanted effect is to have the last but one round key incorrect (and, as a side effect, also the last round key). Furthermore the fault can affect any part of the last two round keys in any possible way, for each affected byte of the next to last round key the attack will recover one of the last round key byte (see Remark 2 below).

When the fault injection is successful, the affected last two round keys will be written as the XOR of the valid round keys and an error:

$$\begin{aligned} \widetilde{K}^9 &= K^9 \oplus E^9 \\ \widetilde{K}^{10} &= K^{10} \oplus E^{10} \end{aligned} \tag{3}$$

Attack Description

1. Encrypt N messages $P^1, \cdots, P^N$.
2. Reset the device and inject an error in the key-schedule.
3. Encrypt $P^1, \cdots, P^N$ once more[1].
4. It produces N pairs of valid-faulty ciphertexts $(C^1, \widetilde{C}^1), \cdots, (C^N, \widetilde{C}^N)$.
5. For each byte index j of the ciphertexts, process separately:

    6. For each hypothetic value $(e_9, e_{10}, k) \in (\mathbb{F}_{2^8})^3$ of the error bytes $E^9_j = e_9$ and $E^{10}_j = e_{10}$ respectively on the 9th and 10th round key, and the subkey byte $K^{10}_j = k$ (of the last round key) create a counter, denoted by $T_{e_9,e_{10},k}$. For each pair of valid-faulty ciphertext $(C^i, \widetilde{C}^i)$, do the following:

        7. Increment the counter $T_{e_9,e_{10},k}$ if

$$\texttt{SubByte}\left(\texttt{SubByte}^{-1}(C^i_j \oplus k) \oplus e_9\right) \oplus k \oplus e_{10} = \widetilde{C}^i_j$$

    8. Find the triplet of error and subkey bytes that led to the highest counter. If the counter is N, then mark the triplet as right. Otherwise the fault injection was not successful.

9. The last round key is retrieved from the different marked subkey bytes when the corresponding error byte e_9 is non-zero.

The complexity of the attack is in $O(N(2^8)^3)$ for N faulty ciphertexts.

[1] In all the sequel, the only constraint on the plaintexts is that they must be used twice, they do not have to be known.

Why the Attack Works. The main idea in the attack above is the ability to inject an error such that any future cipher execution will be impacted by the same error. More precisely, we are here artificially building an unknown but fixed difference in the last round of the cipher. As a matter of fact, for any $0 \leq j \leq 15$, the difference between $\texttt{SubByte}^{-1}(C_j^i \oplus K_j^{10})$ and $\texttt{SubByte}^{-1}(\widetilde{C}_j^i \oplus \widetilde{K}_j^{10})$ is fixed over all $i \leq N$ and equal to E_j^9.

Let us first assume that the fault injection was successful and that only the last two round keys are affected with respectively the error E^9 and E^{10}. After the execution of the attack algorithm, for any $0 \leq j \leq 15$, the counter $T_{E_j^9, E_j^{10}, K_j^{10}}$ is trivially equal to N. On the other hand the probability for a triplet $(e_9, e_{10}, k) \neq (E_j^9, E_j^{10}, K_j^{10})$ to have $T_{e_9,e_{10},k} = N$ in step 8 of the attack algorithm is about $\left(\frac{1}{256}\right)^N$. This result is only a rough estimation considering the AES cipher. The probability is actually dependent on the S-box uniform differentiability, the better the S-box is (in term of resistance against differential attacks), the better the DFA will work[2]. Our simulations confirm that 3 pairs of valid-faulty ciphertexts are enough to get a success rate superior to 90%.

If we consider now that the fault injection impacted more than the last two round keys (*i.e.* the fault model is violated). Then the difference between $\texttt{SubByte}^{-1}(C_j^i \oplus K_j^{10})$ and $\texttt{SubByte}^{-1}(\widetilde{C}_j^i \oplus \widetilde{K}_j^{10})$ (denoted Δ_j^i) is not fixed anymore. This does not change anything for the wrong triplets $(e_9, e_{10}, k) \neq (E_j^9, E_j^{10}, K_j^{10})$, their corresponding counter will have the same probability to reach N. By opposition, the right counter $T_{E_j^9, E_j^{10}, K_j^{10}}$ will have a lower probability to reach N. This probability is in fact dependent on the distribution of the differences Δ_j^i for a fixed j (*e.g.* when the distribution is uniform, the probability to reach N is equal to the other counters).

Remark 1. In the case where the fault model is not respected but still the difference Δ_j^i is fixed with a biased probability, it is possible to adapt the attack algorithm in order to retrieve the key. Indeed, a non-uniform distribution in the Δ_j^i will eventually be distinguished (when N grows) by looking at the $T_{e_9,e_{10},k}$ distribution.

Remark 2. Let us note that if the byte K_j^9 of K^9 is not affected by the fault injection, then the corresponding byte K_j^{10} will not be retrieved by our attack. Hence, the wider the fault, the better the attack. This is in opposition with classical DFA where the fault model restricts the fault size.

The attack described here assumed that the key schedule was pre-computed, let us now consider cipher implementations where the key schedule is re-executed for each cipher run.

4.2 Key Schedule Re-executed at Each Cipher Execution

In the setup where the key schedule is re-computed at each cipher execution, the attack algorithm will have to be modified, now N fault injections being necessary

[2] This is a classical observation in DFA [27].

to get N faulty ciphertexts. The fault model will also have to be modified, two cases can be distinguished whether the key schedule implementation uses masking countermeasures or not. Eventually, in a fully masked implementation, the fault model will be severely restricted.

Unmasked Key Schedule Implementation. In the case of an unmasked key schedule implementation, our attack still targets the key schedule algorithm. The main difference with the attack proposed in previous Section is the fact that a unique error will not be enough anymore to compute N faulty ciphertexts. For each faulty ciphertext computation, the fault injection must be re-applied. As a consequence, the fault injection must possess a new property: a good *repeatability* (*i.e.* two injected faults have a good chance to induce the same error). Indeed, if we consider that, with an identical fault injection setup on the key schedule computation of two cipher executions on the same key, the injected fault is always the same, then the attack here will have the same success rate and same complexity than the attack described in Section 4.1. Let us insist that since the fault is injected on the key schedule, the same values are modified (at each execution the key schedule is manipulating the same intermediate variables), thus, again here, any kind of fault that affected the last two round keys can be written as the XOR of the valid round keys and an error (as in equation 3)

The attack steps are then:

1. Encrypt N messages $P^1, \cdots, P^N$.
2. Encrypt $P^1, \cdots, P^N$ once again, this time with a fault injection during the last but one round of the key schedule.
3. It produces N pairs of valid-faulty ciphertexts $(C^1, \widetilde{C}^1), \cdots, (C^N, \widetilde{C}^N)$.
4. For each byte index j of the ciphertexts, process separately:

   5. For each hypothetic value $(e_9, e_{10}, k) \in (\mathbb{F}_{2^8})^3$ of the errors byte $E^9_j = e_9$ and $E^{10}_j = e_{10}$ respectively on the 9th and 10th round key, and the subkey byte $K^{10}_j = k$ (of the last round key) create a counter, denoted by $T_{e_9,e_{10},k}$. For each pair of valid-faulty ciphertext $(C^i, \widetilde{C}^i)$, do the following:

      6. Increment the counter $T_{e_9,e_{10},k}$ if

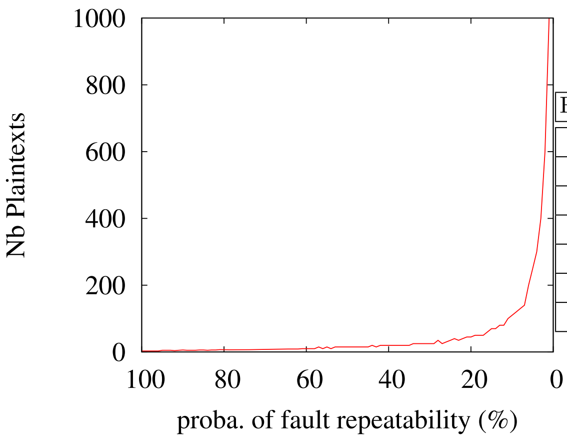
$$\texttt{SubByte}\left(\texttt{SubByte}^{-1}(C^i_j \oplus k) \oplus e_9\right) \oplus k \oplus e_{10} = \widetilde{C}^i_j$$

   7. Find the triplet of error and subkey bytes that led to the highest counter. If the counter is "high enough" compared to the others, then mark the triplet as right. Otherwise, if no triplet can be significantly distinguished from the others, the fault injection was not successful (it might either mean that the fault is not injected at the right location or that the fault repeatability is not high enough).

8. The last round key is retrieved from the different marked subkey bytes when the corresponding error byte e_9 is non-zero.

We believe that the hypothesis about fault injection repeatability is realistic, although of course a 100% repeatability seems a too strong assumption. Hence we simulated the attack (at each cipher execution, the injected fault is either fixed or random with a given probability). Figure 1 shows the exponential cost in the number of plaintexts (N) to get a 90% success rate attack as a function of the repeatability probability (the experiment has been done 100 times for each probability of fault repeatability).

Furthermore, we implemented the AES-128 on a FPGA platform and performed the described attack. We induced the faults by clock glitches during the next to last round of the key schedule and verified that the fault repeatability was good enough: the attack required 15 faulty ciphertexts to recover the 11 round key bytes that have been affected by the fault. According to the table given besides Figure 1, this means that the fault repeatability of our attack is about 50%. The setup and attack are detailed in Annex A.



Fault repeatability	Nb plaintexts
100%	3
90%	6
50%	15
10%	110
5%	250
1%	1000
0%	∞

Fig. 1. Number of valid-, faulty-ciphertext pairs for 90% success rate with respect to the probability of fault injection repeatability (in %)

Remark 3. The attack can be seen as an artificially constructed differential cryptanalysis. As a matter of fact, it works the same way and is based on the same assumption: there exists an output differential that occurs more often than the others. Hence, the classical results on differential cryptanalysis success rate can be straightforwardly applied and formalize our results here (see, for instance, [5]). Let us note here that the value of the output differential (before the last round) is unknown in our case, which does not change much, it shows that the knowledge of the existence of a good differential is what matters and the knowledge of the difference value is useless.

Remark 4. We have considered here that the key schedule is unmasked, it is interesting to add that the attack would also work if the key schedule was masked with a fixed mask (*i.e.* at each key schedule execution the same mask is used). Such fixed mask key schedule is very attractive from the performances point-of-view and yet resistant against 1st-order SPA or profiled attacks on key values.

Masked Implementation. In the case where the implementation is fully masked, *i.e.* the intermediate variables of the whole cipher computation is masked with randomly generated values, the fault model will have to be drastically restricted, we still believe it is relevant when considering certain categories of hardware/injection materials (*e.g.* using a laser beam [24]). We consider here that a Boolean masking scheme is used, the masking may be of any fixed order d (see Section 2.1 for definitions and details about masking countermeasures).

Fault Model. As mentioned in the previous attack scenarios, the gist of the attack is to get pairs of valid-faulty ciphertexts with fixed difference in the last round. Let us now consider that an attacker injects the same fault in two different executions during the last but one round of the masked key schedule, for our attack to work, we would like the following equations 4 and 5 to be satisfied with fixed (a priori unknown) values E^9 and E^{10}:

$$\begin{aligned} \widetilde{K}^9 &= K^9 \oplus M^9 \oplus E^9 \\ \widetilde{K}^{10} &= K^{10} \oplus M^{10} \oplus E^{10} \end{aligned} \tag{4}$$

$$\begin{aligned} \widetilde{K}'^9 &= K^9 \oplus M'^9 \oplus E^9 \\ \widetilde{K}'^{10} &= K^{10} \oplus M'^{10} \oplus E^{10} \end{aligned} \quad , \tag{5}$$

where K^9 and K^{10} are the last two round keys, M^9 and M^{10} (resp. M'^9 and M'^{10}) are the random masks of round keys K^9 and K^{10} for the first (resp. second) cipher execution. $\widetilde{K}^9$ and $\widetilde{K}^{10}$ (resp. $\widetilde{K}'^9$ and $\widetilde{K}'^{10}$) are faulty round keys for the first (resp. second) cipher execution.

Equations 4 and 5 lead to

$$\begin{aligned} \widetilde{K}'^9 \oplus \widetilde{K}^9 &= M^9 \oplus M'^9 \\ \widetilde{K}'^{10} \oplus \widetilde{K}^{10} &= M^{10} \oplus M'^{10} \end{aligned} \tag{6}$$

As these equations must be satisfied for any values of M^i and M'^i ($i \in \{9; 10\}$), it is easy to see[3] that only the XOR error function (error by bit-flip) is acceptable:

$$F_e : x \mapsto x \oplus e \ ,$$

where F_e is the *fault injection function*: $\forall X, F_e(X) = \widetilde{X}$.

[3] The first derivative of the fault injection function F_e is 1: Equation 6 implies that

$$\forall (K, M, M'), \frac{F_e(K \oplus M) \oplus F_e(K \oplus M')}{M \oplus M'} = 1.$$

In the two previous attack scenarios, we *modeled* the fault by a XOR operation; this was possible since the fault was injected on fixed values; Now the target values are varying (because of masking), to write the fault as a XOR, the fault injection needs really to flip the bits.

Assuming that the fault injection function satisfies the constraints (*i.e. bit-flip*), the attack algorithm is similar to the unmasked case, with same success rate as a function of the fault injection repeatability.

Remark 5. The fault model may seem unrealistic, much more than satisfying very good fault repeatability. In fact a trade-off can be made between those two parameters: many fault models will coincide with the *bit-flip* model for a majority of its input space. A good example is the so-called *stuck-at* fault model. Let us consider that the fault injection function is of the following form[4]:

$$F_e : x \mapsto x \& e \ ,$$

where & is the AND bit wise operator. Even though this corresponds to stuck some bits at 0, it can be seen as a *bit-flip* fault injection function with a certain probability of success (*i.e.* they coincide on a fraction of their inputs). This probability of success will add itself to the fault repeatability. Moreover the probability of success increases with the hamming weight of e. When e is the vector made of all 0 bits, no information is leaked about the value of x through the fault and the proposed attack will not be successful. In such a case of fault model (full stuck-at) we point out that other types of combined DFA-SCA attacks exist [14].

Remark 6. When assuming that the fault injection follows the *bit-flip* model (exactly or by approximation), it is equivalent to consider the fault to be injected in the key schedule or directly in the state. Therefore:

– when the key schedule is computed in parallel of the state computation, the fault can touch both parts altogether.
– if the fault targets the state only, the attacker can apply exactly the same attack algorithm as before without the prediction of the error value on the last round subkeys. This simply decreases by a factor of 2^8 the offline complexity of the attack.

5 Combined Attack Against (HO-)Masked and DFA Resistant Implementation of AES

In this section is introduced a combined attack that bypasses both higher order masking countermeasures and integrity check by the simultaneous use of faults injection and side-channel observation.

The combined attack is based on what we point out to be a weakness in the way the error detection mechanism is usually performed. Such a mechanism implies the manipulation of the unmasked faulty ciphertext (or plaintext), therefore allowing to mount a classical 1st-order DSCA.

[4] We present the case of stuck-at 0 but all the following is also true for stuck-at 1 or a mix of both.

First we come back to the different implementations that have been presented in the previous section, now integrating a DFA countermeasure by computation redundancy. Then, in Section 5.2, we propose a generic attack procedure by adding a 1st-order DSCA to the fault injection attack.

5.1 Fault Injection Resistant Implementations

We have proposed in Section 4 a new kind of DFA based on the ability to reproduce a fixed fault for different cipher executions. We presented the attack for several cipher implementations — considering the pre-computation of the key schedule algorithm and the use of masking schemes to thwart SCA attacks — and, for each of them, different fault models led to successful perturbation attacks. Namely:

- The key-schedule is pre-computed once for several cipher executions.
  - Fault Model: Single injection inside the next to last round of the key schedule computation.
- The key-schedule is unmasked (or masked with a fixed mask) and computed at each cipher execution.
  - Fault Model: Injection inside the last but one round of each key schedule computation, does not touch the state computation. The fault needs a good repeatability.
- The key-schedule is masked and computed for each cipher execution.
  - Fault Model: Injection inside the last but one round of each key schedule computation and/or the state computation. It can be modeled, with good approximation, as a *bit-flip* error and possesses good repeatability.

We will moreover assume that one of the countermeasures described in Section 2.2 against DFA has been implemented.

5.2 Combined Attack Description

The idea follows the fact that the faulty ciphertexts are not received by the adversary anymore. However he is still able to observe the computation through a side-channel, and more precisely the manipulation of the (potentially faulty) ciphertext when its integrity is checked[5].

We propose to transform the round key retrieving part of the DFA algorithms in a standard CPA attack (as presented in a generic way in Section 2.1). The attack is straightforwardly extended to fault injection resistant implementations, assuming the knowledge of good approximation of the leakage model (for purpose of clarity we will use here the Hamming weight model as presented in Section 2.1, but, any kind of leakage model could be used). The attack becomes then:

[5] We can add here that such a location in the cipher algorithm is usually easy to spot out from side-channel leakage traces as it corresponds to the end of encryption.

1. Produce N pairs of valid-faulty ciphertexts $(C^1, \widetilde{C}^1), \cdots, (C^N, \widetilde{C}^N)$ (for any of the implementations and their fault models described in Section 4). The faulty ciphertexts are not returned by the chip, the attacker has to record the side-channel traces during the faulty computations (by monitoring the power consumption, the electromagnetic radiation or any exploitable side-channel). We will consider that for each trace Ω_i, there is a known instant t_i such that $\Omega_i[t_i]$ is the side channel observation of the faulty ciphertext $\widetilde{C}^i$ manipulation. Hence, according to the notations introduced in Section 2.1, we have $\Omega_i[t_i] = \phi(s) + \mathfrak{B}$, where s is the value manipulated at time t_i (*i.e.* $\widetilde{C}^i$ or a subpart of it depending on the data register size), ϕ is a deterministic leakage function (*e.g.* Hamming Weight function) and $\mathfrak{B}$ denotes the noise (*e.g.* a Gaussian noise with mean μ and standard deviation σ).
2. For each byte index j of the ciphertexts (included in s), process separately:

    3. For each hypothetic value (e_9, e_{10}, k) of the errors byte $E_j^9 = e_9$ and $E_j^{10} = e_{10}$ respectively on the 9th and 10th round key, and the subkey byte $K_j^{10} = k$ compute the correlation value $\rho_{e_9,e_{10},k}$:

        4. For each ciphertext C^i, compute the *prediction* values:

        $$pred_i = HW(\texttt{SubByte}\left(\texttt{SubByte}^{-1}(C_j^i \oplus k) \oplus e_9\right) \oplus k \oplus e_{10}) \ .$$

        Then compute the Pearson correlation coefficient between the predictions and the observations (the function $\rho(,)$ is recalled in Section 3):

        $$\rho_{e_9,e_{10},k} = \rho(\{pred_i\}_{1 \le i \le N}, \{\Omega_i[t_i]\}_{1 \le i \le N})$$

    5. Find the triplet of error and subkey bytes that led to the highest correlation. If the correlation value is "high enough" compared to the others, then mark the triplet as right. Otherwise, if no triplet can be significantly distinguished from the others, the fault injection was not successful (it might either mean that the fault is not injected at the right location or that the fault repeatability is not high enough with respect to the noise and the number of samples).
6. The last round key is retrieved from the different marked subkey bytes when the corresponding error byte e_9 is non-zero.

The success rate of the attack described above is dependent on the noise in the side-channel measure, the repeatability of the fault injection and, of course, the number of plaintexts (N). A heuristic evaluation of the success rate as a function of these different parameters is given in the next section.

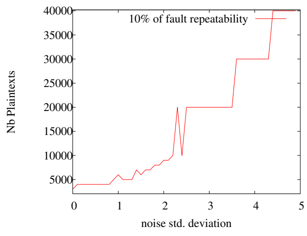
5.3 Evaluation of the Attack Success Rate

In this section we present the results of the attack simulations to show the rough evolution of complexity of the attack (in terms of number of plaintexts) as a function of fault injection repeatability and noise strength in order to reach a fixed success rate of 90%. As in previous experiments, for each choice of parameters, we reached 90% of success over 100 attacks.

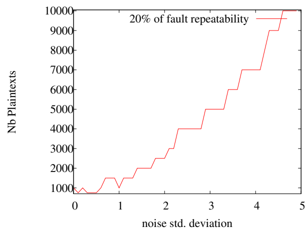
The simulations assumed a leakage function to be, for each ciphertext byte $\widetilde{C}_j^i$, of the form:

$$L(\widetilde{C}_j^i) = HW(\widetilde{C}_j^i) + \mathcal{N}(\mu, \sigma) \ ,$$

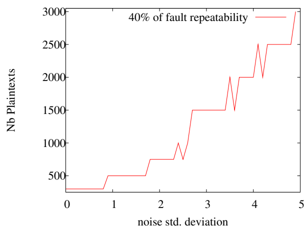
with $\mu = 0$ and σ going from 1 to 5. Figures 2(a) to 2(f) show the results, each of them for a fixed fault injection repeatability. The exponential growth of complexity as a function of the noise standard deviation is clearly visible on the figures, this is also the case with respect to the fault injection repeatability. Figures 2(c) to 2(f) capture practical scenarios where the complexity is limited ($<$ 4000 plaintexts), σ up to 5 and fault repeatability down to 40%. Recall that in the first attack setup (with pre-computed key schedule), the repeatability would be 100%.



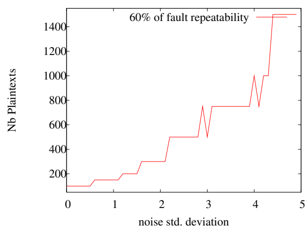
(a) Fault repeatability: 10%



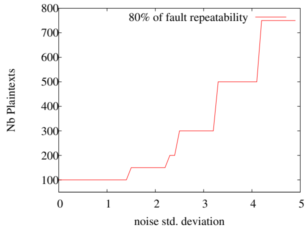
(b) Fault repeatability: 20%



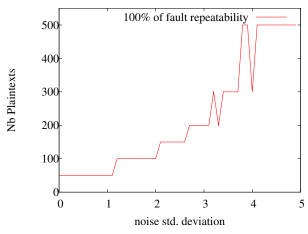
(c) Fault repeatability: 40%



(d) Fault repeatability: 60%



(e) Fault repeatability: 80%



(f) Fault repeatability: 100%

Fig. 2. Number of valid-, faulty-ciphertext pairs for 90% success rate with respect to the gaussian noise standard deviation for each probability of fault injection repeatability (from 10% to 100%)

6 Countermeasures

Some countermeasures against classical SCA and FA attacks are still applicable to the combined attack presented in the previous Section. For instance, the insertion of random delays during the cryptographic computation. This countermeasure is not new, it is already used by industrials and several papers provide a study of its efficiency [15, 31]. Nevertheless, this countermeasure does not make the attack infeasible, it just makes it harder. Moreover there exists methods to bypass this countermeasure, which can be applied in our case (for instance [13]).

Another possible countermeasure could be to keep the ciphertext masked before applying the integrity check. For instance, in the case of the redundancy countermeasure, if C^1 and C^2 are the two ciphertexts obtained from the same encryption and M^1 and M^2 the two different masks used in each instance of the encryption, one gets $C^1 \oplus M^1, C^2 \oplus M^2, M^1$ and M^2. Then one has to XOR $C^1 \oplus M^1$ with M^2 and $C^2 \oplus M^2$ with M^1, and compare the two values. If they are equal, one can unmask the ciphertext without risk of vulnerability. A similar method could be used in the case of the DFA countermeasure consisting of encryption/decryption to check the validity of the ciphertext.

7 Conclusion

In this paper we have introduced a new DFA on AES (or any SP-Network). This attack is particularly interesting when the key schedule is unmasked or masked with a fixed mask. In such cases, we believe this FA attack to have the least restrictive fault model in terms of fault pattern and fault location compared to state-of-the-art attacks. A relaxation on the fault model means that the attacker needs less knowledge about the implementation or less precise fault injection material. The price we pay for this relaxation is the constraint of repeatability of the fault (not necessary in the case of a pre-computed key schedule), we think that in an overwhelming majority of cases, this hypothesis will be easily satisfied. However, when the key-schedule is masked with fresh random values at each execution, the types of faults that lead to a successful attack are severely restricted and the attack becomes harder but still possible in some context.

In a second part, the DFA is combined with a 1^{st}-order SCA to defeat block cipher implementations including not only a masking countermeasure but also an error detection mechanism on the cipher result. Simulations of the attacks verify that, even though the complexity (in term of plaintext number) grows exponentially with the probability of error in the fault repeatability, the attack is still practical for low repeatability ($\sim 40\%$) and realistic noise strength. Finally, we propose some simple countermeasures which can thwart our combined attack.

Future directions will be first to mount such combined attack on an off-the-shelf device and secondly to study the injection of *bit-flip* errors with good repeatability (which lead to the attack of fully masked block cipher implementations).

References

1. Data Encryption Standard, FIPS PUB 46-3 (1999)
2. Advanced Encryption Standard, FIPS PUB 197 (2001)
3. Akkar, M.-L., Giraud, C.: An Implementation of DES and AES, Secure against Some Attacks. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 309–318. Springer, Heidelberg (2001)
4. Amiel, F., Villegas, K., Feix, B., Marcel, L.: Passive and Active Combined Attacks: Combining Fault Attacks and Side Channel Analysis. In: Breveglieri, L., Gueron, S., Koren, I., Naccache, D., Seifert, J.-P. (eds.) FDTC, pp. 92–102 (2007)
5. Selçuk, A.A., Biçak, A.: On Probability of Success in Linear and Differential Cryptanalysis. In: Cimato, S., Persiano, G., Galdi, C. (eds.) SCN 2002. LNCS, vol. 2576, pp. 174–185. Springer, Heidelberg (2003)
6. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. IACR Eprint archive (2004), http://eprint.iacr.org/2004/100
7. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997)
8. Blömer, J., Seifert, J.-P.: Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In: Wright, R.N. (ed.) FC 2003. LNCS, vol. 2742, pp. 162–181. Springer, Heidelberg (2003)
9. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)
10. Brier, E., Clavier, C., Olivier, F.: Correlation Power Analysis with a Leakage Model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)
11. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards Sound Approaches to Counteract Power-Analysis Attacks. In: Wiener (ed.) [33], pp. 398–412
12. Chen, C.-N., Yen, S.-M.: Differential Fault Analysis on AES Key Schedule and Some Coutnermeasures. In: Safavi-Naini, R., Seberry, J. (eds.) ACISP 2003. LNCS, vol. 2727, pp. 118–129. Springer, Heidelberg (2003)
13. Clavier, C., Coron, J.-S., Dabbous, N.: Differential Power Analysis in the Presence of Hardware Countermeasures. In: Koç, Ç.K., Paar, C. (eds.) CHES 2000. LNCS, vol. 1965, pp. 252–263. Springer, Heidelberg (2000)
14. Clavier, C., Feix, B., Gagnerot, G., Roussellet, M.: Passive and Active Combined Attacks on AES: Combining Fault Attacks and Side Channel Analysis. In: Breveglieri, L., Joye, M., Koren, I., Naccache, D., Verbauwhede, I. (eds.) FDTC, pp. 10–19. IEEE Computer Society (2010)
15. Coron, J.-S., Kizhvatov, I.: An Efficient Method for Random Delay Generation in Embedded Software. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 156–170. Springer, Heidelberg (2009)
16. Dusart, P., Letourneux, G., Vivolo, O.: Differential Fault Analysis on A.E.S. In: Zhou, J., Yung, M., Han, Y. (eds.) ACNS 2003. LNCS, vol. 2846, pp. 293–306. Springer, Heidelberg (2003)
17. Gierlichs, B., Batina, L., Tuyls, P., Preneel, B.: Mutual Information Analysis. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 426–442. Springer, Heidelberg (2008)

18. Giraud, C.: DFA on AES. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) AES 2005. LNCS, vol. 3373, pp. 27–41. Springer, Heidelberg (2005)
19. Giraud, C., Thillard, A.: Piret and Quisquater's DFA on AES Revisited. IACR Eprint archive (2010), http://eprint.iacr.org/2010/440
20. Goubin, L., Patarin, J.: DES and Differential Power Analysis (The "Duplication" Method). In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 158–172. Springer, Heidelberg (1999)
21. Kim, C.H., Quisquater, J.-J.: New Differential Fault Analysis on AES Key Schedule: Two Faults Are Enough. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 48–60. Springer, Heidelberg (2008)
22. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener (ed.) [33], pp. 388–397
23. Messerges, T.S.: Using Second-Order Power Analysis to Attack DPA Resistant Software. In: Koç, Ç.K., Paar, C. (eds.) CHES 2000. LNCS, vol. 1965, pp. 238–251. Springer, Heidelberg (2000)
24. Mirbaha, A.P., Dutertre, J.-M., Ribotta, A.-L., Agoyan, M., Tria, A., Naccache, D.: Single-Bit DFA Using Multiple-Byte Laser Fault Injection. In: Technologies for Homeland Security (HST), pp. 113–119 (2010)
25. Mukhopadhyay, D.: An Improved Fault Based Attack of the Advanced Encryption Standard. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 421–434. Springer, Heidelberg (2009)
26. Oswald, E., Mangard, S., Pramstaller, N., Rijmen, V.: A Side-Channel Analysis Resistant Description of the AES S-Box. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 413–423. Springer, Heidelberg (2005)
27. Piret, G., Quisquater, J.-J.: A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 77–88. Springer, Heidelberg (2003)
28. Rivain, M., Prouff, E.: Provably Secure Higher-Order Masking of AES. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 413–427. Springer, Heidelberg (2010)
29. Robisson, B., Manet, P.: Differential Behavioral Analysis. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 413–426. Springer, Heidelberg (2007)
30. Skorobogatov, S.P.: Optically Enhanced Position-Locked Power Analysis. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 61–75. Springer, Heidelberg (2006)
31. Tunstall, M., Benoit, O.: Efficient Use of Random Delays in Embedded Software. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 27–38. Springer, Heidelberg (2007)
32. Tunstall, M., Mukhopadhyay, D., Ali, S.: Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In: Ardagna, C.A., Zhou, J. (eds.) WISTP 2011. LNCS, vol. 6633, pp. 224–233. Springer, Heidelberg (2011)
33. Wiener, M. (ed.): CRYPTO 1999. LNCS, vol. 1666. Springer, Heidelberg (1999)

A An FPGA Implementation Tampered with Clock Glitches

A.1 Architectural Details

In order to validate the assumption of a good repeatability for the fault injection (both for the key-schedule and the encryption state) a machine prototype was developed which allowed us to induce glitches into the clock signal of an AES hardware accelerator. We used an Altera Cyclone II FPGA[6] board involving an AES hardware engine and a main control unit, we will now describe the architecture thereof.

One should note that even though our design is rather specific, as it was tailored to the purpose of the feasibility study/demonstration, it clearly manages to demonstrate the proof of concept of a good repeatability. Indeed, no more assumptions were made on the implementation other than those described in the present paper and which are specifically required for the attack to succeed.

More precisely:

- the AES engine is a hardware unit performing the ten AES rounds in ten successive cycles in a combinational way, therefore using a sole 128-bit register to store the AES state;
- for each encryption run the key-schedule is computed separately from the encryption process, beforehand, allowing the fault to be injected at the same instant each time, and its effect on the last two subkeys to be appreciated according to what matters: its repeatability.
- a control unit is present in the design aside from the AES engine, which handles I/O serial communications with a remote PC and scheduling of computations;
- the AES engine and the control unit each occupies a private clock-domain; an on-chip PLL allows generating a separated clock for each, which share nonetheless both frequency (250 MHz) and phase alignment;
- the control unit drives a clock-enable signal that is fed to a global clock buffer placed ahead the clock network of the AES engine; this clock-enable signal is asserted one cycle out of eight, thus emulating an approx. 30 MHz clock to the encryption core.
- for each encryption run, the user can specify the plaintext and the master-key, along with the precise cycle in which he desires the AES engine clock to be cut short during the key-schedule computation. In the cycle specified by user, the control unit will assert, for two consecutive periods of its own clock, the clock-enable signal of the AES-engine, thereby emulating a transient clock of 250 MHz for an isolated period.

A.2 Attack Description

The above implementation was used to put the attack method described in Section 4.2 to the test:

[6] This is a 130 nm low-cost family.

– we randomly chose 100 plaintexts and performed their encryption without any fault injection;
– the same plaintexts were then encrypted again, this time with a clock-glitch at the next to last cycle of the key-schedule;
– this provided us with 100 pairs of valid-faulty ciphertexts;
– we ran the attack on an increasing number of such pairs. Eventually, only 15 pairs were sufficient to retrieve 11 round key bytes. The other 5 round key bytes were not affected by the fault injection; they could be obtained by exhaustive search.

Note that, according to Figure 1, one can deduce from this result that the probability of fault repeatability is of 50% in this experiment.

Memory-Efficient Fault Countermeasures

Marc Joye and Mohamed Karroumi

Technicolor, Security & Content Protection Labs
1 avenue de Belle Fontaine, 35576 Cesson-Sévigné Cedex, France
{marc.joye,mohamed.karroumi}@technicolor.com

Abstract. An efficient countermeasure against fault attacks for a right-to-left binary exponentiation algorithm was proposed by Boscher, Naciri and Prouff (WISTP, 2007). This countermeasure was later generalized by Baek (Int. J. Inf. Sec., 2010) to the 2^w-ary right-to-left algorithms for any $w \geqslant 1$ (the case $w = 1$ corresponding to the method of Boscher, Naciri and Prouff). In this paper, we modify theses algorithms, devise new coherence relations for error detection, and reduce the memory requirements without sacrificing the performance or the security. In particular, a full register (in working memory) can be gained compared to previous implementations. As a consequence, the implementations described in this paper are particularly well suited to applications for which memory is a premium. This includes smart-card implementations of exponentiation-based cryptosystems.

Keywords: Fault attacks, countermeasures, exponentiation, memory-constrained devices, smart cards.

1 Introduction

Implementation of exponentiation-based cryptosystems needs to be resistant against side-channel attacks. Simple Power Analysis (SPA) or Differential Power Analysis (DPA) target unprotected exponentiation algorithms like the classical square-and-multiply technique [20]. It has been shown that a private key used in RSA can be retrieved by observing the microprocessor's power consumption [21]. A possible countermeasure against SPA consists in always computing a squaring operation followed by a (sometimes dummy) multiplication, regardless the value of the exponent bit. The resulting algorithm is known as the "square-and-multiply always" algorithm [9]. Protection against DPA is usually achieved thanks to randomization techniques.

Fault attacks (FA) constitute another threat for public-key algorithms such as RSA [5]. Different methods have been proposed to protect RSA against fault analysis. There exist basically three main types of countermeasures.

The first type relies on a modification of the RSA modulus. This approach was initiated by Shamir [23] and gave rise to several follow-up papers, e.g. [1,4,18,24]. When these methods are applied to RSA with Chinese remaindering, the main difficulty resides in the protection of the re-combination.

The second type of countermeasures uses the corresponding public operation to check the result before outputting it; for example, one can check the validity of an RSA signature using public exponent e. This however assumes the availability of e and is specific to a given cryptosystem; see [13] for a recent survey on these techniques.

The last type of countermeasures exploits coherence properties between internal variables during the exponentiation algorithm. Such algorithms are sometimes referred to as *self-secure* exponentiation methods. We focus our attention on this third type of countermeasures as they tend to be more generic.

Self-secure secure methods. In 2005, Giraud proposed an exponentiation algorithm, which is secure against SPA and FA [11]. His key idea was to perform a coherence check at the end of the Montgomery powering ladder. Indeed, when evaluating x^d, both values $z := x^{d-1}$ and $y := x^d$ are available at the end of the computation. The check consists then in verifying that $z \cdot x = y$ before outputting the result. We note that this technique nicely combines with Chinese remaindering for RSA.

Boscher, Naciri and Prouff subsequently proposed another SPA-FA resistant exponentiation algorithm [7]. Their method built on an SPA-resistant version of the *right-to-left* binary algorithm[1] for evaluating x^d, that uses three registers. The authors observed that in each iteration of the main loop, the product of two registers is equal to the third one multiplied by the input value x. Their countermeasure consists in using this relation to derive a coherence check in the last iteration. Recently, Baek showed how the coherence check can be adapted to the Yao's right-to-left m-ary algorithm [2].

Yet another self-secure exponentiation method was proposed by Rivain [22]. The underlying idea is different. It relies on a double exponentiation, that is, an algorithm taking on input a pair of exponents (a, b) and returning (x^a, x^b). Applied to RSA, the pair of exponents is $(d, \phi(N) - d)$ and the coherence check verifies that $x^d \cdot x^{\phi(N)-d} = 1$ (modulo N). This method has the advantage of reducing the number of multiplications: on average, 1.65 modular multiplications per bit are required compared to the 2 modular multiplications for Giraud's or Boscher *et al.*'s binary methods. On the down side, the algorithm requires knowledge of $\phi(N)$ (or a multiple thereof like $ed - 1$), which is not necessarily available.

Contributions of the paper. This paper deals with countermeasures against SPA and FA for right-to-left exponentiation algorithms. A drawback of Boscher *et al.*'s and Baek's proposals consists in requiring, on top of the internal registers for the computation of x^d, an additional register for storing the value of the input x, which is needed for the coherence check at the end of the algorithm. The main contribution of this paper is an optimized version of the protected right-to-left m-ary exponentiation algorithm. The optimizations consist of a rearrangement

[1] This scan direction may be preferred as it usually eases the implementation (note that the Montgomery ladder processes the exponent bits from the left to the right).

of Baek's algorithm by modifying the initialization steps and restructuring the inner operations so that a minimal number of registers in memory is used. Plugging $m = 2$ into our general algorithm yields a right-to-left binary method with *one register less* than Boscher *et al.*'s method. As a result, we obtain a *right-to-left* binary algorithm equally efficient (speed- *and* memory-wise) as Giraud's left-to-right algorithm. In the higher-radix case also (i.e., for $m > 2$), we also gain *one full memory register* over the best known right-to-left methods. Finally, as a side result, we offer a detailed memory analysis of Baek's algorithm and a slight variant thereof.

While the main application we had in mind was RSA (in standard and CRT modes), our implementations readily extend to any (finite) abelian group $\mathbb{G}$. For RSA, one has $\mathbb{G} = (\mathbb{Z}/N\mathbb{Z})^\times$ or $(\mathbb{Z}/p\mathbb{Z})^\times \times (\mathbb{Z}/q\mathbb{Z})^\times$. We give a fully *generic* treatment and consider the general problem of computing $y = x^d$ in $\mathbb{G}$. We assume that exponent d is given in a standard format. We do not make *a priori* assumptions on $\mathbb{G}$ so that the presented algorithms can be used in various settings, including elliptic curve cryptosystems — though we present certain shortcuts when for example inverses are easy to compute in $\mathbb{G}$. In particular, we do not assume that the order of $\mathbb{G}$ is known and available to the implementation. We note that memory issues are of paramount importance for devices with limited resources as the amount of working memory generally constitutes the limiting factor in the development of efficient implementations.

Outline. The rest of this paper is organized as follows. In the next section, we review Baek's 2^w-ary exponentiation algorithm, generalizing a binary exponentiation algorithm due to Boscher, Naciri and Prouff. In Section 3, we present a slightly modified variant thereof. Section 4 is the core of the paper. We detail how a full register can be saved, reducing the memory requirements to their minimum: no additional memory is needed for fault detection. Finally, we conclude in Section 5.

2 Exponentiation and Fault Countermeasures

2.1 Yao's *m*-Ary Exponentiation

Consider the m-ary expansion of some positive integer d, $d = \sum_{i=0}^{\ell-1} d_i m^i$ where $0 \leqslant d_i \leqslant m-1$ and $d_{\ell-1} \neq 0$, and an element x in a (multiplicatively written) group $\mathbb{G}$. The goal is to efficiently compute $y = x^d$, that is, $x \cdot x \cdots x$ (d times), for some (secret) exponent d. When $m = 2$, the classical right-to-left binary exponentiation method proceeds from the relation $x^d = \prod_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i = 1}} x^{2^i}$.

This was extended to a general radix $m \geqslant 2$ by Yao [25]. The evaluation of $y = x^d$ is then carried out from the relation

$$y = x^{\sum_{i=0}^{\ell-1} d_i m^i} = \prod_{j=1}^{m-1} \left(\prod_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i = j}} x^{m^i} \right)^j . \tag{1}$$

In more detail, Yao's algorithm makes use of an accumulator A and of $(m-1)$ temporary variables $\mathsf{R}[j]$ $(1 \leqslant j \leqslant m-1)$. Accumulator A is used to contain the successive m^{th} powers of the input element x. Specifically, at the beginning of step i, A contains the value x^{m^i} and is then updated as $\mathsf{A} \leftarrow \mathsf{A}^m$ to contain the value $x^{m^{i+1}}$ for the next step. Temporary variables $\mathsf{R}[j]$ are initialized to $1_{\mathbb{G}}$ (the neutral element in $\mathbb{G}$). At step i, provided that $d_i \neq 0$, the temporary variable corresponding to digit d_i (i.e., $\mathsf{R}[d_i]$) is updated as $\mathsf{R}[d_i] \leftarrow \mathsf{R}[d_i] \cdot \mathsf{A}$; the other temporary variables remaining unchanged. At the end of the computation, all the temporary variables $\mathsf{R}[d_i]$ are aggregated to get the final result as $y = \prod_{j=1}^{m-1} (\mathsf{R}[j])^j$.

The original version of Yao's algorithm, as previously described, is prone to SPA-type attacks since an attacker may distinguish zero digits from nonzero ones. Indeed, when at step i, digit d_i is zero, there is no update of a temporary variable, and thus no multiplication occurs. An easy way to make the algorithm regular is to insert a dummy multiplication when $d_i = 0$ so that the digit 0 is treated as the other digits; see e.g. [9]. This is achieved by using an additional temporary variable, $\mathsf{R}[0]$, which is updated as $\mathsf{R}[0] \leftarrow \mathsf{R}[0] \cdot \mathsf{A}$ when $d_i = 0$. The resulting implementation is depicted in Alg. 1.

Algorithm 1. Yao's algorithm (with dummy multiplication)

Input: $x \in \mathbb{G}$ and $d = (d_{\ell-1}, \dots, d_0)_m \in \mathbb{N}$
Output: $y = x^d$

```
    /* Initialization */
 1: for $i = 0$ to $m-1$ do $\mathsf{R}[i] \leftarrow 1_{\mathbb{G}}$
 2: $\mathsf{A} \leftarrow x$
    /* Main loop */
 3: for $i = 0$ to $\ell - 1$ do
 4:     $\mathsf{R}[d_i] \leftarrow \mathsf{R}[d_i] \cdot \mathsf{A}$
 5:     $\mathsf{A} \leftarrow \mathsf{A}^m$
 6: end for
    /* Aggregation */
 7: $\mathsf{A} \leftarrow \mathsf{R}[m-1]$
 8: for $i = m-2$ down to 1 do
 9:     $\mathsf{R}[i] \leftarrow \mathsf{R}[i] \cdot \mathsf{R}[i+1]$
10:     $\mathsf{A} \leftarrow \mathsf{A} \cdot \mathsf{R}[i]$
11: end for
12: return $\mathsf{A}$
```

Although protected against SPA-type attacks, the algorithm becomes now vulnerable to safe-error attacks [26]. By timely inducing a fault during the multiplication at step i, an attacker may guess whether the operation is dummy or not (and thus whether the corresponding digit is zero or not) from the output value: a correct output value indicates that digit d_i is 0. Again, countermeasures exist. For example, exponent d can be recoded prior entering Yao's exponentiation algorithm in such a way that all digits are nonzero. This is the approach followed in [14] where a regular recoding algorithm with digits in the set $\{1, \ldots, m\}$ is presented.

2.2 Protecting against Faults

This section addresses the more general question of protecting against fault attacks (which encompasses protecting against safe-error attacks).

Baek, generalizing an earlier method due to Boscher, Naciri and Prouff [7], showed in a recent paper [2] how Algorithm 1 can be adapted so as to resist fault attacks. Defining

$$L_j = \prod_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i = j}} x^{m^i} \qquad (\text{for } 0 \leqslant j \leqslant m-1)\,,$$

Equation (1) can be rewritten as $y = \prod_{j=1}^{m-1} (L_j)^j$. Hence, defining

$$T = \prod_{j=0}^{m-2} (L_j)^{m-1-j}\,,$$

it follows that

$$\begin{aligned} y \cdot T &= \prod_{j=0}^{m-1} (L_j)^j \cdot \prod_{j=0}^{m-1} (L_j)^{m-1-j} = \left(\prod_{j=0}^{m-1} L_j \right)^{m-1} = \left(\prod_{0 \leqslant i \leqslant \ell-1} x^{m^i} \right)^{m-1} = \left(x^{m-1} \right)^{\sum_{0 \leqslant i \leqslant \ell-1} m^i} \\ &= x^{m^\ell - 1} \end{aligned}$$

and therefore

$$y \cdot T \cdot x = x^{m^\ell}\ . \tag{2}$$

This relation is the basic idea behind the protection against faults. It serves as a coherence check between the different values involved in the computation. If the content of one of the temporary variables or of the accumulator is corrupted during the computation, then the coherence check will very likely fail and therefore the faulty computation can be detected and notified.

Binary Case. This case corresponds to the method of Boscher, Naciri and Prouff. When $m = 2$, the value of T simplifies to $T = L_0$. Further, noting that

$$d = \sum_{i=0}^{\ell-1} d_i\, 2^i = \sum_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i = 1}} 2^i$$

the binary case (i.e., $m = 2$) also implies $y = L_1$. As a result, the coherence test (Eq. (2)) becomes

$$L_0 \cdot L_1 \cdot x \overset{?}{=} x^{2^\ell}\ . \tag{3}$$

Algorithmically, this translates into:

Algorithm 2. Binary SPA-FA resistant algorithm [7]

Input: $x \in \mathbb{G}$ and $d = (d_{\ell-1}, \ldots, d_0)_2 \in \mathbb{N}$
Output: $y = x^d$

```
   /* Initialization */
1: $\mathsf{R}[0] \leftarrow 1_{\mathbb{G}}$; $\mathsf{R}[1] \leftarrow 1_{\mathbb{G}}$
2: $\mathsf{A} \leftarrow x$
   /* Main loop */
3: for $i = 0$ to $\ell - 1$ do
4:     $\mathsf{R}[d_i] \leftarrow \mathsf{R}[d_i] \cdot \mathsf{A}$
5:     $\mathsf{A} \leftarrow \mathsf{A}^2$
6: end for
   /* Error detection */
7: $\mathsf{R}[0] \leftarrow \mathsf{R}[0] \cdot \mathsf{R}[1]$
8: if ($\mathsf{R}[0] \cdot x \neq \mathsf{A}$) then return 'error'
9: return $\mathsf{R}[1]$
```

Remark 1. As presented, the previous implementation is not protected against second-order fault attacks. If an attacker induces a first fault and next a second fault during the error detection (at Line 8 in Alg. 2), then a faulty result may be returned and possibly exploited to infer information on secret value d. Such attacks were reported in [17]. An efficient countermeasure relying on the so-called lock-principle was later described in [10]. It is assumed that the error detection is done in this way or in an equivalent way so as to make it effective against second-order attacks.

Higher-Radix Case. For the case $m = 2^w$, Baek suggests to evaluate, just after the main loop in Alg. 1, the quantities $y \leftarrow \prod_{j=1}^{m-1} \mathsf{R}[j]^j$ and $T \leftarrow \prod_{j=0}^{m-2} \mathsf{R}[j]^{m-1-j}$, and then to check whether $T \cdot y \cdot x = \mathsf{A}$ (note that A contains x^{m^ℓ} output of the main loop). The update of accumulator A, $\mathsf{A} \leftarrow \mathsf{A}^m$ with $m = 2^w$, is done via w repeated squarings. The evaluations of y and T are done as the aggregation in Alg. 1, the total cost of which amounts to $2 \times 2(m-2)$ multiplications. Therefore, as the coherence check requires two multiplications, if M and S respectively represent the cost of a multiplication and a squaring in $\mathbb{G}$, Baek's original algorithm requires altogether $\underline{(\ell + 4(2^w - 2) + 2)\mathsf{M} + \ell w \mathsf{S}}$, where $\ell = \lceil |d|_2 / w \rceil$ and $|d|_2$ is the bit-length of d.

3 A Variant of Baek's Algorithm

This section offers a detailed memory analysis for Baek's algorithm. We show how rearranging the operations allows a better management of the working memory. As a bonus, the total cost is also slightly reduced.

Let $m = 2^w$ for some $w > 1$. In a nutshell, using the notation of §2.2, Baek's method proceeds in the following way:

1. Compute $y = x^d$ using Algorithm 1;
2. Compute T and next the product $S := y \cdot T$;
3. Check whether $S \cdot x \overset{?}{=} x^{m^\ell}$;
4. If so, return y.

The computation of y is evaluated in the aggregation step as $y = \prod_{i=1}^{m-1} \mathsf{R}[i]^i$; cf. Lines 7–11 in Alg. 1. First, we note the temporary variable $\mathsf{R}[m-1]$ can serve as the accumulator for the aggregation. This allows us to save A, which contains the value of x^{m^ℓ} output of the main loop — the value of x^{m^ℓ} being needed for the coherence check, $S \cdot x \overset{?}{=} x^{m^\ell}$. More explicitly, we rewrite the aggregation step as:

```
/* Aggregation */
for $i = m - 2$ down to 1 do
    $\mathsf{R}[i] \leftarrow \mathsf{R}[i] \cdot \mathsf{R}[i+1]$
    $\mathsf{R}[m-1] \leftarrow \mathsf{R}[m-1] \cdot \mathsf{R}[i]$
end for
```

After the aggregation, the temporary variable $\mathsf{R}[m-1]$ contains the value of y. A close inspection shows also that $\mathsf{R}[1]$ contains $\prod_{i=1}^{m-1} L_i$. Further, since as shown in §2.2

$$S = y \cdot T = \left(\prod_{i=0}^{m-1} L_i\right)^{m-1}$$

and since $\mathsf{R}[0]$ contains L_0, the value of S can be obtained as $(\mathsf{R}[0] \cdot \mathsf{R}[1])^{m-1}$. Therefore, instead of computing the quantity $T = \prod_{j=0}^{m-2} \mathsf{R}[j]^{m-1-j}$ as done in [2], we suggest to directly compute the product $y \cdot T$ by raising $\mathsf{R}[0] \cdot \mathsf{R}[1]$ to the power $m - 1$. This avoid the use of any additional temporary variables. Furthermore, as in the binary representation of $m - 1 = 2^w - 1$, the bits are all equal to one, the powering to $m-1$ can be carried out through $w-1$ squarings and multiplications. This trades $2(2^w - 2)\mathsf{M}$ in Baek's original algorithm against $(w-1)\mathsf{S} + (w-1)\mathsf{M}$. The complete algorithm is depicted in Alg. 3.

The total cost of our variant drops to $\underline{(\ell + 2(2^w - 2) + w + 1)\mathsf{M} + (\ell w + w - 1)\mathsf{S}}$. It requires $2^w + 1$ registers (i.e., 2^w temporary variables $\mathsf{R}[j]$, $0 \leqslant j \leqslant 2^w - 1$, and accumulator A) as well as input value x for the coherence check.

Remark 2. When $m = 2^w$, the powering to $m - 1$ for the computation of $S = R^{m-1}$ where $R := \prod_{0 \leqslant j \leqslant m-1} L_j$ could be optimized. In [8], Brauer explains how to obtain a short addition chain for $2^w - 1$ from an addition chain for w. However, as the optimal value for w is rather small for typical cryptographic sizes (i.e., $w \leqslant 6$ as shown in Table 1), we do not discuss this issue further and stick to a simple binary algorithm for the computation of S. Alternatively, when the computation of an inverse in $\mathbb{G}$ is not expensive, R^{2^w-1} can be evaluated as $R^{2^w} \cdot R^{-1}$.

Algorithm 3. Modified Baek's algorithm

Input: $x \in \mathbb{G}$ and $d = (d_{\ell-1}, \ldots, d_0)_{2^w} \in \mathbb{N}$, $w > 1$
Output: $y = x^d$

```
    /* Initialization */
 1: for $i = 0$ to $2^w - 1$ do $\mathsf{R}[i] \leftarrow 1_{\mathbb{G}}$
 2: $\mathsf{A} \leftarrow x$
    /* Main loop */
 3: for $i = 0$ to $\ell - 1$ do
 4:     $\mathsf{R}[d_i] \leftarrow \mathsf{R}[d_i] \cdot \mathsf{A}$
 5:     $\mathsf{A} \leftarrow \mathsf{A}^{2^w}$
 6: end for
    /* Aggregation */
 7: for $i = 2^w - 2$ down to 1 do
 8:     $\mathsf{R}[i] \leftarrow \mathsf{R}[i] \cdot \mathsf{R}[i+1]$
 9:     $\mathsf{R}[2^w - 1] \leftarrow \mathsf{R}[2^w - 1] \cdot \mathsf{R}[i]$
10: end for
    /* Error detection */
11: $\mathsf{R}[0] \leftarrow \mathsf{R}[0] \cdot \mathsf{R}[1]$; $\mathsf{R}[1] \leftarrow \mathsf{R}[0]$
12: for $i = 1$ to $w - 1$ do
13:     $\mathsf{R}[1] \leftarrow \mathsf{R}[1]^2$
14:     $\mathsf{R}[0] \leftarrow \mathsf{R}[0] \cdot \mathsf{R}[1]$
15: end for
16: if ($\mathsf{R}[0] \cdot x \neq \mathsf{A}$) then return 'error'
17: return $\mathsf{R}[2^w - 1]$
```

The next table compares the proposed variant for various sizes of d and w.

Table 1. Number of multiplications for various sizes of d and w

		1024 bits		1536 bits		2048 bits	
		S/M=1	S/M=.8	S/M=1	S/M=.8	S/M=1	S/M=.8
Boscher *et al.* [7]	$w = 1$	**2050**	**1845**	**3074**	**2767**	**4098**	**3688**
Baek [2]	$w = 2$	1546	1341	2314	2007	3082	2672
	$w = 3$	1391	1187	2074	1767	2757	2347
	$w = 4$	**1338**	**1133**	1978	1671	2618	2208
	$w = 5$	1351	1146	**1965**	**1658**	**2580**	**2170**
	$w = 6$	1445	1230	2042	1735	2639	2230
Our variant	$w = 2$	1544	1339	2312	2005	3080	2670
	$w = 3$	1383	1178	2066	1758	2749	2339
	$w = 4$	1316	1111	1956	1648	2596	2186
	$w = 5$	**1299**	**1093**	**1913**	**1605**	2528	2117
	$w = 6$	1331	1125	1928	1620	**2525**	**2114**

4 Memory-Efficient Methods

We have seen in the previous section that a memory-optimized variant of Baek's algorithm requires $m+1$ registers *together with the input value* x for the coherence check:

$$S \cdot x \overset{?}{=} x^{m^\ell} \quad \text{where } S = y \cdot T$$

(see Eq. (2)). Likewise, in the binary case, Boscher *et al.*'s algorithm requires $2+1=3$ registers together with input value x.

When the computation of an inverse is not expensive in $\mathbb{G}$, a classical trick to avoid the storage of the complete value of x consists in computing a κ-bit digest thereof, say $h = H(x)$ for some function $H : \mathbb{G} \to \{0,1\}^\kappa$, at the beginning of the computation, and to replace the coherence check with

$$h \overset{?}{=} H\Big(S^{-1} \cdot x^{m^\ell}\Big) \ . \tag{4}$$

Such a method is mostly useful when $\kappa \ll |x|_2$. Note also that κ cannot be chosen too small, otherwise the coherence check (Eq. (4)) could be satisfied with a non-negligible probability, even in the presence of faults.

In this section, we consider *generic* countermeasures in the sense that they work equally well in any group $\mathbb{G}$ (even of unknown order). In particular, they do not require that computing inverses is fast. Moreover, they do not need further memory requirements: $m+1$ registers will suffice to get a protected implementation. Finally, they do not degrade the security level: the error detection probability remains unchanged compared to Baek's algorithm or to Boscher *et al.*'s algorithm in the binary case.

4.1 SPA-FA Resistant Right-to-Left m-Ary Exponentiation

The main observation in Baek's algorithm (or in the binary version) for the computation of $y = x^d$ is that the product of the temporary variables is independent of exponent d.

In Algorithm 3, accumulator A is initialized to x and the temporary variables, $\mathsf{R}[j]$, to $1_\mathbb{G}$. In each step of the main loop, exactly *one* temporary variable is updated as $\mathsf{R}[d_i] \leftarrow \mathsf{R}[d_i] \cdot \mathsf{A}$ and the accumulator is updated as $\mathsf{A} \leftarrow \mathsf{A}^m$ (with $m = 2^w$). To avoid confusion, we let $\mathsf{R}[j]^{(i)}$ (resp. $\mathsf{A}^{(i)}$) denote the content of the temporary variable $\mathsf{R}[j]$ (resp. accumulator A) before entering step i. We have:

$$\begin{cases} \mathsf{R}[j]^{(i+1)} = \mathsf{R}[j]^{(i)} \cdot \mathsf{A}^{(i)} & \text{for } j = d_i \\ \mathsf{R}[j]^{(i+1)} = \mathsf{R}[j]^{(i)} & \text{for } j \neq d_i \\ \mathsf{A}^{(i+1)} = \left(\mathsf{A}^{(i)}\right)^m & \end{cases}$$

and consequently the product of all the temporary variables satisfies

$$\begin{aligned}
R^{(i+1)} := \prod_{j=0}^{m-1} \mathsf{R}[j]^{(i+1)} &= \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(i)}\right) \cdot \mathsf{A}^{(i)} \\
&= \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(i-1)}\right) \cdot \mathsf{A}^{(i-1)} \cdot \mathsf{A}^{(i)} \\
&\vdots \\
&= \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(0)}\right) \cdot \mathsf{A}^{(0)} \cdots \mathsf{A}^{(i-1)} \cdot \mathsf{A}^{(i)} \\
&= \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(0)}\right) \cdot \prod_{k=0}^{i} \mathsf{A}^{(k)} = \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(0)}\right) \cdot \left(\mathsf{A}^{(0)}\right)^{1+m+m^2+\cdots+m^i} \\
&= \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(0)}\right) \cdot \left(\mathsf{A}^{(0)}\right)^{\frac{m^{i+1}-1}{m-1}} \quad .
\end{aligned}$$

We observe that if the accumulator A is initialized to x^α and the temporary variables are initialized so that their product is equal to x^β (i.e., if $\mathsf{A}^{(0)} = x^\alpha$ and $\prod_{0\leqslant j\leqslant m-1} \mathsf{R}[j]^{(0)} = x^\beta$) then the previous relation becomes

$$R^{(i+1)} = x^{\beta + \frac{\alpha(m^{i+1}-1)}{m-1}} \quad . \tag{5}$$

Since Equation (5) is independent of d, our idea consists in using it to build a coherence check for appropriately chosen values for $\alpha \neq 0$ and β. There are several options.

Basic Case: $\boldsymbol{\alpha = 1}$ and $\boldsymbol{\beta = 0}$. In this case, we get $R^{(i+1)} = x^{\frac{m^{i+1}-1}{m-1}}$ and in particular

$$R := R^{(\ell)} = x^{\frac{m^\ell-1}{m-1}} \quad .$$

This corresponds to the countermeasures proposed by Boscher *et al.* for the binary exponentiation and by Baek for higher radices. Indeed, setting $S = R^{m-1}$ leads to

$$S \cdot x = R^{m-1} \cdot x = x^{m^\ell - 1} \cdot x = x^{m^\ell} \quad .$$

Fractional Case: $\boldsymbol{\alpha = 1}$ and $\boldsymbol{\beta = \frac{1}{m-1}}$. Plugging these values in Eq. (5) yields $R^{(i+1)} = x^{\frac{m^{i+1}}{m-1}}$ and thus

$$R := R^{(\ell)} = x^{\frac{m^\ell}{m-1}} \quad .$$

In this case, it turns out that the numerator in the power of x is m^ℓ (and not $m^\ell - 1$ as in the basic case). Therefore, a simple coherence check is to compare $S := R^{m-1}$ with x^{m^ℓ}. The main advantage is that the value of x is no longer involved.

On the minus side, since $\beta = \frac{1}{m-1}$ is not an integer, the initialization of the temporary variables requires the computation of roots in $\mathbb{G}$. In general, this is a rather expensive operation but there exist cases where it is not. Examples include point halving (i.e., square roots in $\mathbb{G}$) in odd-order subgroups of binary elliptic curves [19] or cube roots in [the multiplicative group of] a finite field of characteristic three [3].

Generic Case: $\alpha = \beta(m-1)$. This setting generalizes the fractional case. However, to avoid the computation of roots in $\mathbb{G}$, we restrict α and β to integer values. We have:

$$R := R^{(\ell)} = x^{\beta\, m^{\ell}} \implies R^{m-1} = x^{\alpha\, m^{\ell}} \ .$$

It is worth noting here that at the end of the main loop, A contains the value of $x^{\alpha\, m^{\ell}}$. The relation $R^{m-1} = x^{\alpha\, m^{\ell}}$ therefore provides a coherence check to protect against faults. The main advantage over the previous methods is that R^{m-1} appears as the exact value present in the accumulator and so there is no need to keep the value of x.

It remains now to show how to compute $y = x^d$ when A is initialized to $x^{\alpha} = x^{\beta(m-1)}$ and $\prod_{0\leqslant j\leqslant m-1} \mathsf{R}[j]$ is initialized to x^{β}. We write:

$$d = \alpha \cdot q + r \qquad \text{with } q = \left\lfloor \frac{d}{\alpha} \right\rfloor \text{and } r = d \bmod \alpha \ . \tag{6}$$

Hence, if $\sum_{i=0}^{\ell'-1} q_i\, m^i$ denotes the m-ary expansion of q then Yao's method yields

$$x^{d-r} = (x^{\alpha})^{\sum_{i=0}^{\ell'-1} q_i\, m^i} = \prod_{j=1}^{m-1} (\boldsymbol{L}_j)^j \quad \text{where } \boldsymbol{L}_j = \prod_{\substack{0\leqslant i\leqslant \ell'-1\\ q_i = j}} X^{m^i} \text{ and } X = x^{\alpha} \ . \tag{7}$$

Remember from Section 3 that the temporary variable $\mathsf{R}[0]$ is not used in the computation of y. Now, assume in Algorithm 3 that temporary variables $\mathsf{R}[j]$ are initialized to x^{e_j} for some integer e_j, for $0 \leqslant j \leqslant m-1$, so that

$$\sum_{j=1}^{m-1} j \cdot e_j = r \quad \text{and} \quad \sum_{j=0}^{m-1} e_j = \beta \ . \tag{8}$$

In that case, it can be easily verified that if A is initialized to $x^{\beta(m-1)}$, then Algorithm 3 (with the error detection adapted as above explained) will return the value of $y = x^d$ (or an error notification). Indeed, from Eq. (7), we get

$$y = x^r \cdot \prod_{j=1}^{m-1} (\boldsymbol{L}_j)^j = \prod_{j=1}^{m-1} (x^{e_j} \cdot \boldsymbol{L}_j)^j$$

(and $R^{m-1} = X^{m^{\ell'}}$).

Several strategies are possible in order to fulfill Eq. (8). The simplest one is to select $\beta = 1$ (and thus $\alpha = m - 1$). Since $0 \leqslant r < \alpha$ ($= m - 1$), a solution to Eq. (8) is then given by

$$\begin{cases} e_r = 1 \\ e_j = 0 \quad \text{for } 0 \leqslant j \leqslant m-1 \text{ and } j \neq r \end{cases} .$$

We detail below the corresponding algorithm for $m > 2$. The case $m = 2$ is presented in §4.3.

Algorithm 4. Memory-efficient SPA-FA resistant algorithm

Input: $x \in \mathbb{G}$ and $d \in \mathbb{N}$
Output: $y = x^d$

```
    /* Initialization */
 1: $\mathsf{A} \leftarrow x^{m-1}$
 2: for $i = 0$ to $m - 1$ do $\mathsf{R}[i] \leftarrow 1_{\mathbb{G}}$
 3: $\mathsf{R}[d \bmod (m-1)] \leftarrow x$
 4: $d \leftarrow \lfloor d/(m-1) \rfloor = (d'_{\ell'-1}, \ldots, d'_0)_m$
    /* Main loop */
 5: for $i = 0$ to $\ell' - 1$ do
 6:     $\mathsf{R}[d'_i] \leftarrow \mathsf{R}[d'_i] \cdot \mathsf{A}$
 7:     $\mathsf{A} \leftarrow \mathsf{A}^m$
 8: end for
    /* Aggregation */
 9: for $i = m - 2$ down to 1 do
10:     $\mathsf{R}[i] \leftarrow \mathsf{R}[i] \cdot \mathsf{R}[i+1]$
11:     $\mathsf{R}[m-1] \leftarrow \mathsf{R}[m-1] \cdot \mathsf{R}[i]$
12: end for
    /* Error detection */
13: $\mathsf{R}[0] \leftarrow \mathsf{R}[0] \cdot \mathsf{R}[1]$;
14: $\mathsf{R}[0] \leftarrow \mathsf{R}[0]^{m-1}$
15: if ($\mathsf{R}[0] \neq \mathsf{A}$) then return 'error'
16: return $\mathsf{R}[m-1]$
```

Remark 3. For the sake of clarity and in order not to focus to a specific implementation, the error detection in Alg. 4 is written with an if-branching. This may be subject to second-order fault attacks. In practice, if second-order attacks are a concern, precautions need to be taken and the if-branching should be rewritten with appropriate measures; cf. Remark 1.

4.2 Dealing with the Neutral Element $1_{\mathbb{G}}$

In certain cases, multiplication by neutral element $1_{\mathbb{G}}$ may be distinguished, which is turn, may leak information on the secret exponent d.

Small order elements. To avoid this, the temporary variables $\mathsf{R}[j]$ can be multiplied by an element of small order in $\mathbb{G}$ in the initialization step. As an illustration, suppose that they are all multiplied by some element h of order 2. More specifically, suppose that the initialization in Alg. 4 (where $\alpha = m - 1$ and $\beta = 1$) is

```
/* Initialization */
$\mathsf{A} \leftarrow x^{m-1}$
**for** $i = 0$ to $m - 1$ **do** $\mathsf{R}[i] \leftarrow h$
$\mathsf{R}[d \bmod (m-1)] \leftarrow h \cdot x$
$d \leftarrow \lfloor d/(m-1) \rfloor = (d'_{\ell'-1}, \ldots, d'_0)_m$
```

for some $h \in \mathbb{G}$ such that $h^2 = 1_{\mathbb{G}}$. Then in each iteration, it is easily seen that the product of all $\mathsf{R}[j]$'s will contain a surplus factor h^m, or from Eq. (5) that

$$R^{(i+1)} := \prod_{j=0}^{m-1} \mathsf{R}[j]^{(i+1)} = h^m \cdot x^{\beta + \frac{\alpha(m^{i+1}-1)}{m-1}} = h^m \cdot x^{m^{i+1}} \implies R := R^{(\ell)} = h^m \cdot x^{m^\ell} \ .$$

When m is even (which is always the case for $m = 2^w$) then $h^m = 1$ and so the coherence check is unchanged: $R^{m-1} \overset{?}{=} x^{\alpha\, m^\ell}$ with $\alpha = (m-1)$. Furthermore, the computation of y is also unchanged when $m = 2^w$ and $w > 1$. Indeed, letting $r = d \bmod (m-1)$, we have from Eq. (7):

$$\prod_{j=1}^{m-1} \left(\mathsf{R}[j]^{(\ell)}\right)^j = \left(\prod_{\substack{1 \leqslant j \leqslant m-1 \\ j \neq r}} h^j \right) \cdot (h \cdot x)^r \cdot x^{d-r} = h^{\sum_{j=1}^{m-1} j} \cdot x^d = h^{\frac{m(m-1)}{2}} \cdot y = y$$

since $m(m-1)/2$ is even when $m = 2^w$ and $w > 1$. In the binary case (i.e., when $w = 1$), we have $m(m-1)/2 = 1$ and consequently the above product needs to be multiplied by h to get the correct output; see also §4.3 for alternative implementations. Note that for the RSA cryptosystem with a modulus N, we can take $h = N - 1$ which is of order 2 since $(N-1)^2 = 1 \pmod N$. The technique can also be adapted to other elements of small order.

Invertible elements. Prime-order elliptic curves obviously do not possess elements of small order. We present below a solution for groups $\mathbb{G}$ wherein the computation of inverses is fast (as it is the case for elliptic curves).

A method used in [13] consists in initializing all the registers to x. This initialization method does not work as is and has to be adapted here. In order to verify Eq. (5), $\mathsf{R}[j]$ should be initialized to x^{1+e_j} so that

$$\sum_{j=1}^{m-1} j \cdot e_j = r' \quad \text{and} \quad \sum_{j=0}^{m-1} (1 + e_j) = 1 \ .$$

Again there are several possible solutions to the previous relation. We may for example define:

– if $r' = 0$

$$\begin{cases} (e_0,\ e_{m-1}) = (2-m,\ -2) & \\ e_j = 0 & \text{for } 1 \leqslant j \leqslant m-2 \end{cases};$$

– if $r' \neq 0$

$$\begin{cases} (e_0,\ e_{m-1}) = (1-m,\ -2) & \\ e_j = 0 & \text{for } 1 \leqslant j \leqslant m-2 \text{ and } j \neq r' \\ e_{r'} = 1 & \end{cases}.$$

$\mathsf{R}[j]$ (for $1 \leqslant j \leqslant m-2$) are then initialized to x and $\mathsf{R}[m-1]$ is set to $x^{1+e_{m-1}} = x^{-1}$. Since $\mathsf{R}[j]$ is raised to the power j within the aggregation step, we subtract $\sum_{j=1}^{m-2} j - (m-1) = \frac{(m-1)\cdot(m-4)}{2}$ from d prior to the exponentiation, i.e., we compute $d' = d - \frac{(m-1)\cdot(m-4)}{2}$. Next, we write $d' = (m-1)\cdot q' + r'$ with $0 \leqslant r' < m-1$. Finally, $\mathsf{R}[0]$ is initialized to $x^{1+e_0} = x^{-(m-2)}$ and $\mathsf{R}[r']$ is initialized to

$$x^{1+e'_r} = \begin{cases} x^{-(m-3)} & \text{if } r' = 0 \\ x^{1+1} = x^2 & \text{otherwise} \end{cases}.$$

Noting that $x^{-(m-3)} = x^{-(m-2)} \cdot x$ and $x^2 = x \cdot x$, and since $r' \leqslant m-2$ the above procedure can be implemented in a regular fashion by replacing the initialization of Alg. 4 with

```
/* Initialization */
$\mathsf{A} \leftarrow x^m \cdot x^{-1}$
$\mathsf{R}[0] \leftarrow \mathsf{A}^{-1} \cdot x$
**for** $i = 1$ to $m-1$ **do** $\mathsf{R}[i] \leftarrow x$
$d \leftarrow d - \frac{(m-1)\cdot(m-4)}{2}$
$r' \leftarrow d \bmod (m-1)$
$\mathsf{R}[r'] \leftarrow \mathsf{R}[r'] \cdot \mathsf{R}[r'+1]$;
$\mathsf{R}[m-1] \leftarrow \mathsf{R}[m-1]^{-1}$
$d \leftarrow \lfloor d/(m-1) \rfloor = (d'_{\ell'-1}, \ldots, d'_0)_m$
```

This initialization works for all $w > 1$. The next section proposes an efficient alternative in the binary case for the RSA exponentiation. (i.e. d is odd)

4.3 Binary Case

In the binary case, we have $m - 1 = 1$ and thus $q = d$ and $r = 0$. We can use Algorithm 4 as is, where m is set to 2.

The resulting algorithm is very simple. It is slightly faster than Boscher *et al.*'s algorithm (Alg. 2) as it saves one multiplication. When d is odd (as is the case for RSA), A can be initialized to x^2, $\mathsf{R}[1]$ to x, and the for-loop index at

$i = 1$; this allows to save one more multiplication — and not to involve $1_{\mathbb{G}}$. More importantly, Algorithm 5 saves one register compared to Algorithm 2 as the value of x is no longer needed for the error detection. We so obtain a right-to-left algorithm as efficient as Giraud's algorithm. Being based on Montgomery ladder, Giraud's method scans however the exponent in the opposite direction which may be less convenient.

Algorithm 5. Binary right-to-left SPA-FA resistant algorithm

Input: $x \in \mathbb{G}$ and $d = (d_{\ell-1}, \ldots, d_0)_2 \in \mathbb{N}$
Output: $y = x^d$

```
   /* Initialization */
1: A ← $x$
2: R[0] ← $x$; R[1] ← $1_{\mathbb{G}}$
   /* Main loop */
3: for $i = 0$ to $\ell - 1$ do
4:     R[$d_i$] ← R[$d_i$] · A
5:     A ← A$^2$
6: end for
   /* Error detection */
7: R[0] ← R[0] · R[1]
8: if (R[0] $\neq$ A) then return 'error'
9: return R[1]
```

4.4 Efficiency

The initialization phase in Alg. 4 for the higher-radix case involves seemingly cumbersome operations. We detail below efficient implementations.

Algorithm 4 starts with the evaluation of x^{m-1}. We suggest to rely on a binary algorithm similarly to the implementation of the error detection in our variant of Baek's algorithm (i.e., Lines 12-15 in Alg. 3). For $m = 2^w$, this costs at most $(w - 1)$ multiplications and squarings in $\mathbb{G}$ (see Remark 2).

The initialization in Alg. 4 also requires the integer division (with remainder) of exponent d by $m - 1$. One may argue that these two values could be pre-computed and d represented by the pair (q, r) with $q = \lfloor d/(m-1) \rfloor$ and $r = d \bmod (m - 1)$. We rule out this possibility as it supposes that d is fixed (which is for example not the case for ECDSA). Further, even for RSA with a *a priori* fixed exponent, such a non-standard format for d, $d = (q, r)$, is not always possible as it may be incompatible with the personalization process or with certain randomization techniques used to protect against DPA-type attacks.

Let $m = 2^w$. We first remark that given the m-ary expansion of d, $d = \sum_{i=0}^{\ell-1} d_i\, m^i$, the value of $r = d \bmod (m-1)$ can simply be obtained as $r = \sum_{i=0}^{\ell-1} d_i \pmod{m-1}$. For the computation of the quotient, $q = \lfloor d/(m-1) \rfloor$, the schoolboy method can be significantly sped up by noting that the divisor (i.e., $m - 1 = 2^w - 1$) has all its bits set to 1 [20, p. 271] (see also [28]). It is also possible to evaluate q without resorting on a division operation. A common approach is the division-free Newton-Raphson method [16]. Being quadratically convergent, the value of q is rapidly obtained after a few iterations, i.e., roughly $\log_2(\log_2 d)$ iterations. The cost is typically upper-bounded to 2 multiplications in $\mathbb{G}$. The algorithm is presented in Appendix A. Of course, when available, the pair (q, r) can be computed with the co-processor present on the smart card; some of them come with an integer division operation.

To sum up, noting that Algorithm 4 saves a few multiplications, we see that all in all its expected performance is globally the same — when not faster — as our variant of Baek's algorithm. But the main advantage of Algorithm 4 resides in its better usage of memory.

5 Conclusion

This paper presented several memory-efficient implementations for preventing fault attacks in exponentiation-based cryptosystems. Furthermore, they are by nature protected against SPA-type attacks and can be combined with other existing countermeasures to cover other classes of implementation attacks. Remarkably, the developed methodology is fully generic (i.e., applies to any abelian group) and allows one to save one memory register (of size a group element) over previous implementations. This last feature is particularly attractive for memory-constrained devices and makes the proposed implementations well suited for smart-card applications.

Acknowledgments. We are grateful to an anonymous reviewer for useful comments.

References

1. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.-P.: Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 260–275. Springer, Heidelberg (2003)
2. Baek, Y.-J.: Regular 2^w-ary right-to-left exponentiation algorithm with very efficient DPA and FA countermeasures. International Journal of Information Security 9(5), 363–370 (2010)
3. Barreto, P.S.L.M.: A note on efficient computation of cube roots in characteristic 3. Cryptology ePrint Archive, Report 2004/305 (2004), http://eprint.iacr.org/
4. Blömer, J., Otto, M., Seifert, J.-P.: A new CRT-RSA algorithm secure against Bellcore attacks. In: Jajodia, S., Atluri, V., Jaeger, T. (eds.) 10th ACM Conference on Computer and Communications Security (CCS 2003), pp. 311–320. ACM Press (2003)
5. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. Journal of Cryptology 14(2), 101–119 (2001); Earlier version published in EUROCRYPT 1997

6. Boscher, A., Handschuh, H., Trichina, E.: Blinded exponentiation revisited. In: Breveglieri, L., et al. (eds.) Fault Diagnosis and Tolerance in Cryptography - FDTC 2009, pp. 3–9. IEEE Computer Society (2009)
7. Boscher, A., Naciri, R., Prouff, E.: CRT RSA algorithm protected against fault attacks. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 229–243. Springer, Heidelberg (2007)
8. Brauer, A.: On addition chains. Bulletin of the American Mathematical Society 45(10), 736–739 (1939)
9. Coron, J.-S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999)
10. Dottax, E., Giraud, C., Rivain, M., Sierra, Y.: On second-order fault analysis resistance for CRT-RSA implementations. In: Markowitch, O., Bilas, A., Hoepman, J.-H., Mitchell, C.J., Quisquater, J.-J. (eds.) WISTP 2009. LNCS, vol. 5746, pp. 68–83. Springer, Heidelberg (2009)
11. Giraud, C.: An RSA implementation resistant to fault attacks and to simple power analysis. IEEE Transactions on Computers 55(9), 1116–1120 (2006)
12. Joye, M.: Highly regular m-ary powering ladders. In: Jacobson Jr., M.J., Rijmen, V., Safavi-Naini, R. (eds.) SAC 2009. LNCS, vol. 5867, pp. 350–363. Springer, Heidelberg (2009)
13. Joye, M.: Protecting RSA against fault attacks: The embedding method. In: Breveglieri, L., et al. (eds.) Fault Diagnosis and Tolerance in Cryptography – FDTC 2009, pp. 41–45. IEEE Computer Society (2009)
14. Joye, M., Tunstall, M.: Exponent recoding and regular exponentiation algorithms. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 334–349. Springer, Heidelberg (2009)
15. Joye, M., Yen, S.-M.: The Montgomery powering ladder. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 291–302. Springer, Heidelberg (2003)
16. Karp, A.H., Markstein, P.W.: High-precision division and square root. ACM Transactions on Mathematical Software 23(4), 561–589 (1997)
17. Kim, C.H., Quisquater, J.-J.: Fault attacks for CRT based RSA: New attacks, new results, and new countermeasures. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 215–228. Springer, Heidelberg (2007)
18. Kim, C.H., Quisquater, J.-J.: How can we overcome both side channel analysis and fault attacks on RSA-CRT? In: Breveglieri, L., et al. (eds.) Fault Diagnosis and Tolerance in Cryptography – FDTC 2007, pp. 21–29. IEEE Computer Society (2007)
19. Knudsen, E.W.: Elliptic scalar multiplication using point halving. In: Lam, K.-Y., Okamoto, E., Xing, C. (eds.) ASIACRYPT 1999. LNCS, vol. 1716, pp. 135–149. Springer, Heidelberg (1999)
20. Knuth, D.E.: The Art of Computer Programming, 2nd edn. Seminumerical Algorithms, vol. 2. Addison-Wesley (1981)
21. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
22. Rivain, M.: Securing RSA against fault analysis by double addition chain exponentiation. In: Fischlin, M. (ed.) CT-RSA 2009. LNCS, vol. 5473, pp. 459–480. Springer, Heidelberg (2009)
23. Shamir, A.: Method and apparatus for protecting public key schemes from timing and fault attacks. US Patent #5,991,415 (November 1999) Presented at the rump session of EUROCRYPT 1997

24. Vigilant, D.: RSA with CRT: A new cost-effective solution to thwart fault attacks. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 130–145. Springer, Heidelberg (2008)
25. Yao, A.C.-C.: On the evaluation of powers. SIAM Journal on Computing 5(1), 100–103 (1976)
26. Yen, S.-M., Joye, M.: Checking before output not be enough against fault-based cryptanalysis. IEEE Transactions on Computers 49(9), 967–970 (2000)
27. Yen, S.-M., Kim, S., Lim, S., Moon, S.-J.: A countermeasure against one physical cryptanalysis may benefit another attack. In: Kim, K. (ed.) ICISC 2001. LNCS, vol. 2288, pp. 414–429. Springer, Heidelberg (2002)
28. Yungui, C., Xiaodong, Y., Bingshan, W.: A fast division technique for constant divisors $2^m(2^n \pm 1)$. Scientia Sinica (Series A), vol. XXVII(9), pp. 984–989 (1984)

A Newton-Raphson Iterated Division Algorithm

Algorithm 6. Integer division by $m - 1$ (initialization step)

```
Input: $d = \sum_{i=0}^{\ell-1} d_i m^i$ where $m = 2^w$
Output: $q = \lfloor d/(m-1) \rfloor$ and $r = d \bmod (m-1)$
   /* Remainder */
1: $r \leftarrow d_0$
2: for $i = 1$ to $\ell - 1$ do $r \leftarrow (r + d_i) \bmod (2^w - 1)$
   /* Quotient */
3: $d \leftarrow d - r$; $B \leftarrow |\ell(w-1)|_2$
4: $q \leftarrow 1$; $s \leftarrow 2^w - 1$
5: for $i = 1$ to $B$ do $q \leftarrow q \cdot (2 - s \cdot q) \bmod 2^{2^i}$
6: $q \leftarrow q \cdot d \bmod 2^{2^B}$
7: return $(q, r)$
```

Redundant Modular Reduction Algorithms

Vincent Dupaquis and Alexandre Venelli

Inside Secure
Avenue Victoire, 13790 Rousset, France
{vdupaquis,avenelli}@insidefr.com

Abstract. We present modular reduction algorithms over finite fields of large characteristic that allow the use of redundant modular arithmetic. This technique provides constant time reduction algorithms. Moreover, it can also be used to strengthen the differential side-channel resistance of asymmetric cryptosystems. We propose modifications to the classic Montgomery and Barrett reduction algorithms in order to have efficient and resistant modular reduction methods. Our algorithms are called dynamic redundant reductions as random masks are intrinsically added within each reduction for a small overhead. This property is useful in order to thwart recent refined attacks on public key algorithms.

1 Introduction

Modular reduction is at the heart of many asymmetric cryptosystems. Its principle is to evaluate the remainder of the integer division x/m. However, the division of two large multi-precision integers is very costly. Modular reduction algorithms were proposed in order to compute efficiently a remainder. Barrett reduction [5] and Montgomery reduction [15] are the two main methods. Both algorithms have a pre-computational step where either the inverse of the modulus or its reciprocal is computed. If the modulus is fixed amongst many operations, this pre-computed value is used in order to efficiently obtain a modular reduction. In asymmetric cryptosystems, the modulus is generally fixed at the very beginning. These techniques are then extremely efficient. Note that we leave out of our study the use of interleaved reduction and multiplication.

The implementation of cryptography algorithms on embedded devices is particularly sensitive to side-channel attacks. An attacker is often able to recover secret information from the device simply by monitoring the timing variations [11] or the power consumption [12]. Side-channel analysis include two main families of attacks: simple side-channel analysis (SSCA) and differential side-channel analysis (DSCA).

In this paper, we aim at protecting asymmetric cryptosystems against DSCA. Particularly, we focus on securing the exponentiation algorithm when the exponent is secret. Messerges *et al.* in [14] first detailed DSCA attacks applied to a modular exponentiation. Powerful attacks on public key algorithms were also proposed recently by Amiel *et al.* in [2,1]. These attacks can combine information obtained from SSCA, DSCA or fault analysis. There are typically two families of countermeasures against DSCA: randomized addition chains type or data randomization type. This second family is very interesting as it provides a countermeasure independent of the choice of the algorithm. Well known protections consist in message blinding [14], exponent blinding [11] and exponent splitting methods [7]. We propose here an alternative DSCA countermeasure that thwarts most of the attacks published in the literature.

We study in detail a technique called redundant modular arithmetic that allows integers modulo m to be kept modulo m plus some random multiples of m such that their representation is not unique. This idea has been introduced before in order to avoid timing attacks [20]. However, it can be improved to thwart also differential side-channel analysis. Golić and Tymen [10] proposed a differential side-channel countermeasure, based on the idea of redundant arithmetic, in the context of a masked Advanced Encryption Standard (AES) implementation. A brief study of Montgomery redundant arithmetic as a differential side-channel countermeasure is presented by Smart *et al.* in [18].

We extend this work to a so-called dynamic redundant modular arithmetic applied to Barrett's and Montgomery's reduction. Compared to [18], our algorithms offer flexibility in their usage as a randomization parameter can be adjusted. Our solution also allows dynamic randomization inside the exponentiation algorithm whereas other classical countermeasures often use only one randomization at the beginning of the exponentiation algorithm. This property can be useful in order to have a protection against the attack of Amiel *et al.* in [1] or even combined attacks as presented in [3].

The remainder of this paper is structured as follows. Section 2 describes the Montgomery and Barrett modular reduction algorithms. In Section 3, we detail the static redundant arithmetic method previously introduced in [18]. Then, we propose dynamic redundant reduction algorithms based on Montgomery's and Barrett's reductions in Section 4. We evaluate the performance and the security of our propositions in Section 5. Finally, Section 6 concludes the paper.

2 Modular Reduction Algorithms

We first introduce some notations. Long integers are represented as arrays of w-bit digits. Generally, one choose w as the word-size of the processor. The bit length of the integers is noted l and n is the number of digits necessary to store them, hence $n = \lceil l/w \rceil$. A long integer is then noted as $u = (u_{n-1}, \ldots, u_0)_b$ with $0 \leq u_i < b$ and $b = 2^w$. We note that, on a processor which word-size is w, the division by b or a power of b is simply a right shift, hence virtually free. Let $m = (m_{n-1}, \ldots, m_0)_b$ be a prime modulus of size n, *i.e.* n digits in the b-basis representation. Let u and v be two integers strictly lower than m, hence of size at most n. Let $x = (x_{2n-1}, \ldots, x_0)_b = uv < m^2$ be the integer to reduce modulo m.

2.1 Montgomery Reduction

The Montgomery reduction method [15] consists in using divisions by a power of b instead of multi-precision divisions. Let $R > m$ an integer coprime to m such that operations modulo R are easy to compute. A classical choice is $R = b^n$. In this method, integers $u < m$ are represented as a m-residues with respect to R, *i.e.* $uR \bmod m$. This representation of an integer is often called a Montgomery form. The Montgomery reduction of u is then defined as $uR^{-1} \bmod m$ where R^{-1} is the inverse of R modulo m. This reduction supposes that integers are in the Montgomery form. Take two integers $u < m$ and $v < m$. Consider their transformation in Montgomery form $uR \bmod m$ and $vR \bmod m$. Their multiplication gives $x = uvR^2$. Then, using Algorithm 1, one obtains $x = uvR \bmod m$ which is still in Montgomery form. This method uses a pre-computed value $\beta = -m^{-1} \bmod R$. Note also that the reduction requires, at most, only one final subtraction by m. Let $\mathsf{MontRed}(x, m, R, \beta)$ be the Montgomery reduction algorithm presented in Algorithm 1.

Algorithm 1. Montgomery reduction algorithm

```
Input: positive integers $x = (x_{2n-1}, \ldots, x_0)_b, m = (m_{n-1}, \ldots, m_0)_b$ and $\beta = -m^{-1} \bmod R$ where $R = b^n$, $\gcd(b, m) = 1$ and $x < mR$
Output: $xR^{-1} \bmod m$
1: $s_1 \leftarrow x \bmod R,\ s_2 \leftarrow \beta s_1 \bmod R,\ s_3 \leftarrow m s_2$
2: $t \leftarrow (x + s_3)/R$
3: if $(t \geq m)$ then
4:     $t \leftarrow t - m$
5: end if
6: return $t$
```

2.2 Barrett Reduction

Introduced by Barrett [5], this method is based on the idea of fixed point arithmetic. The principle is to estimate the quotient x/m with operations that can either be pre-computed or are less expensive than a multi-precision division. The remainder r of x modulo m is equal to $r = x - m\lfloor x/m \rfloor$. Using the fact that divisions by a power of b are virtually free, we have:

$$r = x - m\left\lfloor \frac{\frac{x}{b^{n-1}} \frac{b^{2n}}{m}}{b^{n+1}} \right\rfloor = x - m\left\lfloor \frac{\frac{x}{b^{n-1}} \mu}{b^{n+1}} \right\rfloor$$

with $\mu = \left\lfloor \frac{b^{2n}}{m} \right\rfloor$ a pre-calculated value that depends on the modulus. Let $\hat{q}$ be the estimation of the quotient of x/m. Barrett improves further the reduction using only partial multi-precision multiplication when needed. The estimate $\hat{r}$ of the remainder of x modulo m is:

$$\hat{r} = (x \bmod b^{n+1} - (m\hat{q} \bmod b^{n+1})) \bmod b^{n+1}.$$

This estimation implies that at most two subtractions of m are required to obtain the correct remainder r. Barrett's algorithm is described in Algorithm 2. Note that in this algorithm, the estimated quotient $\hat{q}$ corresponds to the variable q_3 and the estimated remainder $\hat{r}$ is computed with the operation $r_1 - r_2$ in Line 2.

Algorithm 2. Barrett reduction algorithm

```
Input: positive integers $x = (x_{2n-1}, \ldots, x_0)_b, m = (m_{n-1}, \ldots, m_0)_b$ and $\mu = \lfloor b^{2n}/m \rfloor$
Output: $x \bmod m$
1: $q_1 \leftarrow \lfloor x/b^{n-1} \rfloor,\ q_2 \leftarrow \mu q_1,\ q_3 \leftarrow \lfloor q_2/b^{n+1} \rfloor$
2: $r_1 \leftarrow x \bmod b^{n+1},\ r_2 \leftarrow mq_3 \bmod b^{n+1},\ r \leftarrow r_1 - r_2$
3: if $(r \leq 0)$ then
4:     $r \leftarrow r + b^{n+1}$
5: end if
6: while $(r \geq m)$ do
7:     $r \leftarrow r - m$
8: end while
9: return $r$
```

3 Static Redundant Modular Arithmetic

Redundant field representation can form an interesting defense against differential side-channel attacks as a given element can have different representations. We can cite the recent work by Smart *et al.* [18] on this topic. The authors briefly present a redundant Montgomery arithmetic. If one wants to work with integers modulo m, a standard representation is to take elements in $\mathbb{Z}/m\mathbb{Z} = \{0, \ldots, m-1\}$. The principle of redundant arithmetic is to consider elements in the range $\{0, \ldots, C-1\}$ where $C > m$ and keep integers modulo m within this range.

We recall this method that can be denoted as static redundant modular arithmetic.

Let $C = cm$ where $c > 1$ is an integer coprime to m. Instead of working in $\mathbb{Z}/m\mathbb{Z}$, we work in $\mathbb{Z}/(cm)\mathbb{Z}$, *i.e.* modulo C. Let $R_C = b^j$ with $j > n$ a certain integer such that $R_C > C$. Let $\beta_C = -C^{-1} \bmod R_C$ a pre-computed value similar to β defined in §2.1. The Montgomery form of an element u modulo C becomes $uR_C \bmod C$. We apply this method in a classical modular exponentiation algorithm (see Algorithm 3).

We call this method static redundant modular arithmetic as only one random mask k is applied at the beginning of the algorithm. In the next section, we present two propositions of dynamic redundant modular reduction algorithms for the methods of Montgomery and Barrett.

Algorithm 3 . Square-and-multiply exponentiation using static redundant Montgomery arithmetic

```
Input: positive integers $e = (e_{l-1}, \dots, e_0)_2, x, m, C, \beta_C$ and $R_C$
Output: $x^e \bmod m$
 1: $X \leftarrow x + km$, where $k$ is a random integer
 2: $R_0 \leftarrow R_C$
 3: $R_1 \leftarrow XR_C \bmod C$
 4: for $i = l - 1$ down to 0 do
 5:     $R_0 \leftarrow \mathsf{MontRed}(R_0^2, C, R_C, \beta_C)$
 6:     if $(e_i = 1)$ then
 7:         $R_0 \leftarrow \mathsf{MontRed}(R_0 R_1, C, R_C, \beta_C)$
 8:     end if
 9: end for
10: $R_0 \leftarrow R_0 R_C^{-1} \bmod m$
11: return $R_0$
```

4 Dynamic Redundant Modular Arithmetic Propositions

We propose modular reduction algorithms that are slight modifications of Montgomery's and Barrett's techniques. Our methods offer a so-called dynamic redundant modular arithmetic in which random masks are refreshed intrinsically within the reduction method.

4.1 Dynamic Redundant Montgomery Reduction

Consider we want to reduce an integer $x = uv$ modulo m with $u < m$ and $v < m$ two integers modulo m. Hence, if m is a n-word integer, we want to reduce a $2n$-word integer x. Working with redundant modular arithmetic, several multiples of the modulo m can be added to an integer. Thus, its size will grow depending on the number of multiples considered. If we have a $(2n+2i)$-word integer at the input of the reduction algorithm, we need a $(n+i)$-word output so that further multiplications between reduced elements can be computed.

We first look in details at the operations of the Montgomery reduction (Algorithm 1). Recall that $\beta = -m^{-1} \bmod R$ is a pre-computed value and R is a power of b, generally $R = b^n$. We know from [13, Fact 14.29 and Note 14.30] that $(x + m(x\beta \bmod R))/R = (xR^{-1} \bmod m) + \epsilon m$ with $\epsilon \in \{0, 1\}$ at the end of the reduction. If we develop the formula, we have:

$$\begin{aligned}(x + m(x\beta \bmod R))/R &= (x + m(x\beta - k_1 R))/R && (1)\\ &= (x + x(-1 + k_2 R) - k_1 R m)/R && (2)\\ &= k_2 x - k_1 m, && (3)\end{aligned}$$

for k_1, k_2 some integers. In our context, we want to have ϵ greater than 1 in order to have several multiples of the modulus at the end of the reduction. By slightly modifying the Montgomery reduction algorithm, we can achieve this property. Consider now the following steps for a modified Montgomery reduction algorithm:

```
1: $s_1 \leftarrow x \bmod R$
2: $s_2 \leftarrow \beta s_1 \bmod R$
3: $s_2 \leftarrow s_2 + kR$, with $k$ some random positive integer
4: $s_3 \leftarrow m s_2$
5: $t \leftarrow (x + s_3)/R$
```

We now have from Eq. (3) and Step 3 that

$$(x + m(x\beta \bmod R))/R = k_2 x - (k_1 - k)m.$$

Hence, the error at the end of the reduction becomes $k + \epsilon$. We also note that the final subtraction step of Algorithm 1 is removed so that the implementation is resistant to timing attacks. If t is the output of this modified Montgomery reduction, we have:

$$(xR^{-1} \bmod m) + km \leq t \leq (xR^{-1} \bmod m) + (k+1)m. \tag{4}$$

For a practical implementation, we need to consider the size of the operands at the input and output of the reduction algorithm. Instead of fixing the constant $R = b^n$ for a modulus of size n, we consider a larger constant $R' = b^{n+2i}$ for some integer i. We recall that the Montgomery reduction requires that the input x is such that $x < mR$. Hence, with the constant R' we can process larger integers $x < mR' < b^{2n+2i}$ which means that the output of the reduction can be integers $t < b^{n+i}$. From Eq. (4), we deduce that the random integer k needs to be chosen such that $k < b^i - 1$. We define a dynamic redundant Montgomery reduction algorithm (see Algorithm 4).

Algorithm 4. Dynamic redundant Montgomery reduction algorithm

```
Input: positive integers $x = (x_{2n+2i-1}, \ldots, x_0)_b, m = (m_{n-1}, \ldots, m_0)_b$ and $\beta' = -m^{-1} \bmod R'$ where $R' = b^{n+2i}$, $\gcd(b, m) = 1$, $x < mR'$ and for some integer $i$
Output: $t \leq (xR'^{-1} \bmod m) + (k+1)m$
1: Let $k$ be a random integer such that $0 \leq k < b^i - 1$
2: $s_1 \leftarrow x \bmod R'$, $s_2 \leftarrow \beta' s_1 \bmod R'$
3: $s_2 \leftarrow s_2 + kR'$, $s_3 \leftarrow m s_2$
4: $t \leftarrow (x + s_3)/R'$
5: return $t$
```

Each reduction adds a random number of multiples of the modulus to the remainder. If a dynamic redundant reduction technique is used in an asymmetric cryptosystem such as RSA, a final normalization step is needed at the very end of the exponentiation algorithm for example in order to output a result strictly inferior to m. We note that the boundary $b^i - 1$ on the random integer k can be parametrized in the reduction algorithm. Hence, we could fix $k = 0$ and obtain a time constant Montgomery reduction without final reduction if we want effectiveness instead of differential side-channel resistance. Let $\mathsf{DRMontRed}(x, m, R', \beta')$ be the dynamic redundant Montgomery reduction algorithm presented in Algorithm 4.

4.2 Dynamic Redundant Barrett Reduction

We know from [13, Note 14.44] that, with the original parameters of Barrett used in § 2.2, the estimated quotient $\hat{q}$ satisfies $q - 2 \leq \hat{q} \leq q$ where q is the correct quotient. It can be shown that for about 90% of the values of $x < m^2$ and m, the value of $\hat{q}$ will be equal to q and in only 1% of cases $\hat{q}$ will be 2 in error [6]. Dhem's work [9, Section 2.2.4] provides a general parametrization of the Barrett reduction. In particular, with appropriate choices, the error on the quotient can be reduced. In our context, we are interested in adding more errors to the estimated quotient so that multiples of the modulus are left in the reduced integer. However, using Dhem's parametrization to bound the errors is not interesting as, in practice, only a very small number of integers will reach the upper bound.

We first recall some notations of Dhem's work. The estimated quotient $\hat{q}$ in the reduction of x modulo m can be evaluated as:

$$\hat{q} = \left\lfloor \frac{\frac{x}{b^{n+\beta}} \frac{b^{n+\alpha}}{m}}{b^{\alpha-\beta}} \right\rfloor = \left\lfloor \frac{\frac{x}{b^{n+\beta}} \mu_\alpha}{b^{\alpha-\beta}} \right\rfloor ,$$

with $\mu_\alpha = \lfloor b^{n+\alpha}/m \rfloor$ the parametrized constant used in the Barrett reduction and α, β two integers.

In Barrett's algorithm, the estimated quotient is undervalued. Hence, we need to further undervalue it in order to have a reduced integer with some multiples of the modulus left. If we have $\hat{q} = (q-k)$ the undervalued quotient for some positive integer k, then the estimated remainder will be $\hat{r} = x - m\hat{q} = x - m(q - k) = x - mq + km$. Consider the following steps for a modified Barrett reduction algorithm :

```
1: $q_1 \leftarrow \lfloor x/b^{n+\beta} \rfloor$
2: $q_2 \leftarrow \mu_\alpha q_1$
3: $q_3 \leftarrow \lfloor q_2/b^{\alpha-\beta} \rfloor$
4: $q_3 \leftarrow q_3 - k$, with $k$ some random positive integer
5: $r_1 \leftarrow x \bmod b^\alpha$
6: $r_2 \leftarrow mq_3 \bmod b^\alpha$
7: $r \leftarrow r_1 - r_2$
8: if $(r \leq 0)$ then
9:     $r = r + b^\alpha$
10: end if
```

In practice, we note that the last conditional addition is not required on most processors that use the two's-complement system. We also remove the final subtraction loop in Algorithm 2 as we want multiples of the modulus left in the remainder. As in the Montgomery reduction case, we consider integers x of size $2n + 2i$ to be reduced modulo m of size n. Hence, the output of the reduction algorithm needs to be at most of size $n + i$. We know from Dhem's work that the error ϵ in the estimation of the quotient is such that :

$$\epsilon \leq \lfloor 2^{n+2i-\alpha} + 2^{\beta+1} + 1 - 2^{\beta-\alpha} \rfloor.$$

We choose $\alpha = n + 2i$ and $\beta = -1$, hence the estimated quotient is, at most, undervalued by 2, *i.e.* $\epsilon \leq 2$. We do not need to minimize the error for our proposition as it would add complexity to the reduction algorithm. However the error can not be too large as for the final normalization step we need to subtract the remaining multiples of the modulus relatively quickly. If r is the output of this modified Barrett reduction, we have:

$$(x \bmod m) + km \leq r \leq (x \bmod m) + (k+2)m. \tag{5}$$

From Eq. (5), we deduce that the random integer k needs to be defined such that $k < b^i - 2$. We simply note $\mu' = \mu_{n+2i} = \lfloor b^{2n+2i}/m \rfloor$ in the following. We define a dynamic redundant Barrett reduction algorithm (Algorithm 5). As previously, we note that the randomization can be parametrized such that if we fix $k = 0$, we obtain an efficient time constant Barrett reduction.

Algorithm 5. Dynamic redundant Barrett reduction algorithm

```
Input: positive integers $x = (x_{2n+2i-1}, \ldots, x_0)_b, m = (m_{n-1}, \ldots, m_0)_b$ and $\mu' = \lfloor b^{2n+2i}/m \rfloor$
Output: $r \leq (x \bmod m) + (k+2)m$
 1: Let $k$ be a random integer such that $0 \leq k < b^i - 2$
 2: $q_1 \leftarrow \lfloor x/b^{n-1} \rfloor$, $q_2 \leftarrow \mu' q_1$
 3: $q_3 \leftarrow \lfloor q_2/b^{n+2i+1} \rfloor$, $q_3 \leftarrow q_3 - k$
 4: $r_1 \leftarrow x \bmod b^{n+2i}$, $r_2 \leftarrow mq_3 \bmod b^{n+2i}$, $r \leftarrow r_1 - r_2$
 5: return $r$
```

5 Efficiency and Security Evaluation

As previously stated, we only study reduction algorithms and leave out of our study the use of interleaved multiplication and reduction. Hence, we compare the complexity of reduction algorithms alone. Each algorithm needs a multi-precision multiplication for its intermediate results. We consider Comba's multiplication [8] as it is very interesting in embedded devices. This method is often called column-wise multiplication or product scanning method. Note that an evolution of Comba's multiplication, called hybrid multiplication, is presented in [17]. The authors combine column-wise and line-wise multiplication techniques for better efficiency. In order to compare the performance of the algorithms, we consider the number of base multiplications, *i.e.* multiplication of two b-bit operands.

We first analyze the standard Montgomery reduction (Algorithm 1). The operation $s_2 \leftarrow \beta s_1 \bmod R$ can be computed as a partial multiplication. In fact, only the lower words of βs_1 are needed for the result s_2. We recall that $R = b^n$ with n the size of the modulus in words. The multiplicands β and s_1 are two n-words values. Instead of n^2 base multiplications, this operation can be computed with $\binom{n+1}{2} = (n^2+n)/2$ base multiplications. The next multiplication, $s_3 \leftarrow ms_2$ has to be fully computed using n^2 base multiplications. Hence the standard Montgomery reduction requires $(3n^2+n)/2$ base multiplications.

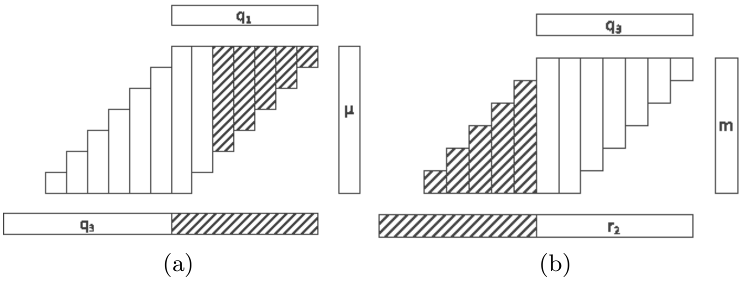


Fig. 1. Partial multiplications used in a standard Barrett reduction. Figure 1a represents the computation of q_3. Figure 1b represents the computation of r_2. The striped lines correspond to operations that are not needed for these partial multiplications.

The standard Barrett reduction (Algorithm 2) benefits also from partial multiplications. Only the highest words of the multiplication $q_2 \leftarrow \mu q_1$ are needed to compute $q_3 \leftarrow \lfloor q_2/b^{n+1} \rfloor$. As noted in [13, Note 14.45], the $n+1$ least significant words of q_2 are not needed. If $b > n$, we can compute the multiplication starting at the $n-1$ least significant word and we will have an error at most 1. The complexity is $(n+1)^2 - \binom{n}{2} = (n^2+5n+2)/2$ base multiplications. The operation $r_2 \leftarrow mq_3 \bmod b^{n+1}$ can also be computed using a partial multiplication as only the $n+1$ least significant words are required. Its complexity is $\binom{n+1}{2} + n = (n^2+3n)/2$. Finally, the complexity of the standard Barrett reduction is n^2+4n+1 base multiplications. The two partial multiplications used in the Barrett reduction are represented in Figure 1.

Using similar considerations as for the standard reduction algorithms, we analyze the complexity of our dynamic redundant reduction algorithms. The dynamic redundant Montgomery reduction (Algorithm 4) requires $\binom{n+2i+1}{2} + n(n+2i+1) = (3n^2+3n)/2 + i(4n+2i+1)$ base multiplications for a given i. We recall that i bounds the maximum number of multiples of the modulus that can be added. The dynamic redundant Barrett reduction (Algorithm 5) requires $n^2+3n+1+i(4n+2i+5)$ base multiplications. Table 1 summarizes the complexities.

We evaluate a practical implementation of the different reduction algorithms on a AVR 8-bit ATmega 2561 processor [4] running at 16 MHz. A 512-bit modulus is chosen randomly and fixed. Each reduction is implemented in assembly

Table 1. Summary of the complexities of the different modular reduction algorithms in the number of base multiplications for a n-word modulus

Algorithm	**Number of base multiplications**
Standard Montgomery	$(3n^2 + n)/2$
Dynamic redundant Montgomery	$(3n^2 + 3n)/2 + i(4n + 2i + 1)$
Standard Barrett	$n^2 + 4n + 1$
Dynamic redundant Barrett	$n^2 + 3n + 1 + i(4n + 2i + 5)$

and use the same column-wise multiplication code. The results are summarized in Table 2. Using either Montgomery's or Barrett's technique, we first remark that redundant arithmetic allows time constant reductions as the final subtractions are removed of both standard algorithms. We also note our proposition is more efficient using Barrett reduction. In fact, the dynamic redundant Barrett reduction with $i = 1$ is faster than the standard Barrett. If the randomization is not needed, we can fix the random $k = 0$ in Algorithm 5 and compute a constant time dynamic redundant Barrett with $i = 1$ more efficiently than the classic Barrett. Note that even if the complexity in base multiplication of our algorithm with $i = 1$ is slightly higher than the standard Barrett (see Table 1), our propositions remove the final loop of the Barrett reduction. Hence, for small values of i, we can obtain a dynamic redundant Barrett reduction faster than the standard one.

Table 2. Execution time of each reduction algorithm on a ATmega 2561 processor running at 16 MHz for a fixed modulus of 512 bits and random inputs

Algorithm	**Time (in ms)**
Standard Montgomery	6.1 or 6.3
Dynamic redundant Montgomery with $i = 1$	8.7
Dynamic redundant Montgomery with $i = 2$	9.3
Standard Barrett	6.4 or 6.6
Dynamic redundant Barrett with $i = 1$	6.3
Dynamic redundant Barrett with $i = 2$	6.6

We define the function $\mathsf{Normalize}(x, m) = x \bmod m$ as a simple loop that computes subtractions $x = x - m$ as long as $x \geq m$. Algorithm 6 illustrates the use of dynamic redundant modular arithmetic in an exponentiation algorithm. We first note that the static redundant arithmetic exponentiation (Algorithm 3) requires one more n-word parameter, *i.e.* we need to store both m and $C = cm$, whereas the dynamic version only needs the modulus m. Moreover, in Algorithm 6, for given pre-computed values β' and R' with a fixed i, one can choose to use a function $\mathsf{rand}()$ that generates random integers in $[0, b^{i'} - 1[$ such that $0 \leq i' \leq i$.

In particular, we can fix $i' = 0$ in the exponentiation algorithm if no randomization is needed. In the static redundant case, values C, β_C and R_C need to be pre-computed again if the amount of random needs to be lower for a particular exponentiation.

Algorithm 6. Multiply always exponentiation using dynamic redundant Montgomery arithmetic

```
Input: positive integers $e = (e_{l-1}, \ldots, e_0)_2, x, m, \beta'$ and $R'$. Let $\mathsf{rand}()$ be a function
    that generates a random integer in $[0, b^i - 1[$ for some integer $i$.
Output: $x^e \bmod m$
 1: $X \leftarrow x + \mathsf{rand}()m$
 2: $R_0 \leftarrow \mathsf{DRMontRed}(\mathsf{rand}()m, m, R', \beta')$
 3: $R_1 \leftarrow \mathsf{DRMontRed}(XR', m, R', \beta')$
 4: $i \leftarrow l - 1$, $t \leftarrow 0$
 5: while $i \geq 0$ do
 6:     $R_0 \leftarrow \mathsf{DRMontRed}(R_0(R_t + \mathsf{rand}()m), m, R', \beta')$
 7:     $t \leftarrow t \oplus e_i$, $i \leftarrow i - 1 + t$
 8: end while
 9: $R_0 \leftarrow \mathsf{DRMontRed}(R_0 R'^{-1}, m, R', \beta')$
10: $R_0 \leftarrow \mathsf{Normalize}(R_0, m)$
11: return $R_0$
```

Amiel *et al.* showed in [1] that the Hamming weight of the output of a multiplication $x \times y$ can be distinguished whether $y = x$ or $y \neq x$. Hence, they can defeat atomic implementations of exponentiation algorithms. Using redundant arithmetic as in Algorithm 6, we note that the representation of R_t can be easily randomized prior to the multiplication $R_0 \times R_t$. Hence, even if $t = 0$, the output of the multiplication $R_0 \times R_t$ cannot be distinguished anymore by this attack.

Left-to-right atomic exponentiation algorithms seems particularly vulnerable to combined attacks [3,16]. In left-to-right algorithms, the value of one register, *e.g.* R_1, is generally fixed to the value of the input message x during the exponentiation. If a precise fault is injected in R_1, an attacker can detect with a simple side-channel analysis when this register is used during the exponentiation, hence the attacker can recover the exponent. Using our dynamic redundant modular propositions, as in the left-to-right Algorithm 6, we can note that the representation of register R_1 is randomized during the exponentiation. Hence, combined attacks as presented in [3] are no longer able to recover the full exponent.

Dynamic redundant reduction algorithms provide a countermeasure against differential side-channel attacks for public key algorithms. Our solution is an alternative to the classical message blinding [14]. However, as pointed out by Smart *et al.* [18], redundant arithmetic may not be as interesting in elliptic curve cryptography. In fact, if the modulus is a generalized Mersenne prime [19], masking using multiples of the modulus is not as suitable. However, for random modulus, as in RSA, this technique provides a very good defense against differential side-channel attacks for a minimal overhead.

6 Conclusion

We study in this paper the use of redundant arithmetic as a differential side-channel countermeasure. We extend the work of Smart *et al.* [18] by proposing dynamic redundant modular reduction algorithms based on Montgomery's and Barrett's techniques. Our algorithms are parametrized and offer a good flexibility in order to control the amount of randomization as well as the size of the operands. The dynamic randomization of the data inside an exponentiation algorithm can also thwart more refined side-channel attacks. As we remove the final subtraction steps in both standard algorithms, our propositions are time constant and efficient.

Acknowledgments. The authors would like to thank François Dassance, Vincent Verneuil and the anonymous referees of Cardis 2011 for their helpful comments.

References

1. Amiel, F., Feix, B., Tunstall, M., Whelan, C., Marnane, W.P.: Distinguishing Multiplications from Squaring Operations. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) SAC 2008. LNCS, vol. 5381, pp. 346–360. Springer, Heidelberg (2009)
2. Amiel, F., Feix, B., Villegas, K.: Power Analysis for Secret Recovering and Reverse Engineering of Public Key Algorithms. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876, pp. 110–125. Springer, Heidelberg (2007)
3. Amiel, F., Villegas, K., Feix, B., Marcel, L.: Passive and Active Combined Attacks: Combining Fault Attacks and Side Channel Analysis. In: FDTC 2007, pp. 92–102. IEEE (2007)
4. ATMEL: ATmega 2561 Data Sheet, http://www.atmel.com/dyn/resources/prod_documents/doc2549.pdf
5. Barrett, P.: Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 311–323. Springer, Heidelberg (1987)
6. Bosselaers, A., Govaerts, R., Vandewalle, J.: Comparison of Three Modular Reduction Functions. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 175–186. Springer, Heidelberg (1994)
7. Clavier, C., Joye, M.: Universal Exponentiation Algorithm A First Step towards Provable SPA-Resistance. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 300–308. Springer, Heidelberg (2001)
8. Comba, P.: Exponentiation Cryptosystems on the IBM PC. IBM Syst. J. 29, 526–538 (1990)
9. Dhem, J.F.: Design of an efficient public-key cryptographic library for RISC-based smart cards. Ph.D. thesis, Université Catholique de Louvain (1998)
10. Golić, J.D., Tymen, C.: Multiplicative Masking and Power Analysis of AES. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 198–212. Springer, Heidelberg (2003)
11. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)

12. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
13. Menezes, A.J., Vanstone, S.A., Van Oorschot, P.C.: Handbook of Applied Cryptography. CRC Press, Inc. (1996)
14. Messerges, T.S., Dabbish, E.A., Sloan, R.H.: Power Analysis Attacks of Modular Exponentiation in Smartcards. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 144–724. Springer, Heidelberg (1999)
15. Montgomery, P.: Modular Multiplication Without Trial Division. Mathematics of computation 44(170), 519–521 (1985)
16. Schmidt, J.-M., Tunstall, M., Avanzi, R., Kizhvatov, I., Kasper, T., Oswald, D.: Combined Implementation Attack Resistant Exponentiation. In: Abdalla, M., Barreto, P.S.L.M. (eds.) LATINCRYPT 2010. LNCS, vol. 6212, pp. 305–322. Springer, Heidelberg (2010)
17. Scott, M., Szczechowiak, P.: Optimizing Multiprecision Multiplication for Public Key Cryptography. Cryptology ePrint Archive, Report 2007/299 (2007)
18. Smart, N., Oswald, E., Page, D.: Randomised Representations. Information Security IET 2(2), 19–27 (2008)
19. Solinas, J.: Generalized Mersenne Numbers. Technical report (1999)
20. Walter, C.: Montgomery Exponentiation Needs no Final Subtractions. Electronics letters 35(21), 1831–1832 (2002)

Fresh Re-keying II: Securing Multiple Parties against Side-Channel and Fault Attacks

Marcel Medwed, Christoph Petit, Francesco Regazzoni, Mathieu Renauld, and François-Xavier Standaert

UCL Crypto Group, Université catholique de Louvain.
Place du Levant 3, B-1348, Louvain-la-Neuve, Belgium

Abstract. Security-aware embedded systems are widespread nowadays and many applications, such as payment, pay-TV and automotive applications rely on them. These devices are usually very resource constrained but at the same time likely to operate in a hostile environment. Thus, the implementation of low-cost protection mechanisms against physical attacks is vital for their market relevance. An appealing choice, to counteract a large family of physical attacks with one mechanism, seem to be protocol-level countermeasures. At last year's Africacrypt, a fresh re-keying scheme has been presented which combines the advantages of re-keying with those of classical countermeasures such as masking and hiding. The contribution of this paper is threefold: most importantly, the original fresh re-keying scheme was limited to one low-cost party (e.g. an RFID tag) in a two party communication scenario. In this paper we extend the scheme to n low-cost parties and show that the scheme is still secure. Second, one unanswered question in the original paper was the susceptibility of the scheme to algebraic SPA attacks. Therefore, we analyze this property of the scheme. Finally, we implemented the scheme on a common 8-bit microcontroller to show its efficiency in software.

Keywords: Side-channel attacks, Fault attacks, Re-keying, Masking, Shuffling.

1 Introduction

Ensuring security against physical (e.g. side-channel and fault) attacks is an increasingly important challenge for cryptographic embedded devices. It is specially critical in applications requiring low-cost implementations. Indeed, most solutions that have been introduced in the literature to prevent physical attacks imply significant performance penalties, that may be too high for certain applications. For example, improving security against side-channel attacks is frequently obtained by applying masking [4,10] or hiding [28,29] to the implementations. Additionally, preventing fault attacks requires to include fault detection mechanisms in the circuits [2,12]. In both cases, implementing these countermeasures implies significant area or time overheads. Unfortunately, low-cost devices such as smart cards, RFID tags or sensor nodes are also the ones for which the threat of a physical attack is the most realistic, when operated in hostile environments.

Since directly protecting cryptographic algorithms, such as the AES Rijndael, against side-channel and fault attacks is difficult, an alternative approach is to design encryption mechanisms that can be more easily protected in this case. One important line of research, denoted as leakage-resilient cryptography, aims at combining such new designs with a proof of security, using the formalism of modern cryptography. For example, this approach has been applied to new stream cipher constructions [6,21,31]. But as discussed in [26,27], present proof techniques have limited practical relevance, as they need to rely on assumptions that may be difficult to fulfill by hardware designers. More important for our present focus, these schemes are also quite inefficient, as their initialization implies the execution of a pseudorandom function [5,27], which typically requires the execution of n AES encryptions, with n the bit size of the initialization vector. A more practically-oriented line of research has been trying to embed the block cipher in some protocol that makes it easier to protect. An example of such an approach is the application of "all-or-nothing" transforms [17]. Here, the idea is to modify the plaintexts and ciphertexts according to a (low-cost) mapping, therefore preventing attacks based on known plaintexts/ciphertexts. Whereas the scheme is efficient for long messages, the initialization effort might render it impractical for smartcard and RFID applications. More recently, a fresh re-keying scheme was presented at Africacrypt 2010 [19]. It combines the re-keying used in leakage-resilient cryptography with easy to protect low-cost mappings in order to remove the initialization overhead.

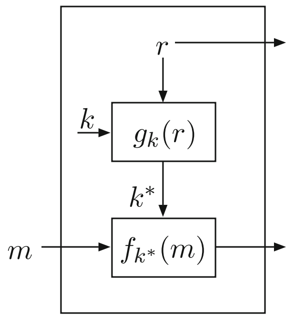


Fig. 1. Original (one party) fresh re-keying scheme

The basic principle of this fresh re-keying scheme, that we further investigate in this paper, is pictured in Figure 1. It essentially encrypts every message block m under a fresh session key k^*, with a block cipher. The session key is generated from the master key k and a public random nonce r, with a function g. At first sight, it may seem that one just shifts the problem of protecting the block cipher f against physical attacks to the one of protecting the function g. Interestingly, it is argued in [19] that a proper selection of g may lead this scheme to be significantly easier to protect against such attacks than the underlying block cipher f.

Namely, g does not need to be cryptographically strong. It should only ensure a strong diffusion between the master key k and the session keys k^*, while being easy to protect against Differential Power Analysis (DPA). Assuming that these conditions are respected, the only additional requirement for the global scheme of Figure 1 to be secure against side-channel attacks is that the block cipher f resists against SPA (i.e. side-channel attacks exploiting a single measurement) - an easier task than preventing DPA. In addition, the re-keying mechanism naturally prevents Differential Fault Analysis (DFA) and was shown to have limited hardware cost. Following these interesting features, the present paper aims at extending this analysis in three main directions:

1. In view of the difficulty to design leakage-resilient pseudorandom permutations [5], the claim that a simple fresh re-keying scheme could be secure against a wide class of side-channel and fault attacks appears quite provocative. In this respect, one important problem that was left open in the previous work of Africacrypt is to evaluate if the cryptographically weak function g could not be the target of advanced side-channel attacks (such as [20,23]), taking advantage of its simple algebraic expression. We evaluate this concern and suggest that it can be efficiently prevented by shuffling the implementation [11] (which is anyway required to prevent SPA).
2. The original Africacrypt scheme was limited to the protection of one party in a communication protocol. This is because in Figure 1, it is crucial that the random nonce r is chosen inside the protected device and cannot be manipulated from outside. For example, keeping r constant would completely break the re-keying. While there exist many practical scenarios in which such an asymmetric type of security is realistic (e.g. RFID readers can be protected with expensive means, only tags have strong cost constraints), there also exist many where protecting multiple parties is necessary. For instance, trends can be observed in which also constrained devices become readers, e.g. mobile phones in NFC applications [22]. Furhtermore, automotive applications, where many low-cost devices need to communicate securely [13], provide a strong motivation for developing such tools. As a result, we extend the re-keying scheme to multiple parties. In particular, this allows all involved parties to derive a common session key in a side-channel protected manner[1]. We show that the security of the new proposal is similar to the one of the single-party case, while its performances only decrease linearly with the number of parties.

[1] This context can be seen as reminiscent of group key distribution. However, our objectives are different in the sense that group key distribution typically aims at ensuring cryptographic properties such as forward security, whereas our fresh re-keying scheme "only" aims at preventing successful side-channel attacks against the master key. It is an interesting open question to investigate whether one could combine strong physical security guarantees and, e.g. forward secrecy. We note that it would be surprising, as the relatively simple (linear) nature of the g function is central in making it easy to protect against side-channel attacks.

3. Finally, since the use of dedicated solutions for protecting block cipher implementations is mainly justified by strong cost constraints, we evaluate the performances of our proposal in an AVR microcontroller. Our implementations consider different levels of masking and shuffling. We confirm their low-cost nature by comparing them with the masked AES Rijndael software implementation proposed at CHES 2010 [24]. These results nicely complement the ones in [19], where hardware implementations were considered.

2 Background: The Africacrypt 2010 Scheme

The original scheme, as depicted in Figure 1, describes a physically secure encryption which is for instance carried out inside an RFID tag. It consists of the re-keying function g and a cryptographically secure encryption function f. At every invocation of the scheme, g uses a symmetric master key k and a fresh random nonce r to obtain a session key $k^* = g_k(r)$. The random nonce r is generated inside the device but is made public afterwards. The session key is then used by f to perform an encryption. Thus, the ciphertext c is obtained as $c = f_{k^*}(m)$. The function f is instantiated with a standardized algorithm, in our examples we use the AES Rijndael. For such algorithms, it is also well known how to protect them against SPA attacks (e.g. by shuffling [7,11]). Thus, the main concern is a careful choice of g. In [19], g was chosen as:

$$\begin{aligned} g &: \left(\mathrm{GF}(2^8)[y]/(y^{16}+1)\right)^* \times \left(\mathrm{GF}(2^8)[y]/(y^{16}+1)\right) \to \mathrm{GF}(2^8)[y]/(y^{16}+1) \\ &: (k, r) \to k \cdot r. \end{aligned}$$

That is, g takes two 128-bit operands, represented by polynomials in y of degree 15 and coefficients in $\mathrm{GF}(2^8)$, and performs a polynomial multiplication to obtain the session key. The key k is constrained to the invertible elements of $\mathrm{GF}(2^8)[y]/(y^{16}+1)$; otherwise any master key k that is a divisor of 0 would only lead to session keys that are also divisors of 0. As shown in the original paper, this choice of g has some very advantageous properties. Most important, it provides sufficient diffusion such that "divide-and-conquer" attacks on the master key become computationally infeasible. Second, the function has homomorphic properties which allows sound protection against higher-order differential attacks [4]. In particular, the key can be split into $t+1$ shares:

$$\left(k_1 = b_1, k_2 = b_2, \cdots, k_t = b_t, k_{t+1} = k \oplus \bigoplus_{i=1}^{t} b_i\right),$$

in order to obtain the session key as:

$$k^* = \bigoplus_{i=1}^{t+1} r \cdot k_i.$$

Such a masking thwarts t^{th}-order attacks[2]. Third, as the arithmetic in the proposed algebra is carry-free, shuffling can be efficiently applied to thwart SPA attacks. Finally, the function can also benefit from hardware countermeasures (such as secure logic styles) at low-costs thanks to its regular structure.

3 Security of g against Algebraic SPA

In this section, we analyze the security of the original fresh re-keying scheme from [19] against side-channel attacks that exploit the algebraic structure of the target algorithm. We first show that, if no attention is paid, block ciphers protected by a re-keying scheme can still be vulnerable to algebraic side-channel attacks, due to the simple algebraic structure of the re-keying function g. Next, we suggest that a shuffling of the operations would prevent this kind of attacks (as it prevents most algebraic side-channel attacks, in fact).

In the rest of the paper, we will assume that the encryption function f is the AES Rijndael, and that the side-channel attacks are performed in a known plaintext context. In this setting, the re-keying function g can be written as a system of linear equations over GF(2^8). Let us call $\mathbf{k}$ the vector containing the 16 bytes $k_{.,j}$ $(0 \leq j \leq 15)$ of the master key k, and $\mathbf{k_i^*}$ (resp. $\mathbf{r_i}$) the vector containing the 16 bytes $k^*_{i,j}$ (resp. $r_{i,j}$) of the i^{th} session key generated from the same master key (resp. nonce used). The bytes of the nonces are known, the bytes of the master and session keys are unknown. Hence, the system of equations linking a session key k_i^* to the master key k and the nonce r_i is:

$$\mathbf{R_i}.\mathbf{k} = \mathbf{k_i^*} \tag{1}$$

$$\begin{bmatrix} r_{i,0} & r_{i,15} & r_{i,14} & \cdots & r_{i,1} \\ r_{i,1} & r_{i,0} & r_{i,15} & \cdots & r_{i,2} \\ r_{i,2} & r_{i,1} & r_{i,0} & \cdots & r_{i,3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{i,15} & r_{i,14} & r_{i,13} & \cdots & r_{i,0} \end{bmatrix} . \begin{bmatrix} k_{.,0} \\ k_{.,1} \\ k_{.,2} \\ \vdots \\ k_{.,15} \end{bmatrix} = \begin{bmatrix} k^*_{i,0} \\ k^*_{i,1} \\ k^*_{i,2} \\ \vdots \\ k^*_{i,15} \end{bmatrix} . \tag{2}$$

This system can be represented using a block matrix:

$$\left[\mathbf{R_i} \mid \mathbf{I}\right] . \left[\frac{\mathbf{k}}{\mathbf{k_i^*}}\right] = \left[0\right] , \tag{3}$$

with $\mathbf{I}$ the identity matrix. It is an homogeneous system of 16 linear equations in 32 unknown variables (the bytes of $\mathbf{k}$ and $\mathbf{k_i^*}$). For each additional session key

[2] In the original scheme, the nonce is shared rather than the key. However, this leads to a first-order leakage if a master key byte is zero, similar as observed in [9].

produced from the same master key, we can add 16 new equations in 16 new variables (the bytes of $\mathbf{k^*_{i+1}}$) to the system:

$$\begin{bmatrix} \mathbf{R_1} & \mathbf{I} & 0 & \cdots & 0 \\ \mathbf{R_2} & 0 & \mathbf{I} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{R_n} & 0 & 0 & \cdots & \mathbf{I} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{k} \\ \mathbf{k^*_1} \\ \mathbf{k^*_2} \\ \vdots \\ \mathbf{k^*_n} \end{bmatrix} = \begin{bmatrix} 0 \end{bmatrix} . \tag{4}$$

As such, this system is underdefined. We need at least 16 bytes of information about the session keys $\mathbf{k^*_i}$ in order to identify the value of the master key k[3]. The easiest way to solve this system is thus to find 16 additional linear equations, involving one or more bytes of $\mathbf{k^*_i}$. For this purpose, a straightforward approach is to use side-channel leakage in order to learn session key bytes. Usually, side-channel leakages do not provide the exact data processed by a device, but some information about it. For example, one could assume that three leakage points are obtained, for each session key byte, as illustrated in Figure 2. They correspond to (1) the output byte of the fresh re-keying multiplication itself ($k^*_{i,j}$), (2) the XOR operation between this byte and a known plaintext byte ($P_{i,j} \oplus k^*_{i,j}$), and (3) the output of the AES S-box ($\mathsf{S}(P_{i,j} \oplus k^*_{i,j})$). By combining these three Hamming weight values, it is possible to identify a unique valid value for the session key byte in approximately 16% of the cases. As a result, only 11 encryptions are required to get 16 session key bytes with probability higher than 0.99. Naturally, these simple estimations assume that the Hamming weights are perfectly recovered, while actual attacks may be affected by noise. Nevertheless, they show that algebraic attacks must be considered in the analysis of fresh re-keying schemes.

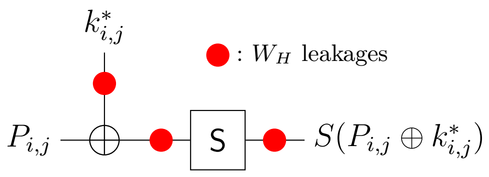


Fig. 2. Three interesting leakage points in the first AES round

Another possibility is to use Side-Channel Collision Attacks (proposed against the DES in [25], and enhanced in [14]). For example, in [3], the author introduces the notion of *generalized internal collisions* for the AES. A generalized internal collision occurs when two AES S-boxes are evaluated on the same input. These two S-boxes can be located in the same round, in two different rounds or even in two different encryptions. If all the AES S-boxes are implemented in a similar way, the computation of the same value should give rise to similar power

[3] Ignoring the fact that the entropy of k is slightly less than 16 bytes.

consumption traces, making the collision detection possible. If a collision is detected between two S-boxes of the first AES round, we translate this information into:

$$P_{i_1,j_1} \oplus k^*_{i_1,j_1} = P_{i_2,j_2} \oplus k^*_{i_2,j_2} \Leftrightarrow k^*_{i_1,j_1} = P_{i_1,j_1} \oplus P_{i_2,j_2} \oplus k^*_{i_2,j_2},$$

Hence, 16 collisions would be sufficient to solve the system of Equation (4). The average number of collisions for n_e encryptions can be computed using the birthday paradox (see [1]): $\mathbf{E}(N(n_e)) = 16n_e - 256 + 256(1 - 1/256)^{16n_e}$. Simulations show that less than 10 encryptions are sufficient to reach the 16 required collisions with high probability.

Summarizing, a straightforward implementation of the re-keying scheme is susceptible to algebraic type of attacks. Importantly, this does not contradict the security analysis in [19], as these attacks can be seen as SPA against the AES function. However, they clearly exhibit the importance of explicit SPA resistance of the block cipher used in the scheme. In a very similar way, fresh re-keying implies executing the AES key scheduling algorithm, in which case Mangard's SPA attack is another possible threat [15] (or similarly [30]).

On the positive side, and as discussed in this previous work, SPA attacks are relatively easy to counteract (compared to DPA). First, they are typically applicable in software implementations with small (e.g. 8-bit) buses. Second, in case small data buses are considered, they are efficiently prevented by shuffling the operations [7,11]. In this respect, it is worth mentioning that, in order to allow secure implementations, the interaction between the f and g functions needs to remain shuffled (which will be ensured in the software implementations of Section 5). Eventually, we also note that the AES Rijndael may not be the most suitable block cipher for efficient shuffling, because of its non-regular and sequential key scheduling algorithm.

4 Extending the Africacrypt Scheme to n Parties

In this section, we present two possible extensions in order to enable the use of fresh re-keying amongst n parties. The first scheme is a straightforward extension of the original scheme and uses n master keys. The second one allows the same functionality but uses only one master key. We show that their security is similar to the one of the original scheme. This will be done in two steps. First, we demonstrate that the session keys are still uniformly distributed and cannot be biased by an adversary controlling all but one of the nonces involved in the re-keying. Second, we argue that the requirement for the diffusion between k and k^* which is the core of the security argument in [19] is still fulfilled.

4.1 Scheme 1: Using n Master Keys

The first solution to the extension problem consists in instantiating n original fresh re-keying schemes with independent master keys. Every party possesses all n master keys and serves as master (that is, it generates the nonce) for

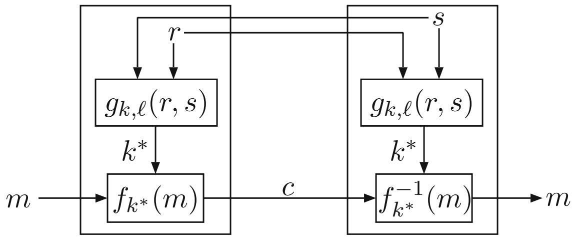


Fig. 3. Basic extension to two parties using two master keys

one instance and as slave (that is, it receives the nonce from outside) for all other $n-1$ instances. The shared session key is derived as the sum of all the n independent session keys, from the n instances. Like in the original scheme, the master keys are constrained to the invertible elements of $\mathrm{GF}(2^8)[y]/(y^{16}+1)$. Figure 3 illustrates the principle for two master keys k and ℓ. In a first step, the parties generate and exchange the nonces r and s. Next, the session key is computed as $k^* = k \cdot r + \ell \cdot s$.

4.2 Scheme 2: Using a Single Master Key

The second scheme is similar to the first scheme, except that only one master key $k \in \left(\mathrm{GF}(2^8)[y]/(y^{16}+1)\right)^*$ is shared amongst the parties. In particular, every party is enumerated with a unique value $i \in [1, n]$. The session key is derived as:

$$k^* = \bigoplus_{i=1}^{n} r_i \cdot k^i,$$

where r_i denotes the nonce generated by party i and k^i denotes the i^{th} power of the master key. Note, that in this scheme, the order of k needs to be greater n. For two parties the session key is derived as $k^* = k \cdot r + k^2 \cdot s$. As for the i values, we assume that the parties are enumerated statically. Such an assumption already covers many scenarios. In a tag-reader scenario, the roles can be assigned in the specification. In car like scenarios, the devices are rarely replaced and thus can be set up by the manufacturer or a certified garage. However, ad-hoc negotiation of the i values would be interesting to cover arbitrary applications. We leave the evaluation and selection of appropriate negotiation protocols for further research.

4.3 Security Model

The challenge when designing an n-party fresh re-keying scheme lies in the fact that each party is provided with $n-1$ nonces from outside. Thus, an adversary potentially has control over $n-1$ out of n nonces. Compared to the single-party case, there are two main properties that need to be verified. First, it should still hold that the adversary controlling the external nonces is not able to significantly bias the session key distributions (e.g. he should not be able to set them to a constant value). Second, it should also hold that there is a strong diffusion between the master key and the session keys (i.e. it should not be possible to guess one byte of session key, excepted by guessing most of the master key). In the following, we show that these conditions are respected in a model where the adversary can eavesdrop the communications and modify the nonce values received by each party. By contrast, he cannot access the master key and he cannot change the nonce value that is generated internally by the target device.

4.4 Security of Scheme 1

Our analysis will be done in two steps. First, and for illustration, we will consider an adversary trying to set the session key to a constant value. We show in Lemma 1, that this is impossible without already possessing substantial information about the master keys. In other words, in order to fix the session key, the quotient of the two master keys has to be guessed correctly. This gives intuition that biasing the session keys is a difficult problem. Next, we consider a more realistic adversary who is just trying to bias the session key distribution. Lemma 2 shows that, whatever the distribution of the external nonces, the session key is close to uniformly distributed given that the nonce generated within the target device is uniformly distributed.

For simplicity, the following lemmata consider the two-party case. Proofs can be extended easily to the general case. Let r_j and s_j be the nonce values generated during the jth execution of the re-keying, and let k_j^* be the corresponding session key. We prove the security of the party generating s_j.

Lemma 1. *If the adversary is able to keep k^* constant, then he also knows ℓ/k. Reciprocally, if the adversary knows k/ℓ, then he can keep k^* constant.*

Proof: Suppose:

$$k^* := kr_1 + \ell s_1 = kr_2 + \ell s_2.$$

Then assuming $s_1 + s_2$ is an invertible element[4], we get:

$$\ell/k = (r_1 + r_2)/(s_1 + s_2),$$

and the adversary can compute this value since he knows r_1, r_2, s_1, s_2. On the other hand, if the adversary knows ℓ/k, then he can choose:

$$r_2 := r_1 + (s_1 + s_2)\ell/k, \tag{5}$$

in order to keep k^* constant. □

[4] The proof can be slightly adapted when $y + 1$ divides $s_1 + s_2$, either by considering the quotient ℓ/k instead, or by dividing both $r_1 + r_2$ and $s_1 + s_2$ by a common power of $y + 1$. Alternatively since the nonces are randomly chosen and a large majority of them produce an invertible $s_1 + s_2$, the adversary can simply wait until $s_1 + s_2$ is invertible.

More generally, let us now define $\delta := (r_1 + r_2)/(s_1 + s_2)$. Since the adversary knows r_1, s_1 and s_2, choosing r_2 is equivalent to choosing δ (from his point of view). In the following lemma, we show that the distribution of $k_2^* + k_1^*$ cannot be biased by the adversary unless he has some information on the value $\delta + \ell/k$.

Lemma 2. *Let the nonces be uniformly distributed and further let* $0 \le e < 16$ *be the maximal power of* $y + 1$ *dividing* $\ell/k + \delta$. *Then the value* $k_{12}^* := k_2^* + k_1^*$ *is uniformly distributed in the set:*

$$\mathcal{K}_e := \{(y+1)^e p(y) | \deg(p) < 16 - e, (y+1) \not| p\},$$

given that the nonces s_i *are uniformly distributed.*

Proof: We have:

$$\begin{aligned} k_2^* &= kr_2 + \ell s_2 \\ &= kr_1 + \ell s_1 + k(r_2 + r_1) + \ell(s_2 + s_1) \\ &= k_1^* + k(\ell/k + \delta)(s_1 + s_2), \end{aligned}$$

hence:

$$k_2^* + k_1^* = (y+1)^e k \frac{\ell/k + \delta}{(y+1)^e}(s_2 + s_1).$$

By definition, $k\frac{\ell/k+\delta}{(y+1)^e}$ is an invertible element of $\mathrm{GF}(28)[y]/(y^{16}+1)$. As a consequence, for any $k_{12}^* = \tilde{k}_{12}(y+1)^e \in \mathcal{K}_e$, we have:

$$k_2^* + k_1^* = \tilde{k}_{12}(y+1)^e \Leftrightarrow s_2 \in \left\{ s_1 + \tilde{k}_{12}k^{-1}\frac{(y+1)^e}{\ell/k+\delta} + (y+1)^{16-e}q(y) | \deg(q) < e \right\}.$$

We see that each k_{12} value corresponds to 2^e values for s_2. Since s_2 is chosen randomly and cannot be controlled by the adversary, we obtain the result. $\square$

Intuitively, Lemma 2 implies that the adversary cannot affect the distribution of $k_2^* + k_1^*$ without affecting the distribution of ℓ/k. Since the adversary has a priori no information at all about ℓ/k, all he can do is to try random values for δ and hope that he guessed ℓ/k up to some large power of $y + 1$. This way, the adversary has no way to decrease the entropy of $k_2^* + k_1^*$. Indeed, $k_2^* + k_1^*$ only belongs to $\mathcal{K}_e$ (that has size $(2^8 - 1)2^{8(15-e)}$) with a probability $2^{-8e} - 2^{-8(e+1)}$.

4.5 Security of Scheme 2

Scheme 2 can be seen as a particular case of Scheme 1 where $\ell = k^2$:

$$k_i^* = kr_i + k^2 s_i.$$

Since the adversary could target both parties, we consider two cases:

1. s is chosen randomly and then r is chosen by the adversary.
2. r is chosen randomly and then s is chosen by the adversary.

The arguments of Section 4.4 also apply to both cases. In the first case, we define $\delta := (r_1 + r_2)/(s_1 + s_2)$ and obtain:

$$k_2^* + k_1^* = k(s_1 + s_2)\,(\delta + k)\,.$$

Using an argument similar to the one in Lemma 2, we have that the adversary cannot modify the distribution of $k_2^* + k_1^*$ without modifying the distribution of $\delta + k$, which requires knowing some information about k. In the second case, we define $\delta := (s_1 + s_2)/(r_1 + r_2)$ and obtain:

$$k_2^* + k_1^* = k^2(s_1 + s_2)\left(\delta + k^{-1}\right).$$

Again by adapting Lemma 2, we see that the adversary cannot modify the distribution of $k_2^* + k_1^*$ without modifying the distribution of $\delta + k^{-1}$, which requires knowing some information about k.

4.6 Security against Divide-and-Conquer Attacks

Most DPA attacks considered in the literature are based on a divide-and-conquer approach. In [19], it is argued that fresh re-keying prevents the application of such a strategy, because every bit of the session key is a sum of half of the master-key bits on average. In addition, since DPA attacks usually require more than one power trace to be successful, this subset changes for every encryption, as long as the nonces are uniformly distributed. Thus, after only a few encryptions, the union of those subsets almost covers the whole master key. In practice, this means that an adversary needs to guess almost all 128 master key bits in order to build 8-bit hypotheses for the session key. More generally, it was shown in [19] that, whatever are the traces selected by the adversary (e.g. those obtained from low Hamming weight nonces), it remains computationally intensive to guess one byte of the session keys. We already showed in the previous section that the control over the external nonces does not allow the adversary to efficiently bias the session keys. As a result, the argument of security against divide-and-conquer attacks for the single party case directly extends to the multi-party case. The best adversarial strategy is to set all the nonces under control to zero and, for the remaining (uncontrolled) nonce, to apply the strategy described in the Africacrypt 2010 paper.

4.7 SCA Security of the Extended Function *g*

Since the function g has a homomorphic property it can be easily shown that there is no leakage of order smaller $t + 1$ with t as the masking order. Furthermore, as shown in the previous subsection, for the final session key, we can rely on the diffusion property after unmasking. However, what has to be considered separately is the unmasking of the different session key contributions. In particular, if an adversary chooses a nonce s in such a way that only one byte is non-zero, every byte of the product involving s depends only on one master-key byte. Thus, after unmasking the product, it is possible to directly build hypotheses for a single master key byte. When implementing the full scheme, one would process the uniform nonce first and then accumulate the shares of the biased product to obtain the final session key. For example let $x = r * k$ and $y = s * \ell$ and further let r be the uniformly distributed nonce and s be the biased nonce with only one non-zero byte. Finally, let x be shared as $x = x_1 + x_2 + x_3$. Then after the first multiplication, the session-key register will hold the intermediate values $(1)\, x, (2)\, x + y_1, (3)\, x + y_1 + y_2$, and finally $(4)\, x + y$. This shows that independently of the masking order, if we implement the scheme straightforwardly there will always be a second-order leakage, namely the joint leakage of $(1)\, x$ and $(4)\, x + y$. Furthermore, these leakage samples are only shuffled over 16 positions.

In order to fix this flaw, the unmasking has to be done in an interleaved manner. That is, first all shares with index one are accumulated, afterwards all shares with index two and so on. For the above example, this means that the session key register will hold the values $(1)\, x_1, (2)\, x_1 + y_1, (3)\, x_1 + y_1 + x_2, (4)\, x_1 + y_1 + x_2 + y_2, (5)\, x + y_1 + y_2$, and finally $(6)\, x + y$. In order to attack y an adversary would need to for instance use the joint distribution of $(6)\, x + y$, $(1)\, x_1$, x_2, and x_3, the dimension of which is greater than t.

5 Software Implementation in an AVR Microcontroller

In this section we will discuss an 8-bit software implementation of the fresh re-keying scheme. As a target platform we chose the AVR microcontroller architecture and in particular the model ATmega128. We target this architecture since it is very common in constrained platforms such as smart cards, which are the ones which will benefit most from our protection scheme.

The selected AVR microcontroller features an 8-bit datapath, 32 general purpose registers and four kilobytes of SRAM. Furthermore, we can rely on four kilobytes of EEPROM and 128 kilobytes of Flash program memory. Finally, most of the instructions finish within a single clock cycle. Unfortunately, our target platform does not feature a hardware random number generator (RNG). Therefore, we state the performance of our implementation in two numbers. First the number of clock cycles assuming that reading a random byte takes 2 clock cycles and second the explicit number of RNG calls needed throughout the execution. This allows a cycle estimation for platforms which do provide such an RNG.

5.1 Multiplication

The basic implementation of the function g relies on a product scan algorithm to implement the polynomial multiplication. That is, every byte c_i of the product $c = a * b$ is calculated as $c_i = \sum_{j=0}^{15} a_{(i-j \mod 16)} * b_j$. The GF$(2^8)$ multiplication is implemented using two lookup tables, a 256 byte large logarithm table (`LOG`) and the corresponding inverse logarithm table (`ILOG`). In the logarithm table, $-\infty$ (`LOG[0x00]`) is encoded as `0xff`. In order to get a conditional-branch free multiplication, we rely on Algorithm 1.

Algorithm 1. Branch-free multiplication

```
Require: a' ← LOG[a], b' ← LOG[b] with a, b ∈ $GF(2^8)$
Ensure: pp = a ∗ b
1: $(\mathrm{carry}, \mathrm{c}) \leftarrow \mathrm{a}' + \mathrm{b}'$
2: pp ← ILOG [c − carry] // mod 255, ILOG [255] := ILOG [0]
3: zfa ← ZF(a′ − 0xff), zfb ← ZF(b′ − 0xff) // ZF ... AVR's zero flag
4: pp ← pp · (1 − (zfa OR zfb))
5: return pp
```

5.2 Shuffling of the Fresh Re-keying

The function g needs to be shuffled for two reasons. First, the noise introduced by shuffling is vital for the impact of masking. Second, in order to prevent attacks as described in Section 3, the position of the leakage samples within the power trace needs to be uncertain. Since g relies on carry-free arithmetic, shuffling can be done efficiently. In this section we describe the two different parts of the polynomial multiplication where we apply shuffling and also discuss two different shuffling techniques. Thus, we provide four different levels of shuffling.

The most important part of the algorithm to shuffle is the processing of the product bytes itself, that is the order in which the c_i bytes are computed. This significantly reduces the information an SPA adversary can learn about the product. In particular, it leaves such an adversary with a probability of 1/16 for gaining information about a specific byte. In addition, it is also possible to shuffle the processing of the partial products corresponding to one product byte. Also here, 16 positions can be shuffled, which results in a total randomization of the partial products over 256 positions.

Internally, both parts of the algorithm are implemented as loops where each iteration operates on independent data. Thus, the simplest way of shuffling is to add an offset modulo 16 to the loop counter. For the bytes of the product, we just start from c_r and run through to $c_{(r+15 \mod 16)}$, for a randomly chosen r. For the partial products of one product byte, and a randomly chosen starting offset s, this would mean $c_i = \sum_{j=0}^{15} a_{(i-(j+s \mod 16) \mod 16)} * b_{(j+s \mod 16)}$. From an overhead point of view, shuffling using a random starting index is negligible.

As a second shuffling method, we indirectly address the product bytes and the operand bytes. The bytes of the product are thus addressed by c_{r_i} where r is now a random permutation of the sequence $0, \ldots, 15$. For the partial products of one product byte this would mean $c_i = \sum_{j=0}^{15} a_{(i-s_j \mod 16)} * b_{s_j}$, for a random permutation s. The generation of such a random permutation is significantly more expensive than generating a random starting offset as in the first strategy. In particular, we start with the sequence $0, \ldots, 15$ and sample n random bytes. For every such random byte we take the four MSbs as position one and the four LSbs as position two. Afterwards, we swap the entries at those positions. In general, the generation of a permutation has a quadratic complexity, thus $n = 16^2$. However, this results in 7 160 clock cycles just to generate one permutation of which we need 17 in total. In order to allow a more flexible tradeoff, we also introduce an intermediate solution which sets n to 16^2 for the first permutation and derives all consecutive ones from the first permutation by swapping 16 random pairs.

For the generation of noise, the two approaches yield similar results, however, if the aim is to prevent SPA attacks, there is a significant difference. In general, if all 16 product bytes need to be recovered within a single trace, the time complexity is 16 for the random starting index and 16! for the full permutation. Since the described SPA attack needs $t > 1$ traces, we can estimate the time complexity of the attacks as 16^t and $16!^t$ respectively. Therefore, for small values of t, a random starting index might not be sufficient to shuffle the product bytes.

5.3 Shuffling of the AES

The AES Rijndael algorithm features a 16 bytes state, thus shuffled implementations usually shuffle these 16 positions plus sometimes additional dummy operations. However, when it comes to the key schedule, only four positions can be shuffled. Therefore, one option in a standard implementation is to store all the round keys in the device instead of computing them on the fly. Unfortunately, we cannot rely on this strategy as we never reuse the same key. As a result we have to introduce three additional key schedules which operate on random dummy data. This allows us again to shuffle over 16 positions and thus achieve the same security level as for the remainder of the algorithm. We implemented a version of the AES which uses fully indirect addressing for all the operations. This allows having complete control over the processed bytes, but negatively affects the performance. In fact our implementation can complete a whole encryption, including one permutation generation, in 30,713 clock cycles.

5.4 Performance Results

Table 1 summarizes the different performance numbers for a single polynomial multiplication when implemented using different levels of shuffling. RSI stands for random starting index and RP-n for random permutation, the generation of which used n swap operations. In addition we state the number of RNG calls. In our implementation, those calls are included with a factor of two in the number of clock cycles. It can be seen that the execution time of one multiplication heavily depends on the way the shuffling is implemented. In order to prevent SPA attacks and to add sufficient noise against DPA attacks we rely on the third solution (RP-256 + RSI) for all further evaluations.

Implementing the masking is equivalent to performing $t + 1$ multiplications for t^{th}-order masking. The reason why first-order masking does not take twice as long as the unmasked version is that we reuse the permutation for the product bytes. We settled for this solution as it provides a good security vs. performance tradeoff. However, the scheme could be implemented using various other tradeoffs, which generally compare favorably with the straightforward protection of a block cipher against side-channel attacks, as will be discussed next.

Table 1. Implementation results for the polynomial multiplication for different levels of shuffling: RSI = random starting index, RP-16 = random permutation generated with 16 swap operations, and RP-256 = random permutation generated with 256 swap operations

Product bytes shuffling	Partial products shuffling	Clock cycles	Calls to RNG	Code size (bytes)	RAM usage (bytes)
-	-	13,400	16	754	48
RSI	RSI	15,032	33	760	48
RP-256	RSI	22,199	288	904	64
RP-256	RP-16	29,688	528	1008	80
RP-256	RP-256	137,208	4368	1008	80

In order to finalize the two n-party schemes, we need to perform n masked multiplications for Scheme 1 and $2n - 1$ multiplications for Scheme 2, if the powers of k are computed on the fly. If the powers of k are precomputed, the execution times of the schemes are equivalent. As for the RAM usage, we need n times the RAM, as the n multiplications need to be performed in parallel to allow interleaving the shares (as discussed in Section 4.7). Only the 16 bytes for the session key accumulation can be shared.

Finally, we compare the total performance of our scheme with the provable secure AES implementation of CHES 2010 [24]. Their results have been recently improved [8] but this paper states only figures for the masked S-box and not for the entire cipher, therefore we still use the figures from the CHES publication. For third-order masking, the implementation of Rivain and Prouff takes 470k clock cycles. Compared to that, our implementation with third-order masking and shuffling takes 125k+31k=156k clock cycles if either Scheme 1 is used or Scheme 2 with precomputed powers of k. This shows the experimental evidence that fresh re-keying is also attractive in software. Especially, because the performance scales linearly with the masking order. For a larger number of involved parties though, it would be favorable to have a hardware polynomial multiplier as in [19].

Table 2. Implementation results for the different masking orders with randomly permuted product bytes (RP-256) and random starting indices for the partial products (RSI)

masking order	single multiplication	Scheme 1	Scheme 2
w/o masking	22,199	48,398	66,597
1^{st}-order	35,559	71,118	106,677
2^{nd}-order	48,919	97,838	146,757
3^{rd}-order	62,279	124,558	186,837

6 Future Research and Conclusions

In this paper we analyzed and extended the fresh re-keying scheme as introduced at Africacrypt 2010. In particular, we extended their research on three lines. First, we analyzed the scheme's susceptibility to algebraic SPA attacks. We showed that on the one hand, if no precautions are taken, such attacks can be easily applied. On the other hand, the assumptions for such attacks are also non-trivial, thus they can be efficiently prevented by shuffling. Second, we extended the scheme to n parties and showed that the security of the two proposed extensions is similar to the one of the original scheme. From a performance point of view the extensions scale linearly in n and our second scheme does not even need additional key material. Third, we implemented the scheme on an 8-bit microcontoller architecture in order to show its efficiency in software. Thus, the paper shows that the fresh re-keying scheme seems to be an appealing choice to provide side-channel security for automotive applications. That is, applications where many low-cost parties need to communicate in a secure way.

Eventually, while the security versus performance results of the fresh re-keying are pretty strong, it is important to note that the analysis of the scheme combines several components. Namely, the overall resistance against side-channel attacks relies on the provable secure masking for the multiplications, the impossibility of biasing the session key, the interleaved recombination of the session key shares and finally, the SPA security of all components. While present evaluation does not reveal obvious weaknesses, a more formal evaluation of the proposed solution, allowing to precisely understand and argue about the interaction between these components, would be a nice scope for further research. Besides, another interesting open problem would be to study the use of a randomness extractor as re-keying function. As recently discussed in [18], such extractors have interesting properties for leakage resilience, when implemented in hardware. Since they are central elements in proofs of leakage resilience such as [6], their use for improving the formal analysis of fresh re-keying schemes could be investigated as well.

Acknowledgements. The work was partly funded by the 7th framework European project TAMPRES, the Belgian Fund for Scientific Research (FNRS-F.R.S.) and the Walloon Region projects S@T Skywin and SCEPTIC.

References

1. The Department of Computer Science at Duke University, Discrete Mathematics for Computer Science lecture, Chapter 18: Probability in hashing (2009), http://www.cs.duke.edu/courses/cps102/spring09/Lectures/L-18.pdf
2. Bertoni, G., Breveglieri, L., Koren, I., Maistri, P., Piuri, V.: Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard. IEEE Trans. Computers 52(4), 492–505 (2003)
3. Bogdanov, A.: Improved Side-Channel Collision Attacks on AES. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876, pp. 84–95. Springer, Heidelberg (2007)
4. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards Sound Approaches to Counteract Power-Analysis Attacks. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999)

5. Dodis, Y., Pietrzak, K.: Leakage-Resilient Pseudorandom Functions and Side-Channel Attacks on Feistel Networks. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 21–40. Springer, Heidelberg (2010)
6. Dziembowski, S., Pietrzak, K.: Leakage-Resilient Cryptography. In: FOCS, pp. 293–302. IEEE Computer Society (2008)
7. Feldhofer, M., Popp, T.: Power Analysis Resistant AES Implementation for Passive RFID Tags. In: Lackner, C., Ostermann, T., Sams, M., Spilka, R. (eds.) Proceedings of Austrochip 2008, Linz, Austria, October 8, pp. 1–6 (2008) ISBN 978-3-200-01330-8
8. Genelle, L., Prouff, E., Quisquater, M.: Montgomery's Trick and Fast Implementation of Masked AES. In: Nitaj, A., Pointcheval, D. (eds.) AFRICACRYPT 2011. LNCS, vol. 6737, pp. 153–169. Springer, Heidelberg (2011)
9. Golic, J.D., Tymen, C.: Multiplicative Masking and Power Analysis of AES. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 198–212. Springer, Heidelberg (2003)
10. Goubin, L., Patarin, J.: DES and Differential Power Analysis (The "Duplication" Method). In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 158–172. Springer, Heidelberg (1999)
11. Herbst, C., Oswald, E., Mangard, S.: An AES Smart Card Implementation Resistant to Power Analysis Attacks. In: Zhou, J., Yung, M., Bao, F. (eds.) ACNS 2006. LNCS, vol. 3989, pp. 239–252. Springer, Heidelberg (2006)
12. Karri, R., Wu, K., Mishra, P., Kim, Y.: Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. IEEE Trans. on CAD of Integrated Circuits and Systems 21(12), 1509–1517 (2002)
13. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S.: Experimental Security Analysis of a Modern Automobile. In: IEEE Symposium on Security and Privacy, pp. 447–462. IEEE Computer Society (2010)
14. Ledig, H., Muller, F., Valette, F.: Enhancing Collision Attacks. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 176–190. Springer, Heidelberg (2004)
15. Mangard, S.: A Simple Power-Analysis (SPA) Attackon Implementations of the AES Key Expansion. In: Lee, P.J., Lim, C.H. (eds.) ICISC 2002. LNCS, vol. 2587, pp. 343–358. Springer, Heidelberg (2003)
16. Mangard, S., Standaert, F.-X. (eds.): CHES 2010. LNCS, vol. 6225. Springer, Heidelberg (2010)
17. McEvoy, R.P., Tunstall, M., Whelan, C., Murphy, C.C., Marnane, W.P.: All-or-Nothing Transforms as a Countermeasure to Differential Side-Channel Analysis. Cryptology ePrint Archive, Report 2009/185 (2009), http://eprint.iacr.org/
18. Medwed, M., Standaert, F.-X.: Extractors against Side-Channel Attacks: Weak or Strong? In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 256–272. Springer, Heidelberg (2011)
19. Medwed, M., Standaert, F.-X., Großschädl, J., Regazzoni, F.: Fresh Re-keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices. In: Bernstein, D.J., Lange, T. (eds.) AFRICACRYPT 2010. LNCS, vol. 6055, pp. 279–296. Springer, Heidelberg (2010)
20. Oren, Y., Kirschbaum, M., Popp, T., Wool, A.: Algebraic Side-Channel Analysis in the Presence of Errors. In: Mangard, Standaert (eds.) [16], pp. 428–442
21. Pietrzak, K.: A Leakage-Resilient Mode of Operation. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 462–482. Springer, Heidelberg (2009)

22. Plos, T., Feldhofer, M.: Hardware Implementation of a Flexible Tag Platform for Passive RFID Devices. In: Proceedings of the 14th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2011), Oulu, Finland, August 2010, pp. xxx–xxx. IEEE Computer Society (2011)
23. Renauld, M., Standaert, F.-X., Veyrat-Charvillon, N.: Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 97–111. Springer, Heidelberg (2009)
24. Rivain, M., Prouff, E.: Provably Secure Higher-Order Masking of AES. In: Mangard, Standaert (eds.) [16], pp. 413–427
25. Schramm, K., Wollinger, T., Paar, C.: A New Class of Collision Attacks and Its Application to DES. In: Johansson, T. (ed.) FSE 2003. LNCS, vol. 2887, pp. 206–222. Springer, Heidelberg (2003)
26. Standaert, F.-X.: Leakage Resilient Cryptography: a Practical Overview. Invited talk, ECRYPT Workshop on Symmetric Encryption (SKEW 2011), Copenhagen, Denmark (February 2011), `http://perso.uclouvain.be/fstandae/PUBLIS/96_slides.pdf`
27. Standaert, F.-X., Pereira, O., Yu, Y., Quisquater, J.-J., Yung, M., Oswald, E.: Leakage Resilient Cryptography in Practice. In: Towards Hardware Intrinsic Security: Foundation and Practice (book chapter), pp. 105–139. Springer, Heidelberg (2010)
28. Tiri, K., Verbauwhede, I.: Securing Encryption Algorithms against DPA at the Logic Level: Next Generation Smart Card Technology. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 125–136. Springer, Heidelberg (2003)
29. Tiri, K., Verbauwhede, I.: A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In: DATE, pp. 246–251. IEEE Computer Society (2004)
30. VanLaven, J., Brehob, M., Compton, K.J.: Side Channel Analysis, Fault Injection and Applications - A Computationally Feasible SPA Attack on AES via Optimized Search. In: Sasaki, R., Qing, S., Okamoto, E., Yoshiura, H. (eds.) SEC, pp. 577–588. Springer, Heidelberg (2005)
31. Yu, Y., Standaert, F.-X., Pereira, O., Yung, M.: Practical leakage-resilient pseudorandom generators. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) ACM Conference on Computer and Communications Security, pp. 141–151. ACM (2010)

Fast Key Recovery Attack on ARMADILLO1 and Variants

Pouyan Sepehrdad, Petr Sušil*, and Serge Vaudenay

EPFL, Lausanne, Switzerland
{pouyan.sepehrdad,petr.susil,serge.vaudenay}@epfl.ch

Abstract. The ARMADILLO cryptographic primitive is a multi-purpose cryptographic primitive for RFID devices proposed at CHES'10. The main purpose of the primitive is to provide a secure authentication in a challenge-response protocol. It has two versions, named ARMADILLO (subsequently denoted by ARMADILLO1) and ARMADILLO2. However, we found a fatal weakness in the design which allows a passive attacker to recover the secret key in polynomial time, of ARMADILLO1 and some generalizations. We introduce some intermediate designs which try to prevent the attack and link ARMADILLO1 to ARMADILLO2. Considering the fact that the attack against ARMADILLO1 is polynomial, this brings about some concerns into the security of the second version ARMADILLO2, although it remains unbroken so far.

1 Introduction

ARMADILLO is a hardware oriented multi-purpose cryptographic primitive presented at CHES'10 [2]. It was built for RFID applications. It can be used as a PRF/MAC, e.g. for a challenge-response protocol as a MAC, and also as a hash function for digital signatures, or a PRNG for making a stream cipher. It has two versions, named ARMADILLO (subsequently denoted by ARMADILLO1) and ARMADILLO2.

During the review process of CHES'10 we found an attack against ARMADILLO1 and its variants. The ARMADILLO2 includes a quick fix to resist it. The attack and its variants are presented in this paper.

To fix the vulnerability of ARMADILLO1 and simultaneously shrink the design, we define multiple intermediate versions of ARMADILLO and we investigate their security with respect to the original attack and illustrate that they are still vulnerable to a key recovery or a forgery attack. Although our attack is not applicable against ARMADILLO2, the step by step approach in the design of other variants would give a concern behind the security of ARMADILLO2. These intermediate designs reveal that the security bounds on ARMADILLO2 might be insufficient.

We introduce a generalized version ARMADILLOgen and we explain when the key recovery or forgery attack is possible. Finally, we come to the definition of ARMADILLO2.

The attacks we show have always complexity polynomial in the size of input. Specifically, the attack against ARMADILLO1 has complexity $O(k^2 \log k)$ and it can be performed "by hand", as the actual key recovery algorithm is very simple.

* Supported by a grant of the Swiss National Science Foundation, 200021_134860/1.

1.1 Related Work

In [1] the authors found an attack against ARMADILLO2 based on parallel matching. The key recovery attack against FIL-MAC application of ARMADILLO2-A and ARMADILLO2-E using single challenge-response pair is 2^7 and 2^{18} times faster than exhaustive search respectively. The techniques presented in our paper may help to reduce the time complexity if the attacker uses multiple samples.

2 Description of ARMADILLO

ARMADILLO relies on data-dependent bit transpositions. Given a bitstring x with bit ordering $x = (x_\ell \| \cdots \| x_1)$, fixed permutations σ_0 and σ_1 over the set $\{1, 2, \ldots, \ell\}$, a bit string s, a bit $b \in \{0, 1\}$ and a permutation σ, define $x_{\sigma_s} = x$ when s has length zero, and $x_{\sigma_{s\|b}} = x_{\sigma_s \circ \sigma_b}$, where x_σ is the bit string x transposed by σ, that is,

$$x_\sigma = (x_{\sigma^{-1}(\ell)} \| \cdots \| x_{\sigma^{-1}(1)})$$

The function $(s, x) \mapsto x_{\sigma_s}$ is a data-dependent transposition of x. The function $s \mapsto \sigma_s$ can be seen as a particular case of the general semi-group homomorphism from $\{0, 1\}^*$ to a group G.

Notations. Throughout this document, $\|$ denotes the concatenation of bitstrings, $\oplus$ denotes the bitwise XOR operation, $\bar{x}$ denotes the bitwise complement of a bitstring x; we assume the little-endian numbering of bits, such as $x = (x_\ell \| \cdots \| x_1)$.

In this section, we give the description of two variants ARMADILLO1 and ARMADILLO2. Then, we introduce a common generalized version ARMADILLOgen and show how it relates to all versions. We show the attack against ARMADILLOgen for many different choices of parameters.

2.1 ARMADILLO1

ARMADILLO1 maps an initial value C and a message block U to two values (see Fig. 1).

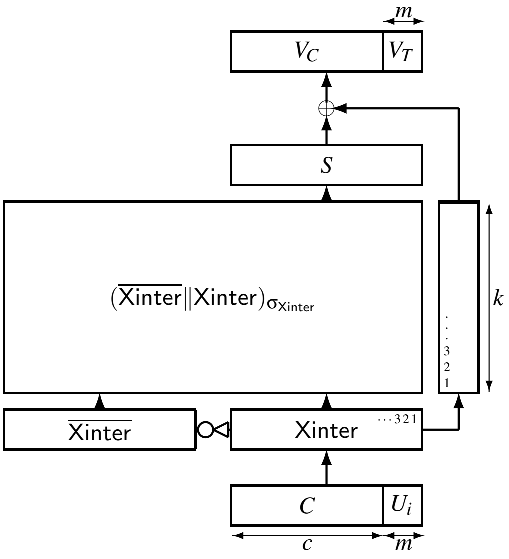
$$(V_C, V_T) = \text{ARMADILLO1}(C, U)$$

ARMADILLO1 works based on a register Xinter. By definition, C and V_C are of c bits, V_T as well as each block U_i are of m bits, Xinter is of $k = c + m$ bits. ARMADILLO1 is defined by integer parameters c, m, and two fixed permutations σ_0 and σ_1 over the set $\{1, 2, \ldots, 2k\}$. Concretely, we consider $m \geq 40$ and $k = c + m$. To initialize ARMADILLO1, Xinter is set to $C \| 0^m$ where 0^m is a null padding block, and C is an initial value. ARMADILLO1 works as follows (Fig. 1).

1: in the i-th step, replace the rightmost m-bit block of Xinter by the block U_i;
2: set a $\ell = 2k$ bits register $x = \overline{\mathsf{Xinter}} \| \mathsf{Xinter}$;
3: x undergoes a sequence of bit permutations which we denote by P. The output of this sequence of bit permutations is truncated to the rightmost k bits, denoted S, by

$$S = \mathsf{tail}_k((\overline{\mathsf{Xinter}} \| \mathsf{Xinter})_{\sigma_{\mathsf{Xinter}}})$$

4: set Xinter to the value of $S \oplus$ Xinter.
5: after processing the last block U_n, take $(V_C \| V_T) =$ Xinter as the output.



m
V_C
V_T
S
$(\overline{\text{Xinter}} \| \text{Xinter})_{\sigma_{\text{Xinter}}}$
k
$\overline{\text{Xinter}}$
Xinter
···321
3
2
1
C
U_i
c
m

Fig. 1. Scheme of ARMADILLO1

2.2 ARMADILLO2

For completeness, we now provide the description of ARMADILLO2 [2] here. The ARMADILLO2 is mostly based on ARMADILLO1b (be defined later) with an additional pre-processing mechanism. As the reader see later in the paper, the pre-processing prevents our attack. We note that the pre-processing step outputs a sequence of bits that defines the data dependent permutation and ensures that the data dependent permutation σ_{Xinter} cannot be easily controlled by the attacker (see Fig. 2).

1: in the i-th step, replace the rightmost m-bit block of Xinter by the block U_i;
2: set a $\ell = k$ bits register $x =$ Xinter;
3: x undergoes a sequence of bit permutations, σ_0 and σ_1 and a constant γ addition, which we denote by P. In fact, P maps a bitstring of m bits and a vector x of k bits into another vector of k bits as $P(s\|b,x) = P(s, x_{\sigma_b} \oplus \gamma)$, where $b \in \{0,1\}$ and x_{σ_b} is a permutation of bits of x (transposition). The output of this sequence of k bit permutations and constant addition is denoted $Y = P(U_i, x)$. We call this step pre-processing, since it is used to define the permutation for the consequent step.
4: x undergoes a sequence of bit permutations and constant addition P defined by Y. The output of this sequence of k bit permutations and constant addition is denoted $S = P(Y, x)$.
5: set Xinter to the value of $S \oplus$ Xinter.
6: after processing the last block U_n, take $(V_C \| V_T)$ as the output.

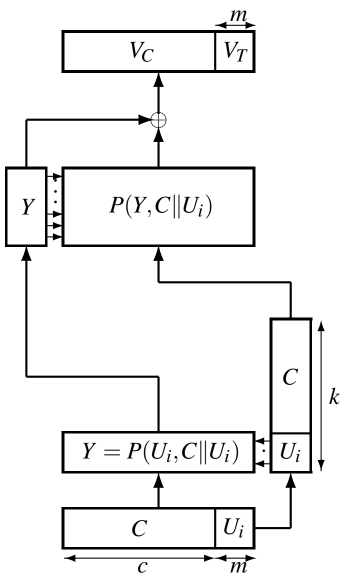


Fig. 2. Scheme of ARMADILLO2

3 General ARMADILLOgen Algorithm

We define various intermediate versions of ARMADILLO. These intermediate versions show the relation between ARMADILLO1 and ARMADILLO2 and give a security concern on ARMADILLO2. We explain step by step how the weakness in the design of ARMADILLO1 relate to a possible weaknesses in design of ARMADILLO2.

All these versions are based on data-dependent permutation P. They all can be covered under ARMADILLOgen as a parametrized version of distinct variants, and by setting corresponding parameters we obtain ARMADILLO1, ARMADILLO1b, ARMADILLO1c, ARMADILLO1d and ARMADILLO2. We show an attack against ARMADILLOgen for some choices of parameters.

ARMADILLOgen is defined as

$$\mathsf{ARMADILLOgen}(X) = T_4(P(T_1(X), T_2(X)), X)$$

where

$$P(s\|b, Y) = P(s, T_3(b, Y))$$
$$P(\lambda, Y) = Y$$

λ denotes the empty string, T_1, T_2, and T_4 are some linear functions, and T_3 in its most general form is

$$T_3(b, Y) = L(Y)_{\sigma_b} \oplus \gamma$$

where L is linear and γ is a constant.

Then, ARMADILLO1 is defined as ARMADILLOgen for

$$\begin{aligned}T_1(X) &= X\\ T_2(X) &= \overline{X}\|X\\ T_3(b,X) &= X_{\sigma_b}\\ T_4(X,Y) &= \mathsf{tail}_k(X)\oplus Y\end{aligned}$$

ARMADILLO2 is defined as ARMADILLOgen for

$$\begin{aligned}T_1(X) &= P(\mathsf{tail}_m(X),X)\\ T_2(X) &= X\\ T_3(b,X) &= X_{\sigma_b}\oplus\gamma\\ T_4(X,Y) &= X\oplus Y\end{aligned}$$

3.1 ARMADILLO1b: Shrinking the Xinter Register

The ARMADILLO1b is a compact version of ARMADILLO1 which prevents the preservation of Hamming weight by adding a constant. However, it does not prevent the attack against ARMADILLO1. According to [2], the ARMADILLO1 design prevents a distinguishing attack based on constant Hamming weight by having the double sized internal register and the final truncation, assuming the output of P transposition looks pseudorandom. We see later in this paper (see section 4) that this proof does not hold in standard attack model and ARMADILLO1 can be broken in polynomial time. First, we define ARMADILLO1b and then demonstrate an attack against this version and explain how the same attack can be used against ARMADILLO1.

ARMADILLO1b is defined as ARMADILLOgen for

$$\begin{aligned}T_1(X) &= X\\ T_2(X) &= X\\ T_3(b,X) &= X_{\sigma_b}\oplus\gamma\\ T_4(X,Y) &= X\oplus Y\end{aligned}$$

In the design of ARMADILLO1b the state size is reduced to k bits to save more gates. So, there is only the register Xinter and not its complement, and there is no truncation. To avoid Hamming weight preservation, after each permutation there is an XOR of the current state with a constant γ (see Fig. 3).

3.2 ARMADILLO1c: Adding a Linear Layer in T_3

To investigate whether a more complex layer in T_3 can prevent the attack on ARMADILLO1b we define ARMADILLO1c. It is defined as ARMADILLOgen for

$$\begin{aligned}T_1(X) &= X\\ T_3(b,X) &= L(X)_{\sigma_b}\oplus\gamma\end{aligned}$$

for a linear transformation L, with arbitrary linear T_2 and T_4.

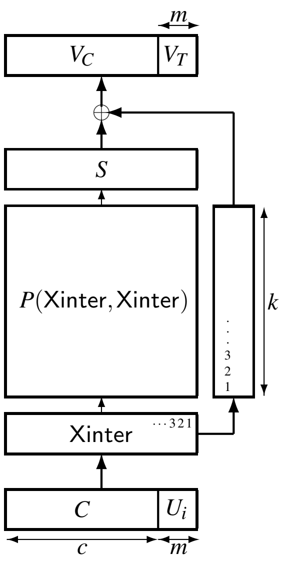


Fig. 3. Scheme of ARMADILLO1b

3.3 ARMADILLO1d: Adding a Fixed Transposition in T_1

We will see later that ARMADILLO1c is still vulnerable. To prevent the attack on ARMADILLO1c, a fixed transposition was added in T_1 to mix the bits of the secret and the challenge. ARMADILLO1d is defined as ARMADILLOgen for

$$T_1(X) = X_\pi$$
$$T_3(b,X) = X_{\sigma_b} \oplus \gamma$$

for a fixed permutation π and with arbitrary linear T_2 and T_4.

4 Key Recovery Attack against ARMADILLO1 and ARMADILLO1b

In this section, we describe an attack against two versions of ARMADILLO. We first explain the attack on ARMADILLO1b and then setting $\gamma = 0$ and extending the initial state to $(\overline{\mathsf{Xinter}}\|\mathsf{Xinter}) = (\overline{C\|U}\|C\|U)$, the same attack can be directly used against ARMADILLO1.

Since ARMADILLO has more than one applications, we just briefly explain how it is deployed in the challenge-response application. We refer the reader to [2] for more details. The objective is to have a fixed input-length MAC. Suppose that C is a secret and U is a one block challenge. The value V_T is the response or the authentication tag. We write

$$V_T = \mathsf{ARMADILLO}(C,U)$$

As can be seen from the description of the algorithm, there is no substitution layer. This means that for a fixed key C the permutation σ_C is fixed (but unknown). As we see later in the paper, it can be easily recovered. For the attack it suffices to recover the mapping σ_C of a single index, for instance we recover $\sigma_C(j) = n$ for some value j. If we can recover the mapping $\sigma_C(j)$, we than take challenges U_i so that j-th bit of $P(U_i, C\|U_i)$ contains different bits of the key. This allows us to recover the secret key from literally reading the key from the output of ARMADILLO1b. We also show that the attack can be extended to other scenarios, or can be changed to forgery attack if the key recovery is not possible. More precisely, we consider

$$T_1(X) = X$$
$$T_3(b,X) = X_{\sigma_b} \oplus \gamma$$

with arbitrary linear T_2 and T_4. This includes ARMADILLO1 and ARMADILLO1b.

The attack is based on the fact that a bit permutation is linear with respect to XOR operation, i.e., for a permutation σ, X and Y be two vectors, we have $(X \oplus Y)_\sigma = X_\sigma \oplus Y_\sigma$.

Lemma 1. *For any T_3, C, and U, we have*

$$P(C\|U, C\|U) = P(C, P(U, C\|U))$$

Proof. We easily prove it by induction on the size of C. □

Lemma 2. *For $T_3(b,X) = X_{\sigma_b} \oplus \gamma$, there exists a function $f : 2^{|X|} \to 2^{|X|}$ such that for any $Y = (y_k\|\ldots\|y_1)$ and X, we have*

$$P(Y,X) = X_{\sigma_Y} \oplus f(Y)$$

Proof. Let rewrite

$$P(Y,X) = \left(\left(\left(X_{\sigma_{y_1}} \oplus \gamma \right)_{\sigma_{y_2}} \oplus \gamma \right)_{\sigma_{y_3}} \oplus \gamma \ldots \right)_{\sigma_{y_k}} \oplus \gamma$$

Let define the *prefix* of Y as

$$\text{prefix}(Y) = \{Y_j;\ Y_j = (y_k\|\ldots\|y_j), 1 \le j \le k\}$$

Thus, P can be rewritten as

$$P(Y,X) = (X \oplus \gamma)_{\sigma_Y} \oplus \gamma \oplus \bigoplus_{p \in \text{prefix}(Y)} \gamma_{\sigma_p} = X_{\sigma_Y} \oplus P(Y,0)$$

□

Now we apply the above results to ARMADILLOgen with $T_1(X) = X$ and $T_3(b,X) = X_{\sigma_b} \oplus \gamma$.

$$\begin{aligned}
\text{ARMADILLOgen}(C\|U) &= T_4(P(C\|U, T_2(C\|U)), C\|U) \\
&= T_4(P(C, P(U, T_2(C\|U))), C\|U) \\
&= T_4(P(C, (L_U(C)_{\sigma_U} \oplus f(U))), C\|U) \\
&= T_4\left((L_U(C)_{\sigma_U} \oplus f(U))_{\sigma_C} \oplus f(C), C\|U \right)
\end{aligned}$$

where $L_U(C) = T_2(C\|U)$ and $f(U)$ is given by Lemma 2. The first equality is coming from the definition, the second from Lemma 1 and the last two from Lemma 2. So, we can write

$$\mathsf{ARMADILLOgen}(C\|U) = L\left((L_U(C)_{\sigma_U} \oplus f(U))_{\sigma_C} \oplus g(U) \oplus h(C)\right)$$

for some linear function L and some functions g and h. For all the variants we consider, L is either the identity function or consists of dropping a few bits. For $\mathsf{ARMADILLO1b}$ and $\mathsf{ARMADILLO1}$ the function $h(C) = f(C) \oplus (C\|0^m)$, $g(U) = (0^c\|U)$. Similarly, $L(X) = X$ and $L(X) = \mathsf{tail}_k(X)$ respectively.

In what follows, we consider an arbitrary i and take a vector e_i such that $e_i \cdot L(X) = X[i]$, i.e, the i-th bit of register X. So, we obtain

$$e_i \cdot \mathsf{ARMADILLOgen}(C\|U) = (L_U(C)_{\sigma_U} \oplus f(U))_{\sigma_C^{-1}(i)} \oplus g(U)_i \oplus h(C)_i$$

Clearly, there exists a $j = \sigma_C^{-1}(i)$ such that

$$e_i \cdot \mathsf{ARMADILLOgen}(C\|U) \oplus g(U)_i = L_U(C)_{\sigma_{U_t}^{-1}(j)} \oplus f(U)_j \oplus h(C)_i \qquad (1)$$

In chosen-input attacks against the PRF mode, we assume that the adversary can compute

$$e_i \cdot \mathsf{ARMADILLOgen}(C\|U)$$

for a chosen U and a secret C. In the challenge-response application, we only have access to V_T, but in all considered variants, e_i has Hamming weight one, so we just need to select i so that this bit lies in the V_T window. We introduce an attack (see Fig. 4) which only needs this bit of the response for $n = k \log k$ queries. This algorithm has complexity $O(k^2 \log k)$ to recover the secret C (also see Fig. 5). In fact, the attacker can simply recover the permutation $Y = P(U_i, \mathsf{Xinter})$, since she has control over U_i's. Now, her goal is to find out how $P(C,Y)$ maps the index j to i. The goal of the algorithm is to find this mapping and recovers C. It is exploiting the fact that fixing the i, then $h(C)_i$ is fixed for all challenges and the left side of Eq. (1) can be computed directly by the adversary. Then, it recovers C by solving an overdefined linear system of equations and check it has a solution. If so, it checks whether the recovered C is consistent with other samples.

Attack complexity. The first **for** loop runs $\mathsf{ARMADILLO}$ algorithm $k \log k$ times . The second loop runs ℓ times where $\ell = 2k$ for $\mathsf{ARMADILLO1}$ and $\ell = k$ for $\mathsf{ARMADILLO1b}$. We perform up to $2k \log k$ simple arithmetic operations in the second loop to compute values $L_{U_t}(C)_{\sigma_{U_t}^{-1}(j)}$. Solving the system of n linear equation requires $O(n^3)$ in general case. However, in the case of $\mathsf{ARMADILLO1}$ and $\mathsf{ARMADILLO1b}$ every line contains only one variable of secret C, which comes from the Lemma 2. As we have $k \log k$ equations in c variables, if the mapping $i \to j$ is not guessed correctly we have high probability to obtain contradiction on line 13. So overall, we have complexity of $O(k^2 \log k)$ for attacking both $\mathsf{ARMADILLO1}$ and $\mathsf{ARMADILLO1b}$.

Probability of success. We first choose randomly $k \log k$ challenges U_t and compute $\mathsf{ARMADILLOgen}(C\|U_t)$. That is because, according to *coupon collector problem* [3]

```
1: Pick a random $i$ from 1 to $m$.
2: for $t$ from 1 to $n = k \log k$ do
3:    collect challenge-response pair $(U_t, e_i \cdot \mathsf{ARMADILLOgen}(C\|U_t))$
4:    compute $b_t = e_i \cdot \mathsf{ARMADILLOgen}(C\|U_t) \oplus g(U_t)_i$.
5: end for
6: for $j$ from 1 to $\ell$ do
7:    for each $\beta \in \{0,1\}$ do
8:       set $h(C)_i = \beta$.
9:       for $t$ from 1 to $n$ do
10:         compute $L_{U_t}(C)_{\sigma_{U_t}^{-1}(j)} = b_t \oplus f(U_t)_j \oplus \beta$ for all $c$ bits.
11:      end for
12:      solve the system of $n$ linear equations $L_{U_t}(C)_{\sigma_{U_t}^{-1}(j)}$
13:      if no solution then
14:         break
15:      end if
16:      derive $C$
17:      if $C$ is consistent with samples then
18:         output $C$.
19:      end if
20:   end for
21: end for
```

Fig. 4. The key recovery algorithm against ARMADILLO1 and ARMADILLO1b

the expected number of challenges so that every bit of C is mapped to i-th bit of output is $k\log k$. Therefore, among $k\log k$ challenges all the bits of challenge and all the bits of secret key are mapped to a single bit of the output. The attacker can derive equation for the j-th bit of $P(U_t, C\|U_t)$, and for $k\log k$ distinct challenges U_t the set of equations will have full rank. These equations do not change through the fixed mapping σ_C, only the constant term might change due to term $P(C,0)$. Therefore if the attacker guess $j \to i$ correctly, the set of $k\log k$ equations in c variables has a solution, otherwise the set of $k\log k$ equations in c variables has no solution with probability at least $1-2^{-n}$.

Failure of the previous attack. The previously mentioned attack would fail, if the permutations σ_0, σ_1 map bit indices in the set $[1,m]$ to the set $[1,m]$, i.e., $\sigma_0[1..m] = [1..m]$ and $\sigma_1[1..m] = [1..m]$. So, it might be speculated that picking such permutations in the design makes the cryptosystem secure. However, the ARMADILLO with such permutations is vulnerable to a simple forgery attack.

Let remind the decomposition

$$\begin{aligned}\mathsf{ARMADILLO1b}(C\|U) &= P(C\|U, C\|U) \oplus (C\|U)\\ &= P(C, P(U, C\|U)) \oplus (C\|U)\\ &= P(C, (L(C)_{\sigma_U} \oplus f(U))) \oplus g(U)\\ &= (L(C)_{\sigma_U} \oplus f(U))_{\sigma_C} \oplus g(U) \oplus h(C)\end{aligned}$$

Given m challenges which form a linearly independent system, we can compute the response by solving the set of these linear equations.

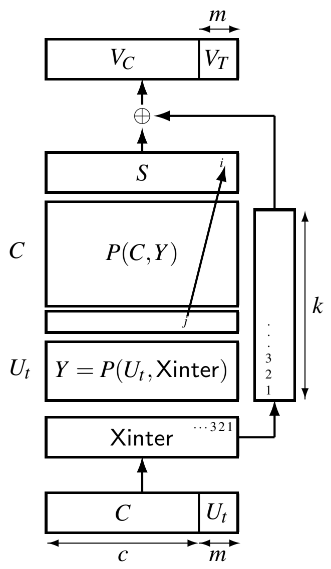


Fig. 5. Scheme of the key recovery algorithm against ARMADILLO1 and ARMADILLO1b

5 Attack Extension with Linear Layer in T_3 (ARMADILLO1c)

T_3 function is very simple in the previous versions. The first attempt to prevent the previous attack is to use a more complex layer but still linear and check whether it prevents the attack. We define another intermediate version and call it ARMADILLO1c. Then, we show that only adding a linear layer L in $T_3(b,X) = L(X)_{\sigma_b} \oplus \gamma$ would not prevent the attack.

Let L be a linear transformation. This attack requires $O(k)$ challenges and three Gaussian eliminations which require $O(k^3)$ operations. We define the new ARMADILLO1c as follows.

$$\mathsf{ARMADILLO1c}(C\|U) = T_4(P(C\|U, T_2(C\|U)), (C\|U))$$

where

$$\begin{aligned} P(s\|b,Y) &= P(s, L(Y)_{\sigma_b} \oplus \gamma) \\ P(\lambda,Y) &= Y \end{aligned}$$

We build a system of equations, where

$$\begin{aligned} \mathsf{ARMADILLO1c}(C\|U_j) &= T_4\left(P(C\|U_j, T_2(C\|U_j)), C\|U_j\right) \\ &= T_4\left(P(C, P(U_j, T_2(C\|U_j))), C\|U_j\right) \\ &= T_4\left(P(C, P(U_j, T_2(C\|0) \oplus T_2(0\|U_j))), C\|U_j\right) \\ &= T_4\left(P(C, P(U_j, T_2(C\|0)) \oplus P(U_j, T_2(0\|U_j))), C\|U_j\right) \\ &= T_4\left(L_C(L_{U_j}(C) \oplus \gamma_j) \oplus P(C,0), C\|U_j\right) \end{aligned}$$

where $L_{U_j}(C) = P(U_j, T_2(C\|0))$ and $\gamma_j = P(U_j, T_2(0\|U_j))$.

We use the fact that a set of $ck+1$ equations $L_{U_j}(C)$ is linearly dependent. Using this we can bypass the unknown mapping L_C. We can find a set J of equations whose sum is 0. Let ε be the parity of the cardinality of J. We obtain

$$\bigoplus_{j\in J} \mathsf{ARMADILLO1c}(C\|U_j) = T_4\left(L_C\left(\bigoplus_{j\in J}\gamma_j\right)\oplus \varepsilon P(C,0), \varepsilon C\|\bigoplus_{j\in J}U_j\right)$$

Using the above expression with several J's, we can recover linear mapping L_C and $P(C,0)$. Using the knowledge of $L_C(X)$ we recover $P(C,2^i)$ for $0<i<k$. We do this by Gaussian elimination on values $\left(\bigoplus_{j\in J}\gamma_j\right)$. Using $P(C,2^i)$ for $0\le i<k$, we can recover $L_{U_j}(C)$ as follow.

$$\begin{aligned}\mathsf{ARMADILLO1c}(C\|U_j) &= T_4\left(L_C(L_{U_j}(C)\oplus\gamma_j)\oplus P(C,0), C\|U_j\right)\\ &= T_4\left(L_C(L_{U_j}(C)), C\|0\right)\oplus T_4\left(L_C(\gamma_j)\oplus P(C,0), 0\|U_j\right)\end{aligned}$$

Therefore, we can compute

$$T_4\left(L_C(L_{U_j}(C)), C\|0\right) = \mathsf{ARMADILLO1c}(C\|U_j)\oplus T_4\left(L_C(\gamma_j)\oplus P(C,0), 0\|U_j\right)$$

Let consider $U_i\neq U_j$, we have

$$\begin{aligned}\Delta(U_i,U_j) =& T_4\left(L_C(L_{U_i}(C)), C\|0\right)\oplus T_4\left(L_C(L_{U_j}(C)), C\|0\right)\\ =& T_4\left(L_C(L_{U_i}(C)\oplus L_{U_j}(C)), 0\right)\\ =& \mathsf{ARMADILLO1c}(C\|U_i)\oplus T_4\left(L_C(\gamma_i)\oplus P(C,0), 0\|U_i\right)\\ &\oplus \mathsf{ARMADILLO1c}(C\|U_j)\oplus T_4\left(L_C(\gamma_j)\oplus P(C,0), 0\|U_j\right)\end{aligned}$$

Hence, we obtain

$$\begin{aligned}L_{U_i}(C)\oplus L_{U_j}(C) = L_C^{-1}(T_4^{-1}(&\mathsf{ARMADILLO1c}(C\|U_i)\oplus T_4\left(L_C(\gamma_i)\oplus P(C,0), 0\|U_i\right)\\ &\oplus \mathsf{ARMADILLO1c}(C\|U_j)\oplus T_4\left(L_C(\gamma_j)\oplus P(C,0), 0\|U_j\right), 0))\end{aligned}$$

Since $L_{U_i}(C)$ is a known transformation linear in C we can recover the secret key by solving the set of linear equations.

6 Attack Extension with a Fixed Transposition in T_1 (ARMADILLO1d)

6.1 Case with no General T_2 and T_4

The previous attack can be prevented by using an S-box layer. However, if the underlying permutation P is not predictable then we can not apply the aforementioned attack. We used $P(U, C\|U)$ to generate linear equations, and then guess the mapping $P(C,Y)$. So, an attempt is to mix bits of C and U. But then, we show that even though we do not know the secret parts of permutation since these parts are fixed, we can guess them one by one. We design a version called ARMADILLO1d that first applies a fixed permutation π on the first register, i.e., T_1 is a transposition. Then, we show that setting T_1 to be a transposition does not prevent a forgery attack. ARMADILLO1d is defined as follows.

$$\mathsf{ARMADILLO1d}(C\|U) = T_4(P((C\|U)_\pi, T_2(C\|U)), (C\|U))$$

where

$$\begin{aligned} P(s\|b,Y) &= P(s, Y_{\sigma_b} \oplus \gamma) \\ P(\lambda, Y) &= Y \end{aligned}$$

We first consider a simple case for T_1 when bits of challenge U form an interval (see Fig. 6), and T_2 is identity and $T_4(X,Y) = X \oplus Y$. Later, we extend the attack to a general transposition T_1, T_2 and T_4.

$$\mathsf{ARMADILLO1d}(C_1\|C_2\|U) = P(C_1\|U\|C_2, C_1\|C_2\|U) \oplus (C_1\|C_2\|U)$$

Let denote $X = (C_1\|C_2\|U)$. We have

$$\begin{aligned} \mathsf{ARMADILLO1d}(C_1\|C_2\|U) &= P(C_1\|U\|C_2, X) \oplus X \\ &= P(C_1\|U, P(C_2, X)) \oplus X \\ &= P(C_1, P(U, P(C_2, X))) \oplus X \end{aligned}$$

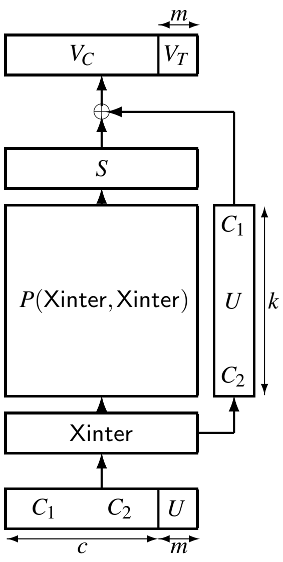


Fig. 6. The compression function of ARMADILLO1d

Concentrating on an arbitrary output bit n and using Lemma 2, we obtain

$$\begin{aligned}
\mathsf{ARMADILLO1d}(C_1\|C_2\|U)[n] &= P(C_1,P(U,P(C_2,C_1\|C_2\|U)))[n] \\
&\quad \oplus(C_1\|C_2\|U)[n] \\
&\overset{t=\sigma_{C_1}^{-1}(n)}{=} P(C_1,0)[n]\oplus P(U,P(C_2,C_1\|C_2\|U))[t] \\
&\quad \oplus(C_1\|C_2\|U)[n] \\
&\overset{l=\sigma_{U}^{-1}(t)}{=} P(C_1,0)[n]\oplus P(U,0)[t]\oplus P(C_2,C_1\|C_2\|U)[l] \\
&\quad \oplus(C_1\|C_2\|U)[n] \\
&\overset{i=\sigma_{C_2}^{-1}(l)}{=} P(C_1,0)[n]\oplus P(U,0)[t]\oplus P(C_2,0)[l] \\
&\quad \oplus(C_1\|C_2\|U)[i]\oplus(C_1\|C_2\|U)[n]
\end{aligned}$$

Re-arranging the above expression, for $a_l = P(C_2,0)[l]$ and $b_n = P(C_1,0)[n]$ we obtain

$$\mathsf{ARMADILLO1d}(C_1\|C_2\|U)[n]\oplus g(U)[t]\oplus X[n]\oplus X[i] = (a_l\oplus b_n) \tag{2}$$

We follow Fig. 7. for the attack scenario. Let $n=\sigma_{C_1}(t)$ and $l=\sigma_{C_2}(i)$, where both permutations are unknown. Set a value q to be determined later. We group $k.q$ distinct challenges as follows: we put the challenge U_j in group $\mathcal{G}_{l\to t}$ if $\sigma_{U_j}(l)=t$. In fact, a challenge appears in k groups. These are groups $\mathcal{G}_{1\to\sigma_{U_j}(1)}$, $\mathcal{G}_{2\to\sigma_{U_j}(2)}$, $\dots$, $\mathcal{G}_{k\to\sigma_{U_j}(k)}$. We obtain k^2 groups with approximately q challenges in each. The right hand side of equation (2) is fixed for all challenges in the same group, since $\sigma_U(l)=t$ for all $U\in\mathcal{G}_{l\to t}$. Hence the left hand side should be fixed for all challenges in the same group as well, since neither a_l not b_n changes. We deploy this property and use the following algorithm to filter out “bad groups” and recover relations which allow us to forge the response. Since C_1 and C_2 are fixed, $\sigma_{C_2}(i)$ and $\sigma_{C_1}^{-1}(n)$ are fixed for (i,n) fixed. So, $\sigma_{C_2}(i)$ for $i\in[1,m]$ can only map to m distinct positions, therefore l can have only m possibilities out of k. The same is true for $\sigma_{C_1}^{-1}(n)$ which can only have m possibilities for t. Intuitively, what we mean by a “bad group” is a group which whether in the mapping $\sigma_{C_2}^{-1}(l)$ maps l out of the corresponding windows of size m or in the mapping $\sigma_{C_1}(t)$ maps t out of the corresponding windows of size m. Term “good groups” is used to recover σ_{C_1}, σ_{C_2}. We now define more precisely what we mean by a “bad group” and how they can be filtered.

Definition 1. *We call a group a “bad group” if the exists no pair $(i,n)\in[1,m]^2$, such that $\sigma_{C_2}(i)=l$ and $\sigma_{C_1}^{-1}(n)=t$.*

Lemma 3. *The group $\mathcal{G}_{l\to t}$ is bad if for every pair $(i,n)\in[1,m]^2$ there exists $U\in\mathcal{G}_{l\to t}$ such that the equation 2 is not satisfied.*

Proof. The equation 2 has to be satisfied for group $\mathcal{G}_{l\to t}$ only if we guess $\sigma_{C_2}(i)=l$ and $\sigma_{C_1}^{-1}(n)=t$ correctly. If $\sigma_{C_2}(i)\neq l$ or $\sigma_{C_1}(t)\neq n$, then given $U\in\mathcal{G}_{l\to t}$ we have $\frac{1}{2}$ probability that the equation 2 would be satisfied even if the group is chosen incorrectly.

Therefore, we drop out a “bad group” $\mathcal{G}_{l\to t}$ if there exists no pair $(i,n)\in[1,m]^2$ such that $(i\to l\to t\to n)$ for which equation (2) is satisfied for all elements (see Fig 9). Following

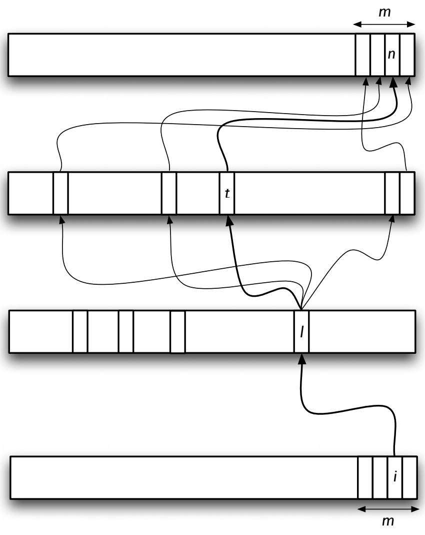


Fig. 7. ARMADILLO1d attack scheme

this step, we also output a correct mapping $(i \to l \to t \to n)$ where $(l,t) \in [1,k]^2$ and $(i,n) \in [1,m]^2$. The probability to accept a given incorrect group is lower than $\frac{k^2}{2^q}$. So, for $k^4 \ll 2^q$ we keep no incorrect group for sure. That is, we need $q \approx 4\log_2 k$.

Now we have m^2 groups left after filtering and at least one mapping $(i_f \to l_f \to t_f \to n_f)$ for a group $\mathcal{G}_{l_f \to t_f}$. These m^2 groups correspond to all groups $\mathcal{G}_{l_i \to e_i}$ for $l_i \in \{\sigma_{C_2}[1],\ldots,\sigma_{C_2}[m]\}$ and $e_i \in \{\sigma_{C_1}^{-1}[1],\ldots,\sigma_{C_1}^{-1}[m]\}$ (see Fig. 8). Since $\sigma_{C_1},\sigma_{C_2}$ are fixed, the correct groups correspond to all mappings between the set of m indices $\{\sigma_{C_2}[1],\ldots,\sigma_{C_2}[m]\}$ and the set of m indices $\{\sigma_{C_1}^{-1}[1],\ldots,\sigma_{C_1}^{-1}[m]\}$.

Now, we fix l to l_f. This way we reduce the number of groups to m. At this stage, we know the exact mapping is $i_f \to l_f$. Depending on which group $\mathcal{G}_{l_f \to t_g}$ we pick at this stage (we have a free choice of t_g), we have m distinct mappings from bit l_f. We pick one of these groups $\mathcal{G}_{l_f \to t_g}$ which maps l_f to t_g. i.e., the mappings on both ends of Fig. 7 are fixed. Then, we go through all m possibilities for n and check for all $U \in \mathcal{G}_{l_f \to t_g}$ whether ARMADILLO1d$(C_1\|C_2\|U)[n] \oplus g(U)[t_g] \oplus X[n] \oplus X[i_f]$ is constant. If yes, we know that t_g maps to n. Using all m groups $\mathcal{G}_{l_f \to \cdot}$, we can recover permutation $\sigma_{C_1}^{-1}$ on $[1,m]$. We can fix t to t_f this time and perform the same procedure to recover σ_{C_2}.

Now we have all we need to forge a response for a new challenge. Let U' be a new challenge. We forge ARMADILLO1d bit by bit. Let consider bit n of the responce R'. We have recovered $\sigma_{C_1}^{-1}[1,m]$ and therefore we know $\sigma_{C_1}^{-1}(n)$ where $n \in [1,m]$ i.e., position to which the nth bit of the response R' is mapped. Now we select l [1] such that $U' \in$

[1] Notice that this l can be in the "bad groups". We only use the "good groups" to recover the secret permutations σ_{C_2} and σ_{C_1}.

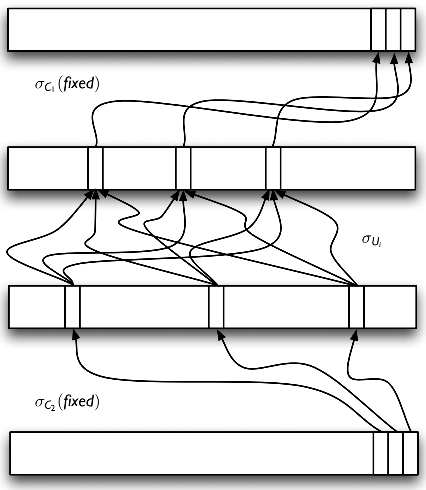


Fig. 8. ARMADILLO1d group filtering scheme

$\mathcal{G}_{l\to\sigma_{C_1}^{-1}(n)}$. If we find such l (i.e., we find the corresponding group) then we forge the $n = \sigma_{C_1}\sigma_U\sigma_{C_2}(i)$-th bit of the response as follows. Let $U \in \mathcal{G}_{l\to\sigma_{C_1}^{-1}(n)}$ be a representative of such a group. From equation (2) we have

$$\begin{aligned}&\mathsf{ARMADILLO1d}(C_1\|C_2\|U)[n] \oplus g(U)[t] \oplus X[n] \oplus X[i] = (a_l \oplus b_n)\\&\mathsf{ARMADILLO1d}(C_1\|C_2\|U')[n] \oplus g(U')[t] \oplus X'[n] \oplus X'[i] = (a_l \oplus b_n)\end{aligned}$$

and therefore

$$\begin{aligned}\mathsf{ARMADILLO1d}(C\|U')[n] = {}&\mathsf{ARMADILLO1d}(C\|U)[n] \oplus (X[i] \oplus X'[i]) \oplus (g(U)[t]\\&\oplus g(U')[t]) \oplus (X[n] \oplus X'[n])\end{aligned}$$

If there is no such l (i.e., we cannot find any corresponding group) we forge the $n = \sigma_{C_1}\sigma_U\sigma_{C_2}(i)$-th bit of response as follows. Compared to the previous case we only drop $(X[i] \oplus X'[i])$, since $X[i] = X'[i]$, since the bit is coming from the secret part.

$$\begin{aligned}\mathsf{ARMADILLO1d}(C\|U')[n] = {}&\mathsf{ARMADILLO1d}(C\|U)[n] \oplus g(U)[t] \oplus g(U')[t]\\&\oplus X[n] \oplus X'[n]\end{aligned}$$

The complexity of the forgery attack is $O(qk^4)$ with qk challenges, where $q \approx 4\log_2 k$.

6.2 Extension with the T_4 and T_2 Transformations

Let now consider the definition for a general case of a T_2 and T_4 and $T_1(C_1\|C_2\|U) = (C_1\|U\|C_2)$, i.e., we assume that π keeps a large piece of consecutive bits of U together.

$$\mathsf{ARMADILLO1d}(C_1\|C_2\|U) = T_4(P(C_1\|U\|C_2, T_2(C_1\|C_2\|U)), (C_1\|C_2\|U))$$

```
for all $\mathcal{G}_{l\to t}$, where $(l,t) \in [1,k]^2$ do
    $\omega = 0$
    for all $(i,n) \in [1,m]^2$ do
        $err = 0$
        pick an arbitrary $U_1 \in \mathcal{G}_{l\to m}$
        $\nu = \mathsf{ARMADILLO1d}(C_1\|C_2\|U_1)[n] \oplus g(U_1)[t] \oplus X[n] \oplus X[i]$
        for all $U \in \mathcal{G}_{l\to m} \backslash U_1$ do
            if $\nu \neq \mathsf{ARMADILLO1d}(C_1\|C_2\|U)[n] \oplus g(U)[t] \oplus X[n] \oplus X[i]$ then
                $\omega = \omega + 1$
                $err = 1$
                break
            end if
        end for
        if $(err == 0)$ then
            output $(i \to l \to t \to n)$.
        end if
    end for
    if $(\omega == m^2)$ then
        drop $\mathcal{G}_{l\to t}$.
    end if
end for
```

Fig. 9. Group filtering algorithm for ARMADILLO1d

The same steps as the previous attacks hold in the general case as well. In the case of T_2, since the secret is fixed it can be derived out as a constant in our computations. So, deploying the same grouping strategy, the attack still works. It is not difficult to see that the same method also holds even if we have the T_4 function.

6.3 Extension to a General π

Let now consider the definition

$$\mathsf{ARMADILLO1d}(X) = P(X_\pi, X) \oplus X$$

for $X = (C_1\|\ldots\|C_t\|U_1\|\ldots\|U_{t-1})$. In the algorithm above, the attacker decomposes the computation of ARMADILLO1d into several stages. Let suppose `wlog` that we permute the bits of secret using permutation π as follows

$$(C_1\|\ldots\|C_t\|U_1\|\ldots\|U_{t-1})_\pi = C_1\|U_1\|C_2\|U_2\|\ldots\|C_{t-1}\|U_{t-1}\|C_t$$

I.e. π mixes C and U bits together, without putting too many consecutive bits of U. We use t times the forgery algorithm on ARMADILLO1d to recover all mappings σ_{C_i}.

Let assume that $2^{|U_i|} \geq kq$ for every i. If this is not the case we can recover the mapping $\sigma_{C_{i-1}}\sigma_{U_i}\sigma_{C_i}$ for all evaluations of U_i in polynomial time. In both cases, we recover each σ_{C_i} and $\sigma_{C_{i+1}}$ recursively. To recover σ_{C_i} and $\sigma_{C_{i+1}}$, we fix $E_{i+1} = U_{i+1}\|C_{i+2}\|\ldots\|C_t$ by fixing $U_{i+1},\ldots,U_{t-1}$, since $C_{i+2},\ldots,C_t$ is already fixed. Then, the problem is reduced to the same situation as the previous attack, where we have a challenge part sandwiched between two intervals of the secret bits C. Finally, we obtain the mapping σ_{C_1} on set $[1,m]$, $\sigma_{C_2},\ldots,\sigma_{C_{t-1}}$ on set $[1,k]$ and $\sigma_{C_t}^{-1}$ on set $[1,m]$. Note that we can recover permutations $\sigma_{C_2},\ldots,\sigma_{C_{t-1}}$ on set $[1,k]$ by setting challenge bits to different values $\frac{k}{m}$ times. All we need is to describe an algorithm to recover all constants $P(C_i,0)$. Then we will have everything we need to forge the response of ARMADILLO1d to any challenge.

We now describe how to recover $P(C_1,0)$. The same method can be used to derive other $P(C_i,0)$ recursively. The same as before, Let fix all U_i's except U_1. We use Lemma 1. and Lemma 2. and rewrite

$$\begin{aligned}\text{ARMADILLO1d}(X)\oplus X &= P(C_1\|U_1\|E_2,X)\\ &= P(E_2,P(U_1,P(C_1,X)))\\ &= P(C_1,X)_{\sigma_{U_1}\sigma_{E_2}}\oplus P(U_1,0)_{\sigma_{E_2}}\oplus P(E_2,0)\\ &= P(C_1,0)_{\sigma_{U_1}\sigma_{E_2}}\oplus X_{\sigma_{U_1}\sigma_{E_2}}\oplus P(U_1,0)_{\sigma_{E_2}}\oplus P(E_2,0)\end{aligned}$$

which gives us set of linear equations and we can vary σ_{U_1} as necessary to obtain a large system of equations and solve it.

Complexity. The general attack may iterate the algorithm in Fig. 9. up to k times. In some iterations, the requirement $2^{|U_i|}\geq k\log k$ does not need to be satisfied. In such case, we need to extend the interval U_i by guessing some bits of key. Such technique would require another $k\log k$ steps. Therefore, the complexity is bounded by $O(k\cdot k^4 q\cdot k\log k)$. We specified before that $q\approx 4\log k$. Hence, the complexity of the offline stage of the attack is $O(k^6\log^2 k)$ and the algorithm requires at most $k^3\log^2 k$ queries.

6.4 Attack Impact on ARMADILLO2

We can see ARMADILLO2 as a successor of ARMADILLO1d with a pre-processing T_1 which is more elaborate than a simple transposition. Such preprocessing makes it resistant against our attack. Our attack is based on decomposition according to Lemma 1 and a guess of a constant value of function $f(U)$ from Lemma 2. The pre-processing phase protects against both the decomposition and the constant value of function $f(U)$. However, the attack we propose points out possible weaknesses in the design of ARMADILLO2.

7 Conclusion

We have shown a devastating key recovery attack against ARMADILLO1 and discussed a potential implication on ARMADILLO2. Although we did not find an attack on ARMADILLO2, we have illustrated that the non-linearity based on data-dependent permutations in both ARMADILLO1 and ARMADILLO2 is not sufficient. The results do not immediately apply on ARMADILLO2 but they allow for better understanding the design and they might be used to improve the attack in [1].

Acknowledgements. This work was sponsored by Oridao [2]. We would like to thank Nicolas Reffé for his kind support. The ARMADILLO algorithms are subject to patent number WO/2008/148784 [4].

[2] http://www.oridao.com/

References

1. Abdelraheem, M., Blondeau, C., Naya-Plasencia, M., Videau, M., Zenner, E.: Cryptanalysis of ARMADILLO2. In: Proceeding of ASIACRYPT 2011. Springer, Heidelberg (2011)
2. Badel, S., Dağtekin, N., Nakahara Jr., J., Ouafi, K., Reffé, N., Sepehrdad, P., Sušil, P., Vaudenay, S.: ARMADILLO: A Multi-purpose Cryptographic Primitive Dedicated to Hardware. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 398–412. Springer, Heidelberg (2010)
3. Kobza, J., Jacobson, S., Vaughan, D.: A Survey of the Coupon Collector's Problem with Random Sample Sizes. Methodology and Computing in Applied Probability 9(4), 573–584 (2007)
4. Reffé, N.: Cryptographic Methods and Devices for the Pseudo-Random Generation of Data Encryption and Cryptographic Hashing of a Message (December 2008)

Implementation and Evaluation of an SCA-Resistant Embedded Processor

Stefan Tillich[1], Mario Kirschbaum[2], and Alexander Szekely[2]

[1] University of Bristol, Computer Science Department, Merchant Venturers Building, Woodland Road, BS8 1UB, Bristol, UK
tillich@cs.bris.ac.uk
[2] Graz University of Technology, Institute for Applied Information Processing and Communications, Inffeldgasse 16a, A–8010 Graz, Austria
{Mario.Kirschbaum,Alexander.Szekely}@iaik.tugraz.at

Abstract. Side-channel analysis (SCA) attacks are a threat for many embedded applications which have a need for security. With embedded processors being at the very heart of such applications, it is desirable to address SCA attacks with countermeasures which "naturally" fit deployment in those processors. This paper describes our work in implementing one such protection concept in an ASIC prototype and our results from a practical evaluation of its security. We are able to demonstrate that the basic principle of limiting the "leaking" portion of the processor works rather well to reduce the side-channel leakage. From this result we can draw valuable conclusions for future embedded processor design. In order to minimize the remaining leakage, the security concept calls for the application of a secure logic style. We used two concrete secure logic styles (iMDPL and DWDDL) in order to demonstrate this increase in security. Unfortunately, neither of these logic styles seems to do a particularly good job as we were still able to attribute SCA leakage to the secure-logic part of the processor. If a better suited logic style can be employed we believe that the overall leakage of the processor can be further reduced. Thus we deem the evaluated security concept as a viable method for protecting embedded processors.

Keywords: Side-channel analysis, SCA countermeasures, embedded processors, iMDPL, DWDDL.

1 Introduction

Two well-known methods for protecting cryptographic workloads on embedded processors against side-channel analysis (SCA) attacks are software countermeasures and cryptographic coprocessors with custom hardware protection. On the one hand, software countermeasures are the most flexible solution but might not address all attacks or bear a significant overhead in terms of running time or resources. On the other hand, coprocessors often only offer a limited range of functionality and inhibit even slight changes of parameters (e.g. new modes of operation). An alternative approach is the incorporation of SCA countermeasures into the processor itself in order to find a trade-off between the flexibility of software countermeasures and the protection of hardware countermeasures.

There have been several proposals for adapting processors directly in order to increase their SCA resistance. Randomized execution of programs is the basic concept in non-deterministic processors as proposed by May *et al.* [11]. Regazzoni *et al.* [16] built an automated design flow for integrating custom functional units (FUs) implemented in a secure logic style into the processor. The MUTE-AES architecture proposed by Ambrose *et al.* [1] uses a second processor to balance out the AES operations of the first with the calculation of the same operations on inverted data. The balancing of the second processor is activated and terminated automatically by detecting certain sequences of instructions and it can be used for other tasks when no cryptographic operations are performed on the first processor. Recently, additional works regarding SCA protection of processors have been published. Nakatsu *et al.* [12] have investigated a processor modification in combination with software countermeasures for eliminating SCA vulnerabilities due to branches, addresses and intermediate values dependent on secret information. They propose to implement the ALU in a specific secure logic style (Random Switching Logic) and use masking of operands and results. Although the authors emphasize that their proposal can be applied to various operations and processor architectures, several issues regarding mask handling remain open. Barthe *et al.* [2] proposed to duplicate the processor data path by adding an additional register file and pipeline registers. Data in the original data path is masked while the second data path carries the corresponding masks. Before entering the FUs, data is unmasked and the according result is freshly masked. Data in memory is protected with a "static" mask.

Protecting a processor from SCA with a combination of masking and the application of secure logic styles has first been proposed by Tillich *et al.* in [19]. The original proposal was targeted at securing a specific set of cryptographic instructions. Based on the original idea, a full security concept covering a wide range of instructions and addressing issues of hardware limitations and secure task switching was then presented in [20]. As this work was conducted in context of the so-called Power-Trust project, we will refer to this security concept as the *Power-Trust security concept* in the remainder of this paper. The work of [20] also produced an FPGA prototype to validate the functionality and a preliminary SCA evaluation to demonstrate the principal protection offered by the concept (although the use of FPGA-style secure logic had been omitted).

In the present paper we present an ASIC implementation of the Power-Trust security concept based on the SPARC V8 Leon3 embedded processor. Our prototype includes several optional features proposed in [20]. For the critical portion of the processor (the so-called *secure zone*) we have used two different secure logic styles. Additionally, we have included a CMOS variant of the secure zone for reference. The goal of our work was threefold:

– Demonstrate the functionality and practicality of the concept.
– Test the utility of the optional features and develop possible improvements both regarding usage in software and simplification in hardware.
– Evaluate the SCA resistance of the prototype in order to estimate the potential protection offered by the Power-Trust security concept.

Although we have observed huge improvements in DPA attack resistance in our practical evaluation, the use of our chosen secure logic styles has turned out to be rather ineffective (leading only to little improvements in resistance at relatively high cost). Nevertheless, we deem our results as valuable insights into the issues of practical deployment of secure logic styles.

The rest of this paper is organized as follows. The Power-Trust security concept is described in Section 2. Our ASIC prototype implementation including a description of features of the secure zone as well as details of the implementation of the logic styles on ASIC is presented in Section 3. Section 4 discusses possible improvements of the whole security concept. A detailed security evaluation and the presentation of our SCA attack results is given in Section 5, followed by analyses of the results in Section 6. Conclusions are drawn in Section 7.

2 Description of the Power-Trust Security Concept

SCA attacks exploit the effect specific key-dependent data has on various observable physical characteristics of a device, e.g. timing or power consumption. We will denote such data as *critical data* in the following. The basic assumption for the Power-Trust security concept is that each handling of critical data is a potential target for an attacker. The general term handling hereby refers to both operations on the critical data as well as the movement of it through various data flow control elements (e.g. multiplexers) and storage units. For example, in a typical modern processor critical data are not only operated upon in a number of functional units but also have to pass through various pipeline stages, feedback paths, and storage elements like pipeline registers, register files, caches and memories. All these activities are reflected in the processor's power consumption and can be ultimately exploited in a power analysis attack.

The first step for counteracting SCA attacks is to minimize the effect of the critical data on the physical characteristics. Leakage in the timing of the processor can be addressed by eliminating key-dependent branches and avoiding other potentially variable-time operations like table lookups. The use of special cryptographic instructions (instruction set extensions) is very helpful in achieving this goal without sacrificing performance. Also, elimination of table lookups is expected to go a long way in reducing leakage in the power side channel, as memory operations are usually a significant contributor to the overall power consumption.

With the help of cryptographic instruction set extensions (ISEs) it is possible to confine actual operations on critical data exclusively to the FUs of the processor, which are usually the main part of its execute stage. Any other part of the processor then just moves around critical data without transforming it.

Based on this observation, the Power-Trust security concept applies a mask to all critical data which circulates outside of the FUs. The operands entering a FU have to be unmasked and the results leaving the FU have to be masked again. Therefore, arbitrary FUs can be supported easily. This is unlike most of the previous masking solutions, which require the development of special FUs to implement non-linear operations which transform masked values and mask.

Thus, direct leakage only emanates from the FUs which are required for implementing the cryptographic algorithm. Leakage from components which handle the masks (mask storage and mask generator) could also lead to higher-order weaknesses when combined with leakage from the masked values. In order to mitigate these vulnerabilities, all these components are implemented in a secure logic style. This portion of the processor is denoted as *secure zone*. From its functionality and interface, the secure zone is very similar to a conventional FU, which facilitates its implementation in secure logic and its integration into the processor. Conceptually, an attacker must overcome the protection of the secure logic style[1] in order to break the security of the processor.

The secure zone can only store a fixed number of masks, which determines the maximum size of the intermediate state of a cryptographic algorithm at any point in time. However, by using pseudo-randomly generated masks it is possible to store masks intermittently outside of the secure zone in a secure redundant representation. This mechanism allows to virtually extend the number of available masks and also to share the secure zone between several processes in a secure fashion. Other issues addressed in [20] are the way that masks and masked values are associated in the processor and the options for dealing with exceptional cases during runtime, e.g. when the maximum number of possible masks is reached. Those issues afford various solutions and we describe our concrete design choices in Section 3.

3 ASIC Prototype Implementation

Our prototype is based on the SPARC V8 Leon3 processor [4], which is a popular platform for academic research due to its high quality and tool support as well as its openness. It is a 32-bit embedded architecture with a large number of configuration options. Our most important enhancements are described in the following.

3.1 Features of the Secure Zone

When masked operands enter the secure zone, the processor needs a mechanism to identify the corresponding mask. Various options are possible but in our implementation we chose to introduce a custom addressing mechanism which is under explicit software control. Our reason for this choice was the huge degree of

[1] Either by attacking the unmasked critical data in the secure zone or with a higher-order attack on the masked values and their masks in the secure zone.

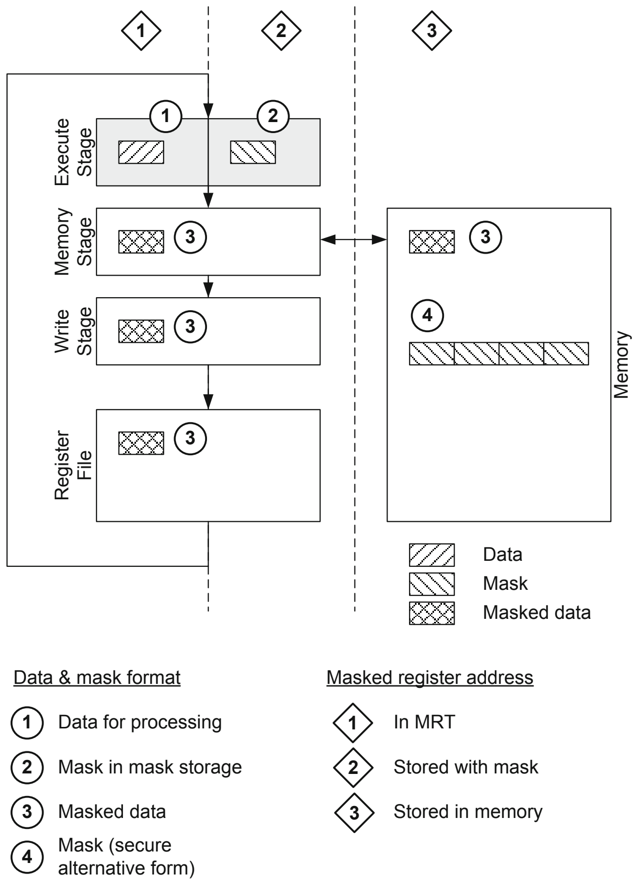


Fig. 1. Data representation and addressing in enhanced processor

flexibility offered by this solution. Each mask and each masked value is associated with a custom 10-bit address, so that the mask and masked value with the same address belong together. As this address acts much like a register address, it is denoted as *masked register address*. Depending on the storage location of mask or masked value, the adjoint masked register address is handled differently.

Figure 1 depicts the different possible data representations of masked data and masks in various storage locations as well as the mechanisms used to keep the masked register address alongside them. Unmasked data only occurs in the secure zone's FUs (circled 1) and masks reside exclusively in the secure zone's mask storage unit (circled 2), where both are protected by the secure logic style (indicated by the gray shading). Masked data can circulate through the other pipeline stages, the register file, and memory (circled 3). Additionally, masks can be held in their secure redundant form in memory (circled 4). Masks and masked data must always be attached to their masked register address, so that the secure zone can associate masked data with its corresponding mask. Masked data in the pipeline stages and register file (diamond 1) are principally attached to a logical register address. A custom hardware table called *Masked Register Table (MRT)* then maps this logical register address to a masked register address. The masks in the mask storage (diamond 2) are held automatically alongside their masked register address. On the other hand, masked data and masks in memory (diamond 3) must have their masked register address stored explicitly with them by the software.

The advantage of this masked register addressing scheme is that it can be used to emulate several different forms of software usage of the secure zone. For example, the software can choose to never write masks or masked values out to memory. In this case, the MRT can be initialized with a fixed mapping and the logical register address of a masked value effectively determines the corresponding mask.

The instruction result coming from the secure zone must be protected by a fresh 32-bit mask. When the processor handles a sequence of such protected instructions, a fresh mask is needed in every clock cycle. Furthermore, masks must be uniformly distributed and unpredictable by an attacker. For dealing with more masks than the mask storage of the secure zone can handle internally, it is also necessary to have a secure representation of masks which allows their subsequent reconstruction.

In order to balance all these (partly conflicting) requirements with an efficient design we have implemented our mask generator as a 127-bit maximum-length LFSR. We used the pentanomial $x^{127} + x^{87} + x^{59} + x^{37} + 1$ as the reduction polynomial because it facilitates parallelization in both forward and reverse direction. In normal operation, the LFSR advances 32 steps per cycle in order to produce a fresh 32-bit mask.

The mask generator also keeps track of the number of steps (denoted as *mask index*) it has taken from a specific initial state. In conjunction with a particular LFSR state, the mask index is a secure alternative representation of a mask. Software can read and write the LFSR state and can run the LFSR in forward or reverse direction towards a specific mask index in order to restore a mask. In a finished device, the LFSR should be seeded from a random number generator (RNG) to make the produced masks unpredictable and to prevent reset attacks. For our prototype, we did not integrate an RNG.

The secure zone can hold up to eight masks in its mask storage component. Metadata about stored mask like masked register address and mask index can be read by software. Exceptional conditions in the secure zone cause traps. Such exceptional conditions are a full mask storage, encounter of a masked operand whose mask is missing from the mask storage, and an overflow of the mask index counter in the mask generator. It is up to the software to either avoid those exceptional conditions altogether or to address them at runtime with specific trap handlers. Protected instructions of the secure zone encompass specific support for AES and ECC over GF(2^m) as well as a range of logic operations for bit-sliced implementations.

3.2 Implementation Details

We have used the UMC 0.18 µm standard-cell library FSA0A_C from Faraday [3] for implementation. Our original plan was to employ memory macros from this library to implement memory components like the caches. However, despite our best efforts we were unable to integrate those memory macros into our design flow within a reasonable time frame. Therefore, we used the synthesizer to infer those memory structures with flip-flops. As such inferred structures are significantly larger compared to macros, we had to reduce the size of instruction and data cache to the minimum of 1 KB each.

We have integrated secure zone blocks in three different logic styles in our prototype. We chose logic styles which are based on standard cell libraries and thus can be relatively easily implemented in a more or less automatic way but which still promised to deliver an increase in protection over CMOS. To this end we chose iMDPL [14] and DWDDL [23]. The third secure zone was implemented in CMOS to serve as a point of reference. Software can select a specific secure zone by writing to a custom configuration register. The inputs (including the clock signal) of the other two secure zones are held at zero in order to minimize their impact on the power profile.

The improved masked dual-rail precharge logic (iMDPL) style combines the dual-rail precharge (DRP) technique, a masking technique, and it takes precautions to prevent the effect of early propagation ([18]). The DRP technique avoids the occurrence of glitches ([15]) which may significantly reduce the side-channel resistance of implementations ([9] and [10]). This is achieved by introducing an evaluation and a precharge phase in each clock cycle. In the precharge phase, both wire rails of each signal are precharged. In the evaluation phase the wires switch to their corresponding values. The masking technique bypasses the need for special routing techniques of the circuitry. In [14] it has been shown that early propagation may cause a significant side-channel leakage in a masked DRP logic style. Thus, each iMDPL gate contains an evaluation-precharge detection unit (EPDU) which tries to prevent the premature evaluation/precharge of the gates.

A basic iMDPL AND gate is built with two three-input majority (MAJ) gates. MAJ gates are part of standard cell libraries, so there is no need to design and verify special security cells in the library. The iMDPL style can thus be implemented by a more or less straight forward search and replace processing step after design synthesis. Each iMDPL gate processes the masked values a_m, b_m $(\bar{a}_m, \bar{b}_m)$ and the mask value m $(\bar{m})$. IMDPL is based on boolean masking, *i.e.* $a_m = a \oplus m$, and the mask value m is derived from a 64-bit maximum-length LFSR counter [13] in each clock cycle.

The area of a pure iMDPL implementation is usually increased by a factor of approximately 20. In our case, the area of the iMDPL secure zone is 406 kGE, compared to 19 kGE of the secure zone implemented in unprotected CMOS logic, which corresponds to a factor of 21. The whole Leon3 processor without any secure zones has 582 kGE[2]. This results in a total area overhead of approximately 64% compared to a pure CMOS implementation of the SPARCV8 embedded processor.

We implemented iMDPL with knowledge that the security of the logic style might suffer from unbalanced wires ([17], [21]). Recent research has shown that the mask value in an iMDPL circuit can most probably be discovered and that also other wires within the circuitry might be affected by imbalances. Nevertheless, investigations of an ASIC implemented in iMDPL have shown that the logic style is able to increase the SCA resistance ([5]).

Double Wave Dynamic Differential Logic (DWDDL) [23] is an enhancement of WDDL [22] with the goal of solving its inherent balancing issues. Whereas WDDL requires techniques of balanced routing of its differential signals, DWDDL duplicates the layout of a normally routed WDDL circuit and inverts both the block's input as well as the inputs and outputs of all logic cells[3]. Thus, for each differential signal, a bit flip at one of the wires in the first WDDL block is counterbalanced by a bit flip on the other wire of this differential signal in the second WDDL block.

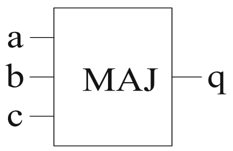
The area of the DWDDL secure zone is 260 kGE, which represents a factor of 14 compared to the secure zone implemented in unprotected CMOS logic. With respect to the whole Leon3 processor, the total area overhead is approximately 40%, which represents a relatively moderate increase in area in return for a significantly increased SCA resistance.

DWDDL has originally been proposed exclusively for use in FPGAs, as they allow to invert the functionality of logic cells simply by changing the contents of their lookup tables (LUTs). In case of ASIC implementations a swapping of cell functionality (*i.e.* substituting an AND cell with an OR cell) would result in significant changes in the circuitry due to differences in the structure of the cells. This would implicate differences between the two WDDL circuits and would hence introduce imbalances in the power consumption of the two circuits. We have developed a simple way to overcome this issues and to adopt DWDDL for ASIC implementations. Our approach is based on the observation that three-input MAJ gates can be seen as configurable two-input AND/OR gates (inputs a and b), where the value of the third MAJ input c determines the functionality, see Figure 2.

We started with the design synthesis utilizing conventional AND and OR gates. Afterwards we implemented WDDL based on MAJ gates by means of a search and replace processing step in which the standard AND and OR cells are replaced by corresponding WDDL cells (based on MAJ cells) and the whole netlist is transformed to a dual-rail circuitry. The third input of each MAJ gate

[2] The area of 582 kGE of the Leon3 processor includes everything except the three secure zones. Note the fact that we implemented the chip using inferred memory structures instead of smaller memory macros. Subtracting the inferred memories (488 kGE) and some conventional AES ISEs (6 kGE), the size of the bare processor is about 88 kGE.

[3] In practice, this means that AND gates become OR gates and vice versa.



MAJ as AND				MAJ as OR			
a	**b**	**c**	**q**	**a**	**b**	**c**	**q**
0	0	0	0	0	0	1	0
0	1	0	0	0	1	1	1
1	0	0	0	1	0	1	1
1	1	0	1	1	1	1	1

Fig. 2. Using a MAJ gate as configurable AND/OR gate. The gate functionality can be controlled by input c of the MAJ gate.

is hooked up to a global configuration signal. This WDDL netlist is then placed and routed. The resulting layout is duplicated and the second instance of the WDDL block is hooked up to the inverted inputs and the inverted configuration signal. This way we achieved two identically placed and routed WDDL circuits with a globally configurable cell functionality. We are aware that such a separation of the two WDDL circuits on the chip opens the door for localized EM measurements and thus represents a potential vulnerability.

A critical point for the SCA resistance of the DWDDL implementation is the timing of the two WDDL circuits. The propagation of the evaluation and precharge waves through the circuits has to start as exact as possible in both circuits, deviations would result in early propagation which would significantly affect the SCA resistance. Thus, we implemented the transformation cells CMOStoWDDL in a way that the clock signal acts as a trigger for the precharge/evaluation phases. Internally, the WDDL circuits are operated with a doubled clock speed due to the master-slave flip-flop structure implemented [22]. So we have two clock domains in the whole design: (1) the standard clock signal for all CMOS circuits and the iMDPL secure zone, which also serves as a precharge/evaluation signal and (2) a double-speed clock signal for the two WDDL circuits. We also had to take precautions when implementing the transformation cells for leaving the dual-rail circuit. Similar to inputs in the CMOStoWDDL cells, in case of WDDLtoCMOS the sensitive outputs of both WDDL circuits have to be synchronized to prevent a data leakage. Thus, we added a register stage to the output transformation which is synchronized by the clock signal.

Known weakness in DWDDL: Before the implementation of DWDDL we performed a theoretical investigation. Unfortunately, at this point we made a mistake and evaluated an incorrectly designed DWDDL cell: we correctly inverted the input signals of the complementary WDDL cell but we missed to exchange the logic gates AND and OR of the complementary WDDL cell. Our mistake disguised the occurrence of early propagation in DWDDL and resulted in an incorrect behavior of the complementary WDDL circuit. We recognized the mistake only shortly before the tape-out date, where we still had the time to correctly exchange the logic gates but we did not have enough time to switch to another logic style or to implement and improved version of DWDDL.

4 Possible Improvements

Experience with software development using the various features of the prototype has led to a number of possible improvements. Most of the features are intended to support manual software development for the enhanced processor. However, we expect that software generation should ultimately be pushed off to the toolchain in order to minimize the risk of programming errors. In such a case, many of the extra features of the secure zone could be simplified or eliminated in order to reduce its size and cost.

The biggest issue is the use of masked register addresses to associate masked data and masks. This leads to a high degree of flexibility for software which allows to move masked data freely between memory and any register. However, as allocation of masked data is expected to be ultimately handled by the toolchain, this flexibility does not seem to be required. Masked register addresses require elaborate address lookup logic and the inclusion of the MRT. By using the already present logical register address as mechanism to associate masked data and masks, the address lookup logic can be greatly simplified and the MRT can be eliminated altogether. This implies that masked data must be loaded back into the same logical register from which it has originally been stored to memory. However, we do not expect this to be a problem for the software.

The hardware traps are mainly intended for a very flexible but also rather slow mode of execution. This mode is intended for software developers who have little knowledge about the characteristics of the secure zone but still want to write code manually. By removing this mode of execution, the hardware trap mechanism can be simplified or even eliminated altogether.

Mask indices are maintained by the mask generator (*i.e.* its current "step" count) and the mask storage (*i.e.* the "step" count of each mask) with the sole purpose of having a secure redundant representation of masks. If the mask storage of the secure zone is deemed large enough to support all required cryptographic algorithms, these mask indexing mechanisms could be removed in order to reduce hardware overhead.

A more minor issue are the management instructions to read out metadata from the mask storage. In our prototype, it is not possible to get the metadata corresponding to a particular mask directly, which makes it necessary to read out all metadata and look for the desired entry. This problem could be easily remedied by a slight modification of the functionality of the management instructions.

5 Security Evaluation

We have compared various implementations of a single round of AES by means of classical correlation-based DPA attacks using Hamming weight (HW) and Hamming distance (HD) as power models. A standard software AES using only native SPARC V8 instructions and incorporating an S-box lookup table serves as our baseline version (SW-AES). Then we have the use of basic AES ISEs, which eliminate the need for table lookups (AESREF). Furthermore, we implemented AES on each of the three secure zones (SZ_CMOS, SZ_IMPDL, SZ_DWDDL). We defined a DPA attack as successful if at least 9 of the 16 key bytes could have been revealed. We assumed that the remaining 7 key bytes can be revealed by a brute force attack within reasonable time. Our approach is based on common brute-force attacks on the 7-byte key of DES ([6] and [7]). Hence, for our estimations of the number of required power traces we used the ninth-highest correlation value of all correctly revealed key bytes.

In case of SW-AES we performed 10 000 measurements. According to our attack success metrics and according to the rule-of-thumb formula from [8] (Chapter 6.4.1) the number of required power traces to distinguish the correct key hypothesis from the incorrect ones is 263 (key byte 9 has the ninth-highest correlation value of 0.315). The SW-AES implemented in unprotected CMOS is highly vulnerable as expected. Due to the involvement of the MixColumns operation in the SW implementation, attacks using the HD power model would have resulted in rather expensive attacks based on 4 key bytes at once, and hence, these attacks have been omitted. Our results show that the AESREF implementation offers a very slightly increased protection compared to SW-AES. The number of required traces of the ninth-highest correlation value (0.0319) in case of HW is 27 200. The attacks using a HD power model resulted in significantly higher correlation values around 0.2, which corresponds to 700 required power traces.

We received very interesting results from the SZ_CMOS: it turned out that the secure zone implementation already has a significant effect on the SCA resistance, even without implementing the secure zone itself in a special logic style. According to our attack success metrics the number of required power traces in case of SZ_CMOS is around 130 000. The results of the attacks on the SZ_IMDPL show a further increase in SCA resistance: around 260 000 power traces are required to distinguish the ninth-highest correlation value. In case of SZ_DWDDL it turned out that the lower 16 bits of the four 32-bit words processed can not be clearly distinguished in a HW attack. As only eight key bytes could be revealed, we used the eighth-highest correlation value of 0.0064 in this case, which corresponds to 675 000 required power traces. By means of an HD attack on SZ_DWDDL, all 16 key bytes could have been revealed: the highest correlation value corresponds to a number of required power traces of approximately 5 200 000.

The results lead to the following conclusions: it shows that the confinement of leaking operations to a small part of the implemented processor significantly enhances the SCA resistance, even though the confined part is not implemented in a secure logic style. The results also show that the implemented logic styles still have an information leakage, therefore we can not conclusively show that the overall security will increase further with a reliably secure logic style. Nevertheless, the assumption that the elimination of the leakage in the confined part of the processor would also have a positive effect on the overall security still seems plausible.

6 Analysis of the Results

The results of the DPA attacks using Hamming weight power model are summarized in Table 1, the results using Hamming distance power model are summarized in Table 2.

The AES ISEs in the FU of the AESREF implementation are identical to the ISEs in the FU of the SZ_CMOS, *i.e.* the only difference between these two implementations is the masking of the result values leaving the SZ_CMOS. Hence, we assume that the significant HD leakage of AESREF is related to storing the unmasked results in the register file of the processor. In other words, the HD leakage of the SZ_CMOS is prevented due to the masking of the output operands.

The results of the SZ_IMDPL support the assumptions that routing imbalances within the circuitry limit the effectiveness of the logic style to a certain degree. Imbalances between complementary wires in a DRP circuit obviously cause signal-dependent differences in terms of power consumption as well as in terms of signal timings. In a conventional digital design which contains considerably large combinational structures such imbalances may influence the overall power consumption to a certain degree, which results in a data leakage.

Unfortunately, at this stage there are several questions unanswered which are marked for future work. During our evaluations it turned out that we are not able to fully resolve the open questions solely by means of measurements. It further turned out that we cannot perform logic simulations of the whole Leon3 processor within a reasonable time. Hence, we plan to perform simpler simulations of individual submodules in order to be able to investigate the processes within the Leon3 processor and the secure zones in more detail.

The proposal of Barthe *et al.* [2] is similar to the Power-Trust security concept: it tries to confine the leaking operations to a specific portion of the processor. However, protection of memory appears relatively weak as a constant mask added to all words is in principle not expected to deliver good protection against DPA attacks. The authors have performed practical evaluation of an FPGA implementation of an enhanced MicroBlaze clone. They used EM measurements at a number of points of the device. However, analysis of the acquired traces seems to have been done with classical difference-of-means DPA, which makes it hard to accurately estimate the minimal required number of traces for a successful attack from the resulting difference-of-means trace. Barthe *et al.* concluded that their proposal significantly reduces leakage from most parts of the pipeline but that leakage from the unprotected combinational logic in the FUs remains.

7 Conclusions

In this paper we presented an ASIC implementation of the Power-Trust security concept. Our prototype is based on the SPARC V8 Leon3 processor and has been produced in a 0.18 µm process technology from UMC. We demonstrated the functionality and practicality of the security concept and we evaluated the SCA resistance of the ASIC prototype.

The results have shown that the use of basic AES ISEs, which eliminate the need for table lookups, already has a positive effect on the SCA resistance of an implementation. A further improvement of the SCA resistance has been achieved by the secure zone concept where critical operations are confined to a small part of the processor and all data outside of the secure zone is strictly masked. Even if the secure zone itself is not implemented in a secure logic style (as it is proposed by the Power-Trust security concept), the SCA resistance is significantly increased. A further increment in security is provided if the secure zone is implemented in a special logic style. Unfortunately, neither iMDPL nor DWDDL provide a very high level of SCA resistance, but our results indicate that the efficiency of the Power-Trust security concept could be further increased when implementing the secure zone in a well-performing logic style. Furthermore, restricting the implementation of a special logic style only to a small part of a processor tremendously reduces the overhead in terms of area and power which is associated with most of such logic styles.

Acknowledgements. The research described in this paper has been supported by the Austrian Science Fund (FWF) under grant number P22241-N23 ("Investigation of Implementation Attacks"), by EPSRC grant EP/H001689/1, and, in part by the European Commission through the ICT Programme under contract ICT-2007-216676 ECRYPT II. The information in this document reflects only the author's views, is provided as is, and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

1. Ambrose, J.A., Parameswaran, S., Ignjatovic, A.: MUTE-AES: A Multiprocessor Architecture to prevent Power Analysis based Side Channel Attack of the AES Algorithm. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD) 2008, pp. 678–684. IEEE (2008)
2. Barthe, L., Benoit, P., Torres, L.: Investigation of a Masking Countermeasure against Side-Channel Attacks for RISC-based Processor Architectures. In: 2010 International Conference on Field Programmable Logic and Applications, pp. 139–144. IEEE Computer Society (2010)
3. Faraday Technology Corporation. Faraday FSA0A_C 0.18 μm ASIC Standard Cell Library (2004), Details, `http://www.faraday-tech.com`
4. Gaisler Research. GRLIB IP Library User's Manual. Version 1.1.0 B4100 (October 2010) `http://www.gaisler.com/products/grlib/grlib.pdf`
5. Kirschbaum, M., Popp, T.: Evaluation of a DPA-Resistant Prototype Chip. In: 25th Annual Computer Security Applications Conference (ACSAC 2009), Honolulu, Hawaii, USA, December 7-11 (2009)
6. Kumar, S.S., Paar, C., Pelzl, J., Pfeiffer, G., Rupp, A., Schimmler, M.: How to Break DES for 8,980. In: Workshop on Special-purpose Hardware for Attacking Cryptographic Systems - SHARCS 2006, Cologne, Germany, April 3-4 (2006)
7. Kumar, S., Paar, C., Pelzl, J., Pfeiffer, G., Schimmler, M.: Breaking Ciphers with COPACOBANA –A Cost-Optimized Parallel Code Breaker. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 101–118. Springer, Heidelberg (2006)

8. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks – Revealing the Secrets of Smart Cards. Springer, Heidelberg (2007) ISBN 978-0-387-30857-9
9. Mangard, S., Popp, T., Gammel, B.M.: Side-Channel Leakage of Masked CMOS Gates. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 351–365. Springer, Heidelberg (2005)
10. Mangard, S., Pramstaller, N., Oswald, E.: Successfully Attacking Masked AES Hardware Implementations. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 157–171. Springer, Heidelberg (2005)
11. May, D., Muller, H.L., Smart, N.P.: Non-deterministic Processors. In: Varadharajan, V., Mu, Y. (eds.) ACISP 2001. LNCS, vol. 2119, pp. 115–129. Springer, Heidelberg (2001)
12. Nakatsu, D., Li, Y., Sakiyama, K., Ohta, K.: Combination of SW Countermeasure and CPU Modification on FPGA against Power Analysis. In: Chung, Y., Yung, M. (eds.) WISA 2010. LNCS, vol. 6513, pp. 258–272. Springer, Heidelberg (2011)
13. Alfke, P.: Xilinx Application note on Shift Registers and LFSR counters (July 1996), http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf
14. Popp, T., Kirschbaum, M., Zefferer, T., Mangard, S.: Evaluation of the Masked Logic Style MDPL on a Prototype Chip. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 81–94. Springer, Heidelberg (2007)
15. Popp, T., Mangard, S.: Masked Dual-Rail Pre-charge Logic: DPA-Resistance Without Routing Constraints. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 172–186. Springer, Heidelberg (2005)
16. Regazzoni, F., Cevrero, A., Standaert, F.-X., Badel, S., Kluter, T., Brisk, P., Leblebici, Y., Ienne, P.: A Design Flow and Evaluation Framework for DPA-Resistant Instruction Set Extensions. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 205–219. Springer, Heidelberg (2009) ISBN 978-3-642-04137-2
17. Schaumont, P., Tiri, K.: Masking and Dual-Rail Logic Don't Add Up. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 95–106. Springer, Heidelberg (2007)
18. Suzuki, D., Saeki, M.: Security Evaluation of DPA Countermeasures Using Dual-Rail Pre-charge Logic Style. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 255–269. Springer, Heidelberg (2006)
19. Tillich, S., Großschädl, J.: Power Analysis Resistant AES Implementation with Instruction Set Extensions. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 303–319. Springer, Heidelberg (2007)
20. Tillich, S., Kirschbaum, M., Szekely, A.: SCA-Resistant Embedded Processors—The Next Generation. In: 26th Annual Computer Security Applications Conference (ACSAC 2010), Austin, Texas, USA, December 6-10, pp. 211–220. ACM (2010)
21. Tiri, K., Schaumont, P.: Changing the Odds Against Masked Logic. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 134–146. Springer, Heidelberg (2007), http://rijndael.ece.vt.edu/schaum/papers/2006sac.pdf
22. Tiri, K., Verbauwhede, I.: A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In: 2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), Paris, France, February 16-20, vol. 1, pp. 246–251. IEEE Computer Society (2004) ISBN 0-7695-2085-5
23. Yu, P., Schaumont, P.: Secure FPGA circuits using controlled placement and routing. In: Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, Salzburg, Austria, September 30 - October 5, pp. 45–50. ACM Press (2007) ISBN 978-1-59593-824-4

A Analysis Results

Table 1. Correlation analysis results for the measurement series (Hamming weight model); correlation values marked with an asterisk indicate unsuccessful attacks; correlation values in bold represent the ninth-highest value (if at least nine key bytes could have been discovered)

Target	SW-AES	AESREF	SZ_CMOS	SZ_IMPDL	SZ_DWDDL
Byte 1	0.227	0.0164*	0.0111	0.0066	0.0028*
Byte 2	0.298	0.162	<0.003*	0.0177	0.0025*
Byte 3	0.574	0.0244	0.0264	**0.0103**	0.0121
Byte 4	0.135	0.177	0.0177	0.0289	0.0243
Byte 5	0.230	0.0334	**0.0146**	0.0067	0.0024*
Byte 6	0.697	0.0162*	0.0074	0.0195	0.0018*
Byte 7	0.672	0.0878	0.0261	0.0080	**0.0064**
Byte 8	0.178	0.0294	0.0213	0.0289	0.0120
Byte 9	**0.315**	0.0222	0.0085	0.0059	0.0026*
Byte 10	0.564	0.0355	0.0067	0.0166	0.0018*
Byte 11	0.511	0.0240	0.0228	0.0078	0.0091
Byte 12	0.556	0.1296	0.0192	0.0268	0.0114
Byte 13	0.409	0.145	0.0050	0.0062	0.0028*
Byte 14	0.294	0.0201	0.0045*	0.0181	0.0019*
Byte 15	0.452	0.168	0.0237	0.0084	0.0155
Byte 16	0.113	**0.0319**	0.0240	0.0279	0.0098

Table 2. Correlation analysis results for the measurement series (Hamming distance model); correlation values marked with an asterisk indicate unsuccessful attacks

Target	AESREF	SZ_CMOS	SZ_IMPDL	SZ_DWDDL
Byte 1 → 5	0.184	<0.003*	<0.002*	0.0019
Byte 3 → 7	0.211	<0.003*	<0.002*	0.0019
Byte 5 → 9	0.184	<0.003*	<0.002*	0.0020
Byte 6 → 10	0.200	<0.003*	<0.002*	0.0023
Byte 7 → 11	0.212	<0.003*	<0.002*	0.0021
Byte 8 → 12	0.221	<0.003*	<0.002*	0.0023
Byte 9 → 13	0.184	<0.003*	<0.002*	0.0017
Byte 10 → 14	0.201	<0.003*	<0.002*	0.0019
Byte 11 → 15	0.214	0.0314	<0.002*	0.0021
Byte 12 → 16	0.221	<0.003*	<0.002*	0.0021
Byte 14 → 2	0.201	<0.003*	<0.002*	0.0017
Byte 16 → 4	0.222	0.0239	<0.002*	0.0021

Evaluating 16-Bit Processors for Elliptic Curve Cryptography

Erich Wenger and Mario Werner

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
`Erich.Wenger@iaik.tugraz.at, M.Werner@student.tugraz.at`

Abstract. In a world in which every processing cycle is proportional to used energy and the amount of available energy is limited, it is especially important to optimize source code in order to achieve the best possible runtime. In this paper, we present a side-channel secure C framework performing elliptic curve cryptography and improve its runtime on three 16-bit microprocessors: the MSP430, the PIC24, and the dsPIC. To the best of our knowledge we are the first to present results for the PIC24 and the dsPIC. By evaluating different multi-precision and field-multiplication methods, and hand-crafting the performance critical code in Assembler, we improve the runtime of a point multiplication by a factor of up to 5.41 and the `secp160r1` field-multiplication by 6.36, and the corresponding multi-precision multiplication by 7.91 (compared to a speed-optimized C-implementation). Additionally, we present and compare results for four different standardized elliptic curves making our data applicable for real-world applications. Most spectacular are the performance results on the dsPIC processor, being able to calculate a point multiplication within 1.7 – 4.9 MCycles.

Keywords: Elliptic Curve Cryptography, ECC, Prime Field, MSP430, PIC24, dsPIC, Assembler Optimization.

1 Introduction

Public-key cryptography is an active research area with a lot of applications. The applications range from desktop computing, over (wireless) smart cards down to low-energy sensor networks. On desktop computers one can use special instructions or graphic processors in order to implement asymmetric cryptography. Even nowadays mobile phones are equipped with powerful 32-bit processors making the job of implementing public-key cryptography easier. However, there are a lot of circumstances which call for cheap, energy saving, and fast solutions.

Especially applications utilizing small, embedded processors are interesting for elliptic curve cryptography (ECC). An ECC implementation for such a microprocessor must be resource-conscious, aware of practically dangerous power, timing, and fault attacks and still deliver a runtime, which is fast enough for real-world applications. Additionally, many applications require the use of standardized elliptic curves [1,5,27] that exceed a defined security threshold. Previous work [20,29,31,36] did only consider a part of those requirements.

This paper focuses on a fast and secure implementation of ECC on three embedded 16-bit processors. By optimizing performance critical field operations in Assembler, we drive the PIC24 [22], dsPIC [23], and MSP430 [32] to their limits. To the best of our knowledge we are the first to report runtime results on the PIC24 and dsPIC. Furthermore we present runtime results on the MSP430 using standardized NIST curves [27]. Especially by taking advantage of the 16-bit multiply-accumulate unit of the dsPIC, we were able to improve its runtime for a point multiplication by a factor of up to 5.41, a field multiplication by 6.36, and a big-integer multiplication by 7.91 (compared to a speed-optimized C-implementation). Thus we reduced a 160-bit multi-precision multiplication to 180 cycles, with 100 cycles being the theoretical minimum.

Our framework is written and verified in C, uses a side-channel aware Montgomery Ladder including y-recovery (formulae by Hutter *et al.* [16]), and verifies the resulting point in order to check for fault attacks (see Ebeid *et al.* [9]). Additionally a projective point randomization [7] was used in order to strengthen the side-channel resistance of the Montgomery ladder. Having fixed our high-level algorithms, we investigate different big-integer multiplication methods (operand-scanning, product-scanning, and hybrid) and field-multiplication methods (Barrett, Montgomery, and fast-NIST-reduction) on the three processors. Our results can be the foundation for choosing suitable embedded processors (*e.g.* smart cards or sensor networks), for the investigation of important ECC-related features for future microprocessors, and for follow-up research on custom ECC-hardware designs.

The remainder of the paper is structured as followed. Section 2 discusses relevant related work. Section 3 gives an introduction to elliptic curve cryptography and the used algorithms. Whereas Section 4 gives a short introduction into the MSP430 processor, Section 5 discusses the performance optimizations made for the Microchip processors. Section 6 gives a comparison of all results and Section 7 concludes the work.

2 Related Work

In the last years, a lot of research has been done on implementing elliptic curve cryptography on embedded devices. Most notable is the work by Gura *et al.* [13], who compared elliptic curve cryptography and RSA for the 8-bit ATmega128 processor [2]. For achieving high performance, they introduced the hybrid multiplication method. The results by Gura *et al.* for the ATmega128 processor have been later improved by Uhsadel *et al.* [33], Scott *et al.* [29], and Liu *et al.* [21]. Further results for ATmega128 processors have been shown by Szczechowiak *et al.* [31]. Using the preceding work of Scott and Szczechowiak [29], Szczechowiak *et al.* [31] presented a comparison of ECC and pairings for the ATmega128 and MSP430 processors. Those processors are specially interesting, because they are used for the Wireless Sensor Network nodes MICA2 [8] and Tmote Sky [26]. They compared the performance of the sensor nodes for the NIST K-163 Koblitz curve over $GF(2^{163})$ and a custom curve with $p = 2^{160} - 2^{112} + 2^{64} + 1$ over $GF(p)$. Their results are based on MIRACL [30], a Multiprecision Integer and Rational Arithmetic C/C++ Library.

Similar work with focus on sensor nodes has been published by Liu *et al.* [20]. They used the by SECG standardized `secp160r1` prime field [4]. Note that for security reasons this curve has been removed from the latest version of the standard [5].

In 2009, Yan *et al.* [36] used the 32-bit DSP processor TMS320C6416 from Texas Instruments to implement the `secp160r1` and `secp224r1` elliptic curves. This very powerful processor fits only partly within the scope of this paper, but uses 16-bit multipliers internally. It operates at 1 GHz, has 64 32-bit registers and 1,024 KB L2 cache.

In contrast to the MSP430 and TMS320C6416 processors, we use the ECC-processors by Kern *et al.* [18], Hutter *et al.* [15], and Wenger *et al.* [35] as references. Those papers present specially optimized semi-custom ASIC designs that use 16-bit multipliers. This makes them perfectly suitable for a comparison with the performance of the dsPIC processor.

The design by Kern *et al.* [18] generates an ECDSA signature within 511 kCycles for the elliptic curve `secp160r1`. Specially notable is their fast reduction technique taking advantage of two carry registers.

Hutter *et al.* [15] are using an 8-bit RISC microcontroller supporting 32 instructions, which is used for higher-level control-flow operations. In order to process ECC efficiently, they use eight microcode ROM tables. Those tables control a 16-bit multiply-accumulate unit and a dual-port 16-bit RAM. For NIST P-192 prime field, without reduction a multiplication only needs 168 clock cycles. This is very close to the optimum of $N^2 = 12^2 = 144$ clock cycles. The design performs ECDSA and has a chip area of only 19,115 gate equivalents.

The focus of the work of Wenger *et al.* [35] was to reduce the area footprint (11,686 gate equivalents) of their processor. They use a custom 16-bit RISC processor, featuring 46 instructions and 12 registers. The most notable feature of their Arithmetic Logic Unit (ALU) is a 16-bit multiply-accumulate unit. In order to reduce the disadvantage from their single-port RAM, they parallelized and combined the arithmetic and memory access instructions. Hence the run time of the same 192-bit multi-precision multiplication needs 252 clock cycles, which is 33 % longer than the design of [15].

3 Elliptic-Curve Cryptography

Elliptic-Curve Cryptography (ECC) is defined over the Weierstrass equation

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1}$$

with $a_{i=1,2,3,4,6}, x, y \in K$. K defines the finite field. If a combination of x and y fulfills Equation (1), it is noted a point $P = (x, y)$ on the elliptic curve. Using formulas for the addition and doubling of points, a point multiplication $Q = kP$ can be derived. Finding k, if Q and P are given, is known as the Elliptic Curve Discrete Logarithm Problem (ECDLP).

Throughout this paper, K is a prime field $\mathbb{F}_p$. $n = \lceil log_2(p) \rceil$ bits or $N = \lceil n/W \rceil$ words are needed to represent an element of $\mathbb{F}_p$. W is the word size of the processor used. The binary representation of a field element a can be stored in an array $A = (A[N-1], \ldots, A[2], A[1], A[0])$ of N W-sized words. The least significant bit of a is stored in the rightmost bit of $A[0]$.

3.1 Algorithms Used

Every point operation uses the underlying field operations, discussed in the following subsection. Speeding up those point operations has been a major goal of a lot of researchers [3,14]. Unfortunately most of those formulas are vulnerable to power, timing and fault-analysis attacks [10,34]. So it is important to use a method that is less vulnerable to such attacks. Such a method is the Montgomery Ladder. This method performs a point addition and doubling for every key bit during a point multiplication. We applied the formula by Hutter *et al.* [16] which only requires 7 n-bit registers. Their formula needs 12 multiplications, 4 squarings and 17 additions per key bit and uses a Montgomery Ladder with common-Z coordinates. By randomly initializing this Z-coordinate [7], an additional, computationally cheap resistance against side-channel attacks can be added. In order to make fault attacks more difficult, the y-coordinate is recovered within the projective coordinates and a point verification [9] is performed.

Having fixed the high-level point-multiplication formula, one can concentrate on the fast and efficient implementation of field operations.

3.2 Modular Multiplication

The most time-critical field operation is the field multiplication [14] and the most time-critical part of the field multiplication is the multi-precision multiplication. Although there exist different multi-precision multiplication methods, for every multiplication, N^2 partial products are needed in order to get a $2N$-word result. Such methods are the operand-scanning, the product-scanning [6], and the by Gura *et al.* [13] introduced hybrid method.

The operand-scanning method is shown on the left of Figure 1. During operand scanning one row is calculated by multiplying one word of the first operand with all words of the second operand. The resulting partial products are immediately added to $C[k]$. Thus this method needs $2N^2 + N$ read operations and $N^2 + N$ write operations. Those are $3N^2 + 2N$ memory operations in total.

By rearranging the fixed number of partial products, the product-scanning method (see right part Figure 1) can be derived. Here the intermediate results are sorted column-wise. By using an accumulator [6,12], the intermediate results can be summed up very efficiently. For this method N^2 read operations are needed to access $A[i]$ and $B[j]$ and only $2N$ write operations are needed to store $C[k]$. So $2N^2 + 2N$ memory operations are needed in total.

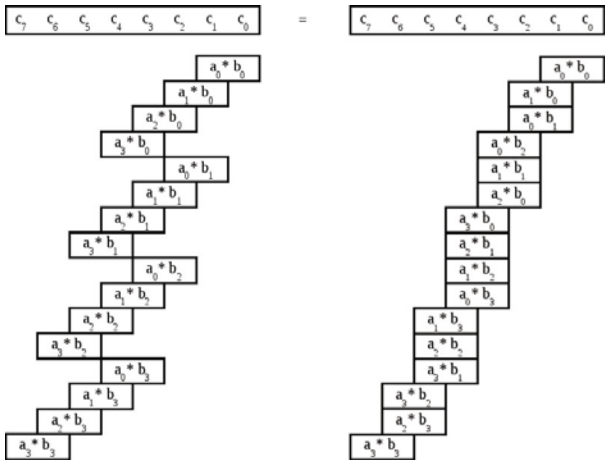


Fig. 1. Partial products during multi-precision multiplication (left: Operand Scanning, right: Product Scanning)

In order to implement the operand and product-scanning multiplication methods, very few registers are needed. Gura *et al.* [13] takes advantage of the large register-set of embedded processors and combines the operand and product-scanning approaches. In an inner loop, d^2 intermediate products are processed row-wise (operand scanning) and in an outer loop, the intermediate sums are added column-wise (product scanning). The larger the parameter d is chosen, the more intermediate products are processed row-wise (d^2) and the more registers are needed ($3d + \lceil log_2(N/d)/W \rceil$) to implement this method efficiently. The advantage of the hybrid-multiplication method is that it can reduce the number of memory operations to $2\lceil N^2/d \rceil + 2N$.

3.3 Field Multiplication

The used multi-precision multiplication method has a significant influence on the runtime of a point and field multiplication. A field multiplication adds a reduction to the multi-precision multiplication. There are three very popular ways to perform this reduction: Barrett reduction, Montgomery reduction, and fast reduction (NIST reduction). We implemented all three methods and the results convinced us to do no further investigation into the Barrett reduction.

The fast reduction method is based on the special design of the primes used within the standards (*e.g.* NIST P-192; $p_{192} = 2^{192} - 2^{64} - 1$). This special Mersenne-like prime allows a reduction by only using addition and shift operations. In practice this method is very fast, because additions are much faster than multiplications. However there are Mersenne-like primes with a lot of (*e.g.* NIST P-256; $p_{256} = 2^{256} - 2^{128} - 2^{96} + 2^{32} - 1$) or odd (*e.g.* `secp160r1`; $p_{160} = 2^{160} - 2^{31} - 1$) exponents, which have a significant influence on the runtime.

A method which is not dependent on the kind of the used prime is the Montgomery multiplication [25]. This method need some special set-up and back transformation of the field elements, but when there are a lot of operations involved, this method definitively pays off. First a $R = 2^{WN}$ is selected at design-time, making sure that $R > p$ and $gcd(R, p) = 1$. The field elements are transformed by

$$\tilde{a} \equiv aR \pmod{p}, \text{ and} \tag{2}$$

$$\tilde{b} \equiv bR \pmod{p}, \tag{3}$$

and further $\tilde{a}$ and $\tilde{b}$ are used instead of a and b. A multiplication is performed as follows:

$$\tilde{c} \equiv Mont(\tilde{a}, \tilde{b}) \equiv \tilde{a}\tilde{b}R^{-1} \equiv (aR)(bR)R^{-1} \equiv abR \equiv cR \pmod{p}. \tag{4}$$

Utilizing the Montgomery multiplication, the transformation of the field elements can be carried out easily:

$$Mont(a, R^2) \equiv aR^2R^{-1} \equiv aR \equiv \tilde{a} \pmod{p}, \tag{5}$$

$$Mont(\tilde{c}, 1) \equiv (cR)R^{-1} \equiv c \pmod{p}. \tag{6}$$

In practice the multi-precision multiplication and Montgomery reduction are interleaved. For the important implementation specific aspects of the Montgomery multiplication we refer to Koç *et al.* [19].

Every one of the previously discussed multiplication and reduction methods has advantages and disadvantages on certain microprocessors and the used prime. Those are discussed in the following sections.

4 Texas Instruments MSP430

The 16-bit MSP430 processor [32] by Texas Instruments is a very popular RISC processor. Especially its vast field of application makes it interesting (*e.g.* Tmote Sky [26]). The processor has 16 fully-addressable, single-cycle CPU registers that can be used with 27 instructions (24 additional instructions are emulated). Four of those registers are special purpose registers (program counter, stack pointer, status register, and constant generator), so not all registers can be used for an Assembler-optimized elliptic-curve implementation. A very important module for the multi-precision multiplication algorithm is the multiplier. The MSP430 does not have a multiplication instruction. Depending on the series, it only offers a 16-bit or 32-bit memory-mapped hardware multiplier. In order to perform a 16-bit multiplication, four memory accesses are necessary (write the two operand words and read the two resulting words).

For the MSP430, we implemented the three major multi-precision multiplication operations, discussed in Section 3.2 in Assembler. The advantage of the operand-scanning method is that in average only three memory accesses are necessary for a single integer multiplication. Remember that one operand stays constant during the calculation of a row. The product-scanning multiplication method takes advantage of the `MAC` instruction. This instruction uses the result register as accumulator and provides a carry bit. So for every partial product, the two operands have to be written and only the carry flag has to be processed. For the hybrid multiplication results in Section 6, we used an implementation with $d = 2$.

In order to summarize our results for the MSP430, the biggest disadvantage in terms of ECC is that its hardware multiplier is memory mapped. Having a processor with a multiplication instruction improves ECC significantly. Such processors are the PIC24 and dsPIC.

5 Microchip PIC24 and dsPIC

The PIC24 and dsPIC microcontroller families by Microchip [22,23] are 16-bit RISC processors that are using a modified Harvard architecture. They are widely used for motor control, signal processing, and intelligent sensor applications. These controllers use 24-bit instruction words with a variable length opcode field. Almost all instructions can be executed in a single cycle. Only commands which change the program flow, table operations and the double word move instruction take two or more cycles. Similar to the MSP430, the used PIC processors have 16 16-bit registers labeled as W0 to W15. W14 and W15 are used as frame and stack pointer.

The dsPIC processors facilitate different features that make them perfectly suited for public-key operations. First of all, the processor comes with a Digital Signal Processing (DSP) engine. This engine provides two 40-bit accumulators and a corresponding Arithmetic Logic Unit (ALU). This ALU is equipped with a multiply and accumulate (`MAC`) instruction. Additionally a second Address Generation Unit (AGU) is integrated. The primary address generation unit is called X, the secondary is called Y. Some of the DSP instructions can utilize these two AGUs and therefore fetch two operands at the same time while processing the data in the registers. However, not the whole address space is available to both AGUs. The X unit can read and write to all addresses while the Y unit can only read from a certain device specific region. Nevertheless it is possible to improve the performance considerable by taking the hassle of placing the data at the correct position in memory.

In order to minimize the overhead of loop constructs, special loop instructions named `REPEAT` and `DO` are provided. While `DO` needs two cycles to set up the loop and can execute several instructions multiple times, `REPEAT` needs one cycle and operates on a single command only. Another feature is the possibility to post increment or decrement pointers when indirect memory addressing is used. With this capability, the pointer arithmetic can be sped up significantly.

When performance is not the first priority using PIC24 processors is an option. Beside the missing DSP engine and the lack of the `DO` instruction the cores of the PIC24 and the dsPIC processors are the same. Due to this compatibility we can give detailed results on the impact of the DSP engine concerning performance.

In order to find the fastest implementation for this processor architecture we start by presenting a generic product scanning algorithm which takes advantage of the DSP instructions. As the code of the inner loop is the most important code segment concerning speed, the following subsection focuses on this part. In the next step additional ideas to further optimize the multi-precision multiplication for speed are explained.

5.1 Generic Product Scanning on the dsPIC

The inner loop of the product scanning algorithm consists of one 16-bit multiplication and one addition (32-bit plus overflow handling). Additionally two load operations are required to fetch the values of $A[i]$ and $B[j]$ from memory. Furthermore the increment or decrement of i and j has to be handled as well. Usually this operations would involve two registers for the operand values, two registers for the multiplication product and three registers for the result of the addition.

As we have the DSP extension we take advantage of one of the 40-bit accumulators. The multiply-accumulate (`MAC`) instruction is used to multiply two 16-bit operands and to add the result to the accumulator. This means that we can handle the multiplication and the addition with the necessary precision in one clock cycle.

Now that we know how the data can be processed efficiently we still need to find a performant way to fetch the operand values. This can be implemented with the `MAC` instruction as well. As this DSP command possesses the capability to pre-fetch values by utilizing the two AGUs, it is possible to load both operands during one clock cycle.

By the use of pointers to address the current operands, the update of i and j can be implemented through applying post increments and/or decrements to those pointers. Due to the fact that this pointer arithmetic is also supported by this DSP operation the complete inner loop can be reduced to a single `MAC` instruction plus `REPEAT` command.

The result of this optimization can be seen in Figure 2 in Lines 14 to 16. At first the number of iterations for the `REPEAT` is calculated by subtracting the current loop count in DCOUNT (addressed via pointer in W0) from the total loop count (stored in W3). Afterwards the loop is set up using `REPEAT` followed by the `MAC` instruction which should be repeated. This `MAC` invocation reads as follows: Multiply the operands stored in *a_temp* and *b_temp* and add them to accumulator A ($A \leftarrow A + W5 * W6$). At the same time fetch the next *a_temp* via AGU X by dereferencing *aPtr_temp* ($W5 \leftarrow [W8]$). Simultaneously, the next *b_temp* is fetched via AGU Y by dereferencing *bPtr_temp* ($W6 \leftarrow [W10]$). The pointers are updated by incrementing *aPtr_temp* ($[W8]+ = 2$) and decrementing *bPtr_temp* ($[W10]- = 2$) by two.

As the `MAC` instruction always operates with the values which are already stored in *a_temp* and *b_temp*, it is mandatory to initialize them before this inner loop is executed. These fetches are done by using the `MOVSAC` operation in Line 12.

```
; W0 = pointer to the DCOUNT register        ; W7 = pointer to the ACCAL register
; W2 = result pointer                        ; W8 = temporary pointer (aPtr_temp)
; W3 = integer length in words (length)      ; W9 = operand pointer (aPtr)
; W4 = temp register (tmp)                   ; W10 = temporary pointer (bPtr_temp)
; W5 = value of operand a (a_temp)           ; W11 = operand pointer (bPtr)
; W6 = value of operand b (b_temp)

      DO W3, loop                            ; main loop, iterates length+1 times
      MOV W9,W8                                        ; aPtr_temp = aPtr
      SUB W11,#2,W10                                   ; bPtr_temp = bPtr − 2
; pre−fetch operands for the inner loop                ; a_temp = *aPtr_temp++
      MOVSAC A, [W8]+=2, W5, [W11]+=2, W6              ; b_temp = *bPtr++
; calculate iteration count and execute the loop
      SUB W3,[W0],W4                                   ; tmp = length − DCOUNT
      REPEAT W4
      MAC W5*W6, A, [W8]+=2, W5, [W10]−=2, W6
; store the resulting word and shift the accumulator
      MOV [W7],[W2++]                                  ; *resPtr++ = ACCAL
loop: SFTAC A, #16
```

Fig. 2. One of the two loops in the generic product scanning implementation

By looking at the pre-fetch behavior it can be seen that the last `MAC` operation loads operands which are not needed for the algorithm. By unrolling this last `MAC` invocation from the `REPEAT` it is possible to use the last pre-fetch to initialize *a_temp* and *b_temp* with the needed data for the next iteration. By doing this, the `MOVSAC` operation in the outer loop can be omitted.

5.2 Unrolled Product Scanning on the dsPIC

By unrolling the code for a fixed n-bit integer multiplication, advantage of the constant input size can be taken to improve the pointer arithmetic. The temporary pointers can be omitted when an appropriate sequence for the partial products is chosen as the post-in/decrement functionality of the `MAC` instruction is sufficient to update the pointers. The resulting zig-zag like pattern is nicely visualized in Figure 3. A part of the resulting source code is shown in Figure 4. Observe that the multiplicands for the multiplication are loaded in the preceding operations $W5 \leftarrow [W9]$ and $W6 \leftarrow [W11]$. In the once more preceding operation, the pointer registers are updated using the post in/decrement feature: $[W9]- = 2$, $[W11]+ = 2$.

At this point, we have to note that there is a prerequisite which has to be fulfilled. The pre-fetch mechanism requires that the operands are in different address spaces (X and Y). We ensure this by copying the input integers into temporary variables at the beginning of the multiplication. Although these copy operations are quite cheap (15 cycles for one 160-bit integer) it is possible to omit them by placing the operands in the correct memory region before the function call. As the generic implementation uses the pre-fetch technique too, the same constraint applies.

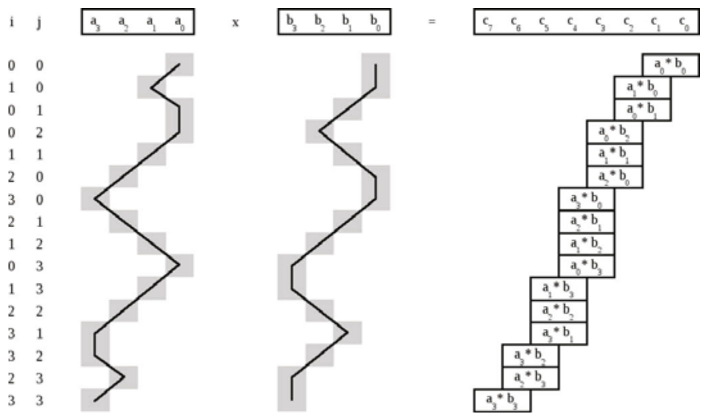


Fig. 3. Note the zig-zag product scanning multiplication method, which is specially suited for the dsPIC (e.g. four word multi-precision multiplication)

5.3 Montgomery Multiplication on the dsPIC

Like proposed by Koç *et al.* [19], various implementations of the Montgomery multiplication are possible which mainly differ in the underlying multiplication method (operand scanning, product scanning, . . .) and the degree of integration of the reduction step. As product scanning provides the best results on the dsPIC architecture we have chosen the Finely Integrated Product Scanning (FIPS) method for our implementation. The interleaving of multiplication and reduction makes it necessary to implement the whole Montgomery multiplication using Assembler. Otherwise no decent performance can be gained following the FIPS approach. Through the high similarity of the product-scanning and the FIPS method most of the optimizations presented in the last sections can be applied as well, resulting in a fast implementation of the field multiplication.

```
MAC W5*W6, A, [W9]-=2, W5, [W11]+=2, W6
MAC W5*W6, A, [W9], W5, [W11]+=2, W6
MAC W5*W6, A, [W9]+=2, W5, [W11]-=2, W6
MOV [W7],[W2++]
SFTAC A, #16

MAC W5*W6, A, [W9]+=2, W5, [W11]-=2, W6
MAC W5*W6, A, [W9]+=2, W5, [W11]-=2, W6
MAC W5*W6, A, [W9]+=2, W5, [W11], W6
MAC W5*W6, A, [W9]-=2, W5, [W11]+=2, W6
MOV [W7],[W2++]
SFTAC A, #16
```

Fig. 4. Unrolled calculation of the result word two and three

6 Comparison Results

For the results shown in this section, we avoided to select the parameters for ECC ourselves. Instead we chose curves that have been standardized [1,5,27]. NIST [28] gave recommendations about the security margin for future applications. They recommend to use elliptic curves that exceed 160 bits. Also SECG removed the 160-bit curve in their latest release of the SEC standard [5]. Nevertheless, we present results for the `secp160r1` curve for comparison with related work. Additionally, the relative performance results shown in Subsection 6.1 are also applicable for larger elliptic curves. Those elliptic curves are discussed in Subsection 6.2.

For the generation of the results, for the MSP430 we used the IAR Embedded Workbench 5.30 [17] with the maximum optimizer settings available '-Ohs'. All results have been generated for the MSP430F1611 device which comes with an embedded 16-bit multiplier.

For the PIC24 and dsPIC results, we used the simulator of the MPLAB IDE v8.63 [24], the MPLAB C30 v3.25 compiler with settings '-o3', '-funroll-loops', and the PIC24FJ96GA006 and dsPIC30F6015 devices.

For the following comparisons it should be noted that we neglect parameters such as chip area, power consumption and cost factors, because they significantly differ from processor to processor. However energy is defined as product of power and time. So by minimizing the runtime, we also optimize the energy consumption.

6.1 Relative Performance

In Table 1, we compare the run times for big-integer multiplication, field multiplication and point multiplication using the `secp160r1` curve [4]. For all presented platforms, we started with a reference implementation in C. In C, the operand-scanning multiplication outperforms the product-scanning multiplication by 12.6 % (MSP430) and 16.4 % (PIC24). Consequently, the point multiplication is 8.2 % and 13.5 % faster. By manually writing the multi-precision multiplication in Assembler, the run time was reduced by a factor of 1.53 to 2.42. Implementing the hybrid multiplication method with $d = 2$ on the MSP430 improved the runtime by just 4.8 %. By only using 3.1 kbytes of program memory and 274 bytes of stack, this implementation is very resource friendly. Unrolling the product-scanning multiplication method improved the multi-precision multiplication by another 15.1 %, but came at the cost of additional 1.5 kbytes (47 %) more program code.

Up to this point, the results on the PIC24 processor are identical to the results of the dsPIC processor. In the following steps, we utilize the special features of the dsPIC to further improve the performance. By making use of the `DO` and `REPEAT` commands, the run-time of a single multi-precision multiplication was reduced by 29 % to 33 %. A more significant performance improvement was achieved by utilizing the DSP part of the dsPIC processor. A speedup of 2.72 has been achieved for the product-scanning multiplication method. Using

Table 1. Comparison of the multi-precision multiplication, the field multiplication and the ECC point multiplication for `secp160r1`

Implementation		**Multi-Prec. Mult.**		$\mathbb{F}_{p_{160}}$ **Mult.**	**Point Mult.**
MSP430	C	op. sc.	4,103	6,069	16,985,654
MSP430	C	pr. sc.	4,699	6,665	18,512,606
MSP430[a]	ASM	op. sc.	2,583	4,127	11,380,361
MSP430[a]	ASM	pr. sc.	1,945	3,489	9,745,805
MSP430[a]	ASM	hybrid	1,851	3,395	9,504,977
MSP430[a]	ASM + unrolled	pr. sc.	**1,570**	**3,112**	**8,779,931**
PIC24	C	op. sc.	1,423	2,393	6,703,476
PIC24	C	pr. sc.	1,702	2,675	7,753,292
PIC24[b]	ASM	op. sc.	**929**	**1,909**	**5,463,648**
PIC24[b]	ASM	pr. sc.	1,031	2,011	5,739,732
dsPIC[c]	ASM + DO/REP.	op. sc.	622	998	2,840,921
dsPIC[c]	ASM + DO/REP.	pr. sc.	727	1,104	3,127,253
dsPIC[c]	ASM + DSP	op. sc.	546	923	2,648,377
dsPIC[c]	ASM + DSP	pr. sc.	267	644	1,932,431
dsPIC[c]	ASM + unrolled	pr. sc.	**180**	557	1,709,537
dsPIC[c]	ASM + DSP	Montgomery	—	554	1,696,433
dsPIC[c]	ASM + unrolled	Montgomery	—	**376**	**1,239,281**
MSP430	Liu *et al.* [20]				12,645,040
MSP430[d]	Scott *et al.* [29,31]		1,746	2,736	5,760,000
TMS320	Yan *et al.* [36]		150	290	810,000
ASIC	Kern *et al.* [18]	pr. sc.		167	511,864

[a] Multi-precision addition and subtraction were manually unrolled in Assembler.
[b] Only multi-precision multiplication was manually written in Assembler.
[c] Multi-precision addition, subtraction, and shift operation were manually written in Assembler.
[d] Did not use `secp160r1`.

the same methodology, the operand-scanning multiplication method improved by only 1.14. So only when we take advantage of the DSP-unit, the product-scanning method is (2.04 times) faster. A further experiment showed that by unrolling the Assembler code and performing the product-scanning in a zig-zag-like fashion, the run time could be further reduced by 32.6 %. At this point it should be noted that although the performance of the multi-precision multiplication has been improved by a factor of 7.91 (unrolled DSP vs. best C version), the performance of the field multiplication improved by 4.30 and the point multiplication improved by 3.92. A reason for that is the slow reduction modulo $p_{160} = 2^{160} - 2^{31} - 1$. The term 2^{31} results into a relatively slow shift operation. So we investigated the Montgomery multiplication technique. Utilizing the FIPS multiplication method from Koç *et al.* [19], the field multiplication was improved by 14.0 %. Unrolling the Assembler code resulted in the fastest field multiplication, just needing 376 cycles. This is 32.5 % faster than the fastest combination of unrolled product-scanning and fast reduction.

By investigating the related results for the MSP430, it becomes obvious that in comparison to Liu *et al.* [20], our point multiplication method is 25 % faster even though they used a memory-hungry sliding-window point multiplication method.

The hybrid multi-precision multiplication method by Scott and Szczechowiak*et al.* [29,31] is 6 % faster than our multiplication method, because they unrolled their hybrid multiplication. However compared to our unrolled product-scanning method they are 11 % slower. The differences within the field and point multiplication come from a different elliptic curve and sliding-window point multiplication formula used. Because the sliding window technique needs additional memory, they need 2.9 kbytes of data memory and 31.3 kbytes of program memory, which is more than 10 times the resources we need.

The results on the dsPIC processor made us confident enough to compare them with the powerful TMS320C6416 processor (Yan *et al.* [36]) as well as the custom designed ASIC by Kern *et al.* [18]. The implementation by Yan *et al.* [36] using the mighty TMS320C6416 processor[1] and the custom designed ASIC ought to be faster. But the difference is only a factor of 1.53 – 2.42.

With this comparison we showed that the field multiplication utilizing a memory-mapped multiplication unit is more than 2 times slower. Also the advantages of having `DO`/`REPEAT` and DSP instructions have been discussed. Utilizing the full potential of the Montgomery multiplication method, the point multiplication has been improved by a factor of 5.41 versus the fastest C-only implementation.

6.2 Scaling of Performance

The last subsection limited the comparison to the `secp160r1` curve. This subsection extends our focuses to the NIST [27] standardized elliptic curves P-192, P-224, and P-256.

The most remarkable feature within Table 2 is the influence of the chosen field prime into the run time of the point multiplication. On the dsPIC processor, the performance of the FIPS Montgomery field-multiplication is better for the `secp160r1` and NIST P-256 fields, but the fast reduction technique utilizing Mersenne-like primes are faster for the NIST P-192 and P-224 prime fields.

The Assembler optimizations on the MSP430 resulted in a speedup of 1.93 to 2.34. The larger the used prime field, the larger is the achieved speedup. Also on the PIC24 architecture, speedups between 1.23 and 1.42 have been achieved. Utilizing the dsPIC, the biggest speedup factors, ranging from 3.89 up to 4.79, have been achieved.

Our results improved the work of Liu *et al.* [20] on the MSP430 processor. Again we compared our results with the powerful TMS320C6416 processor as well as some custom designed ASICs. The TMS320C6416 performs the point multiplication only 1.7 – 2.1 times faster, which is quite small compared to the processing power of the TMS320C6416. Also the ASIC designs ought to be faster.

[1] 64 32-bit registers. Eight independent functional units. 1 GHz.

Table 2. Point multiplication for different field parameters in kCycles

	Implementation		`secp160r1`	**P-192**	**P-224**	**P-256**
MSP430	C	op. sc.	16,986	23,405	35,531	47,455
MSP430	ASM	hybrid	9,505	11,949	18,464	23,973
PIC24	C	op. sc.	6,703	8,985	13,781	18,992
PIC24	ASM	op. sc.	5,464	6,754	10,138	13,379
dsPIC	ASM + DSP	pr. sc.	1,932	**2,178**	**2,880**	5,079
dsPIC	ASM + DSP	Mont.	**1,696**	2,528	3,582	**4,879**
MSP430	Liu *et al.* [20]		12,645			
C6416	Yan *et al.* [36]		810		1,690	
ASIC	Kern *et al.* [18]	pr. sc.	512			
ASIC	Wenger *et al.* [35]	pr. sc.		1,377		
ASIC	Hutter *et al.* [15]	pr. sc.		783		

Although those design focused on area-optimizations the authors [15,18,35] achieved good run time results. Consequently the performance differs by a factor of 1.58 – 3.31.

7 Conclusion

In this paper we presented, evaluated, and optimized an elliptic curve point multiplication for three different processors using four different elliptic curves. Starting with a C reference implementation (which includes several countermeasures against possible attacks), we were able to improve the runtime by simply rewriting the performance critical field operations in Assembler. Therefore we evaluated the most commonly used big-integer and field multiplication methods. We achieved a speedup for a single point multiplication of 1.93 – 2.34 on the MSP430, 1.23 – 1.42 on the PIC24 and 3.89 – 5.41 on the dsPIC. Especially impressing is the possible speedup of 7.91 for a single 160-bit multiplication on the dsPIC which is 10.5 times faster than fastest corresponding operation on the MSP430.

By the best of our knowledge, we are the first to present results for ECC on the PIC24 and dsPIC and those results are especially interesting for future applications in which embedded processors are required.

Acknowledgements. This work has been supported by the Austrian Government through the research program FIT-IT Trust in IT Systems under the project number 825743 (project PIT). Finally we would like to thank Manfred Großmann for some fruitful discussions.

References

1. American National Standards Institute (ANSI). AMERICAN NATIONAL STANDARD X9.62-2005. Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm, ECDSA (2005)

2. Atmel Corporation. 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash (August 2007), http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf
3. Bernstein, D., Lange, T.: Explicit-formulas database, http://www.hyperelliptic.org/EFD
4. Certicom Research. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0 (September 2000), http://www.secg.org/
5. Certicom Research. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2.0 (January 2010), http://www.secg.org/
6. Comba, P.: Exponentiation cryptosystems on the IBM PC. IBM Systems Journal 29(4), 526–538 (1990)
7. Coron, J.-S.: Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999)
8. Crossbow Technology, Inc. MICAz Wireless Measurement System, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf
9. Ebeid, N., Lambert, R.: Securing the Elliptic Curve Montgomery Ladder Against Fault Attacks. In: Proceedings of Workshop on Fault Diagnosis and Tolerance in Cryptography - FDTC 2009, Lausanne, Switzerland, pp. 46–50 (September 2009)
10. Fan, J., Guo, X., Mulder, E.D., Schaumont, P., Preneel, B., Verbauwhede, I.: State-of-the-Art of Secure ECC Implementations: A Survey on known Side-Channel Attacks and Countermeasures. In: Proceedings of 3rd IEEE International Symposium Hardware-Oriented Security and Trust - HOST 2010, California, USA, June 13-14, pp. 76–87. IEEE (2010)
11. Großmann, M.: Optimize Elliptic Curve Cryptography for MSP430 Processor. Bachelor Thesis at Graz University of Technology (May 2011)
12. Großschädl, J., Savaş, E.: Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m). In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 133–147. Springer, Heidelberg (2004)
13. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing Elliptic Curve Cryptography and RSA on 8-Bit CPUs. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 119–132. Springer, Heidelberg (2004)
14. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer, Heidelberg (2004)
15. Hutter, M., Feldhofer, M., Plos, T.: An ECDSA Processor for RFID Authentication. In: Ors Yalcin, S.B. (ed.) RFIDSec 2010. LNCS, vol. 6370, pp. 189–202. Springer, Heidelberg (2010)
16. Hutter, M., Joye, M., Sierra, Y.: Memory-Constrained Implementations of Elliptic Curve Cryptography in Co-Z Coordinate Representation. In: Nitaj, A., Pointcheval, D. (eds.) AFRICACRYPT 2011. LNCS, vol. 6737, pp. 170–187. Springer, Heidelberg (2011)
17. IAR Systems. IAR Embedded Workbench (2011), http://www.iar.com/
18. Kern, T., Feldhofer, M.: Low-Resource ECDSA Implementation for Passive RFID Tags. In: Proceedings of 17th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2010), Athens, Greece, December 12-15, pp. 1236–1239. IEEE (2010)
19. Koç, Ç.K., Acar, T., Kaliski Jr., B.S.: Analyzing and Comparing Montgomery Multiplication Algorithms. IEEE Micro 16(3), 26–33 (1996)

20. Liu, A., Ning, P.: TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In: Proceedings of International Conference on Information Processing in Sensor Networks - IPSN 2008, St. Louis, Missouri, USA, April 22-24, pp. 245–256 (2008)
21. Liu, Z., Großschädl, J., Kizhvatov, I.: Efficient and Side-Channel Resistant RSA Implementation for 8-bit AVR Microcontrollers. In: Proceedings of 1st International Workshop on the Security of the Internet of Things - SOCIOT 2010, Tokyo, Japan, November 29. IEEE Computer Society (2010)
22. Microchip. PIC24FJ128GA010 Family Data Sheet. DS39747E (October 2009), http://www.microchip.com
23. Microchip. dsPIC30F6010A/6015 Data Sheet. DS70150E (March 2011), http://www.microchip.com
24. Microchip. MPLAB Integrated Development Environment (2011), http://www.microchip.com
25. Montgomery, P.L.: Modular Multiplication without Trial Division. Mathematics of Computation 44, 519–521 (1985)
26. Moteiv. The Moteiv Wireless Sensor Networks Website, http://www.moteiv.com/
27. National Institute of Standards and Technology (NIST). FIPS-186-3: Digital Signature Standard, DSS (2009), http://www.itl.nist.gov/fipspubs/
28. National Institute of Standards and Technology (NIST). SP800-57 Part 1: DRAFT Recommendation for Key Management: Part 1: General (May 2011), http://csrc.nist.gov/publications/drafts/800-57/Draft_SP800-57-Part1-Rev3_May2011.pdf
29. Scott, M., Szczechowiak, P.: Optimizing Multiprecision Multiplication for Public Key Cryptography. Cryptology ePrint Archive, Report 2007/299 (2007), http://eprint.iacr.org/
30. Shamus Software. Multiprecision Integer and Rational Arithmetic C/C++ Library (2011), http://www.shamus.ie/
31. Szczechowiak, P., Oliveira, L.B., Scott, M., Collier, M., Dahab, R.: NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. In: Verdone, R. (ed.) EWSN 2008. LNCS, vol. 4913, pp. 305–320. Springer, Heidelberg (2008)
32. Texas Instruments. MSP430C11x1 - Mixed Signal Microcontroller (2008), http://focus.ti.com
33. Uhsadel, L., Poschmann, A., Paar, C.: Enabling Full-Size Public-Key Algorithms on 8-Bit Sensor Nodes. In: Stajano, F., Meadows, C., Capkun, S., Moore, T. (eds.) ESAS 2007. LNCS, vol. 4572, pp. 73–86. Springer, Heidelberg (2007)
34. Walter, C.D.: Simple Power Analysis of Unified Code for ECC Double and Add. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 191–204. Springer, Heidelberg (2004)
35. Wenger, E., Feldhofer, M., Felber, N.: Low-Resource Hardware Design of an Elliptic Curve Processor for Contactless Devices. In: Chung, Y., Yung, M. (eds.) WISA 2010. LNCS, vol. 6513, pp. 92–106. Springer, Heidelberg (2011)
36. Yan, H., Shi, Z.J., Fei, Y.: Efficient Implementation of Elliptic Curve Cryptography on DSP for Underwater Sensor Networks. In: 7th Workshop on Optimizations for DSP and Embedded Systems (ODES- 7), pp. 7–15 (March 2009)

A Hardware Processor Supporting Elliptic Curve Cryptography for Less than 9 kGEs

Erich Wenger and Michael Hutter

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
{Erich.Wenger,Michael.Hutter}@iaik.tugraz.at

Abstract. Elliptic Curve Cryptography (ECC) based processors have gained large attention in the context of embedded-system design due to their ability of efficient implementation. In this paper, we present a low-resource processor that supports ECC operations for less than 9 kGEs. We base our design on an optimized 16-bit microcontroller that provides high flexibility and scalability for various applications. The design allows the use of an optimized RAM-macro block and reduces the complexity by sharing various resources of the controller and the datapath. Our results improves the state of the art in low-resource $\mathbb{F}_{2^{163}}$ ECC implementations (14 % less area needed compared to the best solution reported). The total size of the processor is 8,958 GEs for a 0.13 μm CMOS technology and needs 285 kcycles for a point multiplication. It shows that the proposed solution is well suitable for low-power designs by providing a power consumption of only 3.2 μW at 100 kHz.

Keywords: Low-Resource Hardware Implementation, Elliptic Curve Cryptography, Binary Extension Field, Embedded Systems.

1 Introduction

With the rapid development of more powerful and energy-saving devices, we unwittingly move towards the vision of the Internet of things. The required security services within this vision can be particularly achieved using Elliptic Curve Cryptography (ECC). This paper focuses on a low-resource hardware processor that provides ECC capabilities while meeting the low-area and low-power requirements of embedded systems.

There exist many proposals for low-resource ECC processors. Most of the processors operate on binary-field elliptic curves and use full-precision arithmetic to increase the performance of point multiplication [4,13,25,35]. One of the most efficient solutions in terms of low-resource requirements has been reported by Lee et al. [26].

They presented a processor supporting a small elliptic curve over $\mathbb{F}_{2^{163}}$ which makes use of a tiny 8-bit microcontroller to handle higher-level protocol implementations. The ECC operation of $k \cdot P$ is performed by a separated Modular Arithmetic Logic Unit (MALU). The processor needs 12,506 GEs and 276 kcycles to perform a point multiplication. However, the area estimations do not including program ROM and RAM to store intermediate results and the necessary secret scalar k. Similar datapath architectures have been reported by Batina et al. [2] and Sakiyama et al. [32]. Hein et al. [17] reported a very efficient co-processor (without microcontroller) for the same elliptic curve supporting multi-precision arithmetics. They applied a finite-state machine based control-engine needing 11,904 GEs including a standard-cell based RAM memory.

In this paper, we present a low-resource hardware processor that is based on a 16-bit multi-precision architecture and an area-optimized custom microcontroller. This combination allows several optimizations. First, it allows the use of an efficient RAM-macro block that reduces the area requirements for short-term memory significantly. Second, since both the microcontroller and the datapath use a 16-bit architecture, all resources are shared to minimize the area footprint of the processor. As an outcome, we present a complete solution including memory for short-term (RAM) as well as long-term storage (program ROM), controller, and datapath using a polynomial multiply-accumulate (MAC) unit. In addition, we present results of higher-level protocol implementations of the Elliptic Curve Digital Signature Algorithm (ECDSA) [30] and give results for digital signature generation as well as verification. For a point multiplication, our NIST B-163 based processor needs only 8,958 GEs in total and performs a point multiplication within 285 kcycles. We demonstrate that the proposed solution is also well suitable for low-resource embedded systems by providing a power consumption of only 3.2 μW at 100 kHz.

The rest of the article is structured as follows. In Section 2, a brief introduction into elliptic curve cryptography is given. In Section 3, we face the challenge of low-resource ECC hardware implementations and explore various design possibilities. We evaluate appropriate word sizes of a processor and analyze different memory types. Section 4 presents details about the hardware architecture of our processor. Details about the implementation are given in Section 5. In Section 6, the results are presented. Conclusions are drawn in Section 7.

2 Elliptic Curve Cryptography

Within Elliptic Curve Cryptography (ECC), not only a single number or polynomial is used, but a pair of those. Each pair (x, y) of such numbers that satisfy the general Weierstrass equation

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1}$$

is called a point on an elliptic curve. When a certain type of number is used, in our case binary polynomials within $GF(2^m)$, the Weierstrass equation can be reduced to

$$y^2 + xy = x^3 + ax^2 + b. \tag{2}$$

Among the most critical operation in terms of speed and security is the ECC point multiplication. The implementation of this multiplication has to be secure against various implementation attacks such as side-channel and fault-analysis attacks. The Montgomery ladder [28,21] provides very beneficial properties in this context. We therefore decided to use it for our design and applied the very fast group-operation formulas of López and Dahab [27]. The formulas are based on projective coordinates (which avoid expensive field inversions) that can be nicely combined with proposed countermeasures (see also the work of Junfeng Fan et al. [11]) such as randomized projective coordinates (RPC) [6] or point-validity checks [8].

We use the following notations throughout the paper (similar to [16]). Let $f(z) = z^m + r(z)$ denote an irreducible binary polynomial of degree m. The elements of $\mathbb{F}_{2^m}$ are binary polynomials of degree at most $m-1$. An addition of field elements is the usual addition of binary polynomials. Multiplication is performed modulo $f(z)$. A field element $a(z) = a_{m-1}z^{m-1} + \cdots + a_2z^2 + a_1z + a_0$ is associated with the binary vector $a = (a_{m-1}, \ldots, a_2, a_1, a_0)$ of length m. Furthermore, let $N = \lceil m/W \rceil$ be the number of words with width W needed to store $a(z)$. $A = (A[N-1], \ldots, A[2], A[1], A[0])$, where the rightmost bit of $A[0]$ is a_0, and the leftmost $(WN - m)$ bits of $A[N-1]$ are unused (always set to zero).

For further readings on ECC we refer to several books [1,3,16,23] that discuss the topic extensively.

3 Design-Space Exploration

In this section, we will explore different hardware-design options to obtain best results for a low-resource ECC processor. The design goals have been to meet all requirements of embedded systems which are low area (due to the production costs), low power (due to a possible contactless operation), appropriate speed (required for certain applications), security and flexibility. Due to the latter requirement, we decided to base our design on a customized microcontroller. This has the advantage of being modular in terms of protocol implementations and modifications of already implemented solutions.

By following the principles of hardware/software co-design, it showed that the dominant factors of ECC processors are the finite-field hardware multiplier and the type and size of the applied data memory. In the following, we discuss these factors and explore the design space to find the best solution for our objectives.

3.1 The Hardware Multiplier

One of the most area consuming parts within the ALU of an ECC-hardware design is the finite-field multiplier. The size, speed, and power consumption of such a multiplier largely depends on the word size of the processor and the underlying finite field. Figure 1 shows the hardware architecture of a 4-bit multiplier for binary-field (carry-less multiplier), prime-field (integer multiplier), and dual-field arithmetic. The basic structure of all three types of multiplier is the same. Only the adder structure needs to be adopted.

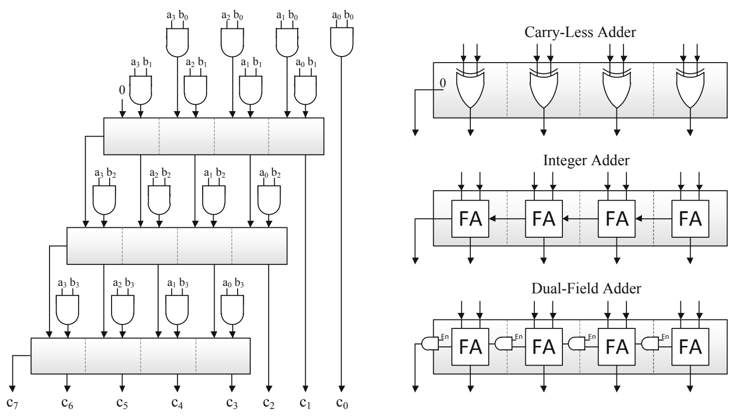


Fig. 1. General 4-bit multiplier structure to the left. Carry-less, integer, and dual-field adder (from top to bottom) on the right.

Table 1 shows the area evaluation of different hardware-multiplier types. We evaluated multipliers for prime-field, binary-field, and dual-field arithmetic for word sizes of 8, 16, 32, and 64 bits (on register-transfer level). For the evaluation we used the UMC-L130 CMOS technology where an AND gate needs 1.25 GEs, a XOR gate needs 2.75 GEs, and a full-adder cell needs 5.5 GEs.

Obviously the area requirement scales quadratically with the given word size and carry-less multipliers provide the lowest area footprint and lowest increase in area for all given word sizes. Runtime approximations for an ECC point multiplication showed that the word size of the carry-less multiplier must be at least 16 bits in order to achieve a sensible runtime.

Next to a carry-less multiplier, an integer multiplier is necessary to provide operations for higher-level protocols (*e.g.* ECDSA). Note that this multiplier is needed only very few times for most protocols (only four prime field multiplications are required for ECDSA signature generation, for instance). Thus, lower word sizes are acceptable since no significant reduction in speed is expected. We therefore decided to implement a 16-bit carry-less multiplier (to provide an appropriate speed for a point multiplication) and an 8-bit integer multiplier instead of a dual-field 16-bit multiplier (which needs 1,946 GEs). This would sum up to 1,226 GEs which is 720 GEs less than for a dual-field multiplier.

Table 1. Area evaluation of different hardware-multiplier types

Finite Field	**Required adder cells per bit**	**8 bit [GE]**	**16 bit [GE]**	**32 bit [GE]**	**64 bit [GE]**
$GF(2^m)$	XOR	211	850	3,389	13,508
$GF(p)$	FA	376	1,616	6,688	26,336
Dual field	AND + FA	458	1,946	8,018	31,514

Table 2. Area evaluation of different 16×128-bit RAM architectures

Type	**Port**	**Storage** [GEs]	**Logic** [GEs]	**Total** [GEs]
Std. cells (registers)	Single	10,281	2,941	13,926
Std. cells (latches)	Single	8,388	3,119	12,221
Macro S-RAM	Dual	-	-	6,737
Macro S-RAM[a]	Single	-	-	6,000
Macro register-file	**Single**	-	-	**2,955**

[a] Approximated based on UMC 180 nm technology.

3.2 The Memory Type and Architecture

One of the most area expensive chip components of ECC processors is the Random Access Memory (RAM). RAM is necessary to store intermediate values (*e.g.* point coordinates during point multiplication $k \cdot P$) and the secret scalar k. The size of the memory varies depending on the requirements of the ECC formulas (the formulas of López Dahab [27] need at least 5 registers of memory for full-precision architectures and 6 registers for multi-precision architectures due to the need of intermediate storage of in-place operations).

In Table 2, we compare different 16×128-bit RAM types concerning their area requirements. We compare standard-cell based implementations with dedicated RAM macro blocks synthesized in CMOS UMC-L130 technology. The standard-cell based RAM implementations (register and latch based) have been designed on RTL-level and synthesized using Cadence RTL compiler [5]. The RAM-macro blocks have been generated using the Standard Memory Compiler FSA0A Memaker 200901.1.1 by the Faraday Technology Corporation [12]. All except of one type of RAM provide a single read-port and a single write-port. There is one S-RAM macro that features a dual-port read/write interface.

It shows that the latch-based RAM is about 12 % smaller than the register-based RAM. This is because the size of a flip-flop is 5 GE and the size of a latch is 4 GE. This 25 % difference in area is debilitated because some additional registers and control logic is required so that the latch-based RAM works the same way as the register-based RAM. Adding a second read port to those RAMs would be relatively cheap in terms of chip area (it would require about 3,000 GEs in addition by introducing a second multiplexer at the output). Note that a dual-port memory would increase the performance of a multi-precision multiplication by a factor of about two.

From the two available single-port RAM macros, the register-file macro is about 50 % smaller than the S-RAM macro. The dual-port S-RAM macro, in contrast, is only 12 % larger than the single-port S-RAM macro, however, it is about 2.3 times larger than the register-file based RAM macro.

The register-file RAM macro provides best performance in our evaluation scenario. We performed several power simulations using Cadence Encounter and obtained similar results for the register-file RAM macro and the standard-cell

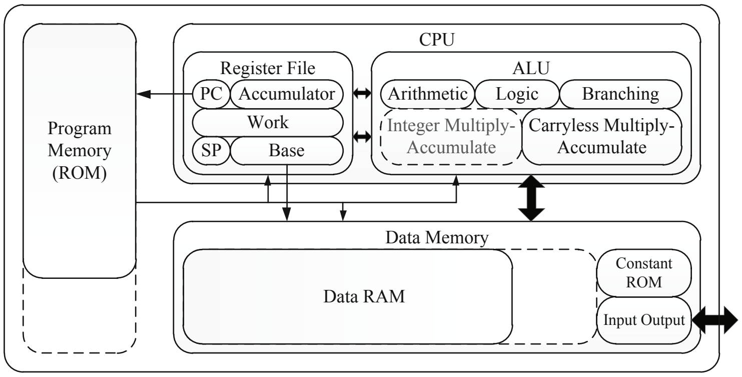


Fig. 2. High-level block diagram of the processor. Components for higher-level protocols are drawn with slashed lines (*i.e.* integer multiplier, program and data memory).

based RAM architectures. The main disadvantages of the register-file macro are the lack of a second read port (speed) and the limit of clock-synchronous read operations. The lack of a second read port can be compensated by using temporary working registers. The lack of an asynchronous read functionality can be balanced with a more difficult control logic.

4 Hardware Architecture

In this section, we introduce the hardware architecture of our processor. It is based on the microprocessor design called Neptun [34], which uses a Harvard architecture. This allows to fetch, decode, execute, and store data within the same clock cycle and allows low-area optimizations due to the choice of different memory types and sizes. Figure 2 shows the block diagram of the architecture. It is mainly composed of a Central Processing Unit (CPU) including register file and Arithmetic Logic Unit (ALU), and memories for program code, constants, and data.

4.1 Central Processing Unit (CPU)

The heart of the processor is the 16-bit CPU. It is composed of several internal registers and an ECC optimized ALU. The register file consists of a program counter (PC), a stack pointer (SP), three base registers, four working registers, and an accumulator register: The program counter is used as index for the program memory. The stack pointer (SP) is needed to store registers on the data memory. The stack is also used to store program-return addresses that are needed for function calls. In order to address certain base addresses within the data memory, three base registers are used. We integrated two source registers and one destination register. They are used together with a 4-bit offset to address data in the memory. The offset address is stored within a program word. We

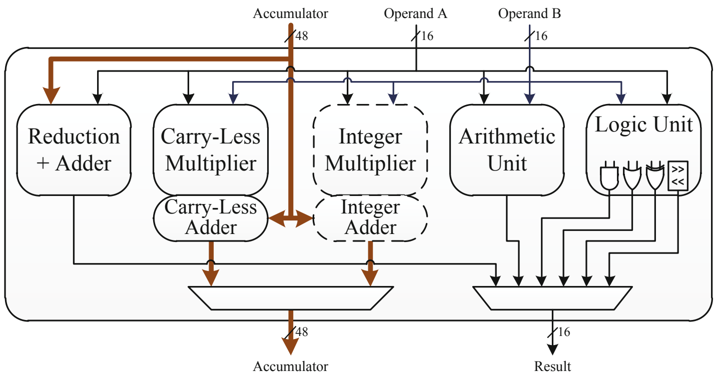


Fig. 3. High-level diagram of the arithmetic logic unit

implemented four 16-bit working registers that can be used as general-purpose registers. The registers are needed for almost any ECC operation and are used to reduce the number of memory-read cycles within the finite-field multiplication. The accumulator register (ACC) is needed for the multiply-accumulate operation of the 163-bit multi-precision multiplication.

We integrated several optimizations to increase the performance of ECC operations. First, the ALU accesses data directly without loading it first into CPU registers (as it is in the case of conventional microcontrollers). In the first clock cycle, the data is addressed in the memory. In the second cycle, the data is processed by the ALU and the result is stored back in memory within the same clock cycle. This increases the performance of memory-access operations significantly. Second, loading and processing of data is done simultaneously by the processor. This avoids unnecessary idle cycles and improves the efficiency of multi-precision arithmetic operations. Those optimizations are described in more detail in [34].

Arithmetic Logic Unit (ALU). The arithmetic logic unit (ALU) mainly consists of a reduction-logic unit, a carry-less multiplier, an arithmetic unit (addition/subtraction), and a logic unit (supporting OR, AND, XOR, and shift operations). For higher-level protocols, an integer multiplier is needed in addition (drawn with dashed lines). Figure 3 shows a high-level diagram of the ALU. We also integrated an operand isolation technique for each submodule which reduces the power-consumption significantly.

4.2 Memory for Program, Data, and Constants

Our processor provides a long-term storage memory that mainly stores the program for ECC point multiplication. The memory provides 72 control signals and contains up to 1,800 entries depending on the implemented algorithms and higher-level protocols. Most of the control signals are used to control the dataflow within the CPU. Best area results have been achieved by directly synthesizing the memory table as Read Only Memory (ROM) using standard cells. Experiments in which a 16-bit instruction set or a ROM macro have been introduced resulted in a larger area requirement.

For short-term data storage, we used a 16-bit RAM macro (register-file based) as discussed in Section 3. Note that in contrast to most processors reported in literature [4,25,26,31], we include the number for the required storage of the secret scalar k. For an ECC point multiplication, 1,296 bits (81 entries) are necessary (we used a 16×84 macro in that case). For higher-level protocols, additional memory is needed (*e.g.* 1,536 bits for ECDSA signature generation (16×96 macro) and 2,384 bits for ECDSA signature verification (16×152 macro)).

ECC constants have been stored in a ROM. The ROM has been implemented as a look-up table and stores between 880 and 2,564 bits such as the x and y coordinate of the base point P, the ECC parameters a and b (see Equation (2)), and the irreducible polynomial $f(z)$.

The input/output of data has been realized via memory mapped I/O. Data can be written and read using a 16-bit parallel interface.

5 Implementation Details

In the following, we give details about the implemented carryless multiply-accumulate unit and the modular arithmetics in order to perform ECC operations.

5.1 Carry-Less Multiply-Accumulate Unit

The multi-precision multiplication over $\mathbb{F}_{2^{163}}$ has been realized following a multiply-accumulate (MAC) approach. There exist several publications that make use of MAC units to increase the performance of modular multiplication (see *e.g.* the work of [9,14,15,17,33]). We implemented the multiplication by a product-scanning form (often referred as Comba multiplication), where each partial product of $A[i] \cdot B[j]$ gets accumulated to a common sum (ACC_1, ACC_0), *i.e.* $(ACC_1, ACC_0) \leftarrow (ACC_1, ACC_0) + A[i] \cdot B[j]$.

Note that for the polynomial MAC unit the handling of carry propagation is not needed. Thus, the accumulator register needs a size of only $(2W - 1)$ bits.

We implemented several improvements to increase the performance. First, the entire multiplication algorithm has been unrolled so that no extra cycles are wasted for loop operations. Second, we reused the working registers as a memory cache to reduce the number of necessary load operations. With each working register used, the total number of read operations has been reduced by about $2N$. Third, we added a third word to the accumulator register (ACC_2, ACC_1, ACC_0) in order to allow efficient reduction of the accumulated sum. Thus, the MAC operation is performed on the words (ACC_2, ACC_1) instead of (ACC_1, ACC_0) and ACC_0 is used to store the previous intermediate result. A detailed description of the reduction method is given in the following subsection.

Algorithm 1. Polynomial multiplication with interleaved reduction

```
Require: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m-1$.
Ensure: $c(z) = a(z) \cdot b(z) \bmod f(z)$.
 1: $ACC \leftarrow 0$
 2: for $i$ from 0 to $N-1$ do
 3:     for each element of $\{(i,j) | i+j=k, 0 \leq i,j \leq N-1\}$ do
 4:         $(ACC_2, ACC_1) \leftarrow (ACC_2, ACC_1) + A[i] \cdot B[j]$.
 5:     end for
 6:     $C[k] \leftarrow ACC_1$.
 7:     $ACC \leftarrow ACC \gg W$.
 8: end for
 9: $ACC \leftarrow higher(ACC)$.
10: for $k$ from $t$ to $2N-2$ do
11:     for each element of $\{(i,j) | i+j=k, 0 \leq i,j \leq t-1\}$ do
12:         $(ACC_2, ACC_1) \leftarrow (ACC_2, ACC_1) + A[i] \cdot B[j]$.
13:     end for
14:     $C[k-N-1] \leftarrow C[k-N-1] + reduce(ACC)$.
15:     $ACC \leftarrow ACC \gg W$.
16: end for
17: $C[N-1] \leftarrow lower(C[N-1]) + reduce(ACC)$.
18: $ACC \leftarrow ACC \gg W$.
19: $C[0] \leftarrow C[0] + reduce(ACC + higher(C[N-1])) \gg W$.
20: $C[N-1] \leftarrow lower(C[N-1]) \gg W$.
21: Return(c).
```

Algorithm 1 shows the algorithm of the implemented polynomial multiplication. The polynomials $a(z)$ and $b(z)$ get multiplied and the reduced result is stored in $c(z)$. In the lines 1 to 8, the lower N words of the result $c(z)$ are calculated. Note that in this phase the ACC_0 register is not used. In line 9, the lower $(m - W(N-1))$ bits of the accumulator need to be cleared. Those are the bits of the results that do not need to be reduced. The lines 10-16 calculate the higher N words of $c(z)$ and reduce them immediately. According to the recommended NIST irreducible polynomial B-163 $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$, the reduction function (line 14) can be written as

$$\begin{aligned} reduce(ACC) = \Big(& ACC \gg (W+3) + ACC \gg W + \\ & ACC \gg (W-3) + ACC \gg (W-4) \Big) \wedge (2^W - 1). \end{aligned} \tag{3}$$

Finally, in lines 17-20 the rest of the accumulator and the higher bits of $C[N-1]$ get reduced.

Polynomial NIST B-163 Reduction Logic. We make use of the recommended NIST irreducible polynomial B-163 to perform a very efficient modular reduction for modular multiplication and squaring. The reduction logic is shown in Figure 4. We hard-wired the output of the appropriate accumulator register according to Equation (3). The reduction logic takes the output of the 48-bit

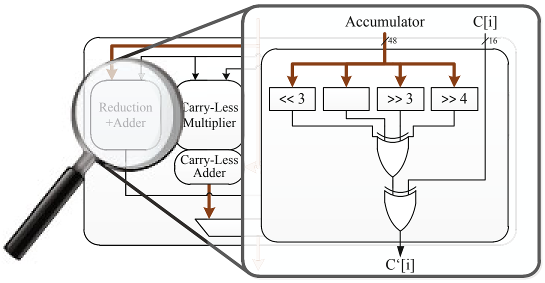


Fig. 4. Using the dedicated reduction logic, the content of the accumulator is reduced and stored in $C'[i]$ (hold in data memory) with index i

accumulator register, performs 4×16 XOR operations and the result is added with the intermediate result $C[i] = C[k - N - 1]$ (see line 14 in Algorithm 1). After the addition (XOR), the variable $C[i]$ is updated with $C'[i]$ in the data memory. Only one clock cycle is needed to reduce the intermediate result of the accumulator and sum of partial products, respectively. Figure 4 shows the dedicated reduction logic.

It should be noted that although the reduction logic has been specially optimized for NIST B-163, the CPU is capable of handling arbitrary irreducible polynomials. Thus requirements such as flexibility and extendability are ensured.

5.2 Modular Arithmetic

Modular Addition. The simplest operation is the modular addition. It is a simple XOR operation. Neither a carry flag nor a finite-field reduction need to be considered. Modular addition over $\mathbb{F}_{2^{163}}$ needs 35 clock cycles on our processor.

Modular Multiplication. Modular multiplication has been realized using the carryless multiply-accumulate unit described in Section 5.1. Our processor needs 222 clock cycles for a 163-bit multiplication.

Modular Squaring. Modular squaring can be performed very efficiently. The binary representation of the polynomial can be easily squared by inserting a 0 between each consecutive bit of the polynomial, e.g. $a(z) = a_{m-1}z^{m-1} + \cdots + a_2z^2 + a_1z + a_0$ would results in $a(z)^2 = a_{m-1}z^{2m-2} + \cdots + a_2z^4 + a_1z^2 + a_0$. This can be realized with only a few additional hardware components. The polynomial-reduction logic can be reused for squaring. One modular squaring needs 41 clock cycles on our processor and thus is 5.4 times faster than a modular multiplication.

Modular Inversion. Modular inversion is required to transform the projective coordinates back into affine. For this operation, we made use of Fermat's little theorem [20] that states that $a = a^{2^m} \bmod f(z) \, \forall a \in \mathbb{F}_{2^m}$. As a result, $a^{-1} \equiv a^{2^m-2} \bmod f(z)$. This exponentiation can be performed using 162 squaring and only 9 multiplications for the NIST B-163 binary field. As a result 11,031 cycles are needed for an inversion.

Table 3. Size and power estimations of our processor for different CMOS technologies using Latch-based RAMs

Technology	**Area** $[\mu m^2]$	**NAND Gate** $[\mu m^2]$	**Total Area** $[GE]$	**Power** $[\mu W$@1MHz$]$	**Leakage** $[\mu W]$
AMS c35b4	693,948	54.600	12,710	696.3	0.63
UMC f180GII	139,469	9.374	14,878	107.1	0.53
UMC f130SP	71,745	5.120	14,013	31.4	1.37
UMC f090SP	39,550	3.136	12,612	70.1	54.32

6 Results

We synthesized our processor using different CMOS technologies from various manufacturers. For synthesis, we used the Cadence RTL compiler [5] Version v08.10. Table 3 shows the total area and power-consumption estimation of the processor using latch-based RAMs[1] (described in Section 3.2). The power-consumption estimations were made using Cadence Encounter Version v08.10. All obtained area results are within a 20 % margin. In view of power consumption, best performance had been obtained for the UMC-L130 technology. For all following approximations we used register-based RAM macros.

In Table 4, the area and power requirements for individual chip components are listed. The memory needs most of the area which is 5,399 GEs. The CPU needs 3,556 GEs in total where only 849 GEs are used for the carry-less multiplier. The total size of the processor sums up to 8,958 GEs.

In Table 5, we compare our results with related work. There exist many publications of ECC processors over $\mathbb{F}_{2^{163}}$. Most of those processors use full-precision arithmetic to perform the point multiplication. For a fair comparison, we listed the results of the authors for different digit sizes (d=1...8). All implementations need between 10,392 GEs and 16,247 GEs of chip area and between 47 and 430 kcycles for the computation of $k \cdot P$. Our implementation needs 8,958 GEs of area which is 1,434 GEs less area than the best reported solution. This is an area improvement by about 14 %. The number of needed clock cycles can be compared with the full-precision solutions with d=1. The power and energy consumption is very low and fulfills most requirements of embedded-system designs.

6.1 Results for Higher-Level Protocol Implementations

As a higher-level protocol, we implemented the Elliptic Curve Digital Signature Algorithm (ECDSA) [30]. In addition to a point multiplication over the binary field $\mathbb{F}_{2^{163}}$, ECDSA needs a hash function and several prime-field arithmetic operations to generate and verify a digital signature. As a hash function, we implemented the 160-bit SHA-1 algorithm according to ISO/IEC FIPS-180-3 [29]. Replacing the SHA-1 algorithm with one of the current SHA-3 candidates [19]

[1] We did not have access to RAM macros for all those technologies.

Table 4. Size and power consumption of individual chip components

Component	**Area** [GE]	**Area** [%]	**Power** [μW@1MHz]	**Power** [%]
Memory	5,399	60.27	11.57	35.77
Program memory	2,471	27.58	4.24	13.10
Data RAM	2,528	28.22	4.66	14.41
Constant ROM	256	2.56	1.62	5.01
CPU	3,556	39.70	18.93	58.54
ALU	1,837	20.51	11.05	34.16
Carry-less multiplier	849	9.48	2.30	7.12
Logic unit	348	3.88	2.15	6.65
Arithmetic unit	93	1.04	0.37	1.15
Register Set	875	9.77	1.48	4.58
Total Area	**8,958**	**100.00**	**32.34**	**100.00**

would be easily possible. For prime-field multiplications and inversion, we decided to implemented Montgomery-arithmetic operations. We implemented the Finely Integrated Product Scanning Form (FIPS) according to Koç et al. [24]. The algorithm is used only four times, so we optimized the code for low area (no loop unrolling etc.). Furthermore, we implemented the Montgomery-inversion algorithm according to Kalinski et al. [22].

For signature verification, we applied Shamir's trick [7,10] to improve the performance of multiple-point multiplication. All described operations for ECDSA have been implemented as Assembler functions for our processor and have been stored in program memory. Table 6 shows the results after synthesizing the processor.

Table 5. Comparison with related work

Related Work	**Area** [GE]	**Cycles** [kCycles]	**Power** [μW@1MHz]	**Energy** [μJ]	**CMOS Technology**
Kumar06 d=1 [25]	15,094	430	-	-	AMI C35
Batina06[a] d=4 [2]	14,816	95	27.00	2.57	130 nm
Batina06[a] d=3 [2]	14,258	125	27.00	3.38	130 nm
Batina06[a] d=2 [2]	13,681	182	27.00	4.91	130 nm
Batina06[a] d=1 [2]	13,104	354	27.00	9.56	130 nm
Bock08 d=8 [4]	16,247	47	148.76	6.99	INF SRF55V01P
Bock08 d=4 [4]	12,876	80	93.27	7.46	INF SRF55V01P
Bock08 d=1 [4]	10,392	280	54.31	15.21	INF SRF55V01P
Lee08 d=4 [26]	15,356	79	37.39	2.95	UMC L130
Lee08 d=3 [26]	14,729	101	38.32	3.87	UMC L130
Lee08 d=2 [26]	14,064	145	36.52	5.30	UMC L130
Lee08 d=1 [26]	12,506	276	32.42	8.95	UMC L130
Hein08 16-bit [17]	11,904	296	101.87	30.15	UMC L180
This work 16-bit	**8,958**	**286**	**32.34**	**9.25**	**UMC L130**

[a] For a fair comparison a RAM approximated with 4,890 GE was added. The power values lack the power consumption of this RAM.

Table 6. Area and power estimations of our processor supporting ECDSA

Program	**Area** [GE]	**Cycles** [kCycles]	**Lines of Code**	**Power** [μW@1MHz]	**Energy** [μJ]
ECC Only	8,958	294	637	32.09	9.43
ECC Protected[a]	9,728	298	828	32.48	9.68
ECDSA Sign[a,b]	15,387	378	1771	41.11	15.54
ECDSA Verify[b]	16,005	605	1784	40.76	24.66

[a] The numbers include y-recovery, randomized projective coordinates (RPC) side-channel countermeasure [6], and ECC point-validity check [8].

[b] Includes the SHA-1 hash function [29], Random Number Generation (RNG) [30], and prime-field arithmetics.

For ECDSA signature generation, our processor needs 15,387 GEs which outperforms existing solutions in terms of area, power, and speed [13,18,34,35]. Signature verification can be realized using a chip area of 16,005 GEs.

7 Conclusions

In this paper, we presented a low-resource implementation of an ECC hardware processor. The processor needs 8,958 GEs and performs a point multiplication within 285 kcycles. The power consumption is about 3.2 μW at 100 kHz. We met the low-resource constraints of embedded systems by applying a very modular microcontroller architecture that allows the execution of higher-level protocols like ECDSA. The elliptic-curve operations have been performed over the NIST $\mathbb{F}_{2^{163}}$ elliptic curve using multi-precision arithmetic. The outcome improves the state of the art in low area ECC hardware designs and provides even a smaller area footprint than most of the proposed SHA-3 candidates [19].

Acknowledgements. We would like to thank Mario Kirschbaum and Martin Feldhofer for several fruitful discussions.

This work has been supported by the Austrian Government through the research program FIT-IT Trust in IT Systems under the project number 825743 (project PIT).

References

1. Avanzi, R.M., Cohen, H., Doche, C., Frey, G., Lange, T., Nguyen, K., Vercauteren, F.: Handbook of Elliptic and Hyperelliptic Curve Cryptography. Chapman & Hall/CRC (2005)
2. Batina, L., Mentens, N., Sakiyama, K., Preneel, B., Verbauwhede, I.: Low-Cost Elliptic Curve Cryptography for Wireless Sensor Networks. In: Buttyán, L., Gligor, V.D., Westhoff, D. (eds.) ESAS 2006. LNCS, vol. 4357, pp. 6–17. Springer, Heidelberg (2006)

3. Blake, I.F., Seroussi, G., Smart, N.P.: Elliptic Curves in Cryptography. London Mathematical Society Lecture Notes Series, vol. 265. Cambridge University Press, Cambridge (1999)
4. Bock, H., Braun, M., Dichtl, M., Hess, E., Heyszl, J., Kargl, W., Koroschetz, H., Meyer, B., Seuschek, H.: A Milestone Towards RFID Products Offering Asymmetric Authentication Based on Elliptic Curve Cryptography. Invited talk at RFIDsec 2008 (July 2008)
5. Cadence Design Systems. The Cadence Design Systems Website, http://www.cadence.com/
6. Coron, J.-S.: Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999)
7. de Rooij, P.: Efficient Exponentiation Using Precomputation and Vector Addition Chains. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 389–399. Springer, Heidelberg (1995)
8. Ebeid, N., Lambert, R.: Securing the Elliptic Curve Montgomery Ladder Against Fault Attacks. In: Proceedings of Workshop on Fault Diagnosis and Tolerance in Cryptography - FDTC 2009, Lausanne, Switzerland, pp. 46–50 (September 2009)
9. Eberle, H., Gura, N., Shantz, S.C., Gupta, V., Rarick, L.: A Public-key Cryptographic Processor for RSA and ECC. In: Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2004), pp. 98–110. IEEE Computer Society (September 2004)
10. El Gamal, T.: A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In: Blakely, G.R., Chaum, D. (eds.) CRYPTO 1984. LNCS, vol. 196, pp. 10–18. Springer, Heidelberg (1985)
11. Fan, J., Guo, X., Mulder, E.D., Schaumont, P., Preneel, B., Verbauwhede, I.: State-of-the-Art of Secure ECC Implementations: A Survey on known Side-Channel Attacks and Countermeasures. In: Proceedings of 3rd IEEE International Symposium Hardware-Oriented Security and Trust - HOST 2010, California, USA, June 13-14, pp. 76–87. IEEE (2010)
12. Faraday Technology Corporation. Faraday FSA0A_C 0.18 μm ASIC Standard Cell Library (2004), http://www.faraday-tech.com
13. Fürbass, F., Wolkerstorfer, J.: ECC Processor with Low Die Size for RFID Applications. In: Proceedings of 2007 IEEE International Symposium on Circuits and Systems. IEEE (May 2007)
14. Großschädl, J.: Full-Custom VLSI Design of a Unified Multiplier for Elliptic Curve Cryptography on RFID Tags. In: Bao, F., Yung, M., Lin, D., Jing, J. (eds.) Inscrypt 2009. LNCS, vol. 6151, pp. 366–382. Springer, Heidelberg (2010)
15. Großschädl, J., Kamendje, G.-A.: Optimized RISC Architecture for Multiple-Precision Modular Arithmetic. In: Hutter, D., Müller, G., Stephan, W., Ullmann, M. (eds.) SPC 2008. LNCS, vol. 2802, pp. 253–270. Springer, Heidelberg (2004)
16. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer, Heidelberg (2004)
17. Hein, D., Wolkerstorfer, J., Felber, N.: ECC Is Ready for RFID – A Proof in Silicon. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) SAC 2008. LNCS, vol. 5381, pp. 401–413. Springer, Heidelberg (2009)
18. Hutter, M., Feldhofer, M., Plos, T.: An ECDSA Processor for RFID Authentication. In: Ors Yalcin, S.B. (ed.) RFIDSec 2010. LNCS, vol. 6370, pp. 189–202. Springer, Heidelberg (2010)
19. IAIK. Hash Function Zoo, http://ehash.iaik.tugraz.at/index.php/HashFunctionZoo

20. Itoh, T., Tsujii, S.: Effective recursive algorithm for computing multiplicative inverses in $GF(2^m)$. Electronic Letters 24(6), 334–335 (1988)
21. Joye, M., Yen, S.-M.: The Montgomery Powering Ladder. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 291–302. Springer, Heidelberg (2003)
22. Kaliski, B.: The Montgomery Inverse and its Applications. IEEE Transactions on Computers 44(8), 1064–1065 (1995)
23. Koblitz, N.: A Course in Number Theory and Cryptography. Springer, Heidelberg (1994) ISBN 0-387-94293-9
24. Koç, Ç.K., Acar, T., Kaliski Jr., B.S.: Analyzing and Comparing Montgomery Multiplication Algorithms. IEEE Micro 16(3), 26–33 (1996)
25. Kumar, S.S., Paar, C.: Are standards compliant Elliptic Curve Cryptosystems feasible on RFID? In: Workshop on RFID Security 2006 (RFIDSec 2006), Graz, Austria, July 12-14 (2006)
26. Lee, Y.K., Sakiyama, K., Batina, L., Verbauwhede, I.: Elliptic-Curve-Based Security Processor for RFID. IEEE Transactions on Computers 57(11), 1514–1527 (2008)
27. López, J., Dahab, R.: Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 316–327. Springer, Heidelberg (1999)
28. Montgomery, P.L.: Speeding the Pollard and Elliptic Curve Methods of Factorization. Mathematics of Computation 48(177), 243–264 (1987) ISSN 0025-5718
29. National Institute of Standards and Technology (NIST). FIPS-180-3: Secure Hash Standard (October 2008), http://www.itl.nist.gov/fipspubs/
30. National Institute of Standards and Technology (NIST). FIPS-186-3: Digital Signature Standard, DSS (2009), http://www.itl.nist.gov/fipspubs/
31. Öztürk, E., Sunar, B., Savaş, E.: Low-Power Elliptic Curve Cryptography Using Scaled Modular Arithmetic. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 92–106. Springer, Heidelberg (2004)
32. Sakiyama, K., Batina, L., Mentens, N., Preneel, B., Verbauwhede, I.: Small-footprint ALU for public-key processors for pervasive security. In: Workshop on RFID Security 2006 (RFIDSec 2006), Graz, Austria, July 12-14 (2006)
33. Tillich, S., Großschädl, J.: VLSI Implementation of a Functional Unit to Accelerate ECC and AES on 32-Bit Processors. In: Carlet, C., Sunar, B. (eds.) WAIFI 2007. LNCS, vol. 4547, pp. 40–54. Springer, Heidelberg (2007)
34. Wenger, E., Feldhofer, M., Felber, N.: Low-Resource Hardware Design of an Elliptic Curve Processor for Contactless Devices. In: Chung, Y., Yung, M. (eds.) WISA 2010. LNCS, vol. 6513, pp. 92–106. Springer, Heidelberg (2011)
35. Wolkerstorfer, J.: Is Elliptic-Curve Cryptography Suitable for Small Devices? In: Workshop on RFID and Lightweight Crypto, Graz, Austria, July 13-15, pp. 78–91 (2005)

A Statistics for ECC Multiplication

During the development of the ECC and ECDSA functions we used a statistics feature of our tool-chain to investigate the code-line and cycle consumption of each function. Table 7 shows the number of times each function is called, the size of each function in code lines and the total runtime of each function. Even though the multiplication algorithm is optimized down to 222 cycles it still covers 74 % of the total runtime.

Table 7. Functions used during ECC point multiplication with y-recovery and point-validity check

Function	**Calls**	**Code Lines**	**Cycles**
B163.Multiplication	990	222	219,780
B163.Square	969	41	39,729
B163.Add	490	35	17,150
PointOperation.Multiplication	1	148	16,636
B163.FermatInverseHelp	7	31	2,041
Utilities.Copy	16	24	384
PointOperation.yRecovery	1	90	90
B163.FermatInverse	1	88	88
PointOperation.isValidPoint	1	44	44
Utilities.CMP	1	35	35
Utilities.Clear	2	13	26
TOTAL	**2,479**	**771**	**296,003**
TOTAL including test functions	**2,480**	**828**	**296,547**

Table 8 shows how often each and every type of instruction is used. The parallelized commands are a combination of other commands. They cover 71 % of the total runtime. Note that only 4.4 % of the total runtime is used for program-flow instructions such as RET, CALL, BRA, and JMP. This overhead would not exist if a dedicated state machine instead of a CPU with instruction set would be used.

Table 8. Instructions used during an ECC point multiplication with y-recovery and point-validity check

Mnemonic	Description	CPI	Cycles	Used
PAR: BMULACC | LD		1	109,869	111
PAR: MOVNF | LD		1	65,707	83
LD	Load from memory	1	35,410	65
PAR: BREDUCE_ADD_ST | BRSACC		1	13,818	14
PAR: BMULACC | ST | BRSACC		1	11,859	12
PAR: BREDUCE_ADDBYTE_ST | BRSACC		1	9,690	10
CALL	Call a function	3	7,440	70
LDI	Load Immediate	1	6,900	105
AND	Logic AND	1	6,038	7
PAR: XOR | ST		1	5,390	11
MOVNF	Copy register to register without flag update	1	5,269	47
RET	Return from function	2	4,960	13
BMULACC	Binary multiply-accumulate	1	3,876	4
STR	Store a register to memory	1	2,725	32
XOR	Logic XOR	1	2,120	3
LDR	Load from memory and store to register	2	1,942	10
ADDI	Add with carry	1	573	11
BRA	Branch if flag is set/cleared	1	488	6
PUSH	Push a value to the stack	2	338	5
POP	Pop a value from the stack	2	336	4
LSI	Left shift by immediate	1	326	4
SUBI	Subtract with carry	1	324	6
ADD	Add	1	163	2
RS	Right shift	1	163	2
RSI	Right shift immediate	1	163	2
SUB	Subtract	1	163	2
ASRI	Arithmetic shift right	1	161	1
JMP	Jump to address	1	160	1
CMP	Compare	1	84	2
MOV	Copy register to register	1	82	1
CMPC	Compare with carry	1	10	10
TOTAL			**296,547**	**656**

Memory Encryption for Smart Cards

Barış Ege[1], Elif Bilge Kavun[2], and Tolga Yalçın[2]

[1] Institute of Applied Mathematics,
Middle East Technical University, Turkey
baris.ege@metu.edu.tr

[2] Chair of Embedded Security, HGI,
Ruhr University Bochum, Germany
{elif.kavun,tolga.yalcin}@rub.de

Abstract. With the latest advances in attack methods, it has become increasingly more difficult to secure data stored on smart cards, especially on non-volatile memories (NVMs), which may store sensitive information such as cryptographic keys or program code. Lightweight and low-latency cryptographic modules are a promising solution to this problem. In this study, memory encryption schemes using counter (CTR) and XOR-Encrypt-XOR (XEX) modes of operation are adapted for the target application, and utilized using various implementations of the block ciphers AES and PRESENT. Both schemes are implemented with a block cipher-based address scrambling scheme, as well as a special write counter scheme in order to extend the lifetime of the encryption key in CTR-mode. Using the lightweight cipher PRESENT, it is possible to implement a smart card NVM encryption scheme with less than 6K gate equivalents and zero additional latency.

Keywords: memory encryption, smart card, low-latency block cipher, AES, PRESENT.

1 Introduction

Smart cards and devices containing smart card ICs, have been playing increasingly important roles in our daily lives for years. Within the last decade, passports, credit cards, ID cards, and even public transport tokens have come to rely on smart card technologies. Non-volatile memories (NVMs) in those smart cards contain frequently updated data such as personal information about the card holder, and less frequently (or never) changing data such as program code, cryptographic keys, etc. Access to the frequently updated data should be executed with very low latency. Due to advances in attack methods, e.g., invasive attacks, it is often realistic to expect that the information would be compromised in case of a stolen or lost card. Therefore it is crucial to store the information securely using state-of-the-art cryptographic algorithms and technologies, and yet in an economic way.

This requires some form of data encryption algorithm to be implemented on the smart card, either in hardware or software. However, encryption (and decryption) of memory data comes with a specific set of requirements, which are not met with standard cipher implementations. For instance, while AES is extremely well tested and secure, it requires either several tens of thousands of gates for a fast hardware implementation, or several hundreds of cycles for software implementation on a standard embedded processor. On the other hand, mechanisms that allow access to data stored on the NVM of a smart card should be lightweight in terms of power consumption, latency and resource usage. This enforces utilization of specifically optimized algorithms and modes.

The memory encryption schemes proposed up to date mainly target real-time hard disk encryption [1,2,3,4,5,6,7], which differs considerably from the security in smart cards. They require lots of temporary storage for predictive processing of data, which makes the overall process seem like zero-latency to the external user.

On smart cards, NVM sizes are usually restricted. Even the future visions for the next two decades predict at most a few gigabytes of NVM storage[8]. In most cases, NVM also stores program memory, which has to be accessed with very low latency for seamless execution of the program. Standard memory encryption techniques usually require several clock cycles in order to decrypt the page header, calculate the tweak, and then use this information to further decrypt consecutive block data. This is unacceptable for low latency access demanding smart card NVM applications. Therefore, either the existing techniques have to be improved, or new memory encryption techniques have to be introduced together with supporting cryptographic primitives.

When considering security of smart card NVMs, the following assumptions have to be considered:

- An adversary can read the raw contents of the NVM at any time (which might be possible with either invasive attacks or attacks targeting the access mechanism).
- An adversary can fool the smart card to encrypt and store arbitrary data of his/her choosing on the NVM.
- An adversary can modify unused sectors on the NVM and then request their decryption.
- Furthermore, most data stored on the NVM has a predefined structure, such as headers, footers, fixed size and address, etc.

In order to resist such an adversary, the security module on the smart card should be able to efficiently encrypt memory contents of NVM and perform memory address scrambling. Scrambling (re-mapping of the address space) prevents the adversary from locating the exact location of the stored data. It is a crucial element of memory encryption, without which most memory encryption schemes become pointless.

The memory encryption on smart cards should be low-cost, security proven, low-latency, and low-complexity. In this work, we this target is accomplished by adapting CTR and XEX-modes of operation for memory encryption, and realizing them using AES[9] and PRESENT[10] block ciphers. The proposed systems are implemented at RTL level, synthesized using standard CMOS technology, and their performance figures are presented.

The rest of the paper is organized as follows: In section 2, an overview of existing and proposed memory encryption schemes is presented together with security discussion and performance analysis of two selected schemes for smart card NVM encryption. System design details and performance results are presented in section 3. Finally in section 4, the results are summarized.

2 Memory Encryption

2.1 Memory Encryption Background

Like most other fixed width data applications, memory encryption also relies on use of block ciphers. However, using a block cipher as the cryptographic primitive for memory encryption makes sense only when it is used with a proper mode of operation. In addition to the conventional counter mode (CTR), new modes of operation specific to memory encryption have been proposed: namely Liskov-Rivest-Wagner (LRW), XOR-Encrypt-XOR (XEX)[7] and XEX-based Tweaked CodeBook mode with ciphertext stealing (XTS)[6]. All three proposed schemes (LRW, XEX, XTS) depend on the tweakable block cipher idea in [11], which is composed of a block cipher and a tweak. This system is proven to be secure against the chosen-ciphertext and chosen-plaintext attack scenarios, provided that the number of plaintext-ciphertext pairs is limited. Basically, a pseudo-random tweak is computed for each block and then XORed with the input and output of the cipher, thereby randomizing any structure in the plaintext and ciphertext and thwarting attacks exploiting these structures. The CTR, LRW, XEX, and XTS modes of operation are briefly summarized below:

CTR-Mode. CTR-mode mode enables the user to have parallel encryption and random access to the encrypted data. This makes it a viable candidate for memory encryption. However, it was shown that this mode is insecure when more than $2^{\frac{n}{2}}$ blocks of data are encrypted with a block cipher with block size n. The idea in the counter mode is to produce a one-time pad (OTP) using the underlying block cipher and XOR the plaintext with this OTP. This approach is quite problematic when used recklessly in a memory encryption scheme, as will be seen in Section 2.3.

LRW-Mode. LRW-mode randomizes the input and output of the block cipher by XORing both with a tweak value, i.e. $C = E_{K_1}(P \oplus T) \oplus T$, where $T = K_2 \otimes I$ ($\otimes$ being multiplication over GF(2^n). Here, K_1 is the key to the block cipher, K_2 is a key of the same size as the block size of the underlying block cipher and I is the counter/index of the data to be encrypted. This mode of operation is directly applicable to memory encryption when I is used as the address of the block to be encrypted. However, for non-sequential data, a full GF(2^n) multiplier is required.

XEX-Mode. Proposed by Rogaway in 2004 [7], XEX-mode enables the user to have an easier computation of the tweak T (Figure 1), i.e. $T = E_{K_1}(N) \otimes 2^I$. As a result, a full finite field multiplier is not required. Tweak computation gets even easier when encrypting sequential data, where it becomes simple doubling over GF(2^n).

XTS-mode. XTS-mode is standardized as IEEE standard 1617 for Cryptographic Protection of Data on Block-Oriented Storage Devices in 2008 [6]. It is basically the same as the XEX-mode. However, a second key, K_2, is used in tweak compution, i.e. $T = E_{K_2}(N) \otimes 2^I$.

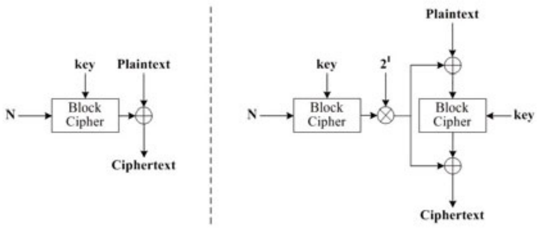


Fig. 1. Graphical illustration of the CTR and XEX-modes of operation

2.2 Previous Work

The previously proposed schemes for memory encryption mostly target everyday PCs or even larger systems, and they use large amounts of resources. In [1], CRYPTOPAGE extension of the HIDE infrastructure is proposed in addition to the address bus protection, memory encryption and memory checking which are combined in a way to provide a very low performance overhead. [2] presents predecryption as a method of providing security with less overhead by using well-known prefetching techniques to collect data from memory and perform decryption before it is needed by the processor. Their results show no increase in execution time despite an extra 128 cycle decryption latency per memory block access. In [4,3], a technique is applied to hide the latency overhead of memory decryption (which is encrypted in CTR-mode) by predicting the sequence number and precomputing the OTP. This technique solves the latency problem by using idle decryption engine cycles to speculatively predict and precompute OTPs before the corresponding sequence number is loaded. Also, an adaptive OTP prediction technique is presented to further improve OTP prediction and precomputation mechanism. This scheme is not only able to predict encryption pads associated with static and infrequently updated cache lines, but also the frequently updated cache lines as well. [4] presents new hardware mechanisms for memory integrity verification and encryption. The integrity verification mechanism offers better performance when the checks are infrequent as in grid computing applications, and the encryption mechanism improves the performance in all cases. [12] shows a hardware implementation of an execute-only memory (XOM) form which allows instructions stored in memory to be executed but not manipulated in another way. In this work, all data that leaves the machine is encrypted, becuse the external memory is assumed to be insecure. However, for efficient operation, hardware assist to provide fast symmetric ciphers is also required. Some other works on memory-bus encryption [13,14] mention hardware engines for bus encryption. [13] describes an engine called "Parallelized Encryption and Integrity Checking Engine (PE-ICE)", which guarantees the confidentiality and integrity of data exchanged between a system-on-chip (SoC) and its external memory. This approach is based on an existing block-encryption algorithm, to which the integrity checking capability is added. According to [13], it results in low performance overhead. In [14], a comprehensive survey on existing techniques for hardware engines which are used in bus encryption is presented.

Apart from these, there exists proprietary data bus encryption algorithms for on-chip NVMs in use by several chip vendors. However, these are not publicly accessible, and to the best of our knowledge, unlike us, none of the existing schemes focus on the cipher block in order to come up with a lightweight and cacheless solution.

2.3 Memory Encryption System Design Issues

So far, we have covered previously proposed modes of operation for memory encryption in the literature. In this study, our main concern is encyption on smart card NVMs. For this purpose, we choose two target modes, CTR and XEX, basically due to the simplicity and low implementation cost of CTR-mode (which make it weak in terms of security), and solid security proof and single key requirement of XEX-mode. In the rest of this section, we shall investigate design and implementation issues for both modes. Additionaly we shall introduce a simple address scrambling scheme, which is a practical requirement for all memory encryption systems.

In CTR-mode, the address α is encrypted using the encryption key, generating the OTP. Part or whole of OTP is XORed with the plaintext, P, or the ciphertext, C, resulting in ciptertext or plaintext outputs, respectively. In cases where OTP size is equal to or more than twice the plaintext/ciphertext block size (i.e. using a 64-bit block cipher with 32-bit memory), it is possible to use only part of OTP to encrypt/decrypt data.

This is not the case for XEX-mode, where a data pair (or even quartet) is required in order to be able to decrypt only a single word within the memory. Referring to the same 64-bit block cipher with 32-bit memory case, we see that updating an address within memory requires first decryption of the old contents of the target address together with the contents of its neigbouring address (which together form a 64-bit block), then writing to both addresses after encryption. This scheme aggravates the average latency. Furthermore, XEX requires both encryption and decryption modules, or a combined module in order to reduce area at the cost of further additional latency.

Clearly, counter mode presents a much simpler and compact solution, and a lot less implementation problems to deal with. This, of course, comes at the cost of lower security with respect to that of XEX-mode. This problem can be remedied with an address scrambling scheme, as we shall see in the next subsection.

Furthermore, we modify the CTR-mode in order to have a cryptographically sound scheme, and introduce a write counter as explained later.

2.4 Address Scrambling

As mentioned before, address scrambling is an integral element of memory encryption and should be handled with care. The most important aspect to address scrambling is that the order of write addresses should not have any visible structure, or in other words, the order that the data is written to the memory should look random to an adversary. In this work, a key dependent address scrambling is proposed using a small scale block cipher. A small scale variant of the well known and secure lightweight block cipher PRESENT[15] is used for this purpose.

In our sample system, 16-bit memory address is encrypted via 4 rounds of the variant cipher, and the resultant ciphertext is used as the scrambled memory address. Simulations show that the auto-correlation of the 4-round output of the 16-bit version of the block cipher PRESENT gives a satisfactory distribution (see Figure 2). It is possible to store the regular 80-bit key and perform key expansion for each of the 4-rounds. Alternatively, 16-bit keys for the 5 rounds (which also sum up to 80-bits) can be stored directly. This way, not only the key expansion logic can be strip off the design, but completely random keys without any dependency in any round can be stored at no addition cost. Naturally, address scrambling requires $(4 + 1) \times 16 = 80$ bits of additional registers for key storage. It is also possible to use part or whole of data encryption keys for address scrambling. However, this requires careful investigation of the security implications.

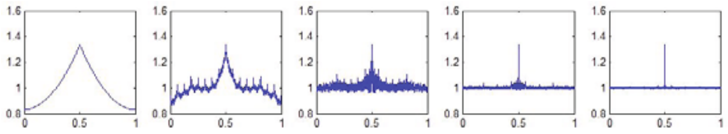


Fig. 2. Auto correlation of the input and the outputs of the small scale PRESENT for rounds 1 to 4

2.5 Performance Issues

In this subsection, we discuss how the performances of CTR and XEX-modes in terms of average latency per memory operation are affected by the cipher latency and the page size (or packet size in case of CTR). First, let's briefly discuss issues with both modes.

Issues with CTR. In CTR-mode, the memory is divided into blocks that are identified by unique headers. For each block, a write counter is used. This scheme works as follows: Target write address width, which is the input to the encryption, is for all practical purposes much smaller than the cipher block size.

Instead of using all zeros to pad it, a predefined address field (for example, most significant 5-bits) is reserved as the write counter. At each update of the block data (i.e. writing of the whole block), the address counter is incremented by one, resulting in a completely new OTP for the same addresses with respect to previous writes. In order to be able to decrypt a block, the header address is decrypted first, starting with zero write counter value, and incrementing it by one until the correct header value is read. The resultant write counter is used for the decryption of the rest of the data within the same memory block. This way, the uniqueness of the IV of each encryption is guaranteed. Although this is an expensive solution in terms of latency, larger packet sizes make up for the initial write counter value search delay.

Issues with XEX. Memory encryption using XEX-mode of operation requires the memory to be divided into pages, so that only one address encryption is enough for each page. The rest of the page is encrypted using the same encrypted page address information and the index of the word within the page (see Figure 1). However, XEX requires both encryption and decryption, which results in doubling of the hardware area of the module, or doubling of the latency in case a single module is used for both operations.

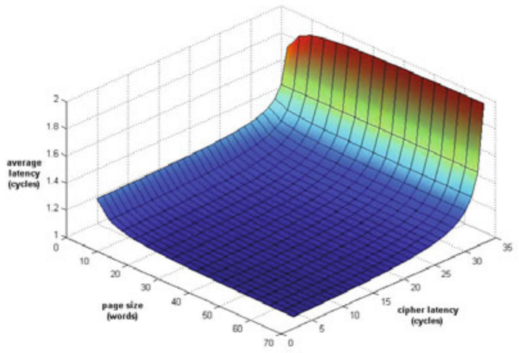


Fig. 3. Average latency of the proposed XEX scheme as a function of page size and cipher latency

For the evaluation of the performance, we assume a generic block cipher which can have up to 32 cycles latency to complete a single encryption/decryption operation. For simplicity, memory word width is assumed to be equal to the cipher block width. In our MATLAB model, we sweep the memory page size (packet size for CTR) from 1 to 64 words, and the cipher latency from 1 to 32, and then measure the average latency per memory operation (assuming equal number of reads and writes). Our simulations clearly indicate that cipher latency has a much higher impact than page and packet sizes over average latency (see Figure 3). As seen in the figure, page size has a significant effect only for very small values. As it goes above 10-15 words, its effect diminishes. However, the effect of cipher latency on the average latency increases almost exponentially. It should also be noted that the figure only shows the average latency for XEX-mode. In CTR-mode, unless the write counter mode is implemented, the average latency is equal to the cipher latency. Clearly, it is imperative that the cipher latency should be kept at minimum.

2.6 System Performance and Block Cipher Selection

Simulation results show that the best way to decrease average latency in memory encryption is to have a low latency cipher. This can be achieved in a number of ways. One is to simply design a low-latency block cipher from scratch, but this requires extensive analysis, expertise and time. Another approach would be to take a well analyzed block cipher and use reduced versions of that cipher, in terms of the number of rounds it requires to encrypt a message block. Although this approach is relatively more plausible, the security analysis done on the full cipher does not directly apply to the reduced round versions of it.

Another approach to achieve low-latency encryption/decryption is to implement a cascade block cipher, i.e. to execute more than one round in one cycle. In the remainder of this work, we implement several rounds of a block cipher (r rounds in one cycle) in order to come up with $\frac{n}{r}$ rounds for the encryption process. Inevitably, this approach brings a trade-off between the latency in terms of the number of rounds an encryption requires and the logical delay of the circuit.

The overall path delay is naturally a function of the underlying cipher function used. In our system, we consider AES and PRESENT as both of these block ciphers have gone through extensive cryptanalysis and remain unbroken. AES-128 requires only 10 rounds per encryption/decryption, but it requires relatively large area in hardware. On the other hand, although it requires 31-rounds, PRESENT-80 is a much more compact cipher, and therefore one can push the cascade implementation to the limit with relatively low area/dealay overhead. Implementation aspects and simulation results for cascade AES and PRESENT are explained in detail in Section 3.

3 Implementation Aspects and Simulations

In this study, we have decided to focus on two candidate systems, XEX and CTR, for reasons explained before. While XEX mode offers higher security compared to CTR mode, it also results in a more complex structure. Most important of all, XEX requires use of both encryption and decryption cores, and therefore is not very suitable for lightweight purposes. In addition, it is not suitable for low-latency requirements, because using 64 and/or 128-bit cipher with 32-bit memories requires reading (and therefore decryption) of neighboring words in order to form the 64/128-bit complete blocks for each write operation, and noticeably increases the average latency. However, CTR system presents more suitable results for both lightweight and low-latency purposes.

In order to give a comparison of both schemes, XEX and CTR-based systems are designed separately. In the first step, variants of AES and PRESENT ciphers, which are basic building blocks for both systems, are implemented in order to obtain performance figures. Furthermore, a reduced round and reduced width version of the PRESENT cipher is utilized as the memory address scrambler block.

Following this step, an XEX-based system is presented together with performance figures. The same is done for a CTR-based system as well. The beforementioned address scrambling block is used in both modules. Additionally, the write counter scheme mentioned in [2] is also implemented as part of the CTR-mode based system. In the following two subsections, design and implementation details of each cipher core for various cascade combinations are explained in detail. Then, the proposed compact XEX and CTR-mode systems and their performance figures for variants of AES and PRESENT are given in the last subsection.

3.1 AES Round Function and Core Design

Advanced Encryption Standard (AES) [9] is a block cipher with a fixed block size of 128 bits and a key size of 128, 192 or 256 bits. In this design, only 128-bit key size (number of rounds $Nr = 10$) is used, as the main concern is to have a lightweight core. AES encryption and decryption pseudocodes are given as:

```
# Encryption
AddRoundKey(state, w[0,3])
for round=1 step 1 to Nr-1
  SubBytes(state)
  ShiftRows(state)
  MixColumns(state)
  AddRoundKey(state, w[4*round,4*(round+1)-1])
end for
SubBytes(state)
ShiftRows(state)
AddRoundKey(state, w[4*Nr,4*(Nr+1)-1])
```

```
# Decryption
AddRoundKey(state, w[4*Nr,4*(Nr+1)-1])
for round=Nr-1 step -1 to 1
  InvShiftRows(state)
  InvSubBytes(state)
  AddRoundKey(state, w[4*round,4*(round+1)-1])
  InvMixColumns(state)
end for
InvShiftRows(state)
InvSubBytes(state)
AddRoundKey(state, w[0,3])
```

Both encryption and decryption paths were designed and combined together in one block with encryption/decryption select. In addition, reverse key generation, which is required for decryption phase, is handled within the module.

AES Encryption Path. As shown in Figure 4, encryption path of AES is a fully parallel implementation of the algorithm. It consists of state round and key round blocks, which handle state operations and key expansion operations, respectively. By connecting several of these paths, it is possible to implement any cascade configuration AES. State round block has a combined ShiftRows & MixColumns module and a SubBytes module. ShiftRows part is the direct mapping of the ShiftRows algorithm and MixColumns is optimized for each coefficient as in [16]. SubBytes module is implemented as a composite S-Box [17]. Key expansion, shown in Figure 4, is performed on-the-fly in parallel to state processing.

AES Decryption Path. Decryption path of AES is designed in a structure similar to encryption path. It is also a fully parallel implementation of the algorithm optimized for area, as shown in Figure 4. It consists of state round and key round blocks as in the previous path. However, here the state round block has a combined InvShiftRows & InvMixColumns module to handle inverse cipher operations. In addition, there is InvSubBytes module, whose S-Box has the same composite inverter structure [17]. Reverse key expansion is performed on-the-fly in parallel to state processing, similar to encryption path, which is shown in Figure 4.

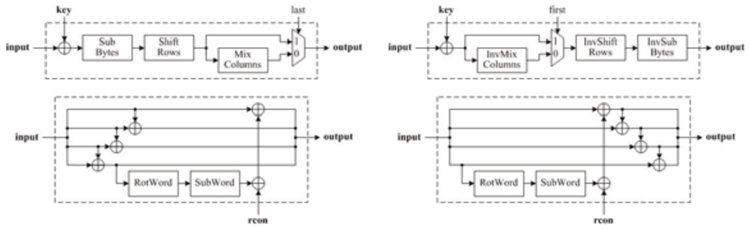


Fig. 4. AES encryption & decryption paths and forward key expansion

Combined AES Core. In the combined AES core, encryption and decryption state processing and key expansion paths are integrated as shown in Figure 5. As the S-Box unit is the most area consuming unit of the core, the block is implemented in a way that the composite inverter [17] is shared between two paths in order to save area. The S-Box block is partitioned as a finite field inverter and pre/post matrix multiplications (transformations), resulting in the block in Figure 5. As a result of this approach, the area of this combined core is smaller than the total of encryption and decryption paths.

Key round block implements the key expansion for both encryption and decryption. Since both modes use the same S-Box, the only additional cost for the combined key expansion is using extra multiplixers, with further area savings (Figure 5).

Decryption Key Generation. Initially, only the encryption key is stored in the key register. However, reverse key expansion for decryption requires use of the last state of key expansion for encryption as the initial "decryption" key. In order to avoid using decryption key registers, a simple key generation scheme is applied: A dummy key expansion for encryption is executed before switching to decryption every time. As a result, the last state of the encryption key is generated and then saved in the key register as the decryption key. When the core needs to switch from decryption to encryption, similar operation is performed: A dummy reverse key expansion is executed, which generates the last state of the decryption key and writes it back on to the key register as the encryption key. Status of the key (encryption/decryption key) is stored in a 1-bit register as the key status, and control logic handles the dummy key expansions.

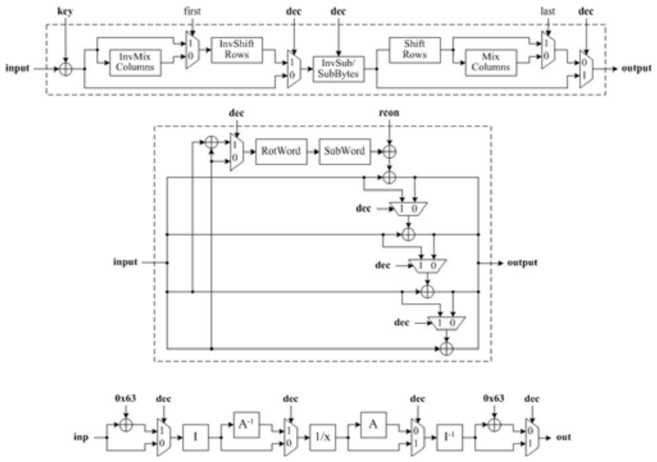


Fig. 5. Combined AES block with key expansion and AES S-Box

Area, Speed, Power Figures. The combined AES core is implemented in various cascade combinations (1, 2, 5, and 10) as shown in 6, which results in varying latencies, gate counts, and operating frequencies (see Table 1). All combinations are synthesized and the obtained performance figures are used in the implementation of the memory encryption unit as guidelines.

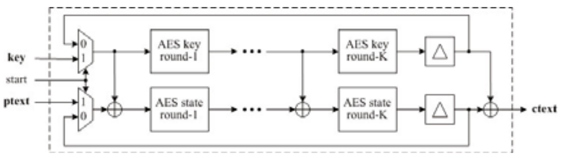


Fig. 6. Cascade implementation of AES

Table 1. Performance of different AES cores

Processor Core	Gate Count (GE)	Max Speed (MHz)	Power (uW/MHz)
acore 1-cyc (enc/dec)	115415	15,7	590,3
acore 1-cyc (enc)	76950	21,0	361,4
acore 2-cyc (enc/dec)	62307	29,5	273,3
acore 2-cyc (enc)	39790	39,6	168,9
acore 5-cyc (enc/dec)	26865	71,7	100,5
acore 5-cyc (enc)	17785	99,4	63,6
acore 10-cyc (enc/dec)	15132	137,2	45,4
acore 10-cyc (enc)	10458	191,2	30,1

3.2 PRESENT Round Function and Core Design

PRESENT [10] is a lightweight block cipher which is an SP-network that consists of 31 rounds. The block length is 64 bits and two key lengths of 80 and 128 bits are supported. For targeted lightweight applications, 80-bit PRESENT provides enough security with smaller area. PRESENT round function is defined as:

```
generateRoundKeys()
for round=1 31
  addRoundKey(state,Ki)
  sBoxLayer(state)
  pLayer(state)
end for
addRoundKey(state,K32)
```

Encryption and decryption paths for PRESENT were designed and combined together in a block. Reverse key generation required for decryption phase is handled within the module, as in the AES design.

PRESENT Encryption Path. PRESENT encryption path is designed to be the fully parallel implementation of the algorithm, as shown in Figure 7. There are two blocks: state round and key round blocks, for state operations and key expansion operations, respectively. State round block has a Permutation and an S-Box module. Key expansion, shown in Figure 7, is performed on-the-fly in parallel with the state processing.

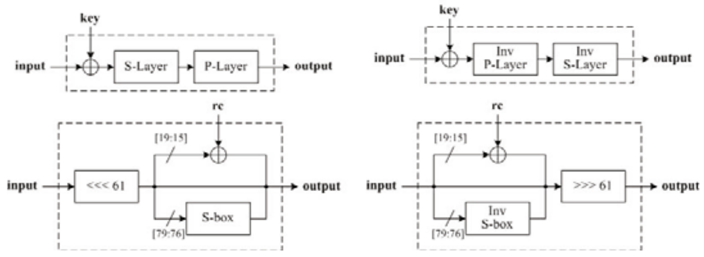


Fig. 7. PRESENT encryption & decryption paths and forward key expansion

PRESENT Decryption Path. Decryption path of PRESENT is designed similar to encryption path as a fully parallel core optimized for area. It consists of reverse state round and reverse key round blocks as in the previous path, as shown in Figure 7. The reverse state round block has an inverse permutation module as well as inverse S-Boxes. The key unit is also adapted for on-the-fly reverse key expansion, which is shown in Figure 7.

Combined PRESENT Core. The combined PRESENT core (Figure 8) puts both encryption and decryption paths together. Unlike the AES S-Box, the PRESENT S-Box can not be formulated in a finite field. Therefore, both regular and inverse S-Boxes are used in the combined core, and selected via multiplexers. The direct result of this fact is the almost doubling of the gate count. The same approach is also applied to the key expansion unit, which has to use both regular and inverse S-Boxes, as well as left and right rotations in order to handle both encryption and decryption. Direct result is again doubled gate count.

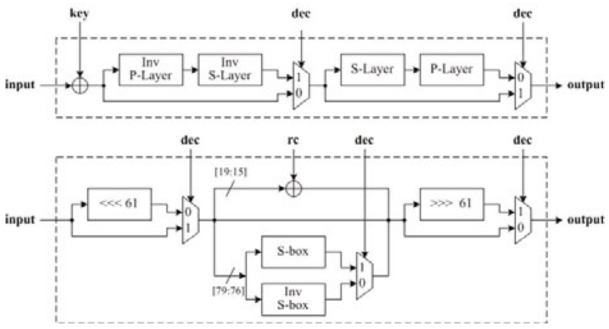


Fig. 8. Combined PRESENT block with key expansion

Decryption Key Generation. The decryption key generation scheme of the AES core is also used in the PRESENT core. That is, in switching from encryption to decryption and from decryption to encryption, dummy forward and reverse key expansions are executed in order to generate and store the decryption and encryption keys, respectively.

Area, Speed, Power Figures. As in the case of AES, various cascade versions (1, 2, 4, 8, 16, and 32) of PRESENT are implemented as shown in Figure 9, and the resulting performance figures (Table 2) are later used as guideline in the design and implementation of memory encryption units.

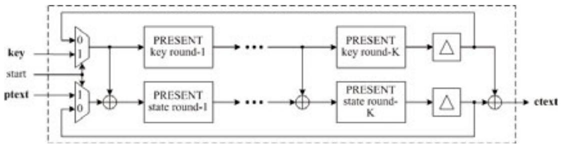


Fig. 9. Cascade implementation of PRESENT

Table 2. Performance of different PRESENT cores

Processor Core	Gate Count (GE)	Max Speed (MHz)	Power (μW/MHz)
pcore 1-cyc (enc/dec)	45700	32,4	172,9
pcore 1-cyc (enc)	20088	48,8	113,2
pcore 2-cyc (enc/dec)	25012	62,3	85,6
pcore 2-cyc (enc)	11863	90,4	47,1
pcore 4-cyc (enc/dec)	13273	122,4	42
pcore 4-cyc (enc)	6744	191,2	22,4
pcore 8-cyc (enc/dec)	7417	228,3	20,1
pcore 8-cyc (enc)	4208	350,9	11,9
pcore 16-cyc (enc/dec)	4547	390,6	10,1
pcore 16-cyc (enc)	2937	429,2	6,4
pcore 32-cyc (enc/dec)	3116	574,7	7,1
pcore 32-cyc (enc)	1839	598,8	3,5

3.3 Memory Encryption Module Design

The I/O signals for the memory encryption module are shown in Figure 10. From the processor point of view, the memory encryption module acts like an ordinary single-port memory. However, there is one important difference: The chip enable signal, *cen*, also acts as a memory access request signal. Once it is asserted, the control module inside the memory encryption unit starts the initialization. In the case of XEX-mode, this corresponds to the encryption of the page address.

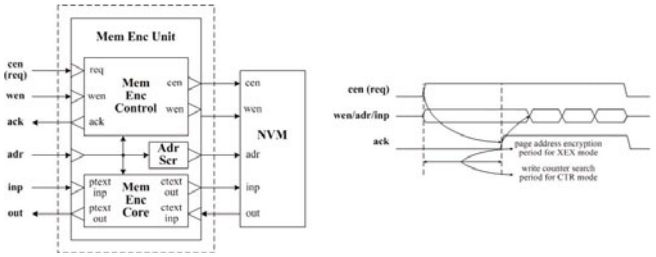


Fig. 10. Memory encryption unit I/O and timing

The situation is a bit different for the CTR-mode. During regular memory accesses, the counter mode requires no initialization, for example when it accesses the program memory code (i.e. only reads data). However, in case of repeatedly written data, the CTR-mode memory encryption unit can be operated in a special mode, where each write to a data packet is counted, and appended to the start of memory address (write counter). This scheme allows use of the same key over and over again until the predefined maximum counter value is reached. However, one has to know the header for every packet. Additionally,, and as the number of newly written data increases, the average memory access time also increases. Therefore, a maximum write counter value has to be determined as part of system design.

Upon completion of the initialization, the memory unit is ready to operate in sequential memory access mode. It asserts the acknowledge signal, *ack*, asking the processor side to send the next query. During the next query, the request (chip enable) signal stays asserted, and only de-asserted during page and/or packet switching (depending on the packet data). The timing for this operation is shown in Figure 10.

During each memory access, the address is scrambled using a reduced width and reduced round of the PRESENT cipher, which is a combined block with no clock cycle losses. From the memory (NVM) point of view, the memory encryption unit acts like an ordinary processor. The memory encryption unit acts as a bridge between the smart card processor and the memory. The most important parameter of this translation process is the average cipher latency. In an ideal case, the memory access latency is a single clock cycle. However, due to the initialization latencies, the average latency increases above the ideal value of single cycle. As shown before, the cipher module latency is the most important parameter in the whole process. Therefore it should be kept as low as possible, ideally at zero additional delay. As will be shown in the following subsections, even in this case, the initialization delay can only be lowered down to 1 cycle.

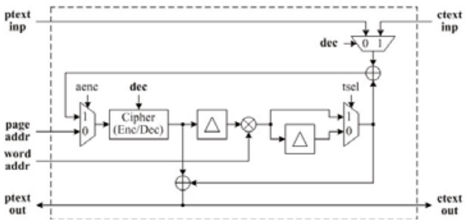


Fig. 11. XEX based memory encryption unit

3.4 XEX-Mode

Figure 11 shows the XEX-mode based memory encryption module. It uses a single cipher core for both page address encryption and data encryption/decryption operations. In page encryption, the cipher core operates in encryption mode, and stores the encrypted page address inside the page address register. It is multiplied with the word address to obtain the tweak value, which is added to both pre and post encrypted/decrypted data. Data encryption (memory write) uses only the instantaneous value of the tweak, whereas decryption (memory read) uses also the delayed value of the tweak. Tweak and page address registers have the same width as the cipher block size.

The XEX-mode based memory encryption unit is implemented using different versions of both AES and PRESENT ciphers. In the implementation, both ciphers are operated in single cycle mode with respect to the memory access clock. Therefore, for different versions of ciphers (except for the fully combinational version), a second cipher clock is required, which should be a multiple of the memory clock with respect to the number of combinational rounds inside the cipher core.

3.5 CTR-Mode

In Figure 12, the CTR-mode based memory encryption module is shown. Its structure is much less complex compared to the XEX-based system. Furthermore, it requires an only-encryption cipher core, which reduces the overall gate count and combinational path delay dramatically. It also does not require the encryption/decryption input data widths to be equal to the cipher width. Therefore, it can be run truly in single cycle mode, resulting in zero additional latency.

The CTR-mode memory encryption module is also implemented using various versions of AES and PRESENT ciphers. As in the case of XEX, it is possible to run the cipher module using a higher rate clock in order to complete the overall encryption process within the same cycle with respect to memory clock.

Table 3 shows the results for the best performing combinations of both modules. As seen in the table, an 8-cycle version of PRESENT, which runs at four times the memory clock rate, gives almost the same throughput result as the 1-cycle version, while occupying only one forth of the area. The result is a security proven ultralight memory encryption scheme implemented in less than 6K gates at zero additional latency.

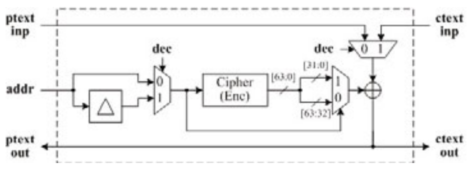


Fig. 12. CTR based memory encryption unit

Table 3. Performance of various XEX and CTR cores

Processor Core	Gate Count	Max Speed
XEX with AES 1-cyc	121,7KGE	15,7MHz
XEX with AES 5-cyc	33,2KGE	14,4MHz
XEX with PRESENT 1-cyc	51,8KGE	32,4MHz
XEX with PRESENT 8-cyc	11,4KGE	28,5MHz
CTR with AES 1-cyc	75,5KGE	21,0MHz
CTR with AES 5-cyc	19,9KGE	19,8MHz
CTR with PRESENT 1-cyc	21,6KGE	48,8MHz
CTR with PRESENT 8-cyc	5,9KGE	43,8MHz

4 Conclusions

In this study, we have investigated the existing memory encryption schemes and their suitability for smart card memory encryption. XEX and CTR-modes were selected as suitable schemes for their proven security and relatively simple structure. Two memory encryption units were designed for each of these schemes, and they were implemented for different versions of AES and PRESENT. PRESENT seems to be an ideal choice for memory encryption when used in CTR-mode. It is possible to implement a PRESENT-based CTR-mode memory encryption module in less than 6K gates with zero additional latency.

The proposed architecture is not only suitable for today's applications, but also capable of fulfilling requirements set forth in the 2023 vision of Eurosmart[8]. We furthermore enhanced our memory encryption module via a block-cipher based address scrambling scheme, where we implemented a small scale variant of

PRESENT. For commercial applications, which require even more security, it is also possible to replace the standard PRESENT S-Boxes with secret proprietary S-Boxes to achieve higher levels of security.

References

1. Duc, G., Keryell, R.: Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection. In: Proceedings of the 22nd Annual Computer Security Applications Conference, pp. 483–492. IEEE Computer Society, Washington, DC, USA (2006) ISBN 0-7695-2716-7
2. Rogers, B., Solihin, Y., Prvulovic, M.: Memory predecryption: hiding the latency overhead of memory encryption. SIGARCH Comput. Archit. News 33, 27–33 (2005)
3. Shi, W., Lee, H.-H.S., Ghosh, M., Lu, C., Boldyreva, A.: High efficiency counter mode security architecture via prediction and precomputation. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture, ISCA 2005, pp. 14–24. IEEE Computer Society, Washington, DC, USA (2005) ISBN 0-7695-2270-X
4. Edward Suh, G., Clarke, D., Gassend, B., van Dijk, M., Devadas, S.: Efficient memory integrity verification and encryption for secure processors. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, p. 339. IEEE Computer Society, Washington, DC, USA (2003)
5. Yang, J., Zhang, Y., Gao, L.: Fast secure processor for inhibiting software piracy and tampering. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, p. 351. IEEE Computer Society, Washington, DC, USA (2003) ISBN 0-7695-2043-X
6. IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. IEEE Std 1619-2007, c1–c32, April 18 (2008)
7. Rogaway, P.: Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 16–31. Springer, Heidelberg (2004) 10.1007/978-3-540-30539-2_2
8. Alyanaklan, E., et al.: The smart & secure world in 2020. Technical report, Eurosmart (2007)
9. Daemen, J., Rijmen, V.: The block cipher rijndael. In: Proceedings of the The International Conference on Smart Card Research and Applications, pp. 277–284. Springer, London (2000) ISBN 3-540-67923-5
10. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: Present: An ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007) ISBN 978-3-540-74734-5
11. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable block ciphers. J. Cryptology 24(3), 588–613 (2011)
12. Lie, D.J.: Architectural support for copy and tamper-resistant software. PhD thesis, Stanford, CA, USA (2004) AAI3111747
13. Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M., Martinez, A.: A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In: Proceedings of the 43rd Annual Design Automation Conference, DAC 2006, pp. 506–509. ACM, New York (2006) ISBN 1-59593-381-6

14. Elbaz, R., Champagne, D., Gebotys, C., Lee, R.B., Potlapally, N., Torres, L.: Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines. In: Gavrilova, M.L., Tan, C.J.K., Moreno, E.D. (eds.) Transactions on Computational Science IV. LNCS, vol. 5430, pp. 1–22. Springer, Heidelberg (2009)
15. Leander, G.: Small scale variants of the block cipher present. Cryptology ePrint Archive, Report 2010/143 (2010), http://eprint.iacr.org/
16. Li, H., Friggstad, Z.: An efficient architecture for the aes mix columns operation. ISCAS (5), 4637–4640 (2005)
17. Paar, C.: Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields. PhD thesis, Institute for Experimental Mathematics, Universität Essen (1994)

Compact FPGA Implementations of the Five SHA-3 Finalists

Stéphanie Kerckhof[1], François Durvaux[1],
Nicolas Veyrat-Charvillon[1], Francesco Regazzoni[1],
Guerric Meurice de Dormale[2], and François-Xavier Standaert[1]

[1] Université catholique de Louvain, UCL Crypto Group,
B-1348 Louvain-la-Neuve, Belgium
{stephanie.kerckhof,francois.durvaux,nicolas.veyrat,
francesco.regazzoni,fstandae}@uclouvain.be
[2] MuElec, Belgium
gm@muelec.com

Abstract. Allowing good performances on different platforms is an important criteria for the selection of the future SHA-3 standard. In this paper, we consider the compact implementations of BLAKE, Grøstl, JH, Keccak and Skein on recent FPGA devices. Our results bring an interesting complement to existing analyzes, as most previous works on FPGA implementations of the SHA-3 candidates were optimized for high throughput applications. Following recent guidelines for the fair comparison of hardware architectures, we put forward clear trends for the selection of the future standard. First, compact FPGA implementations of Keccak are less efficient than their high throughput counterparts. Second, Grøstl shows interesting performances in this setting, in particular in terms of throughput over area ratio. Third, the remaining candidates are comparably suitable for compact FPGA implementations, with some slight contrasts (in area cost and throughput).

Introduction

The SHA-3 competition has been announced by NIST on November 2, 2007. Its goal is to develop a new cryptographic hash algorithm, addressing the concerns raised by recent cryptanalysis results against SHA-1 and SHA-2. As for the AES competition, a number of criteria have been retained for the selection of the final algorithm. Security against cryptanalysis naturally comes in the first place. But good performances on a wide range of platforms is another important condition. In this paper, we consider the hardware performances of the SHA-3 finalists on recent FPGA devices.

In this respect, an important observation is that most previous works on hardware implementations of the SHA-3 candidates were focused on expensive and high throughput architectures, e.g. [17,25]. On the one hand, this is natural as such implementations provide a direct snapshot of the elementary operations' cost for the different algorithms. On the other hand, fully unrolled and pipelined architectures may sometimes hide a part of the algorithms' complexity that is better revealed in compact implementations. Namely, when trying to design more serial architectures, the possibility to share resources, the regularity of the algorithms, and the simplicity to address memories, are additional factors that may influence the final performances. In other words, compact implementations do not only depend on the cost of each elementary operation needed in an algorithm, but also on the number of different operations and the way they interact. Besides, investigating such implementations is also interesting from an application point of view, as the resources available for cryptographic functionalities in hardware systems can be very limited. Consequently, the evaluation of this constrained scenario is generally an important step in better understanding the implementation potentialities of an algorithm.

As extensively discussed in the last years, the evaluation of hardware architectures is inherently difficult, in view of the amount of parameters that may influence their performances. The differences can be extreme when changing technologies. For example, ASIC and FPGA implementations have very different ways to deal with memories and registers, that generally imply different design choices [14,16]. In a similar way, comparing FPGA implementations based on different manufacturers can only lead to rough intuitions about their respective efficiency. In fact, even comparing different architectures on the same FPGA is difficult, as carefully discussed in Saar Drimer's PhD dissertation [12]. Obviously, this does not mean that performance comparisons are impossible, but simply that they have to be considered with care. In other words, it is important to go beyond the quantified results obtained by performance tables, and to analyze the different metrics they provide (area cost, clock cycles, register use, throughput, . . .) in a comprehensive manner.

Following these observations, the goal of this paper is to compare the five SHA-3 finalists on the basis of their compact FPGA implementation. In order to allow as fair a comparison as possible, we applied the approach described by Gaj *et al.* at CHES 2010 [14]. Namely, the IP cores were designed according to similar architectural choices and identical interface protocols. In particular, our results are based on the a priori decision to rely on a 64-bit datapath (see Section 3 for the details). As for their optimization goals, we targeted implementations in the hundreds of slices (that are the FPGAs' basic resources), additionally aiming for the best throughput over area ratio, in accordance with the usual characteristics of a security IP core. In other words, we did not aim for the lowest cost implementations (e.g. with an 8-bit datapath), and rather investigated how efficiently the different SHA-3 finalists allow sharing resources and addressing memories, under optimization goals that we believe reflective of the application scenarios where reconfigurable computing is useful.

As a result, and to the best of our knowledge, we obtain the first complete study of compact FPGA implementations for the SHA-3 finalists. For some of the algorithms, the obtained results are the only available ones for such optimization goals. For the others, they at least compare to the previously reported ones, sometimes bringing major improvements. For illustration purposes, we additionally provide the implementation results of an AES implementation based on the same framework. Eventually, we take advantage of our results to discuss and compare the five investigated algorithms. While none of the remaining candidates leads to dramatically poor performances, this discussion allows us to contrast the previous conclusions obtained from high throughput implementations. In particular, we put forward that the clear advantage of Keccak in a high throughput FPGA implementation context vanishes in a low area one. Performance tables also indicate a good behavior for Grøstl in our implementation scenario, in particular when looking at the throughput over area evaluation metric.

1 SHA-3 Finalists

This section provides a quick overview of the five SHA-3 finalists. We refer to the original submissions for the detailed algorithm descriptions.

BLAKE. BLAKE [3] is built on previously studied components, chosen for their complementarity. The iteration mode is HAIFA, an improved version of the Merkle-Damgard paradigm proposed by Biham and Dunkelman [10]. It provides resistance to long-message second preimage attacks, and explicitly handles hashing with a salt and a "number of bits hashed so far" counter. The internal structure is the local wide-pipe, which was already used within the LAKE hash function [4]. The compression algorithm is a modified version of Bernstein's stream cipher ChaCha [5], which is easily parallelizable. The two main instances of BLAKE are BLAKE-256 and BLAKE-512. They respectively work with 32- and 64-bit words, and produce 256- and 512-bit digests. The compression function of BLAKE relies heavily on the function G, which consists in additions, XOR operations and rotations. It works with four variables : *a, b, c* and *d*. It is called 112 to 128 times respectively for the 32- and 64-bit versions.

Grøstl. Grøstl [15] is an iterated hash function with a compression function built from two fixed, large, distinct but very similar permutations P and Q. These are constructed using the wide-trail design strategy. The hash function is based on a byte-oriented SP-network which borrows components from the AES [11], described by the transforms AddRoundConstant, SubBytes, ShiftBytes and MixBytes. Grøstl is a so-called wide-pipe construction where the size of the internal state (represented by a two 8×16-byte matrices) is significantly larger than the size of the output. The specification was last updated in March of 2011.

JH. JH [26] essentially exploits two techniques : a new compression function structure and a generalized AES design methodology, which provides a simple approach to obtain large block ciphers from small components. The compression function proposed for JH is composed as follows. Half of a 1024-bit hash value $H^{(i-1)}$ is XOR-ed with a 512-bit block message $M^{(i)}$. The result of this operation is passed through a bijective function E8 which is a 42-rounds block cipher with constant key. The output of E8 output is then once again XOR-ed with $M^{(i)}$. This paper considers the round 3 version of the JH specifications submitted to the NIST, in which the number of rounds has been increased from 35.5 to 42.

Keccak. Keccak [6] is a family of sponge functions [7], characterized by two parameters: a bitrate r, and a capacity c. The sponge construction uses $r + c$ bits of state and essentially works in two steps. In a first absorbing phase, r bits are updated by XORing them with message bits and applying the Keccak permutation (called f). Next, during the squeezing phase, r bits are output after each application of the same permutation. The remaining c bits are not directly affected by message bits, nor taken as output. The version of the Keccak function proposed as SHA standard operates on a 1600-bit state, organized in words. The function f is iterated a number of times determined by the size of the state and it is composed of five operations. Theta consists of a parity computation, a rotation of one position, and a bitwise XOR. Rho is a rotation by an offset which depends on the word position. Pi is a permutation. Chi consists of bitwise XOR, NOT and AND gates. Finally, iota is a round constant addition.

Skein. Skein [2] is built out of a tweakable block cipher [23] which allows hashing configuration data along with the input text in every block, and makes every instance of the compression function unique. The underlying primitive of Skein is the Treefish block cipher: it contains no S-box and implements a non-linear layer using a combination of 64-bit rotations, XORs and additions (i.e. operations that are very efficient on 64-bit processors). The Unique Block Iteration (UBI) chaining mode uses Threefish to build a compression function that maps an arbitrary input size to a fixed output size. Skein supports internal state sizes of 256, 512 and 1024 bits, and arbitrary output sizes. The proposition was last updated in October of 2010 (version 1.3) [13].

2 Related Works

Table 1 provides a partial overview of existing low area FPGA implementations for the SHA-3 candidates, as reported in the SHA-3 Zoo [1]. Namely, since our following results were obtained for Virtex-6 and Spartan-6 devices (as will be motivated in the next section), we list only the most relevant implementations on similar FPGAs.

BLAKE has been implemented in two different ways. The first one, designed by Aumasson *et al.* [3], consists in the core functionality (CF) with one G function. This implementation offers a light version of the algorithm but does not really exploit FPGA specificities. On the other hand, the second BLAKE implementation, by Beuchat *et al.* [9], consists in a fully autonomous implementation (FA) and is designed to perfectly fit the Xilinx FPGA architecture : the slice's carry-chain logic is exploited to build a adder/XOR operator within the same slices. The authors also harness the 18-kbit embedded RAM blocks (RAMB) to implement the register file and store the micro-code of the control unit. Table 1 shows Spartan-3 (S3) and Virtex-5 (V5) implementation results.

Jungk *et al.* [20][21] chose to implement the Grøstl algorithm on a Spartan-3 device. They provide a fully autonomous implementation including padding. The similarity between Grøstl and the AES is exploited and AES-specific optimizations

Table 1. Existing compact FPGA implementations of third round SHA-3 candidates (* padding included, ** Altera ALUTs)

	Algorithm	Scope	FPGA	Area [slices]	Reg.	RAMB	Clk cyc.	Freq. [MHz]	Thr. [Mbps]
Aumasson *et al.* [3]	BLAKE-32	CF	V5	390	-	-	-	91	575
Beuchat *et al.* [9]	BLAKE-32	FA	S3	124	-	2	844	190	115
Beuchat *et al.* [9]	BLAKE-32	FA	V5	56	-	2	844	372	225
Aumasson *et al.* [3]	BLAKE-64	CF	V5	939	-	-	-	59	533
Beuchat *et al.* [9]	BLAKE-64	FA	S3	229	-	3	1164	158	138
Beuchat *et al.* [9]	BLAKE-64	FA	V5	108	-	3	1164	358	314
Jungk *et al.* [21]	Grøstl-256	FA*	S3	2486	-	0	-	63	404
Jungk *et al.* [20]	Grøstl-256	FA*	S3	1276	-	0	-	60	192
Jungk *et al.* [20]	Grøstl-512	FA*	S3	2110	-	0	-	63	144
Homsirikamol *et al.* [17]	JH-256	FA	V5	1018	-	-	36	381	5416
Homsirikamol *et al.* [17]	JH-512	FA	V5	1104	-	-	36	395	5610
Bertoni *et al.* [8]	Keccak-256	EM	V5	444	227	-	5160	265	70
Namin *et al.* [24]	Skein-256	CF	AS3	1385**	1858	-	72	574	161

presented in previous works are applied. The table only reports the best and most recent results from [20]. Also, only serial implementations of P and Q are considered, because they better match our low area optimization goal.

No low area implementation of JH has been proposed up to now. In order to have a comparison, the implementation proposed by Homsirikamol *et al.* [17] may be mentioned. It is the high speed FPGA implementation that has the lowest area cost reported in the literature.

A low area implementation of the Keccak algorithm is given by Bertoni *et al.* [8]. In this implementation, the hash function is implemented as a small area coprocessor using system (external) memory (EM). In the best case, with a 64-bit memory, the function takes approximately 5000 clock cycles to compute. With a 32-bit memory, this number increases up to 9000 clock cycles.

Finally, Namin *et al.* [24] presented a low area implementation of Skein. It provides the core functionality and is evaluated on an Altera Stratix-III (AS3) FPGA.

3 Methodology

As suggested by the previous section, there are only a few existing low area FPGA implementations of the SHA-3 candidates up to now. Furthermore, those implementations often lack of similar specifications which make them difficult to compare. We therefore propose to design compact hardware cores of the five third-round candidates using a common methodology, which allows a fair comparison of the performances. The methodology we used mainly follows the one described by Gaj *et al.* [14], which suggests to use uniform interface and architecture, and defines some performance metrics.

First of all, we tried to keep the number of slices in the same range for all the implementations (typically between 150 and 300), with the throughput over area ratio as optimization target. This is a relevant choice for hardware cores, as they often need to be as efficient as possible with a limited resources usage. We then decided to primarily focus on the SHA-3 candidate variants with the 512-bit digest output size, as they correspond to the most challenging scenario for compact implementations - and may be the most informative for comparison purposes. For completeness, we also report the implementation results of the 256-bit versions in appendix, that are based on essentially similar architectures. Next, since we are implementing low area designs, we limited the internal datapath to 64-bit bus widths. This is a natural choice, as most presented algorithms are designed to operate well on 64-bit processors. Therefore, trying to decrease the bus size tends to be cumbersome and provides a limited area improvement at the expense of a significantly decreased throughput. In addition, we specified a common interface for all our designs, in which we chose to have an input message width of 64 bits, as this is the most convenient size to use with our 64-bit internal datapath. It also corresponds to our typical scenario, in which the hash IP cores have to be inserted in larger FPGA designs. Smaller or bigger message sizes would, most of the time, require additional logic in order to reorganise the message in 64-bit words. This is resources consuming and can be added externaly if needed by the user. All our cores have been designed to be fully autonomous, which will help us in the comparison of the total resources needed by each candidate.

Drimer presented in [12] that implementation results are subject to great variations, depending on the implementation options. Furthermore, comparing different implementations with each others can be irrelevant if not made with careful considerations. We therefore specified fixed implementation options and architecture choices for all our implementations. We choose to work on a Virtex-6 and Spartan-6 FPGAs, specifically a XC6VLX75T with speed grade -1 and a XC6SLX9 with speed grade -2, which are the most constraining FPGAs in their respective families, in terms of number of available logic elements. Note that the selection of a high-performance device is not in contradiction with compact implementations, as we typically envision applications in which the hash functionality can only consume a small fraction of the FPGA resources. Also, we believe it is interesting to detail implementation results exploiting the latest FPGA structures, as these advanced structures will typically be available in future low cost FPGAs too. In other words, we expect this choice to better reflect the evolution of reconfigurable hardware devices. Besides, and as will be illustrated by the implementation tables in Section 5, the results for Virtex-6 and Spartan-6 devices do not significantly modify our conclusions regarding the suitability of the SHA-3 finalists for compact FPGA implementations.

We did not use any dedicated FPGA resources such as block RAMs or DSPs. It is indeed easier to compare implementations when they are all represented in terms of slices rather than in a combination of several factors. Additionally, the use of block RAMs is often not optimal as they are too big for our actual needs. All the implementations took advantage of the particular LUT capabilities of the Virtex-6 and Spartan-6, and use shift registers and/or distributed RAMs (or ROMs). The different modules are however always inferred so that portability to other devices is possible, even if not optimal. The design was implemented using ISE 12.1 and for two different sets of parameters. Those two sets are predefined sets available in ISE Design Goals and Strategies project options and are specified as “Area Reduction with Physical Synthesis” and “Timing Performance without IOB Packing”. This choice was mainly motivated by the willing to illustrate the impact of synthesis options on the final performance figures.

We have made the assumption that padding is performed outside of our cores for the same reasons as in [14]. The padding functions are very similar from one hash function to another and will mainly result in the same absolute area overhead. Additionally, complexity of the padding function will depend on the granularity of the message (bit, byte, words,...) considered in each application.

Finally, the performance metrics we used in this text is always the throughput for long message (as defined in [14]). We did not specify the throughput for short message, but the information needed to compute it is present in the result tables of Section 5.

4 Architectures

This section presents the different compact architectures we developed. Because of space constraints, we mainly focus on the description of their block diagrams.

BLAKE. BLAKE algorithm is implemented as a narrow-pipelined-datapath design. The architecture of BLAKE is illustrated in Figure 1. The overall organization is similar to the implementation proposed by Beuchat *et al.*.

BLAKE has a large 16-word state matrix V but each operation works with only two elements of it. Hence, the datapath does not need to be larger than 64 bits. The operations are quite simple, they consist in additions, XOR and rotations. This allows us to design a small ALU embedding all the required operators in parallel, followed by a multiplexer. The way the ALU is build allows computing XOR-rotation and XOR-addition operations in one clock cycle.

Our BLAKE implementation uses distributed RAM memory to store intermediate values, message blocks and C constants. Using this kind of memory offers some advantages. Beyond effective slices occupation, the controller must be able to access randomly to different values. Indeed, message blocks and C constants are chosen according to elements of a permutation matrix. Furthermore, elements of the inner state matrix are selected in different orders during column and diagonal steps.

The 4-input multiplexer in front of the RAM memory is used to load message blocks (M), salt (S) and counter (T) through the *Message* input, to load the initialization vector (IV), to write the ALU results thanks to the feedback loop, and to set automatically the salt to zero if the user does not specify any value. Loading salt or initializing it to zero takes 4 clock cycles. Loading initialization vector takes 8 clock cycles. These two first steps are made once per message.

The two following steps, which are loading the counter and message block, take 18 clock cycles and are carried out at each new message block.

The scheduling is made so that, for each call of the round function G (as described in Section [1]), the variable a is computed in two clock cycles, because it needs two additions between three different inputs. The three other variables (b, c, and d) are computed in one clock cycle thanks to the feedback loop on the ALU. As a result, one call of the G function needs 10 clock cycles to be executed. To avoid pipeline bubbles between column and diagonal steps, the ordering of G functions during diagonal step is changed to G_4, G_7, G_6 and G_5. The BLAKE-64 version needs 16 (rounds) $\times$ 8 (G calls) $\times$ 10 = 1280 clock cycles to process one block through the round function, and 4 more ones to empty the pipeline. The initialization and the finalization steps need each 20 clock cycles. So, complete hashing one message block takes $18 + 1284 + 40 = 1342$ clock cycles. Finally, the hash value is directly read on the output of the RAM and takes 8 clock cycles to be entirely read.

As expected, these results are very close to those announced by Beuchat *et al.* [9] after adjustement (they considered 14 rounds for the BLAKE-64 version rather than 16), since the overall architectures are very similar.

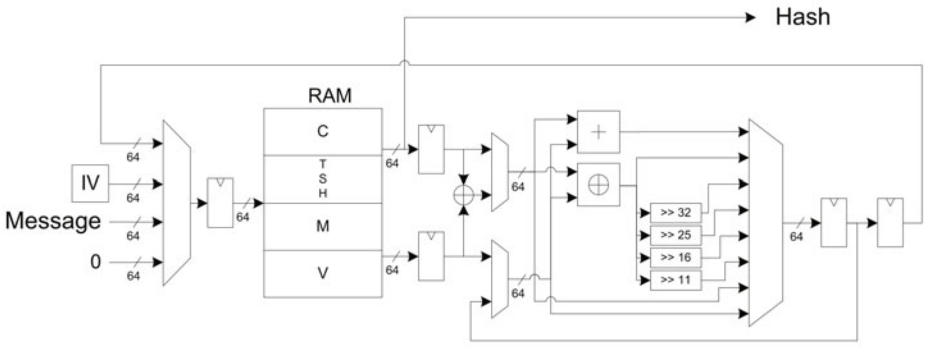


Fig. 1. BLAKE Architecture

Grøstl. The 64-bit architecture of Grøstl algorithm is depicted in Figure [2]. This pipelined datapath implements the P and Q permutation rounds in an interleaved fashion (to avoid data dependency problems). The last round function (Ω) is implemented with the same datapath and only resorts to P. The difference between P and Q lies in slightly different AddRound constants and ShiftBytes shift pattern. Besides the main AES-like functions, there are several circuits. A layer of multiplexers and bitwise XORs is required at the beginning and at the end of the datapath. They implement algorithm initialization, additions necessary at beginning and end of each round, and internal and external data loading. Two distinct RAMs are used to store the P and Q state matrices and input message m_i (RAM *qpm*) and the hash result (RAM *h*). RAM *qpm* is a 64 $\times$ 64-bit dual port RAM. One RAM slot is used to store message m_i and three other slots are used to store current and next P and Q states (slots are used as a circular buffer).

RAM h is a 32×64-bit dual port RAM that stores current and next H (as well as final result).

The four main operations of each P or Q rounds are implemented in the following way. The ShiftBytes operation comes first. It is implemented by accessing bytes of different columns instead of a single column (as if RAM *qpm* was a collection of eight 8-bit RAMs), to save a memory in the middle of the datapath. Different memory access patterns (meaning different initialization of address counters) are required to implement P and Q ShiftBytes as well as no shift (for post-addition with h and hash unloading). Constants of AddRoundConstant are computed thanks to a few small-size integer counters (corresponding to the row and round numbers) and all-zero or all-one constants. Addition of those constants with data is a simple bitwise XOR operation. The eight S-boxes of SubBytes are simply implemented as eight 8×8-bit ROMs (efficiently implemented in 6-input look-up tables-based FPGAs). Finally, the MixBytes operation is similar to the AES MixColumn, except that 8×6 different 8-bit $\mathbb{F}_2$ multiplications by small constants are required, and that eight 64-bit partial products have to be added together. We implemented it as a large XOR tree, with multipliers hardcoded as 8-bit XORs and partial products XORed together.

Hashing a 1024-bit chunk of a message takes around 450 cycles: 16 (loading of m_i) + 14 (rounds) $\times$ 2 (interleaved P and textscq) $\times$ 16 (columns of state matrix) + 8 (ending). The last operation Ω requires around 350 cycles: 14 (rounds) $\times$ (16 (columns) + 6 (pipeline flush)) + 8 (ending) + 8 (hash output).

Roughly speaking, the most consuming parts of the architecture are MixBytes (accounting for 30 % of the final cost), the S-boxes (25 %) and the control of the dual port RAMs (25 %). Note that most pipeline registers are taken from already consumed slices, hence do not increase the slice count of the implementation.

JH. The JH architecture is illustrated in Figure 3 and is composed as follows. Two 16×32-bit single port distributed RAMs (HASH RAM) are used to store the intermediate hash values. Those RAMs are first initialized in 16 clock cycles with IV values coming from a 16×64-bit distributed ROM[1] and are afterwards updated with the output of R8 or the XOR operation output. R8 performs the round functions and is composed of sixteen 5×4 S-boxes, eight linear functions and a permutation. As the permutation layer always puts in correspondence two consecutive nibbles with a nibble from the first half and another from the second half of the permuted state, the output of R8 can be split into two 32-bits words, one coming from the first half and the other from the second half of the intermediate hash value. An address controller (ADDR CONTR), composed of two 16×4-bit dual-port distributed RAMs is then used to reach the wanted location in each HASH RAM, at each cycle. Rotations before and after R8 are needed to organize correctly the hash intermediate values in the two HASH RAMS.

A similar path is designed for constants generation. Two 16×8-bit single-port distributed RAM (CST RAMS) are used to store the constants intermediate values.

[1] IV ROM contains $H^{(0)}$ initial value and not $H^{(-1)}$ as defined in JH specifications. That way, we save 688 cycles of initialization and only loose a couple of slices

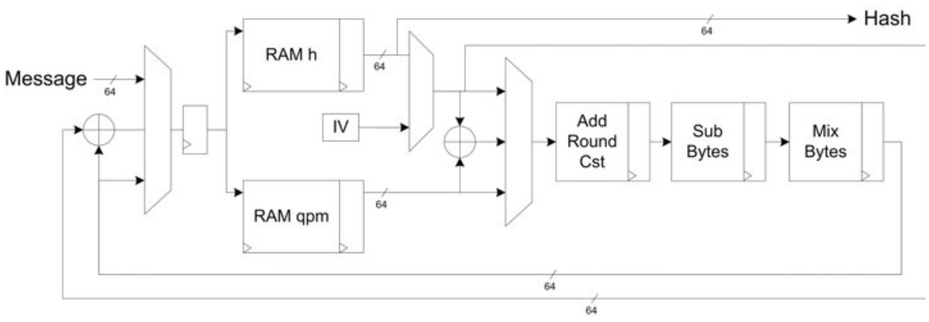


Fig. 2. Grøstl Architecture

The function R6 performs a round function on 16 bits of the constant state. The same address controller as for HASH RAMs is used for CST RAMs.

Finally, a GROUP/DE-GROUP block is used to re-organize the input message. As JH has been designed to achieve efficient bit-slice software implementations, a grouping of bits into 4-bit elements has been defined as the first step of the JH bijective function E8. Similarly, a de-grouping is performed in the last step of E8. When those grouping and de-grouping phases have no impact on high speed hardware implementations (as they result only in routing), this in not the case anymore for low area architectures. Indeed, those steps requires 16 additional clock cycles per message block, as well as more complex controls to access the single port RAMs. To avoid this, we chose to always work on a grouped hash and therefore to perform the data organization on the message with the GROUP/DE-GROUP block. The same component is also used to re-organize the hash final value before sending it to the user.

Our implementation of JH needs 16×42 clock cycles to compute the 42 rounds and 16 additional ones to perform the final XOR operation. In total, 688 clock cycles are required to process a 512-bit message block, 16 for RAM initialization and 20 additional clock cycles are used for the finalization step (4 to empty the pipeline and 16 to output hash from the GROUP/DE-GROUP component).

Keccak. Our architecture, depicted in Figure 4, implements the Keccak version proposed as SHA-3 standard. It works on state of 1600 bits organized into 25 words of 64 bits each. The whole algorithm does not use complex operations, but only XORs, rotations, negations and additions. The basic operations are performed on the 64-bit words, thus our implementation has a 64-bit internal datapath.

We maintained the same organization of Bertoni *et al.*, where the computation was split into three main steps: the first which does part of the theta transformation by calculating of the parity of the columns, the second which completes the theta transformation and performs the rho and pi transformations, and the third which computes the chi and iota steps. This structure requires a memory of 50 words of 64 bits, which are needed to store the state and the intermediate values at the end of the pi transformation.

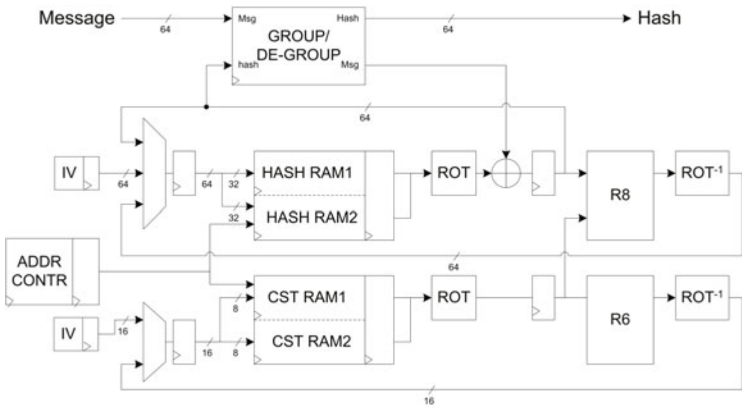


Fig. 3. JH Architecture

To allow parallel read/write operations and to simplify the access to the state, we organized the whole memory into two distinct asynchronous read single port RAM of 32×64-bit (RAM A and RAM B), and we reserved RAM B to store the output of the pi transformation.

Internally, our architecture has 5 registers of 64 bits, connected in order to create a word oriented rotator. During the theta transformation, the registers store the results of the computed parities. The rotator allows to quickly position the correct word for computing the second part of theta, as well as for computing the chi transformation.

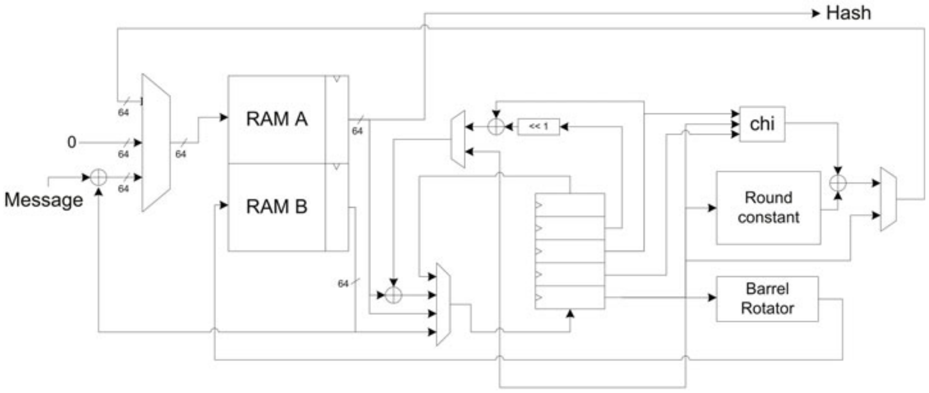


Fig. 4. Keccak Architecture

The most crucial part of Keccak is the rho transformation, which consist of rotation of words with an offset which depends from the specific index. We implemented this step efficiently in FPGA by using a 64-bit barrel rotator and by storing the rotation offsets into a dedicated look up table. The explicit implementation of a barrel rotator allows to significantly reduce the area requirements in comparison to the use of a basic multiplexer. Furthermore, even if the 25 words of the state need to be processed successively by the same component, the use of a single barrel rotator reaches the trade off between the reached performances and the overall area cost which is more suitable for the scope of this paper.

Our implementation of Keccak requires 88 clock cycles to compute a single round. Since Keccak-1600 has 24 rounds, the total number of cycles required to hash a message is 2112, to which is should be added the initial XOR with the current state (25 cycles repeated for each block), the load of the message (9 cycles), and the offloading of the final result (8 cycles).

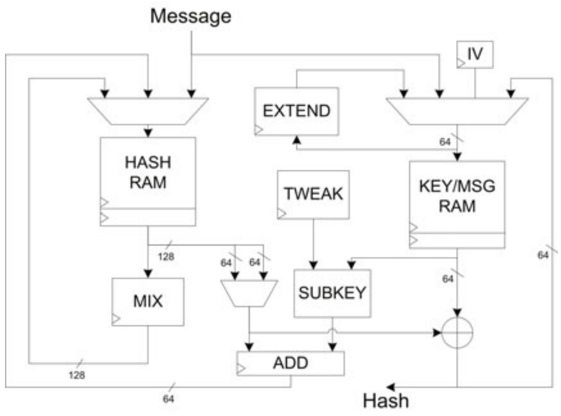


Fig. 5. Skein Architecture

Skein. Our implementation only contains a minimal set of operations necessary to the realization of round computations. In order to provide acceptable performances and memory requirements, the operations are not broken up all the way down to the basic addition, exclusive OR and rotate operations, but rather realize the MIX and subkey addition steps. The architecture is illustrated in figure 5. the initial UBI value is obtained through an 8×64-bit rom (IV) which avoids hashing the first configuration block. Key extension is performed on-the-fly using some simple arithmetic and a 64-bit register (EXTEND). One 17×64-bit RAM memory (KEY/MSG RAM) is used to store both the message block in view of the next UBI chaining, and the keys used for the current block. The hash variables can be memorized in two different 4×64-bit RAMs (HASH RAM), since the permute layer never swaps even and odd words. The permute operation itself is implicitly computed using arithmetic on memory addresses. The MIX operations take two 64-bit values (MIX), and require 4 cycles per round. The subkey addition acts on 64-bit values (ADD), requiring 8 cycles every 4 rounds. Subkeys are computed just before addition, with the help of the tweak registers (SUBKEY and TWEAK). Finally, a 64-bit XOR is used for UBI chaining. After the completion of round operations, the hash digest is read from the key register. Given the variable management in this architecture, only single-port RAMs are needed, rather than the more expensive dual-port RAMs. All these are used asynchronously. When hashing a message, the operator first has to load the initialization vector, taking 9 cycles, followed by 457 cycles per 512-bit message block. Finally, one last block has to be processed before the hash value is output, leading to an overhead of 466 additional cycles.

5 Implementation Results and Discussion

The complete implementation results for our different architectures are given in Tables 2 and 3 for Virtex-6 and Spartan-6 devices, respectively. As expected, one can notice the strong impact of the two sets of options we considered (i.e. area and timing). Still, a number of important intuitions can be extracted.

In the first place, and compared to previous works, we see that our implementation results for BLAKE are quite close to the previous ones of Beuchat *et al.* The main differences are our exploitation of distributed memories (reported in the slices count) rather than embedded memory blocks and the fact that they implemented only 14 rounds, as specified in previous BLAKE-64 version, instead of 16. By contrast, for all the other algorithms, our results bring some interesting novelty. In particular, for Keccak, the previous architecture of Bertoni *et al.* was using only three internal registers, because of its compact ASIC-oriented flavor. This was at the cost of a weak performances, in the range of 5000 clock cycles per hash block. We paid a significant attention in taking advantage of the FPGA structure, in particular its distributed RAMs. As a result, we reduced the number of clock cycles by a factor of more than two. As for the three remaining algorithms, no similar results were known to date, which make them interesting, as first milestones.

Next, this table also leads to a number of comments regarding the different algorithms and their compact FPGA implementations. First, one can notice that Grøstl compares favorably with all the other candidates. While it has quite expensive components, interleaving the P and Q functions allows reducing the logic resources. More importantly, this algorithm proceeds blocks of 1024 bits and has a quite limited cycle count, which leads to significantly higher throughput than our other implementations.

BLAKE and JH also achieve reasonable throughput, but do not reach the level of performance of Grøstl in this case study. For BLAKE, the input blocks are still 1024-bit wide, but our implementation requires three times more cycles per block. For JH, it is rather the reduction of input block size that is in cause.

Skein provides interesting performances too. Its most noticeable limitation is a lower clock frequency, that could be improved by better pipelining the additions involved in our design. As a first step, we exploited the carry propagate adders that are efficiently implemented in Xilinx FPGAs. But this is not a theoretical limitation of the algorithm. One could reasonably assume that further optimization efforts would increase the frequency at the level of the other candidates.

Table 2. Implementation results for the 5 SHA-3 candidates on Virtex-6 (512-bit digests)

		BLAKE	Grøstl	JH	Keccak	Skein	AES
Properties	Input block message size	1024	1024	512	576	512	128
	Clock cycles per block	1342	448	688	2137	458	44
	Clock cycles overhead (pre/post)	12/8	24/354	16/20	9/8	9/466	8/0
Area	Number of LUTs	701	912	789	519	770	658
	Number of Registers	371	556	411	429	158	364
	Number of Slices	192	260	240	144	240	205
	Frequency (MHz)	240	280	288	250	160	222
	Throughput (Mbit/s)	183	640	214	68	179	646
	Efficiency (Mbit/s/slice)	0.95	2.46	0.89	0.47	0.75	3.15
Timing	Number of LUTs	810	966	1034	610	1039	845
	Number of Registers	541	571	463	533	506	524
	Number of Slices	215	293	304	188	291	236
	Frequency (MHz)	304	330	299	285	200	250
	Throughput (Mbit/s)	232	754	222	77	223	727
	Efficiency (Mbit/s/slice)	1.08	2.57	0.73	0.41	0.77	3.08

Table 3. Implementation results for the 5 SHA-3 candidates on Spartan-6 (512-bit digests)

		BLAKE	Grøstl	JH	Keccak	Skein	AES
Properties	Input block message size	1024	1024	512	576	512	128
	Clock cycles per block	1342	448	688	2137	458	44
	Clock cycles overhead (pre/post)	12/8	24/354	16/20	9/8	9/466	8/0
Area	Number of LUTs	719	912	737	525	888	685
	Number of Registers	370	574	338	433	249	365
	Number of Slices	230	343	260	193	292	232
	Frequency (MHz)	135	240	113	166	91	125
	Throughput (Mbit/s)	103	548	84	45	102	364
	Efficiency (Mbit/s/slice)	0.47	1.60	0.32	0.23	0.35	1.57
Timing	Number of LUTs	856	766	1106	640	1059	852
	Number of Registers	594	759	646	476	395	529
	Number of Slices	303	281	362	216	351	274
	Frequency (MHz)	150	265	175	166	111	154
	Throughput (Mbit/s)	114	605	130	45	124	448
	Efficiency (Mbit/s/slice)	0.38	2.15	0.36	0.21	0.35	1.64

Finally, Keccak presents the poorest performances for the 512-bit digests. This is an interesting result in view of the excellent behavior of this algorithm in a high throughput implementation context [14]. Further optimizations could be investigated in order to reduce the number of clock cycles. But as long as a similar architecture as in this paper is used, this would probably be at the cost of a larger datapath (hence, higher slice count). Also, even considering an optimistic

50 cycles per round, the throughput of Keccak would remain 6 times smaller than the one of Grøstl. The main reason of this observation relates different rotations used in this algorithm (that come for free in unrolled implementations but may turn out to be expensive in compact ones) and to the large state that needs to be addressed multiple times when hashing a block. We note that our results are in line with the recent evaluations from CHES 2011 [18], where it is stated that Keccak is not straightforwardly suitable for folding[2]. An interesting alternative and scope for further research would be to change the overall architecture in order to better exploit the bit interleaving techniques described in [19].

Unsurprisingly, the main difference between the Virtex-6 and Spartan-6 implementations consists in a slightly larger number of slices, most likely due to the more constraining FPGA, and a reduction in frequency due to the lower performance of the Spartan-6 FPGAs.

In addition to these results, Table 4 in appendix provides the implementation results for the 256-bit digest versions of the hash algorithms, on Virtex-6. In general, these smaller variants do not exhibit significantly different conclusions. One important reason for this observation is that, when using distributed RAM's in an implementation, reducing the size of a state does not directly imply a gain in slices for a compact implementation (as only the depth of the memories are affected in this case). In fact, the move to 256-bit digests only implied a change of architecture for BLAKE (in the 256-bit version, we used a datapath size of 32 bits). Overall, this move towards smaller digests is positive for Keccak, because of a larger bitrate r, which allows this candidate to be more in line with the other finalists. By contrast, for BLAKE, the processing of 512-bit blocks does not come with a sufficient reduction of the number of rounds, hence leading to smaller throughputs. As for Grøstl, the number of rounds is also reduced by less than a factor 2, but the smaller number of columns in the state matrix allows keeping a higher throughput.

To conclude this work, we finally reported the performance results for an AES-128 implementation, with "on-the-fly" key scheduling, based on a 32-bit architecture. This implementation is best compared with the 256-bit versions of the SHA-3 candidates (because of a 128-bit key). One can notice that the slice count and throughput also range in the same levels as the ones of Grøstl.

Acknowledgements. This work was funded by the Walloon Region (S@T Skywin, MIPSs and SCEPTIC projects) and by the Belgian Fund for Scientific Research (FNRS-F.R.S. and FRIA funding). It also received support from the IAP Program P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and the European Commission through the ICT program under contract ICT-2007-216676 ECRYPT II.

References

1. The sha-3 zoo, `http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo`
2. The skein hash function family, `http://www.skein-hash.info/`

[2] See also the following work to appear at INDOCRYPT 2011 [22].

3. Aumasson, J.-P., Henzen, L., Meier, W., Phan, R.C.-W.: Sha-3 proposal blake, version 1.4 (2011), http://131002.net/blake/
4. Aumasson, J.-P., Meier, W., Phan, R.C.-W.: The Hash Function Family LAKE. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 36–53. Springer, Heidelberg (2008)
5. Bernstein, D.J.: Chacha, a variant of salsa20. In: Workshop Record of SASC 2008: The State of the Art of Stream Ciphers (2008), http://cr.yp.to/chacha.html#chacha-paper
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The keccak sha-3 submission. Submission to NIST, Round 3 (2011)
7. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the Indifferentiability of the Sponge Construction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 181–197. Springer, Heidelberg (2008)
8. Bertoni, G., Daemen, J., Peeters, M., van Assche, G.: Keccak sponge function family main document, version 1.2. April 23 (2009), http://keccak.noekeon.org/
9. Beuchat, J.-L., Okamoto, E., Yamazaki, T.: Compact implementations of blake-32 and blake-64 on fpga. Cryptology ePrint Archive, Report 2010/173 (2010), http://eprint.iacr.org/
10. Biham, E., Dunkelman, O.: A framework for iterative hash functions - haifa. Cryptology ePrint Archive, Report 2007/278 (2007), http://eprint.iacr.org/
11. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer-Verlag New York, Inc., Secaucus (2002)
12. Drimer, S.: Security for volatile FPGAs. Technical Report UCAM-CL-TR-763, University of Cambridge, Computer Laboratory (November 2009)
13. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The skein hash function family. Submission to NIST, round 3 (2011)
14. Gaj, K., Homsirikamol, E., Rogawski, M.: Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 264–278. Springer, Heidelberg (2010)
15. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schlffer, M., Thomsen, S.S.: Sha-3 proposal grøstl, version 2.0.1 (2011), http://www.groestl.info/
16. Henzen, L., Gendotti, P., Guillet, P., Pargaetzi, E., Zoller, M., Gürkaynak, F.K.: Developing a Hardware Evaluation Method for SHA-3 Candidates. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 248–263. Springer, Heidelberg (2010)
17. Homsirikamol, E., Rogawski, M., Gaj, K.: Comparing hardware performance of fourteen round two sha-3 candidates using fpgas. Cryptology ePrint Archive, Report 2010/445 (2010), http://eprint.iacr.org/
18. Homsirikamol, E., Rogawski, M., Gaj, K.: Throughput vs. Area Trade-offs in High-Speed Architectures of Five Round 3 SHA-3 Candidates Implemented Using Xilinx and Altera FPGAs. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 491–506. Springer, Heidelberg (2011)
19. http://keccak.noekeon.org/
20. Jungk, B., Reith, S.: On fpga-based implementations of grøstl. Cryptology ePrint Archive, Report 2010/260 (2010), http://eprint.iacr.org/
21. Jungk, B., Reith, S., Apfelbeck, J.: On optimized fpga implementations of the sha-3 candidate grøstl. Cryptology ePrint Archive, Report 2009/206 (2009), http://eprint.iacr.org/

22. Kaps, J.-P., Yalla, P., Surapathi, K.K., Habib, B., Vadlamudi, S., Gurung, S., Pham, J.: Lightweight implementations of sha-3 candidates on fpgas. To appear in the Proceedings of IndoCrypt (2011)
23. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable Block Ciphers. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 31–46. Springer, Heidelberg (2002)
24. Namin, A.H., Hasan, M.A.: Hardware implementation of the compression function for selected sha-3 candidates. CACR 2009-28 (2009), http://www.vlsi.uwaterloo.ca/~ahasan/hasan_report.html
25. Tillich, S., Feldhofer, M., Kirschbaum, M., Plos, T., Marc-Schmidt, J., Szekely, A.: High-speed hardware implementations of blake, blue midnight wish, cubehash, ECHO, fugue, grøstl, hamsi, jh, keccak, luffa, shabal, shavite-3, simd, and skein. Cryptology ePrint Archive, Report 2010/445 (2010), http://eprint.iacr.org/
26. Wu, H.: The hash function jh. Submission to NIST, round 3 (2011)

A 256-Bit Digest Implementation Results

Table 4. Implementation results for the 5 SHA-3 candidates on Virtex-6 (256-bit digests)

		BLAKE	Grøstl	JH	Keccak	Skein	AES
Properties	Input block message size	512	512	512	1088	256	128
	Clock cycles per block	1182	176	688	2137	230	44
	Clock cycles overhead (pre/post)	12/8	24/122	16/20	17/16	5/234	8/0
Area	Number of LUTs	417	912	789	519	770	658
	Number of Registers	211	556	411	429	158	364
	Number of Slices	117	260	240	144	240	205
	Frequency (MHz)	274	280	288	250	160	222
	Throughput (Mbit/s)	105	815	214	128	179	646
	Efficiency (Mbit/s/slice)	0.90	3.13	0.89	0.89	0.75	3.15
Timing	Number of LUTs	500	966	1034	610	1039	845
	Number of Registers	284	571	463	533	506	524
	Number of Slices	175	293	304	188	291	236
	Frequency (MHz)	347	330	299	285	200	250
	Throughput (Mbit/s)	132	960	222	145	223	727
	Efficiency (Mbit/s/slice)	0.75	3.27	0.73	0.77	0.77	3.08

An Exploration of the Kolmogorov-Smirnov Test as a Competitor to Mutual Information Analysis

Carolyn Whitnall, Elisabeth Oswald, and Luke Mather

University of Bristol, Department of Computer Science,
Merchant Venturers Building, Woodland Road, BS8 1UB, Bristol, UK

Abstract. A theme of recent side-channel research has been the quest for distinguishers which remain effective even when few assumptions can be made about the underlying distribution of the measured leakage traces. The Kolmogorov-Smirnov (KS) test is a well known non-parametric method for distinguishing between distributions, and, as such, a perfect candidate and an interesting competitor to the (already much discussed) mutual information (MI) based attacks. However, the side-channel distinguisher based on the KS test statistic has received only cursory evaluation so far, which is the gap we narrow here. This contribution explores the effectiveness and efficiency of Kolmogorov-Smirnov analysis (KSA), and compares it with mutual information analysis (MIA) in a number of relevant scenarios ranging from optimistic first-order DPA to multivariate settings. We show that KSA shares certain 'generic' capabilities in common with MIA whilst being more robust to noise than MIA in univariate settings. This has the practical implication that designers should consider results of KSA to determine the resilience of their designs against univariate power analysis attacks.

1 Introduction

Differential power analysis (DPA) is a form of side-channel analysis which employs some type of statistic (the *distinguisher*) to identify a correct hypothesis about (part of) the secret key from within a set of possible alternative hypotheses. Popular distinguishers include the Pearson correlation coefficient, the distance-of-means test, and mutual information analysis (MIA). Mutual information (MI) measures the total dependency between two random variables, and was first proposed for use as a distinguisher at CHES 2008 ([1]). MIA's selling point is *genericity*: it is capable of key recovery even when the underlying leakages satisfy few assumptions.

Previous work such as [2] and [3] demonstrated that the (notoriously problematic) estimation of the leakage probability density functions for different key-dependent models is of decisive importance to the performance of MIA in practice. The authors of [2] suggested two alternative distinguishers based on statistics which are conceptually similar to MI but do not require explicit density estimation: the (two-sample) Kolmogorov-Smirnov (KS) test and the Cramér-von-Mises criterion. Each essentially computes some notion of a 'distance' between two distributions. Evaluations of these (and other similar) methods can be found in the statistical literature (for example, [4]): whilst the Cramér-von-Mises statistic performs particularly well (i.e. better than KS) for certain specific distributions, the KS statistic is found to perform well across the board and therefore represents the most generic, distribution-free method.

In this paper we demonstrate how the KS test statistic adapts to the purposes of DPA and investigate the properties and practical performance of such attacks. Alongside, we present an equivalent analysis of MIA—an ideal comparator because of its established role in the existing literature as well as its conceptual similarity to Kolmogorov-Smirnov Analysis (KSA). We assess the distinguishers as applied to key-recovery attacks against implementations of DES in four practically relevant leakage scenarios. Our results are interesting for academics and practitioners alike: from an academic point of view it is interesting to investigate how a conceptually similar approach such as the KS test performs in comparison to MIA. From a practical point of view we are providing information about how to choose the most appropriate distinguisher in certain settings. Specifically, in the setting where the actual power model of a device is unknown to the attacker and does not correspond to a 'nice' Hamming weight leakage, and where a substantial amount of noise distorts the data-dependent signal, we show that KSA actually outperforms MIA and hence is the best choice of a distinguisher (in this setting) at present. This setting is practically relevant as it resembles what can be expected when attacking devices that implement cryptography in hardware and have measures in place to increase the level of noise.

Sect. 2 provides an introduction to differential power analysis (DPA). To explain our comparison criterion we outline some key concepts related to the outcomes of DPA attacks (i.e. the distinguishing vectors) in Sect. 3. We then explain how the KS test adapts to DPA attacks (including considerations for higher-order attacks) in Sect. 4. Section 5 reports the results of our analysis. We conclude thereafter in Sect. 6.

1.1 Our Contributions

In Sect. 3 we adapt the ideas presented in [5] to our purposes and introduce the measure of *nearest-rival distinguishability* to compare distinguishers. We argue that this measure is relevant for practical considerations as it strongly influences the number of traces required for successful key recovery: the smaller the nearest-rival distinguishability score, the more traces will be necessary before the correct key stands out from the alternative hypotheses when the vector comes to be estimated in practice.

In Sect. 4 we show how the KS test statistic can be used to construct a distinguisher for power analysis attacks. We briefly include relevant results from the statistical literature and show how to apply them in the context of univariate and multivariate attacks. An interesting conclusion that we can draw is that whilst

KSA shares many properties with MIA in the univariate setting, its extension to general multivariate settings is problematic [6,7].

In Sect. 5 we analyse the application of the KS distinguisher to four relevant scenarios. An important phenomenon that we observe is that KSA is consistently more robust to noise. Our results give conclusive evidence that it outperforms MIA in univariate scenarios (our study ranges from the optimistic Hamming weight assumption to realistic leakages including the assumption of an unknown highly-nonlinear function). Interesting observations result from our study of bivariate extensions of KSA: here it clearly underperforms MIA both in the masked and unmasked case, irrespective of noise. Our contribution thus gives a balanced view of KSA; it shows both its strengths and weaknesses.

2 Differential Power Analysis

The context for all our analyses is a 'standard DPA attack' scenario as defined in [8]. We assume that the power consumption T of the target cryptographic device depends on some internal value (or state) $f_{k^*}(x)$. The state is a function of some part of the plaintext $x \in \mathcal{X}$, as well as some part of the secret key $k^* \in \mathcal{K}$. Consequently, we have that $T = L \circ f_{k^*}(X) + \varepsilon$, where X is a random variable taking values in $\mathcal{X}$, L is some function which describes the data-dependent component and ε comprises the remaining power consumption which can be modeled as independent random noise. The attacker has N power measurements corresponding to encryptions of N known plaintexts $x_i \in \mathcal{X}$, $i = 1, \ldots, N$ and wishes to recover the secret key k^*. The attacker can accurately compute the internal values as they would be under each key hypothesis $\{f_k(x_i)\}_{i=1}^N$, $k \in \mathcal{K}$ and uses whatever information he possesses about the true leakage function L to construct a prediction model $M : f(\mathcal{X}) \longrightarrow \mathcal{M}$.

DPA is based on the intuition that the modeled power traces corresponding to the correct key hypothesis should bear more resemblance to the true power traces than the modeled traces corresponding to incorrect key hypotheses. An attacker is thus concerned with comparing the degree of similarity between the true and modeled traces. A range of comparison tools—'distinguishers'—can be used, of which mutual information (MI) is an example. MI measures, in bits, the total information shared between two random variables, and is most intuitively expressed in terms of entropies via Shannon's formula: $\mathrm{I}(A;B) = \mathrm{H}(A) - \mathrm{H}(A|B)$.

It is employed as an attack distinguisher to compare the measured traces T with the hypothesis-dependent predictions $M_k = M \circ f_k(X)$:

$$D_{\mathrm{MI}}(k) = \mathrm{I}(T; M_k) = \mathrm{H}(T) - \mathrm{H}(T|M_k) = \mathrm{H}(T) - \underset{m \in \mathcal{M}}{\mathbb{E}} \left[\mathrm{H}(T|M_k = m)\right], \quad (1)$$

and because the 'unexplained' entropy (the second term) is smallest when the predictions are good, we expect (1) to be maximised for the correct key hypothesis $k = k^*$.

MI is particularly appealing for use in DPA because it compares distributions in a general way, detecting not just linear relationships but nonlinear relationships too. Thus MIA has been promoted as a 'generic' distinguisher which potentially remains effective even in the absence of a good power model. It also has natural multivariate extensions, by which it can be straightforwardly adapted to higher-order attacks (see [9] for an overview). However, estimation of MI is notoriously problematic ([10]); all known estimators are biased and no 'ideal' estimator exists (different estimators perform differently depending on the underlying structure of the data). Consequently, MIA outcomes are highly sensitive to the estimation procedure and parameters chosen by the attacker.

3 Evaluation Methodology

The aim of our paper is to compare KSA with MIA in practically relevant scenarios. It is imperative to understand that we are seeking to compare statistical procedures and not attacks or devices: we thus test our methodology in a range of practically meaningful and clearly defined hypothetical scenarios, as characterised by cryptographic function (a non-linear substitution box from the DES standard, as well as the Boolean exclusive-or), device leakage model (Hamming weight, an unevenly weighted sum over the bits, and a highly-nonlinear function) and noise (Gaussian noise of varying size). Our results will be relevant for all devices which share the above mentioned characteristics.

Our approach is based on the recent work published in [5] which proposes to study 'complicated' distinguishers such as MIA by computing and estimating (respectively) so-called theoretic and practical distinguishing vectors. The motivation for this is that distinguishers like MIA do not conform to the easily understood behaviours of 'simple' distinguishers such as correlation, which has a known sampling distribution and responds to noise in a well-understood fashion (e.g. see Chapters 4 and 6 in [11]). We have mentioned before that estimation is notoriously difficult [10]. Studying only practical distinguishing vectors does not, in many cases (as illustrated by previous work such as [9]), allow us to draw any definite conclusions about MIA because it is unclear from the practical vectors whether it is a lack of good estimators or an inherent weakness of MIA that causes its sometimes disappointing performance in practice. By contrast, by studying both theoretic and practical vectors we can assess whether MI itself is the problem or simply the estimation process.

Another contribution of [5] is that of defining measures for distinguishability. This is motivated by the fact that the larger the theoretic (true) margins by which the correct key is distinguished, the fewer traces we expect to require to detect this difference in practice [12]. We use the following subsections to further elaborate on the key concepts relevant to our study (theoretic and practical distinguishing vectors, distinguishability).

3.1 Theoretic vs. Practical Distinguishing Vectors

We adopt the notation of [5], which defines the theoretic attack distinguisher as $\mathbf{D} = \{D(k)\}_{k\in\mathcal{K}} = \{D(L\circ f_{k^*}(X)+\varepsilon, M\circ f_k(X))\}_{k\in\mathcal{K}}$, where the plaintext input X takes values in $\mathcal{X}$ according to some known distribution (usually uniform). The distinguisher D is chosen as some function, e.g. MI. For a defined leakage function L and a power model M, the value $D(k)$ can be precisely calculated. It thus represents the 'true' value of the distinguisher given M, L, and key hypothesis k.

How to compute the 'true' distinguisher values. For each possible input $x \in \mathcal{X}$ to the cryptographic function we obtain a vector evaluating the (variance $\text{Var}(\varepsilon)$) Gaussian density centred at the corresponding data-dependent leakage value $L \circ f_{k^*}(x)$. The average of these vectors, weighted by the input probabilities $\mathbb{P}(X = x)$, then gives the probability density of the power consumption evaluated over the full range of possible leakage values. Conditional densities, corresponding to each possible prediction value $m \in \mathcal{M}$ under each key hypothesis $k \in \mathcal{K}$, are constructed similarly. From these probability densities we are able to directly compute (via numerical integration) MIA distinguishing vectors as per equation (1). The same approach allows us to compute KSA distinguishing vectors (to be defined in Sect. 4, equation (2)).

In practice $\mathbf{D}$ must be estimated as the true distribution of T is unknown (in the unprofiled setting which we are examining). Suppose we have observations corresponding to the vector of inputs $\mathbf{x} = \{x_i\}_{i=1}^N$, and write $\mathbf{e} = \{e_i\}_{i=1}^N$ to be the observed noise (i.e. drawn from the distribution of ε). Then the estimated vector is $\hat{\mathbf{D}}_N = \{\hat{D}_N(k)\}_{k\in\mathcal{K}} = \{\hat{D}_N(L \circ f_{k^*}(\mathbf{x}) + \mathbf{e}, M \circ f_k(\mathbf{x}))\}_{k\in\mathcal{K}}$.

The theoretic distinguishing vector $\mathbf{D}$ can thus be seen as representing the 'best' result one could hope to achieve when performing an analysis in practice.

3.2 Notion of Distinguishability

It follows clearly from the working principle of the distinguishers (as explained in previous sections) that the results of each will be on very different scales: MI is measured in bits and takes values between zero and the total entropy of the measured traces, whereas the KS statistic measures the (absolute) difference between probability distributions and therefore takes values in $[0, 1]$. In order to make meaningful comparisons we need to define an outcome measure which is independent of the numerical results of distinguishers. One approach is to look at how well the correct key hypothesis 'stands out'. Previous work has introduced measures for 'standing out'; for instance a "DPA signal-to-noise ratio" was defined in [13]. We seek to represent, more directly than the "DPA signal-to-noise ratio", the margin to be detected by a practical attack. Thus we look at the distance of the correct key hypothesis from its nearest rival, and to scale this by an appropriate normalising constant. Consequently, we define the *nearest-rival distinguishability* score as the difference between the true-key distinguisher value and the highest incorrect-key value, divided by the standard deviation of the 'optimal' distinguishing vector: the theoretic output of an attack in a noise-free setting with a known power model.

$$\textit{Nearest-rival distinguishability}(\mathbf{D}) = \frac{D(k^*) - \max\{D(k)|k \neq k^*\}}{\sqrt{\text{Var}\{D(L \circ f_{k^*}(X), L \circ f_k(X))\}_{k\in\mathcal{K}}}}.$$

We stress again that this measure of theoretic distinguishability is a meaningful indicator of the practical efficiency of an attack as statistical theory (for example,

[12]) teaches us that the sample size required to detect a difference is strongly related to the true size of that difference: the lower the score, the more traces we expect to require for a successful attack in practice.

4 The Kolmogorov-Smirnov Distinguisher

The Kolmogorov-Smirnov (KS) test has been mentioned in [2] as a seemingly attractive alternative to MIA: it is similarly able to generically compare the distributions of two samples but achieves this without explicit estimation of their probability density functions (PDFs). It also extends fairly straightforwardly to bivariate distributions which makes it adaptable to second-order DPA attacks, although (unlike MI) it becomes problematic in higher dimensions ([7]).

In this paper we are particularly interested in how KSA compares with MIA, in 'typical' scenarios and in some of the more specific scenarios for which MIA has been promoted, namely unknown power model and higher-order attacks. The remainder of this section introduces the KS test and discusses its application to univariate and bivariate (second-order) DPA attacks.

4.1 Kolmogorov-Smirnov Based DPA Attacks

The (two-sample) KS test statistic measures the distance between the empirical cumulative distribution functions (CDFs) of two samples $\mathbf{A} = \{A_i\}_{i=1}^n$ and $\mathbf{B} = \{B_j\}_{j=1}^m$, in order to test whether they have been drawn from the same distribution. It is defined as $\sup_{x \in \mathbf{A} \cup \mathbf{B}} |F_A(x) - F_B(x)|$ where F_A, F_B are the empirical CDFs, i.e. $F_A(x) = \frac{1}{n} \sum_{i=1}^n I_{\{A_i \leq x\}}$ ($I_{\{A_i \leq x\}}$ is the indicator function, taking the value 1 if $A_i \leq x$ and 0 otherwise).

Just as MIA can be understood to operate by comparing the global traces T with the hypothesis-dependent conditional traces $T|M_k$—via the expected change in entropy—a KS-inspired distinguisher measures the maximum distance between the global and the conditional trace distributions, as averaged over the prediction space:

$$D_{\mathrm{KS}}(k) = \mathbb{E}[K(T||T|M_k)] = \mathop{\mathbb{E}}_{m \in \mathcal{M}} \left[\sup_t |F_T(t) - F_{T|M_k=m}(t)| \right]. \qquad (2)$$

Under the correct key hypothesis we expect the test statistic to return a large difference.

The particular appeal of the KS statistic as an alternative to mutual information is that it does not require the explicit estimation of densities, but only the calculation of empirical cumulative distribution functions.

Example: We illustrate the working principle of the KS test via a very simple example consisting of a DES implementation leaking the Hamming weight (HW) of the first S-Box with a signal-to-noise ratio (SNR, defined as $\frac{\mathrm{Var}(L \circ f_{k^*}(M))}{\mathrm{Var}(\varepsilon)}$) of 8. For each key hypothesis we estimate the empirical CDFs of the traces as conditioned on the model predictions and compare them with the 'global' CDF of the traces by computing the expected largest difference between them according to (2).

The left panel of Fig. 1 shows (in grey) the conditional CDFs under the correct key hypothesis, where the 'weight' of the lines indicates the relative contribution of the prediction-specific KS statistics to the expectation which comprises the KS distinguisher as in equation (2). The difference—and most pertinently the maximum (vertical) distance—between these conditional CDFs and the global CDF (in black) is visibly substantial. By comparison the right panel shows the same conditional CDFs as induced by an incorrect key hypothesis. These more closely resemble the global CDF; it is clear to see that the expected maximum distance will be substantially smaller. The same behaviour can be observed for all other incorrect key hypotheses, hence providing the rationale for our KS-inspired distinguisher: we expect only the correct key hypothesis to produce a large average difference.

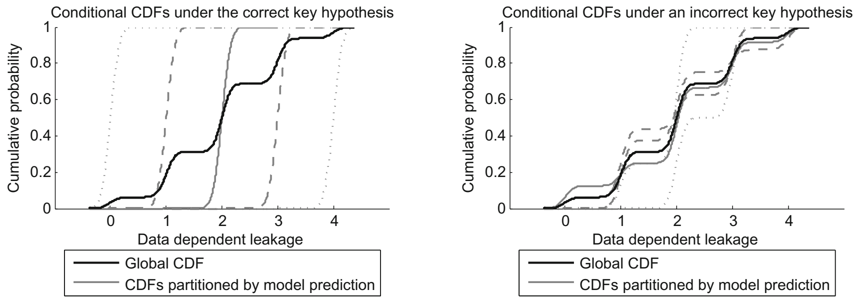


Fig. 1. The KS test is based on the largest distance between the CDFs of two samples. The left and right panels show the CDFs as conditioned on the model predictions under the correct key hypothesis and an incorrect key hypothesis, respectively.

Note that, by design, the test is very sensitive to *any* distributional difference; this is one of the features which makes it popular as a general, non-parametric method of comparison. But for the purposes of DPA there is a potential downside to this sensitivity: the statistic will detect even the subtle differences induced by the incorrect hypotheses, to the detriment of the margin by which the correct key is distinguished.

4.2 Multivariate Extensions

Standard first-order DPA attacks apply a distinguisher to a single point in a trace. It is appealing to suppose that including more than one data point might be beneficial. In the case of attacks against unprotected implementations this could produce better results as more data points potentially imply that more information can be exploited (this has been argued specifically for template attacks [14]). In the case of masked implementations it could provide a way to defeat the masking scheme as the joint distributions of two or more trace points might be related to unmasked model values.

Peacock ([6]) introduces a bivariate KS test statistic for comparing two-dimensional samples $(\mathbf{A_1}, \mathbf{A_2}) = \{(A_{1,i}, A_{2,i})\}_{i=1}^{n}$ and $(\mathbf{B_1}, \mathbf{B_2}) = \{(B_{1,j}, B_{2,j})\}_{j=1}^{m}$, which he defines as:

$$\sup_{(x,y)\in(\mathbf{A_1}\cup\mathbf{B_1})\times(\mathbf{A_2}\cup\mathbf{B_2})} |F_{A_1,A_2}(x,y) - F_{B_1,B_2}(x,y)|.$$

However, this extension is more problematic than the univariate case as it requires a meaningful construction of bivariate empirical CDFs.

The distribution-free property of the KS test rests on being able to map any distribution function on to any other distribution function using a transformation that preserves the ordering of the data. In the one-dimensional case this is trivially fulfilled: there are only two ways of ordering data, namely $\mathbb{P}(A \geq x)$ and $\mathbb{P}(A \leq x)$. As we have that $\mathbb{P}(A \geq x) = 1 - \mathbb{P}(A \leq x)$ the choice is in fact arbitrary.

In higher dimensions the empirical CDF can be defined as:

$$F_{A_1,A_2}(x,y) = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n} I_{A_{1,i}\leq x, A_{2,j}\leq y}$$

for all pairs (x, y). However, in the general case the choice of ordering now *does* affect the test statistic: there is no direct way to map (e.g.) between $\mathbb{P}(A_1 \leq x, A_2 \leq y)$ and $\mathbb{P}(A_1 \geq x, A_2 \leq y)$. In fact for d different random variables, there are 2^d possible orderings we need to consider. The simplest solution to this problem, as suggested by [6], is to find the maximum distributional difference arising from all 2^d possible orderings. The computational complexity of this approach is exponential in the number of variables ($O(2^d * n^d)$). Peacock shows in his work that a bivariate KS test statistic according to his suggestion is close enough to being distribution-free to be useful in practice.

Fasano and Franceschini [7] propose an optimisation whereby the test statistic is evaluated only at the points which are observed in the sample, i.e. at every $(x, y) \in (\mathbf{A_1}, \mathbf{A_2}) \cup (\mathbf{B_1}, \mathbf{B_2})$ rather than every $(x, y) \in (\mathbf{A_1} \cup \mathbf{B_1}) \times (\mathbf{A_2} \cup \mathbf{B_2})$. They are able to show that this leads to a linear increase in speed without compromising on the power of the test or the distribution-free property.

We next explain how this bivariate extension of the test statistic can be adapted to DPA attacks in which two trace points are exploited, and present analogous distinguishers based on multivariate extensions to mutual information. Note that, whilst the latter has natural extensions to dimensions greater than 2, the KS statistic is shown to be problematic in higher dimensions. The authors of [7] *do* present a three-dimensional test but this is not achieved without some difficulty and a substantial increase in complexity (now 2^3 orderings need to be considered); as such we choose not to make use of it ourselves.

Extensions for Masked Implementations. In a second-order attack against a masked implementation we make univariate leakage predictions based on the (unmasked) target value and then exploit what this 'tells' us about the joint distribution of the mask and the target value combined. For the KS distinguisher this means that we are comparing the global joint CDF of the traces with the conditional joint CDFs as partitioned by the model predictions under each key hypothesis:

$$\begin{aligned} D_{2\text{OKS}}(k) &= \mathbb{E}[K(T_1,T_2||T_1,T_2|M_k)] \\ &= \underset{m\in\mathcal{M}}{\mathbb{E}}\left[\sup_{t_1,t_2}\left\{|F_{T_1,T_2}(t_1,t_2) - F_{T_1,T_2|M_k=m}(t_1,t_2)|\right\}\right]. \end{aligned} \tag{3}$$

Previous work (such as [9]) has explored the various ways in which mutual information generalises to higher orders and how these different notions can be adapted to the purposes of DPA. For the purposes of comparison we focus on the extension which is most analogous to the KS distinguisher, namely the information shared between the *pair* of trace points taken jointly and the model prediction, as follows:

$$D_{2\text{OMI}}(k) = \text{I}((T_1,T_2);M_k) = \text{H}(T_1,T_2) - \text{H}(T_1,T_2|M_k). \tag{4}$$

Extensions for Unprotected Implementations. In an unprotected implementation we can use multivariate extensions of our distinguishers to exploit the joint leakage of two target values simultaneously, for example key addition and the output of the first DES S-Box.[1] This approach makes use of a bivariate model prediction and thus calls for slightly different constructions of the distinguishers to those employed in the context of masked implementations.

For the KS distinguisher we simply condition the joint CDFs by the bivariate prediction and proceed as before:

$$\begin{aligned} D_{\text{MKS}}(k) &= \mathbb{E}[K(T_1,T_2||T_1,T_2|(M_1,M_2)_k)] \\ &= \underset{\substack{(m_1,m_2)\in \\ \mathcal{M}_1\times\mathcal{M}_2}}{\mathbb{E}}\left[\sup_{t_1,t_2}\left\{|F_{T_1,T_2}(t_1,t_2) - F_{T_1,T_2|(M_1,M_2)_k=(x_1,x_2)}(t_1,t_2)|\right\}\right]. \end{aligned} \tag{5}$$

Analogously we consider the MI between the pair of trace values and the pair of predictions:

$$D_{\text{MMI}}(k) = \text{I}((T_1,T_2);(M_1,M_2)_k) = \text{H}(T_1,T_2) - \text{H}(T_1,T_2|(M_1,M_2)_k). \tag{6}$$

5 Results

For each scenario that follows we first analyse theoretic KSA and MIA vectors for varying levels of Gaussian noise. These are derived from (respectively) true distributional differences and true entropies, computed directly from the trace density

[1] This choice is meaningful as the model predictions are in this case statistically independent.

functions as explained in Sect. 3. We complement this theoretic analysis—which gives an indication of the underlying potential of a distinguisher—by estimating 'practical' attack vectors against simulated traces and reporting on trace requirements (again as noise varies).[2]

5.1 Optimistic Scenario: DES S-Box with (Known) Hamming Weight Leakage

We first consider the simple and often-studied scenario in which the power consumption comprises a data-dependent component proportional to the Hamming weight of the (first) DES S-Box plus some independent Gaussian noise. Assuming Hamming-weight leakage is realistic for implementations on simple micro-controllers (e.g. [11] use this as their running example).

Theoretic Outcomes

Pure-Signal Leakage: Figure 2 shows the theoretic distinguishing vectors for MIA and KSA attacks using a Hamming weight (HW) power model against noise-free Hamming weight leakage of the first DES S-Box. It also illustrates our notion of distinguishability. Both distinguishers are capable of identifying the correct key; MIA achieves a slightly higher distinguishability score of 5.6 compared with 4.2 for KSA. Equivalent attacks using the identity (ID) power model were less distinguishing, with scores of 3.8 and 3.1 for MIA and KSA respectively: evidently, the generic capabilities of the distinguishers are not useful in this 'known power model' scenario.

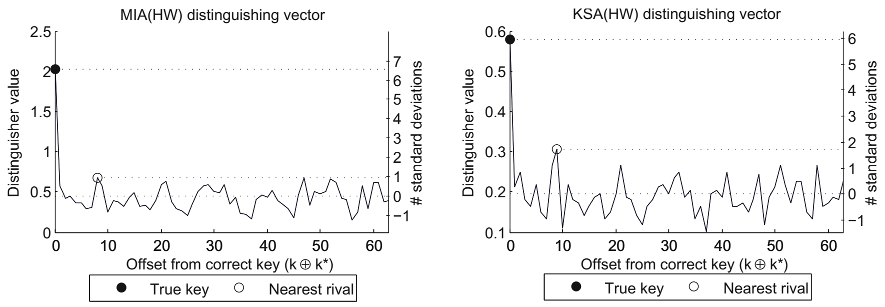


Fig. 2. Theoretic distinguishing vectors for MIA(HW) and KSA(HW) in attacks against HW leakage of the first DES S-Box with zero noise

[2] For MIA estimations we employ the heuristic rule favoured by the literature, and estimate PDFs via histograms with the number of bins equal to the cardinality of the power model image (i.e. 5 for the HW power model, 16 for the identity power model). Therefore, though these are not 'definitive' results (as no universally 'best' estimator exists) they do represent an established methodology and, as such, a meaningful basis for comparison with KSA.

As SNR Varies: Figure 3 shows how the distinguishability scores vary with the strength of the data-dependent signal (relative to the Gaussian noise). The KSA attacks, though less distinguishing than their MIA counterparts in strong-signal scenarios, are more robust to noise and therefore attain a theoretic advantage in weak-signal scenarios.

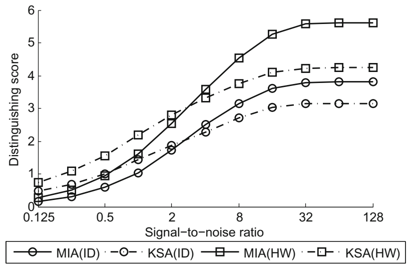


Fig. 3. Theoretic distinguishing power as SNR varies, for attacks against the first DES S-Box with HW leakage

Practical Outcomes (Simulations). The first panel of Figure 4 shows the mean number of traces needed to recover the key; the second panel shows the 90th-percentile, i.e. the number needed to achieve a 90% success rate. KSA(HW) performs almost identically to MIA(HW) (as could be expected from the theoretic vectors), with some evidence of a small advantage in weak-signal settings (again in keeping with the theoretic vectors). The ID attacks are more data intensive in both cases, but KSA(ID) exhibits consistently better performance than MIA(ID), probably due to the heavy estimation overhead incurred by the large number of bins required by the latter.

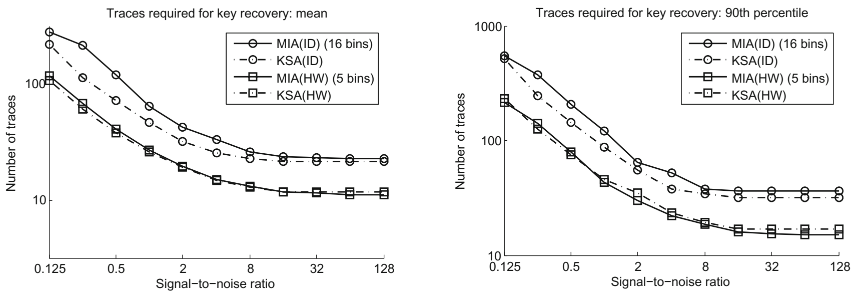


Fig. 4. Mean and 90th percentile of the trace requirement for key recovery, in repeated experiments against simulated HW leakage of the first DES S-Box, as SNR varies

5.2 Realistic Scenario: DES S-Box with Unknown Power Model

We next consider the performance of the two distinguishers in the case that the attacker does not have a precise power model. As motivated by [15] we focus on the case that the device leaks—instead of the Hamming weight—an unevenly weighted sum of the bits. This is realistic for typical micro-processors especially in the low-cost range (as reported by [15]). In our experiments, we assume that the least significant bit dominates in the leakage function with a relative weight of 10; in the experiments of [2] this was sufficient distortion to render MIA more effective than correlation DPA. To extend this analysis we also consider theoretic vectors assuming a highly non-linear power model[3]. This is relevant for hardware implementations, e.g. often non-linear functions are implemented via combinational logic in hardware and it is well known (see [16], [17]) that such implementations show leakage characteristics which are unrelated to linear leakage models.

Theoretic Outcomes

Pure-Signal Leakage: Both the HW and the generic ID variants of KSA are theoretically successful in a noise-free environment, but once again are slightly disadvantaged relative to MIA with distinguishing scores of 2.8 and 3.4 compared with 4.8 and 4.8 respectively.

As SNR Varies: The impact of noise is more marked than that observed for the known power model scenario, as can be seen in Figure 5; all attacks require a stronger signal before converging to their noise-free outcomes.

It is particularly notable that in high-noise settings the KSA attacks are actually more distinguishing than their MIA counterparts. Also of interest is the fact that the ID variants exhibit stronger outcomes and greater robustness to large amounts of noise than attacks using the (now imprecise) HW power model. Thus we confirm the existence of conditions under which KSA has the same 'generic' potential as MIA.

Practical Outcomes (Simulations). The theoretic KSA vectors show more distinguishing power than MIA in noisy scenarios so we have sufficient reason to expect that this translates to a practical advantage in terms of trace requirements, which we test by estimating the practical distinguishing vectors against simulated trace measurements.

Figure 6 plots the results (in terms of sample size requirements) of the practical distinguishing vectors as estimated from simulated traces with Gaussian noise. These tally well with the results of the theoretic vectors: ID attacks substantially outperform HW attacks when the leakage signal is weak, but this advantage is less clear in high-signal settings. KSA(ID) is particularly effective relative to MIA(ID) as estimated with 16 bins (we note that this does not necessarily represent the best-case capabilities of MIA but it is consistent with what one expects given the theoretic distinguishing vectors). KSA(HW) performs similarly to MIA(HW).

[3] To achieve a high-degree of non-linearity we use the Hamming weight of output of the AES SubBytes function.

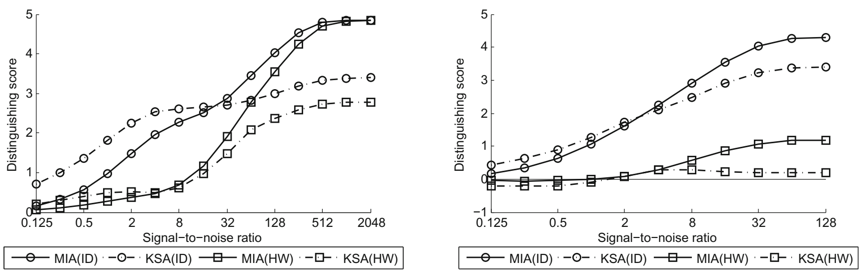


Fig. 5. Theoretic distinguishing power as SNR varies for attacks against the first DES S-Box where the LSB dominates in the leakage with a relative weight of 10 (left panel) and were the leakage is a highly non-linear function (right panel)

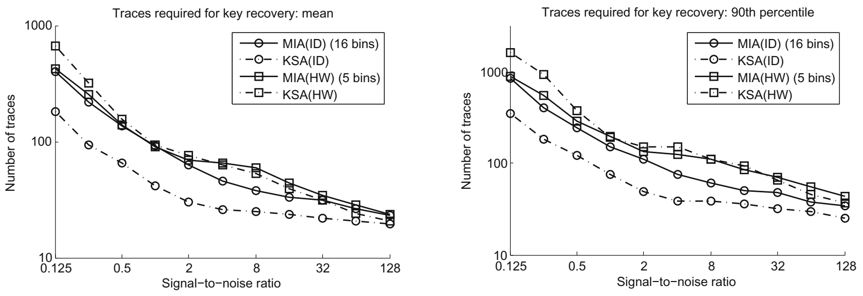


Fig. 6. Mean and 90^{th} percentile of the trace requirement for key recovery, in repeated experiments against simulated S-Box leakage in which the LSB dominates with a relative weight of 10

5.3 Higher-Order Scenario: Second-Order Attacks against a Masked Implementation

As our first example of a multivariate application, we consider second-order attacks on a masked implementation of DES leaking the HW of the mask and the HW of the S-Box output, each with independent Gaussian noise. The second-order extensions for KSA and MIA distinguishers are as described in Sect. 4.2.

Theoretic Outcomes

Pure-Signal Leakage: The noise-free distinguishing score of second-order KSA is just 0.6, compared with 3.2 for the MIA analogue. Thus both are capable of identifying the correct key, though with substantially reduced distinguishability relative to their first-order counterparts in unprotected scenarios, particularly in the case of KSA, as Fig. 7 illustrates.

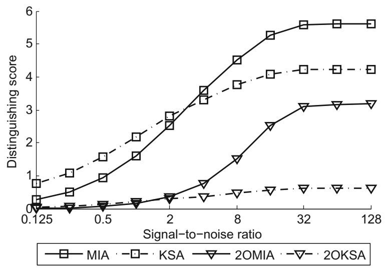


Fig. 7. Theoretic distinguishing power as SNR varies, for second-order HW attacks against a masked implementation of DES with HW leakage

As SNR Varies: Mark once more in Figure 7 that the KSA variant of the second-order attack exhibits greater noise robustness, so that in low-signal settings it shares comparable theoretic distinguishing power with MIA.

Practical Outcomes (Simulations). The first panel of Figure 8 shows the success rates for attacks against a masked DES implementation with noise-free leakage. The second-order KSA attack requires on average 150 traces, with a 90^{th}-percentile of 325, whilst second-order MIA is markedly more efficient, requiring on average only 30 traces with a 90^{th}-percentile of 45.

The remaining three panels show the same for scenarios in which small but increasing amounts of Gaussian noise are added. Even with an SNR as high as 128 the impact on success is substantial for both attack methods but (proportionately) more so for MIA. For an SNR of 32 (the lowest we attempted) the mean and 90^{th}-percentile of the trace requirement for KSA to be successful were 2,450 and 5,500 respectively; the equivalent figures for MIA were 1,440 and 3,200.

The heavy computational demands of the second-order KSA distinguisher mean that, as more noise is added, such attacks quickly become infeasible without enhanced computing power. Our theoretic analysis, and our practical results in other scenarios, indicate that it *could* achieve a small advantage over MIA (in terms of data complexity) when the signal is weak enough, but we are not able to test this and the advantage would likely be far outweighed by the relative computational costs.

5.4 Bivariate Extensions for an Unprotected Implementation

We next investigate whether or not attack outcomes can be improved by the incorporation of a second trace measurement corresponding to a different target function. In particular, we consider exploiting the joint leakage of key addition and the first DES S-Box, in the case that this is comprised of the Hamming weight of the target values plus some independent Gaussian noise.

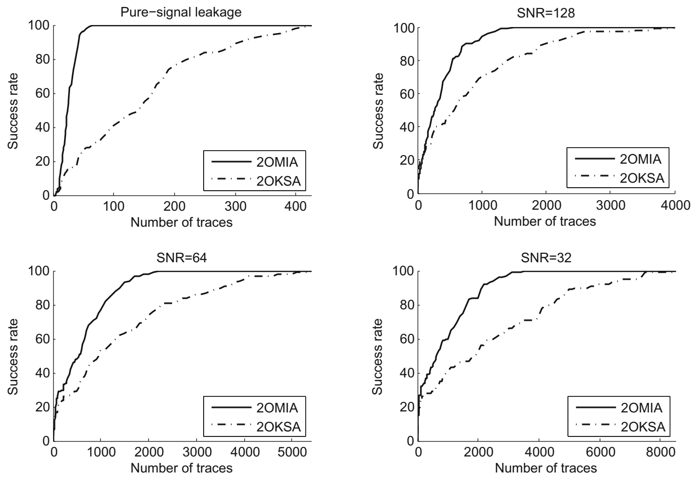


Fig. 8. Success rates of HW attacks against a masked implementation of DES with HW leakage, as the number of traces increases.

Theoretic Outcomes

Pure-Signal Leakage: The noise-free distinguishability scores of bivariate MIA and KSA attacks are 3.6 and 1.7 respectively, compared with 5.6 and 4.2 for the equivalent univariate S-Box attacks. Thus, both methods are actually weakened by the incorporation of key addition leakage; KSA more so than MIA.

However, it is well documented that the resistance of a function to DPA has an inverse relationship with its resistance to cryptanalysis ([18]). In particular, the linearity of key addition makes it hard for DPA to distinguish between similar keys: small changes to the input produce small changes in the output. S-Boxes, on the other hand, are specially designed so that the converse is true, which makes them particularly vulnerable to DPA.

It is not, then, so surprising that key addition information detracts from attack distinguishability. If the leakage of two suitably nonlinear functions could be jointly targeted, our bivariate enhancement may prove more useful—we leave this as an open question.

As SNR Varies: Figure 9 shows the distinguishing scores of the bivariate attacks as compared with the univariate S-Box attacks, for varying levels of Gaussian noise. As with the univariate attacks, the bivariate KSA distinguisher is more robust to noise so that in very low-signal settings it exhibits a slight advantage over the bivariate (and indeed the univariate) MIA distinguisher. As in the application to the masked implementation, for all noise levels (i.e. including the noise-free setting) the bivariate distinguishing vectors are considerably less distinguishing than their univariate counterparts.

Practical Outcomes (Simulations). Figure 10 depicts the performance of practical bivariate attacks (against the DES S-Box and key addition jointly)

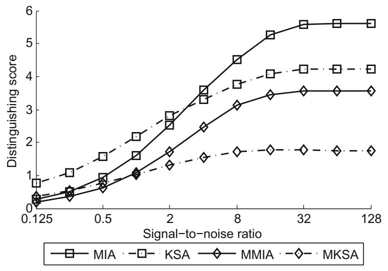


Fig. 9. Theoretic distinguishing power as SNR varies, for bivariate HW attacks against DES with HW leakage

as compared with univariate attacks against the DES S-Box alone. The lower theoretical distinguishing power, coupled with the additional complexity of estimation, mean that the bivariate attacks require more traces to be successful, in all tested noise settings. As with the univariate attacks, bivariate KSA performs very similarly to bivariate MIA.

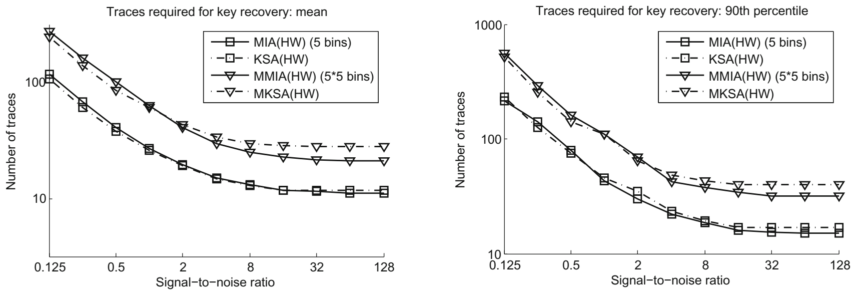


Fig. 10. Mean and 90th percentile of the trace requirement for key recovery, in repeated experiments against simulated HW leakage of AddRoundKey and the first DES S-Box jointly, as SNR varies

These results (w.r.t. both theoretic and practical distinguishing vectors) are an important reminder that it is not the *quantity* of information which contributes to attack outcomes so much as the *quality*: identifying the most vulnerable targets is more likely to be fruitful than combining information from targets with differing degrees of DPA resistance. Moreover, univariate attacks remain less demanding in terms of computational complexity and the sample size required for estimation.

6 Conclusion

We have shown that the (two-sample) KS test statistic can be adapted to the purposes of DPA in a manner which bears considerable resemblance to MI-based

DPA. We explored the theoretic and practical distinguishing vectors of KSA as compared with MIA, with a particular focus on scenarios that are relevant for practice.

Our findings showed that in noise-free or strong-signal univariate settings MIA was consistently the more distinguishing and more efficient attack, but when the signal was sufficiently weak the noise-robustness of KSA enabled it to gain an advantage.

The KSA distinguisher was found to share those characteristics of MIA which make it to some extent 'power model free'; each can be adapted to use the identity power model in the case that an attacker lacks precise knowledge of the true data-dependent leakage (provided the target function is non-injective).

We also showed how a bivariate version of the (two-sample) KS test statistic enables extension to second-order KSA in order to defeat a masking scheme. However, here it was quite substantially outperformed by MIA in strong-signal settings and was so computationally complex as to be unfeasible in weak-signal settings. Moreover, whereas multivariate MI quite naturally incorporates additional data points, extensions of the KS test beyond 2 dimensions quickly become problematic so that there is little scope for third- or higher-order KSA.

A interesting question for future work is whether or not the known distribution of the KS test statistic could be used to formally derive the number of traces required for an attack to be successful, as has been accomplished in the case of correlation DPA (see §6.4 of [11]). Whilst the distribution of the KS test statistic is known it is unclear how it could be used to derive that of the KSA distinguisher (recall that this is defined as an average over several KS test statistics).

Acknowledgements. The first author of this paper has been funded via an Engineering and Physical Sciences Research Council studentship. The second and third authors have been supported by an Engineering and Physical Sciences Research Council Leadership Fellowship I005226.

References

1. Gierlichs, B., Batina, L., Tuyls, P., Preneel, B.: Mutual Information Analysis: A Generic Side-Channel Distinguisher. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 426–442. Springer, Heidelberg (2008)
2. Veyrat-Charvillon, N., Standaert, F.-X.: Mutual Information Analysis: How, When and Why? In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 429–443. Springer, Heidelberg (2009)
3. Prouff, E., Rivain, M.: Theoretical and Practical Aspects of Mutual Information Based Side Channel Analysis. In: Abdalla, M., Pointcheval, D., Fouque, P.-A., Vergnaud, D. (eds.) ACNS 2009. LNCS, vol. 5536, pp. 499–518. Springer, Heidelberg (2009)
4. Stephens, M.A.: EDF Statistics for Goodness of Fit and Some Comparisons. Journal of the American Statistical Association 69(347), 730–737 (1974)
5. Whitnall, C., Oswald, E.: A Comprehensive Evaluation of Mutual Information Analysis Using a Fair Evaluation Framework. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 316–334. Springer, Heidelberg (2011)

6. Peacock, J.: Two-Dimensional Goodness-of-Fit Testing in Astronomy. Monthly notices of the Royal Astronomical Society, 615–627 (1983)
7. Fasano, G., Franceschini, A.: A Multidimensional Version of the Kolmogorov-Smirnov Test. Monthly Notices of the Royal Astronomical Society 225, 155–170 (1987)
8. Mangard, S., Oswald, E., Standaert, F.X.: One for All - All for One: Unifying Standard DPA Attacks. IET Information Security 5(2), 100–110 (2011)
9. Batina, L., Gierlichs, B., Prouff, E., Rivain, M., Standaert, F.X., Veyrat-Charvillon, N.: Mutual Information Analysis: A Comprehensive Study. Journal of Cryptology, 1–23 (2010)
10. Paninski, L.: Estimation of Entropy and Mutual Information. Neural Computation 15(6), 1191–1253 (2003)
11. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smart Cards. Springer, Heidelberg (2007)
12. Kraemer, H.C., Thiemann, S.: How Many Subjects?: Statistical Power Analysis in Research, 1st edn. Sage Publications, Inc. (September 1987)
13. Guilley, S., Hoogvorst, P., Pacalet, R.: Differential Power Analysis Model and Some Results. Smart Card Research and Advanced Applications Vi, 127–142 (2004)
14. Chari, S., Rao, J., Rohatgi, P.: Template Attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 13–28. Springer, Heidelberg (2003)
15. Akkar, M.-L., Bevan, R., Dischamp, P., Moyart, D.: Power Analysis, What Is Now Possible.. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 489–502. Springer, Heidelberg (2000)
16. Mangard, S., Pramstaller, N., Oswald, E.: Successfully Attacking Masked AES Hardware Implementations. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 157–171. Springer, Heidelberg (2005)
17. Renauld, M., Standaert, F.-X., Veyrat-Charvillon, N., Kamel, D., Flandre, D.: A Formal Study of Power Variability Issues and Side-Channel Attacks for Nanoscale Devices. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 109–128. Springer, Heidelberg (2011)
18. Prouff, E.: DPA Attacks and S-Boxes. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 424–441. Springer, Heidelberg (2005)

A High-Performance Implementation of Differential Power Analysis on Graphics Cards

Timo Bartkewitz and Kerstin Lemke-Rust

Department of Computer Science,
Bonn-Rhine-Sieg University of Applied Sciences,
Grantham-Allee 20, 53757 Sankt Augustin, Germany
{timo.bartkewitz,kerstin.lemke-rust}@h-brs.de

Abstract. We present an implementation for Differential Power Analysis (DPA) that is entirely based on Graphics Processing Units (GPUs). In this paper we make use of advanced techniques offered by the CUDA Framework in order to minimize the runtime. In security testing DPA still plays a major role for the smart card industry and these evaluations require, apart from educationally prepared measurement setups, the analysis of measurements with large amounts of traces and samples, and here time does matter. Most often DPA implementations are tailor-made and adapted to fit certain platforms and hence efficient reference implementations are sparsely seeded. In this work we show that the powerful architecture of graphics cards is well suited to facilitate a DPA implementation, based on the Pearson correlation coefficient, that could serve as a high performant reference, e.g., by analyzing one million traces of $20k$ samples in less than two minutes.

Keywords: DPA, CPA, Graphics Cards, CUDA.

1 Introduction

The resistance of a cryptographic device against side channel attacks is defined by the amount of traces that is at least required to recover a secret information that is embedded in the device under a specific adversarial scenario. In commercial applications, many crypto devices which can either consist of a microcontroller or an FPGA (Field Programmable Gate Array), respectively an ASIC (Application-Specific Integrated Circuit) are hardened to resist side channel attacks. In order to fulfill the security requirements for highly resistant devices side channel testing with one million or even more traces are nowadays common for security labs, cf. [7] for the state of the art in testing AES hardware implementations in 2005. Currently, CPA attacks in research are carried out with up to one hundred million traces [8]. Both processes that are implied by a side channel attack, namely trace recording and computational analysis, can be highly time consuming for a smart card evaluation. Contrary to the effort involved in the trace recording that is mostly dictated by the device and done once, the computational analysis is repeatedly carried out to meet different attack scenarios. Hence, an acceleration concerning the analysis part is definitely desirable. Differential Power Analysis (DPA) [4] using the Pearson correlation coefficient [2] is still the most common statistical tool to evaluate the side channel resistance.

Motivation: Graphics cards provide a powerful parallel architecture which became widely accepted in scientific computations. Also cryptography is well established on GPUs with several implementations that were made during the last few years. For instance, cryptosystems like ECC, RSA, and AES were efficiently implemented [13,1,3]. Until now, only little efforts were spent to speed up DPA with the help of graphics cards. The only other proposal we are aware of is [5]. Their approach includes Difference of Means (DOM) as the statistical test and makes use of both, the general purpose CPU and a CUDA related GPU. All in all they achieve a reasonable speed-up factor of about two by parallelizing the summation of samples on the GPU side.

Our contribution: In this paper we make use of advanced techniques offered by CUDA to achieve key benefits for the runtime. Further, we shift any computation to the GPU. We adopt algorithms for DPA to achieve optimal results on graphics cards involving the Pearson correlation coefficient as the statistical test. A major part for the speed-up is the covariance whose implementation is done through a matrix multiplication which performs very well on graphics cards.

Organization of the paper: This paper is organized as follows: Section 2 briefly introduces Differential Power Analysis, especially the formula of the Pearson correlation coefficient that will be adapted in the remainder. Section 3 provides a short overview on modern graphics cards architecture considering their programming and memory model. In Section 4, we describe the chosen implementation approach concerning algorithms and requirements when being applied on a graphics card. Section 5 reports our experimental results before we conclude in Section 6.

2 Differential Power Analysis

Differential Power Analysis (DPA) is a passive implementation attack aiming at key recovery of a cryptographic implementation. The physical leakage of the device that is exploited is usually the power consumption or the electromagnetic emanation of the device while it processes the cryptographic algorithm. DPA is a divide-and-conquer attack, i.e., a cryptographic key is successively compromised by its subkeys.

We assume that the device processes a sensitive variable z which is the conjunction of known input v to the cryptographic computation, i.e., a plaintext or ciphertext and unknown secret information embedded in the device, i.e., a subkey k, such that $z = f_k(v)$. We further assume that the physical leakage of the device under test leaks is modeled by

$$L_t = \delta_t + \mathcal{L}(z) + B_t.$$

Herein, L_t is the leakage at time t that depends on a constant portion δ_t, a certain deterministic leakage function $\mathcal{L}(\bullet)$ that describes how the leaked information depends on the sensitive variable z, and a noise term B_t, a randomly and normally distributed variable centered in zero with standard deviation σ. Note that in practice the leakage function $\mathcal{L}(\bullet)$ is usually not known by the adversary, however, it is well-known that often the Hamming weight of z is a good approximation [6]. Alternatively, the adversary may evaluate single-bit leakage of z.

Measurement. As the first step of DPA the power consumption of the device under attack is measured with a digital storage oscilloscope (DSO). Such a measurement is given by the matrix

$$\mathbf{X} = \left(X_1\ X_2\ X_3 \dots X_s\right) = \begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \cdots & x_{1,s} \\ x_{2,1} & x_{2,2} & x_{2,3} & \cdots & x_{2,s} \\ x_{3,1} & x_{3,2} & x_{3,3} & \cdots & x_{3,s} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ x_{m,1} & x_{m,2} & x_{m,3} & \cdots & x_{m,s} \end{pmatrix}$$

involving m independent measurements (traces) containing s samples per trace, where each $x_{i,j}$ is a sample from trace i at time j (row-major order).

There are usually different (randomly chosen and uniformly distributed) inputs for each trace i, such that

$$\mathbf{V} = \begin{pmatrix} v_{1,1} & v_{1,2} & v_{1,3} & \cdots & v_{1,b} \\ v_{2,1} & v_{2,2} & v_{2,3} & \cdots & v_{2,b} \\ v_{3,1} & v_{3,2} & v_{3,3} & \cdots & v_{3,b} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ v_{m,1} & v_{m,2} & v_{m,3} & \cdots & v_{m,b} \end{pmatrix},$$

where $v_{i,l}$ is the l_{th} input of trace i, for $l \in \{1, 2, \dots, b\}$ and b is the length of the plaintext.

DPA works with hypotheses on subkeys. Let v_i be the partial entry of row i of the matrix $\mathbf{V}$ that enters the computation of $z = f_k(v)$. That is, we get a hypothetical leakage matrix

$$\mathbf{Y} = \left(Y_1\ Y_2\ Y_3 \dots Y_p\right) = \begin{pmatrix} y_{1,1} & y_{1,2} & y_{1,3} & \cdots & y_{1,p} \\ y_{2,1} & y_{2,2} & y_{2,3} & \cdots & y_{2,p} \\ y_{3,1} & y_{3,2} & y_{3,3} & \cdots & y_{3,p} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ y_{m,1} & y_{m,2} & y_{m,3} & \cdots & y_{m,p} \end{pmatrix}$$

$$= \begin{pmatrix} \mathcal{L}(f_1(v_1)) & \mathcal{L}(f_2(v_1)) & \mathcal{L}(f_3(v_1)) & \dots & \mathcal{L}(f_p(v_1)) \\ \mathcal{L}(f_1(v_2)) & \mathcal{L}(f_2(v_2)) & \mathcal{L}(f_3(v_2)) & \dots & \mathcal{L}(f_p(v_2)) \\ \mathcal{L}(f_1(v_3)) & \mathcal{L}(f_2(v_3)) & \mathcal{L}(f_3(v_3)) & \dots & \mathcal{L}(f_p(v_3)) \\ \vdots & \vdots & \vdots & \dots & \vdots \\ \mathcal{L}(f_1(v_m)) & \mathcal{L}(f_2(v_m)) & \mathcal{L}(f_3(v_m)) & \dots & \mathcal{L}(f_p(v_m)) \end{pmatrix},$$

covering all p subkey candidates $i \in \{0, 1, \dots, p\}$.

Pearson Correlation Coefficient. In this paper, we use the Pearson correlation coefficient as the statistical test for DPA. It computes the correlation coefficient of each column of the leakage matrix with each column of the measurement matrix. DPA finally outputs the key hypothesis reaching the absolute maximum of correlation.

For completeness, we provide the explicit formula for the estimated Pearson correlation coefficient:

$$r_{X,Y} = \frac{\sum_{i=1}^{m} x_i y_i - \frac{1}{m}\sum_{i=1}^{m} x_i \cdot \sum_{i=1}^{m} y_i}{\sqrt{\sum_{i=1}^{m} x_i^2 - \frac{1}{m}(\sum_{i=1}^{m} x_i)^2} \cdot \sqrt{\sum_{i=1}^{m} y_i^2 - \frac{1}{m}(\sum_{i=1}^{m} y_i)^2}}. \quad (1)$$

3 Computations on Graphics Cards

General-purpose computing on graphics processing units (GPGPU) is the shift of computations that are traditionally handled by the *central processing unit* (CPU) or *host* processor, to the *graphics processing unit* (GPU), also known as *device*. In this paper, we focus on nVidia GPUs and CUDA [10] that can be programmed with *C for CUDA*, a C language derivative with special extensions.

The main unit of the device is the multiprocessor which is a set of a number of stream processors (depends on the generation) which share memory, caches, and an instruction unit. The multiprocessor creates, manages, and executes *threads* in hardware. As Figure 1 shows, a thread in CUDA is the smallest unit of parallelism that is executed concurrently with other threads (*warps*) on the hardware. Threads are organized in a *thread block*, a group of threads in which

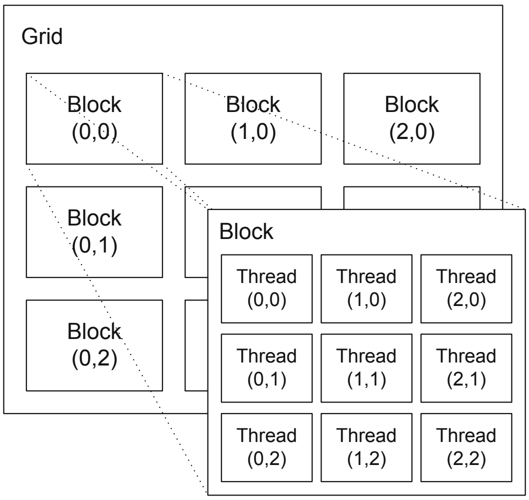


Fig. 1. CUDA thread hierarchy

the threads can communicate with each other and synchronize their state. A group of thread blocks is called a *thread grid*. A thread grid forms the execution unit in the CUDA model since it is not possible to execute a thread or thread block solely.

Threads in the CUDA programming model can access data from various memory spaces that differ in size and access time. The CUDA memory model (Fig. 2) describes the accessible memory spaces from the view point of the thread. At lowest level, a thread has read and write access to its own *registers* and additionally its own copy of *local memory*. Threads within the same block have read and write access to a *shared memory* on the next higher level. Beyond the block, all threads can have read and write access to the largest memory space, the *global memory*. Beside the global memory, there are two further spaces that are read-only, the *constant memory* and the *texture memory*. Usually, memory spaces that are

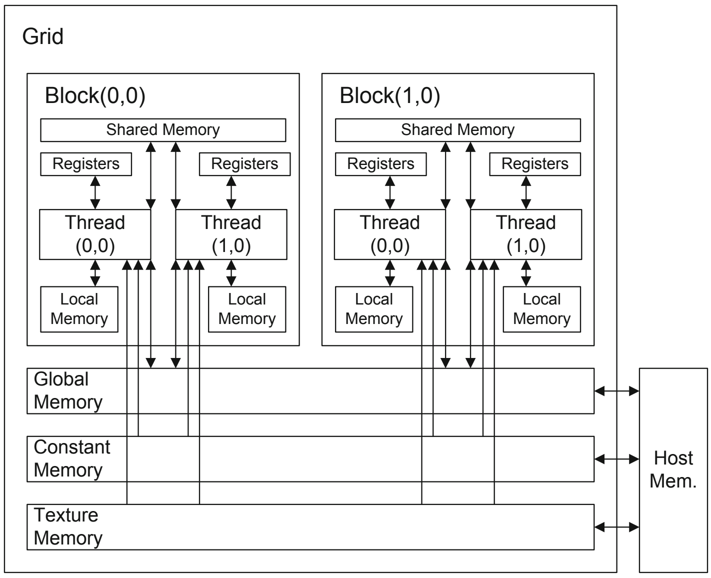


Fig. 2. CUDA memory model

shared by threads contain potential hazards of conflicts such as *read-after-write*, *write-after-read*, or *write-after-write*. Thus, the programming model implements a barrier by defining a synchronization instruction. As a consequence, a large number of divergent threads (i.e., threads which follow a different execution flow of an algorithm) require frequent synchronization, reducing the overall computation time of the entire systems due to wait cycles. Accesses to the global memory are crucial for the overall performance due to its latency. But most of that latency can be hidden if there are enough independent arithmetic instructions that are executed while waiting for access to complete.

4 Differential Power Analysis on Graphics Cards

The implementation of the Pearson correlation coefficient according to its representation (1) requires us to compute five sums:

$$\sum_{\forall i} x_i y_i, \quad \sum_{\forall i} x_i, \quad \sum_{\forall i} y_i, \quad \sum_{\forall i} x_i^2, \quad \text{and} \sum_{\forall i} y_i^2$$

Taking both matrices $\mathbf{X}$ and $\mathbf{Y}$ into account the first sum embodies the matrix multiplication

$$\mathbf{Y}^T * \mathbf{X} = \begin{pmatrix} Y_1 * X_1 & Y_1 * X_2 & Y_1 * X_3 & \ldots & Y_1 * X_s \\ Y_2 * X_1 & Y_2 * X_2 & Y_2 * X_3 & \ldots & Y_2 * X_s \\ Y_3 * X_1 & Y_3 * X_2 & Y_3 * X_3 & \ldots & Y_3 * X_s \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ Y_p * X_1 & Y_p * X_2 & Y_p * X_3 & \ldots & Y_p * X_s \end{pmatrix},$$

where X_i and Y_j are column, respectively row vectors of length m.

Matrix multiplications perform very well on graphics cards, and hence the idea is to build an implementation of the correlation coefficient based upon the matrix multiplication. The other sums could then be computed simultaneously. However, in this case prerequisite is the computation of the hypothetical leakage matrix $\mathbf{Y}$ beforehand. Additionally, we have to face some other issues that arise when we aim for an implementation that can handle arbitrary large measurement matrices. First, we have to keep in mind that the global memory of a graphics card is a constrained resource. Second, we probably run into numerical problems considering the single dot products, respectively the sums of large vectors (arithmetic overflow). Finally, we aim to distribute the computation of the correlation coefficient over an arbitrary number of graphics cards, respectively the computation has to be iteratively issuable if only one graphics card is available in the case of very large measurements. Therefore, our implementation approach consists of three major steps, i.e., CUDA kernels, that are carried out iteratively.

Initially, a kernel that computes the leakage model, next a kernel that performs the computations of the sums, and finally a kernel that computes the correlation coefficients. Apart from this approach it is, of course, possible to have one kernel that computes everything but that depends on the fact how large the matrices are. Here, we merely assume measurements being too large to be processed by one kernel at once but we will also briefly include this point into our considerations.

4.1 Leakage Model Creation

The leakage model is created by a kernel that is given an input vector $V \in \mathbf{V}$. Therefore, the row vectors of $\mathbf{Y^T}$, of which each is based on a copy of V, are distributed over different thread blocks, that is each row is computed among several

$$\begin{pmatrix} \boxed{\mathcal{L}(f_1(v_1))\ \mathcal{L}(f_1(v_2))\ \mathcal{L}(f_1(v_3))\ \ldots\ \mathcal{L}(f_1(v_m))} \\ \boxed{\mathcal{L}(f_2(v_1))\ \mathcal{L}(f_2(v_2))\ \mathcal{L}(f_2(v_3))\ \ldots\ \mathcal{L}(f_2(v_m))} \\ \boxed{\mathcal{L}(f_3(v_1))\ \mathcal{L}(f_3(v_2))\ \mathcal{L}(f_3(v_3))\ \ldots\ \mathcal{L}(f_3(v_m))} \\ \vdots \qquad \vdots \qquad \vdots \qquad \ldots \qquad \vdots \\ \boxed{\mathcal{L}(f_p(v_1))\ \mathcal{L}(f_p(v_2))\ \mathcal{L}(f_p(v_3))\ \ldots\ \mathcal{L}(f_p(v_m))} \end{pmatrix}$$

Fig. 3. Computation of $\mathbf{Y^T}$ among different thread blocks (outlined by solid lines)

threads within a thread block as depicted by Figure 3. As usual, the computation of the leakage prediction function $\mathcal{L}(f_k(v_i))$ is realized using a table, e.g. a S-box with precomputed Hamming weights, which is copied into the constant memory of the graphics card prior to the execution of the kernel and referenced later on. Eventually, this kernel can be omitted if the inputs are directly fed into the matrix multiplication kernel. This saves global memory because the leakage model matrix does not need to be stored. However, in some cases it might be more convenient to have a separated kernel when the leakage model is more complex for instance. Our straightforward approach is represented by Algorithm 1. As stated in the algorithm the integer values *tIdx.x* and *bIdx.x* represent the index of a single thread and thread block in the first dimension complying with the CUDA model.

Algorithm 1. Leakage Model Creation

```
Input: Input vector $V \in \mathbf{V}$
Output: Leakage model matrix $\mathbf{Y^T}$
1: for each block parallel do
2:     for each thread parallel do
3:         $\mathbf{Y}[bIdx.x, tIdx.x] = \mathcal{L}(f_{bIdx.x}(V[tIdx.x]))$
4:     end for parallel
5: end for parallel
```

4.2 Computation of the Sums

The computation of the correlation coefficient sums is, as already mentioned above, based upon the matrix multiplication $\mathbf{Y}^T * \mathbf{X}$. In order to achieve the best performance in the sense of DPA some effort has to be spent. Regarding arbitrary large matrices we have to keep in mind that at some point the matrices exceed the global memory of the graphics card. From this point of view we can follow two approaches, first a kernel that is given the matrices and computes the correlation coefficient directly, because the matrices fit the global memory. Contrary, if the matrices do not fit the memory, a kernel is needed that outputs the sums for a later processing. In the remainder we deal with the latter opportunity but the similarity is high. However, these deliberations also lead us to the next implementation decision. Each sum has to be stored in a single variable and a 32-bit variable may not be sufficient in general because of a potential overflow. This is aggravated by the fact that the CUDA performance involving 64-bit variables is quite low [11]. Nevertheless, the framework is highly optimized to employ 32-bit floating arithmetic. So to address these issues the correlation coefficients, respectively their implied sums, are computed as follows.

$$r_{X,Y} = \frac{\sum_{i=1}^{m} \frac{1}{m} x_i y_i - \sum_{i=1}^{m} \frac{1}{m} x_i \cdot \sum_{i=1}^{m} \frac{1}{m} y_i}{\sqrt{\sum_{i=1}^{m} \frac{1}{m} x_i^2 - (\sum_{i=1}^{m} \frac{1}{m} x_i)^2} \cdot \sqrt{\sum_{i=1}^{m} \frac{1}{m} y_i^2 - (\sum_{i=1}^{m} \frac{1}{m} y_i)^2}} \qquad (2)$$

Shifting the fractal into the sums does not necessarily cause a performance penalty due to potential latency hiding since it also works vice versa by which means additional instructions are covered by the waiting time.

Referring to [12] the resultant matrix of the multiplication is computed by two-dimensional thread blocks, here called *tiles*, that are distributed as shown in Figure 4. Each tile consists of n^2 threads where a single thread is responsible to compute a single dot product of the entire resultant matrix. At the beginning a tile loads portions such that n^2 elements of $\mathbf{X}$ and n^2 of $\mathbf{Y^T}$ are deposited into the shared memory of the thread block. This strategy avoids loading every vector each time it is needed. With these portions a thread can now compute the first n products of a dot product, since the first n elements of each row vector of $\mathbf{Y^T}$ and each column vector of $\mathbf{X}$ are loaded. Afterwards, a tile fetches the next portions to compute the next n products, a procedure which is repeated until all elements are passed through. Obviously, we can exploit synergies and compute all other correlation coefficient sums concurrently since all necessary elements are already loaded. Figuratively, the tile move rightwards regarding

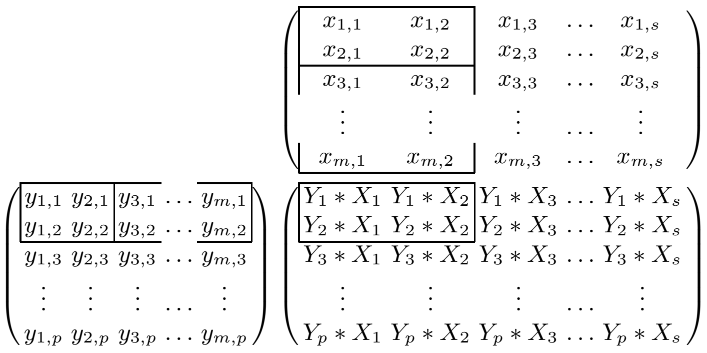

$x_{1,1}$ $x_{1,2}$ $x_{1,3}$ … $x_{1,s}$
$x_{2,1}$ $x_{2,2}$ $x_{2,3}$ … $x_{2,s}$
$x_{3,1}$ $x_{3,2}$ $x_{3,3}$ … $x_{3,s}$
$x_{m,1}$ $x_{m,2}$ $x_{m,3}$ … $x_{m,s}$
$y_{1,1}$ $y_{2,1}$ $y_{3,1}$ … $y_{m,1}$
$y_{1,2}$ $y_{2,2}$ $y_{3,2}$ … $y_{m,2}$
$y_{1,3}$ $y_{2,3}$ $y_{3,3}$ … $y_{m,3}$
$y_{1,p}$ $y_{2,p}$ $y_{3,p}$ … $y_{m,p}$
$Y_1 * X_1$ $Y_1 * X_2$ $Y_1 * X_3$ … $Y_1 * X_s$
$Y_2 * X_1$ $Y_2 * X_2$ $Y_2 * X_3$ … $Y_2 * X_s$
$Y_3 * X_1$ $Y_3 * X_2$ $Y_3 * X_3$ … $Y_3 * X_s$
$Y_p * X_1$ $Y_p * X_2$ $Y_p * X_3$ … $Y_p * X_s$

Fig. 4. Computation of $\mathbf{Y^T} * \mathbf{X}$ among different two-dimensional thread blocks, here called tiles. Exemplarily, only one tile is emphasized to show which portions of the matrices $\mathbf{X}$ and $\mathbf{Y}$ are involved to compute the resultant sub-matrix covered by that tile. Actually, the whole resultant matrix is covered by several tiles.

$\mathbf{Y^T}$ and downwards regarding $\mathbf{X}$ (Fig. 4). The kernel, constituted by its optimized version, is depicted in Algorithm 2. Additionally, the algorithm reveals two mandatory optimizations that were not mentioned so far. The portions are prefetched by the tile threads into their registers first and then deposited into shared memory with the effect that the sum calculations only consume already fetched tile elements while the next elements are being loaded. This enables latency hiding. The second optimization considers the workload balance within kernel. Therefore, a number of tiles of matrix $\mathbf{X}$, instead of one, are loaded horizontally to compute multiple dot products involving the loaded single tile of $\mathbf{Y^T}$.

Algorithm 2. Computation of correlation coefficient sums

Input: Leakage model matrix $\mathbf{Y^T}$ and measurement matrix $\mathbf{X}$
Output: Sums of corr. coef.: $\sum_{\forall i} \frac{1}{m} x_i y_i, \sum_{\forall i} \frac{1}{m} x_i, \sum_{\forall i} \frac{1}{m} y_i, \sum_{\forall i} \frac{1}{m} x_i^2$, and $\sum_{\forall i} \frac{1}{m} y_i^2$

```
 1: for each block parallel do
 2:    for each thread parallel do
 3:       prefetch first tile of $\mathbf{Y^T}$ and first horizontal tiles of $\mathbf{X}$ into registers: $y_i$, $x_i$.
 4:    end for parallel
 5: end for parallel
 6:
 7: for k = 1 to $\frac{m}{n}$ do
 8:    for each block parallel do
 9:       for each thread parallel do
10:          $i \leftarrow$ thread position within column vectors of $\mathbf{X}$
11:          $j \leftarrow$ thread position within row vectors of $\mathbf{Y^T}$
12:          deposit prefetched tiles into shared memory: $X^{shared}[tIdx.x, tIdx.y] = \frac{x_i}{\sqrt{m}}$,
             $Y^{shared}[tIdx.x, tIdx.y] = \frac{y_j}{\sqrt{m}}$
13:          prefetch next tiles into registers: $x_{i+k}$, $y_{j+k}$
14:          for l = 1 to $n$ do
15:             $\sum_{thread} x_i y_i = \sum_{thread} x_i y_i + X^{shared}[tIdx.x, l] \cdot Y^{shared}[l, tIdx.y]$
16:             $\sum_{thread} x_i = \sum_{thread} x_i + X^{shared}[tIdx.x, l] \cdot \frac{1}{\sqrt{m}}$
17:             $\sum_{thread} y_i = \sum_{thread} y_i + Y^{shared}[l, tIdx.y] \cdot \frac{1}{\sqrt{m}}$
18:             $\sum_{thread} x_i^2 = \sum_{thread} x_i^2 + X^{shared}[tIdx.x, l]^2$
19:             $\sum_{thread} y_i^2 = \sum_{thread} y_i^2 + Y^{shared}[l, tIdx.y]^2$
20:          end for
21:       end for parallel
22:    end for parallel
23: end for
```

4.3 Computation of the Correlation Coefficient Matrix

This kernel implementation is similar to that of the leakage model creation. The matrix of the correlation coefficients is segmented in the same way (Fig. 3). Every thread block is responsible for samples that are related to one key hypothesis, hence a block is given the corresponding correlation coefficient sums which result from the measurement matrix and any sum from the leakage model matrix. The implementation is shown in Algorithm 3.

Algorithm 3. Computation of correlation coefficient matrix

```
Input: Sums of corr. coef.: $\sum_{\forall i} \frac{1}{m} x_i y_i$, $\sum_{\forall i} \frac{1}{m} x_i$, $\sum_{\forall i} \frac{1}{m} y_i$, $\sum_{\forall i} \frac{1}{m} x_i^2$, and $\sum_{\forall i} \frac{1}{m} y_i^2$
Output: Correlation coefficient matrix $\mathbf{R}$
1: for each block parallel do
2:    for each thread parallel do
3:       $\mathbf{R}[bIdx, tIdx] = r_{X_{\mathrm{tIdx}}, Y_{\mathrm{bIdx}}}$, complying with (2) and (3)
4:    end for parallel
5: end for parallel
```

4.4 Special Case: Hamming Weight Model

In the case of the Hamming Weight model, a further approach to achieve an even better performance outcome is to estimate the mean and standard deviation of the leakage model. This would save computational effort, above all divisions and square root calculations which should be avoided. These operations only offer one fourth, respectively one eighth of the performance of a multiplication [11]. In addition to that, estimation also saves global memory due the avoided sums that do not need to be stored anymore.

Presuming that the inputs, contained in $\mathbf{V}$, are randomly chosen and uniformly distributed the mean $\sum_{\forall i} \frac{1}{m} y_i$ can be estimated with

$$\sum_{\forall i} \frac{1}{m} y_i = E[HW(Z)] = E[\sum_{i=1}^{b} Z(i)] = b \cdot E[Z(i)] = \frac{b}{2},$$

where $Z(i)$ is the i_{th} bit of random variable Z, i.e., Z is a b-bit variable. Whereas the mean of the squares $\sum_{\forall i} \frac{1}{m} y_i^2$ can be estimated with

$$\sum_{\forall i} \frac{1}{m} y_i^2 = E[HW(Z)^2] = E[\sum_{i,j=1}^{b} Z(i) \cdot Z(j)] = \sum_{i \neq j}^{b} E[Z(i) \cdot Z(j)] + \sum_{i}^{b} E[Z(i)]$$
$$= b \cdot (b-1) \cdot E[Z(i)Z(j)] + b \cdot E[Z(i)] = \frac{b \cdot (b-1)}{4} + \frac{b}{2} = \frac{b^2 + b}{4}.$$

Eventually, we obtain the correlation coefficient with leakage estimation being expressed as

$$r_{X,Y} = \frac{\sum_{i=1}^{m} \frac{1}{m} x_i y_i - \frac{b}{2} \cdot \sum_{i=1}^{m} \frac{1}{m} x_i}{\sqrt{\frac{b}{4}} \cdot \sqrt{\sum_{i=1}^{m} \frac{1}{m} x_i^2 - (\sum_{i=1}^{m} \frac{1}{m} x_i)^2}}. \quad (3)$$

5 Experimental Results

For our experiments, we used nVidia Tesla C2070 graphics cards with 6 *GiB* video RAM and an Intel Xeon X5660 at 2.8 *GHz* running Windows 7 64-bit. The results were obtained using the CUDA toolkit and SDK of version *3.2*, the CUDA driver *270.61*, and the Microsoft Visual C++ compiler.

We presume attacking a sequential 8-bit implementation of AES [9] to recover one subkey byte in the Hamming Weight model. In order to gain meaningful results the inputs are composed of byte values that are randomly chosen and uniformly distributed. Since in most cases employing a DSO with a vertical resolution of eight bit suffices, we set $\mathbf{X} \in_R [-127, 127]^{m \times s}$.

First of all the best kernel configuration has to be figured out, more precisely the thread and thread block numbers. For the leakage creation and the correlation coefficient kernel it is quite obvious that they have to be launched with p thread blocks (one block per key hypothesis). Regarding the correlation coefficient sum kernel two constraints show up, the maximum number of threads a kernel can take and the shared memory in use that is dictated by this number. Actually, we have at most 1536 threads per kernel on our graphics card, thus the tile size could be $n_{max} = \lfloor \sqrt{1536} \rfloor = 39$, but due to the restricted shared memory and the horizontal tiles (one tile computes more than one portion of $\mathbf{X}$) we need to find out the optimal trade-off. Through empirical testing, it turned out that the kernel performs best with $n = 28$, that is $n^2 = 784$ threads per block and four horizontal tiles while barely not exceeding the available shared memory. The number of tiles (thread blocks) can be obtained by simply dividing each dimension, the number of key hypotheses p and the number of given samples s, by n. Another limitation is the total global memory M_{global} which accommodates the measurement matrix, the input vector, and the five correlation coefficient sums. Presuming single precision variables for the sums and 8-bit variables for the samples and inputs, the global memory usage consists of $m \cdot s$ bytes for the measurement matrix, m bytes for the inputs, $p \cdot s \cdot 4$ bytes for the covariance, and $2 \cdot p \cdot 4$ bytes, respectively $2 \cdot s \cdot 4$ bytes for the variances and means. Eventually, we obtain the inequality

$$ms + m + 4ps + 8p + 8s < M_{global}.$$

The runtime was then measured in steps of $10k$ traces and a number of samples fixed to $20k$. Figure 5 shows the results for the following variations of the sums kernel: the kernel is given the precomputed leakage model, the kernel is given the input vector directly (cf. Sec. 4.1), and further both variants with the leakage estimation (cf. Sec. 4.4). Additionally, we show the runtime for the data transfer between the host memory and the device memory.

As it can be seen the kernel applying the leakage estimation performs slightly better and further it can be seen that the effort increases linearly with the number of traces. It is not worthwhile to compute the leakage model beforehand which is most likely caused by the frequent global memory accesses. Furthermore, we get the same results if we fix the number of traces and iteratively increase

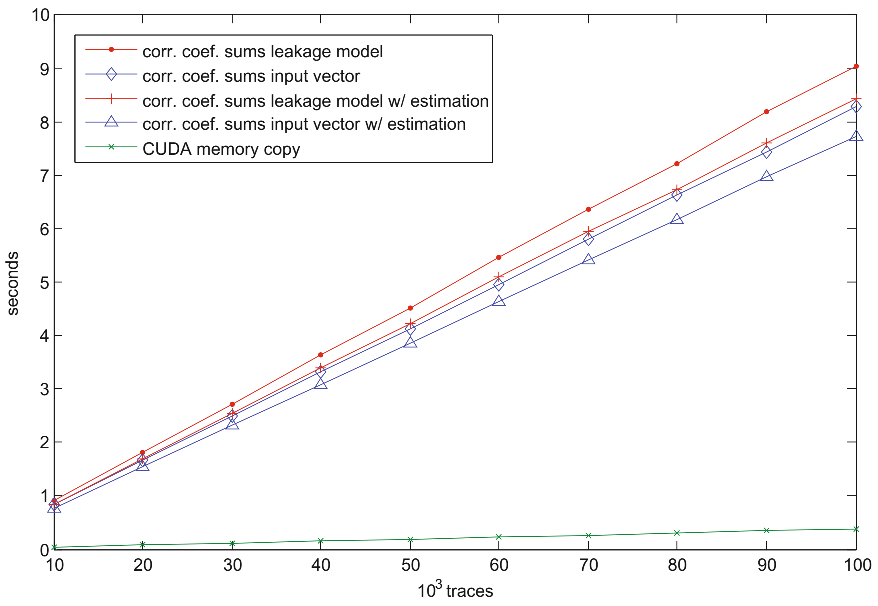


Fig. 5. Runtime of the correlation coefficient sums kernel. The kernel can either be given the leakage model or the input vector directly.

the number of samples by which means the measurement matrix can be cut at any point to make it fit the graphics cards global memory in the case of very large measurements. Since the effort increases linearly with both, the number of traces and the number of samples, the implementation is perfectly scaled with additional graphics cards. The runtimes of the remaining two kernels, namely leakage model creation and correlation coefficient computation, are absolutely negligible while being in the range of a few milliseconds. It is not surprising that their runtimes hardly contribute to the overall runtime since the elements of the resultant matrices are independent of each other, in marked contrast to the sums kernel where as well thread synchronization is vital due to the usage of shared memory. That is also true for the overhead, i.e., transferring data from the host memory to the device memory and vice versa, respectively the kernel launches. However, data transfers are dependent on the data and the time increases linearly with the amount of bytes. Figure 5 thus shows the respective transfers of the full measurement matrix. The influence of I/O, i.e., loading traces from a hard disk, is not considered because this also affects a CPU implementation in the same way. However, measurements containing over one million traces can be analyzed in less than two minutes employing just one graphics card.

Furthermore, we provide a comparison between the CPU and the GPU. Therefore, we implemented and optimized the sums kernel with leakage estimation on the CPU which does exactly the same as its GPU counterpart. Table 1 compares the results. As expected, the CPU implementation is, as well, linearly scaled with the number of traces. Hence, we can derive a speed-up factor of about almost 100 taking a common processor with four cores into account, a performance gain that obviously suggests CUDA as a very promising platform for DPA.

Table 1. Runtime comparison between CPU and GPU where one thread runs the CPU implementation. The number of samples is fixed to $20k$.

	$10k$ traces	$20k$ traces	$50k$ traces	$100k$ traces
GPU	0.774 s	1.545 s	3.861 s	7.733 s
CPU	302.72 s	622.11 s	1531.69 s	3152.21 s

6 Conclusion and Future Work

In this paper we presented a highly performant implementation of Differential Power Analysis on graphics cards. The implementation can handle arbitrary large measurement matrices which can be split up at any point to make them fit into the graphics cards memory. Large measurements can be analyzed within a few minutes. Our ongoing work will deal with the implementation of DPA based higher order attacks.

Acknowledgements. The work described in this paper was supported by the German Federal Ministry of Education and Research (BMBF) through the EXSET project (promotional reference 01IS10026B). The information in this document reflects the views only of the authors.

References

1. Bernstein, D.J., Chen, T.R., Cheng, C.M., Lange, T., Yang, B.Y.: ECM on Graphics Cards. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 483–501. Springer, Heidelberg (2009)
2. Brier, E., Clavier, C., Olivier, F.: Correlation Power Analysis with a Leakage Model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)
3. Harrison, O., Waldron, J.: AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 209–226. Springer, Heidelberg (2007)
4. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
5. Lee, S.J., Seo, S.C., Han, D.G., Hong, S., Lee, S.: Acceleration of Differential Power Analysis through the Parallel Use of GPU and CPU. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E93.A(9), 1688–1692 (2010)
6. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks. Springer, Heidelberg (2007)

7. Mangard, S., Pramstaller, N., Oswald, E.: Successfully Attacking Masked AES Hardware Implementations. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 157–171. Springer, Heidelberg (2005)
8. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 69–88. Springer, Heidelberg (2011)
9. National Institute of Standards and Technology: Advanced Encryption Standard (AES). Federal Information Processing Standards Publications 197 (2001), http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
10. nVidia: NVIDIA CUDA Development Tools (2010), http://developer.download.nvidia.com/compute/cuda/3_2/docs/Getting_Started_Windows.pdf
11. nVidia: NVIDIA CUDA Programming Guide (2010), http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf
12. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Longman, Amsterdam (2010)
13. Szerwinski, R., Güneysu, T.: Exploiting the Power of GPUs for Asymmetric Cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 79–99. Springer, Heidelberg (2008)

RAM: Rapid Alignment Method

Ruben A. Muijrers[1], Jasper G.J. van Woudenberg[2], and Lejla Batina[1,3]

[1] Radboud University Nijmegen, ICIS/Digital Security group
Heyendaalseweg 135, 6525 AJ Nijmegen, The Netherlands
rubenmuijrers@yahoo.com, lejla@cs.ru.nl
[2] Riscure BV, 2628 XJ Delft, The Netherlands
vanwoudenberg@riscure.com
[3] K.U.Leuven ESAT/SCD-COSIC and IBBT
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
lejla.batina@esat.kuleuven.be

Abstract. Several countermeasures against side-channel analysis result in misalignment of power traces, in order to make DPA more difficult. In this paper we propose a new algorithm to align the measurements after this desynchronizing through the variations of the internal clock, random delays, etc. The algorithm is based on the ideas of SIFT and U-SURF algorithm that were originally proposed for image recognition. The comparison with other known methods favors our solution in terms of efficiency and computational complexity.

Keywords: smart cards, side-channel analysis, DPA, random delay countermeasure, alignment methods.

1 Introduction

Small electronic devices such as smart phones, PDAs and smart cards are becoming increasingly popular. As a side effect of this trend, more critical information is stored in these devices and therefore it is crucial that they are secure. As algorithms used in modern chips are mathematically rather secure, attackers shift focus to specific implementations and the secret information that leaks through physics.

One of the focus areas is the power consumption of a device. About a decade ago Kocher et al. proposed DPA, a method that allows attackers to extract the secret key used for cryptographic operations from a small device such as a smartcard [7]. Since then many countermeasures have been developed, as well as ways to reduce the effects of these countermeasures.

Kocher's method and its many variants depend on the assumption that during the encryption, the timing of each operation during the cryptographic operation is constant between multiple executions of the algorithm. By adding dummy operations at random points in time or using an unstable clock this assumption is broken, and thereby DPA attacks become less effective. However, there exist algorithms that attempt to recover DPA information even in this case of the traces being misaligned e.g. Static Alignment [9] and Elastic Alignment [10].

The power traces are altered such that the target is susceptible to a DPA attack again. Static alignment is fast, but it only aligns one point of the cryptographic operation and does not take into account differences in the timing of the individual operations. Elastic Alignment performs continuous alignment, but it is computationally intensive.

Our proposed algorithm is designed to perform continuous alignment, without the drawback of the high computational requirements of Elastic Alignment. The algorithm is inspired by U-SURF [2] which, given a reference picture, is used to recognize pictures of the same scene/object taking into account differences in angle and light. The proposed algorithm uses several techniques used in U-SURF, among most notably the use of block wavelets for detection and identifying of specific points in the recorded power signal. The main advantage of block wavelets is that they can be applied in $O(1)$, which improves the running time substantially.

This paper is organized as follows. Section 2 describes relevant prior work. In Sect. 3 we explain the main ideas behind wavelets as a starting point for the new method. We present the algorithm and its main components in Sect. 4. Our experiments and results are given in Sect. 5. Finally, Sect. 6 concludes this work.

2 Related Work

Alignment algorithms try to reduce the effects of an unstable clock and dummy operations by aligning all traces in a set to a reference trace, which is a common approach. In this paper we refer to the trace we align to as the *reference trace* and the traces that need to be aligned as the *target traces*.

Static alignment was described by Mangard et al. in 2007 [9], see Ch. 8. The idea is as follows: the attacker chooses a fragment in a reference trace close to the area where the attack takes place. Then the algorithm aims to find this same fragment in the other traces and shifts the other trace so that the reference fragments are aligned. Although this does not fully counter an unstable clock or random delays, it often does reduce the number of traces needed to successfully perform a DPA attack.

Elastic Alignment [10] attempts to match each sample in a target trace to samples in the reference trace. It can match sequences of samples in the target trace to a single sample in the reference trace, and vice versa. The algorithm minimizes the sum of the different sample values between matched samples. This match is used to stretch and compress samples in the target trace to match the reference trace. The result is an algorithm that can perform a continuous synchronization of any two traces. The authors show Elastic Alignment runs in $O(n)$, with n the number of samples in the trace to align. However, the continuous matching procedure is still quite computationally expensive.

Sliding Window DPA was proposed by Clavier et al. in 2000 [6]. as an algorithm that is specifically designed to counter random process interrupts (RPI). When RPI are used as a countermeasure the position of the leakage that is exploited by DPA can shift a few clock cycles. Each clock cycle in a power trace is replaced by the average of itself and a number of previous clock cycles. The two main parameters of this method are the number of cycles to average (the window size) and the number of samples per one clock cycle. In [10] it was shown that SW-DPA performs fairly well. However when an unstable clock is used the performance drops drastically. This is not surprising since the algorithm assumes a stable clock of which the frequency is set in parameters of the algorithm.

SIFT stands for Scale Invariant Feature Transform. It is a feature generation method proposed by Lowe in 1999 [8]. The features generated are used to recognize objects in images. Object recognition algorithms have to be robust against scaling, translation, rotation and noise in the images, which are similar problems to those due to the misalignment. U-SURF [2] stands for Upright-SURF where SURF stands for Speeded Up Robust Features. U-SURF does the same as SURF, except that it skips a step where orientations of the points of interest (POI). Our solution is inspired by both, as we translate the ideas of SIFT and U-SURF from the 2 dimensions (images) to one dimension (power traces).

In this work we aim at creating a continuous alignment algorithm, with a strong focus on improving computational complexity.

3 Alignment with Wavelets

To align two power traces one first needs to choose points in these power traces to align upon. In Elastic Alignment, the points used for alignment are all the samples in every resolution of each trace. This results in the calculation being computationally demanding. The proposed algorithm uses far less points for alignment and interpolates in between these points.

The points that are used to align on must be efficient to compute and recognizable in each trace. To find these *points of interest* (POI) in multiple traces the proposed algorithm searches for certain patterns in the traces. These patterns are identified with the use of wavelets. Wavelets give information about a power trace with excellent temporal resolution, as opposed to a Fourier Transform which has a trade-off between frequency resolution and temporal resolution. The benefits of wavelets in getting rid of noise were already presented in [5], but they were not used for re-synchronization. Another important advantage is that the wavelets used in the proposed algorithm can be applied in $O(1)$. This is further explained in the remainder of this section.

3.1 Wavelets

The term "wavelet" means small wave. It is a signal with an amplitude that starts out at zero, than increases, and then decreases back to zero. An example of this can be created by multiplying a sine wave with the Gaussian distribution function. This will result in a Morlet wavelet as can be seen in Fig. 1. Given a wavelet function the wavelet response $WR(t, \psi)$, of a wavelet defined by function $\psi(t)$ at a point t in a signal defined by function $f(t)$, can be computed by

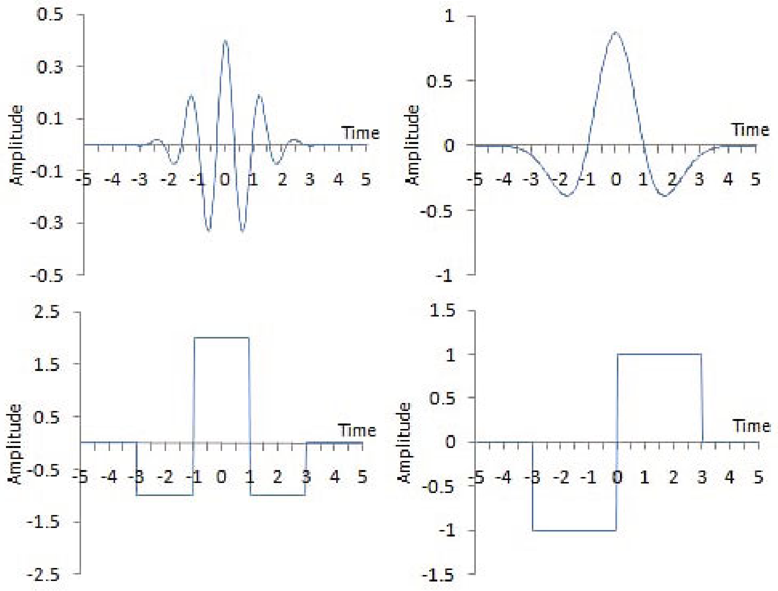


Fig. 1. Examples of a Morlet wavelet (top-left), Mexican Hat wavelet (top-right), Mexican Hat block wavelet (bottom-left) and Haar wavelet (bottom-right)

calculating the convolution of the wavelet and the signal at that point as is shown in equation (1):

$$WR(t, \psi) = \int f(x) \cdot \psi_{\alpha,t}(x)dx, \tag{1}$$

where $\psi_{\alpha,\beta}$ is the wavelet function ψ scaled with α and translated with β as is shown in equation (2):

$$\psi_{\alpha,\beta}(t) = \frac{1}{\sqrt{\alpha}}\psi(\frac{t-\beta}{\alpha}). \tag{2}$$

When working with power traces the signal is not defined by a continuous function. Instead it is defined by a set of discrete data points. The wavelet response function reduces then to a summation which is shown in the following sum:

$$WR(t, \psi) = \sum_{x=0}^{len} f(x) \cdot \psi_{\alpha,t}(x). \tag{3}$$

where len is the number of sample points in the power trace.

For detection of POIs, wavelet responses are calculated at several scales and at several sample points in the trace. Two patterns have been tried, the first being a peak pattern which was identified with the Mexican Hat wavelet shown in Fig. 1. The second was the slope pattern which was identified with the Haar wavelet also shown in Fig. 1. To be able to compare wavelet responses of different scales the wavelet responses are divided by $\sqrt{s}$ where s is the scale of the wavelet.

3.2 Block Wavelets

The calculation of a wavelet response as defined in equation (3) takes $O(n)$ to compute, where n is the length of the power trace. By using the 1-dimensional variant of summed-area tables [1] this can be reduced to $O(1)$ when using block wavelets such as the Haar wavelet or the approximation of the Mexican Hat wavelet. First, the power traces that need to be aligned are transformed in a *summed trace*. Each element at position t in the summed trace is the sum of all the samples before and including sample t in the power trace. The formal definition of the summed trace S of power trace P is given in equation (4):

$$\begin{aligned} S[0] &= P[0] \\ S[t] &= S[t-1] + P[t]. \end{aligned} \tag{4}$$

The convolution of the Haar wavelet and the block version of the Mexican Hat wavelet with scale s at position t in summed trace S can now be calculated efficiently: the Haar wavelet now requires only 3 read operation regardless of its scale and the Mexican Hat requires 4. This follows directly from their definitions applied to summed traces, the result of which is shown in equations (5) and (6):

$$\begin{aligned} Haar(S,t,s) &= \frac{-(S[t] - S[t-\frac{s}{2}]) + (S[t+\frac{s}{2}] - S[t])}{\sqrt{s}} \\ &= \frac{-2 \cdot S[t] + S[t-\frac{s}{2}] + S[t+\frac{s}{2}]}{\sqrt{s}}, \end{aligned} \tag{5}$$

$$\begin{aligned} MexicanHat(S,t,s) &= \frac{-(S[t-\frac{s}{6}] - S[t-\frac{s}{2}]) + 2 \cdot (S[t+\frac{s}{6}] - S[t-\frac{s}{6}]) - (A[t+\frac{s}{2}] - S[t+\frac{s}{6}])}{\sqrt{s}} \\ &= \frac{3 \cdot (S[t+\frac{s}{6}] - S[t-\frac{s}{6}]) - (S[t+\frac{s}{2}] - S[t-\frac{s}{2}])}{\sqrt{s}}. \end{aligned} \tag{6}$$

4 New Algorithm

The proposed algorithm is inspired by U-SURF [2]. It has several tunable parameters but in order to ease the tuning attackers need to do, every parameter in the algorithm is related to properties of the trace.

The algorithm consists out of four components: Detector, Descriptor, Matcher and Warper. First the Detector finds *points of interest* (POI) in the reference trace and the target traces. The Descriptor then generates a feature vector for each POI based on their context. Using these vectors the Matcher will match POIs in the reference trace with POIs in the target trace. Finally, the Warper uses the matched points to stretch and shrink the target trace and to align them with the reference trace. This is a recursive process starting with POIs detected with large wavelets. Each recursive step the algorithm repeats itself on trace parts in between the POIs from the previous step with a reduced wavelet scale.

A *POI* in this paper is a data structure containing its position in a trace, the wavelet scale which was used to detect it and a feature vector. The position and scale are set by the Detector and the feature vector is generated by the Descriptor. An *area* in the code consists of two positions marking the start (inclusive) and end (inclusive) of the area. A *match* consists of a position in the reference trace, a position in the target trace and a confidence value.

In the following sub-sections a detailed description is given for each of the four components. The main procedure of the algorithm is shown in Algorithm 1.

Algorithm 1. The main loop of RAM

```
procedure RAMInner($m$:matches, $ref$:sumtrace, $tar$:sumtrace, $aref$:area, $atar$:area, $ws$:wavsize)
    if $ws < Det_{mw} \vee atar.size = 0 \vee aref.size = 0$ then
        return
    end if

    $refpois = \text{DETECT}(ref, aref, ws)$
    $tarpois = \text{DETECT}(tar, atar, ws)$
    $\text{DESCRIBE}(ref, refpois)$
    $\text{DESCRIBE}(tar, tarpois)$
    $mat = \text{MATCH}(refpois, tarpois)$

    if isEmpty(mat) then
        $\text{RAMInner}(m, ref, tar, aref, atar, ws/2)$
        return
    end if
    for All areas $(mref, mtar)$ between neighbouring matches in $mat$ do
        $\text{RAMInner}(m, ref, tar, mref, mtar, ws/2)$
        $\text{addMatch}(m, mref.end, mtar.end)$
    end for
end procedure

procedure RAM($ref$:trace, $tar$:trace)
    $mat = (0,0)$
    $ws = \text{argmax}_x\,(x = Det_{mw} \cdot 2^k \mid k \in \mathbb{N} \wedge x < ref.size)$
    $\text{RAMInner}(mat, MakeSumTrace(ref), MakeSumTrace(tar), \{0, ref.size - 1\}, \{0, tar.size - 1\}, ws)$
    return $\text{WARP}(tar, mat)$
end procedure
```

4.1 Detector

The Detector detects POIs in the reference and target trace. A POI is a pattern in the trace that can be quickly and repeatedly recognized in other traces of the same set. The entire Detector procedure is shown in Algorithm 2.

POIs are found by performing a wavelet transform. The response for wavelets of various scales is calculated for the trace. The algorithm starts with the largest wavelet scale that satisfies this predicate $Det_{mw} \cdot 2^k < l$ with k being a natural number, l being the trace length and Det_{mw} being the minimum wavelet scale. For each iteration the scale is halved until Det_{mw} is reached. Due to the type of wavelet used (Mexican Hat) Det_{mw} should be a multiple of 3.

For performance reasons the wavelet responses are not calculated for every sample but with a step size of 10% of the wavelet scale. A small step size slows down the algorithm, whereas a big step size does not detect sufficient POIs. Preliminary testing shows a value of 10% yields a good tradeoff. To be able to compare the responses to a constant threshold, the responses are normalized with respect to the wavelet scale. Samples with an absolute wavelet response less than Det_{th} times the standard deviation of the trace are discarded. The constant Det_{th} will be a parameter of the algorithm.

A *best-of-its-neighbors* filter is then applied to the remaining samples. If the absolute wavelet response of a sample is not greater than the absolute wavelet responses of its neighbors, the sample is discarded. Neighbors are defined here as every sample within 3 times the step size from the sample under evaluation. Early testing showed that at least 3 times the step size is needed here, as otherwise too many POIs survive the filter, whereby the matcher will not be able to differentiate them properly.

The samples that remain after the filter will be the POIs. Since these are the points that will be used for alignment later on it is important that they are located as accurate as possible. To achieve this the remaining responses will be pinpointed by searching for the highest wavelet response in the vicinity (the step size) of the point.

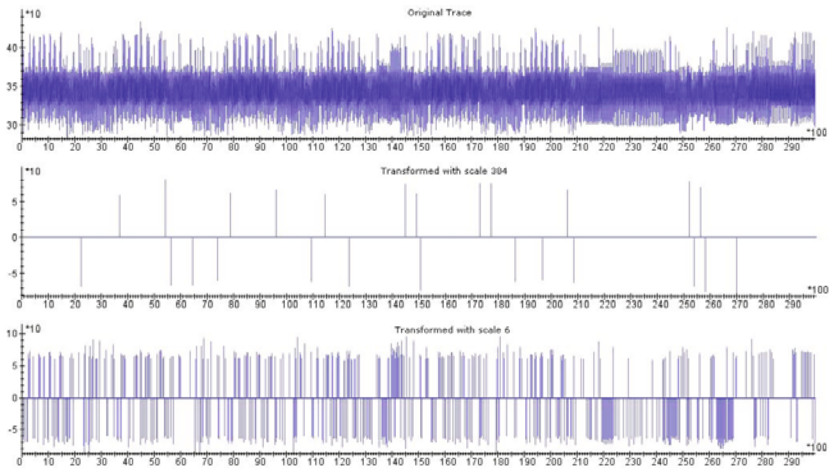


Fig. 2. A power trace (top) with its points of interest for two scales (middle=384, bottom=6)

Figure 2 shows the result of the Detector. The first image is the original trace followed by two images that show the POIs found by the detector at different scales. As is expected, large wavelets result in much fewer POIs than small wavelets. Therefore it is more efficient to match the POIs from large scale wavelets to POIs from another trace. On the other hand, smaller wavelets give more information on how to align the traces.

The type of wavelet used here defines on which patterns the responses will be high. Searching for slope patterns (by using Haar wavelets) and for peak patterns (by using Mexican hat block wavelets) was tried. Both wavelets are shown in Fig. 1. We found that in the traces we analyzed that the Mexican Hat block wavelet outperformed the Haar wavelet in finding the same POIs in similar traces.

Algorithm 2. The main loop of the Detector

```
procedure DETECT($S$:sumtrace, $a$:area, $ws$:wavsize)
    $B$ = buffer trace
    $step = max(1, ws/10)$
    $pois$ = empty list
    for $i = a.start; i < a.end; i+ = step$ do
        $B[i]$ = MEXICANHAT($S, i, ws$)
        if $abs(B[i]) < Det_{th} * stddev(RefTrace)$ then        ▷ Below threshold?
            $B[i] = 0$
        end if
    end for

    for $i = a.start; i < a.end; i+ = step$ do
        if $B[i] \neq 0 \wedge$ BESTOFNEIGHBORS($B, i$) then        ▷ If best, add
            ADDPOI($pois, PinPoint(S, i), ws$)
        end if
    end for
    return $pois$
end procedure
```

4.2 Descriptor

The descriptor (Algorithm 3) aims to uniquely describe each POI by its context. It generates a feature vector which will be used by the matcher to calculate the distance between two POIs. These features must be robust to noise and because there are many POIs coming from one single trace, the features must be fast to calculate. We use the same approach as in U-SURF: an area of Des_{as} times the scale around each POI is divided in several sections as can be seen in Fig. 3. Des_{as} is a tunable parameter of the algorithm. For each of the sections a few simple features are calculated. The sections are used to retain temporal information.

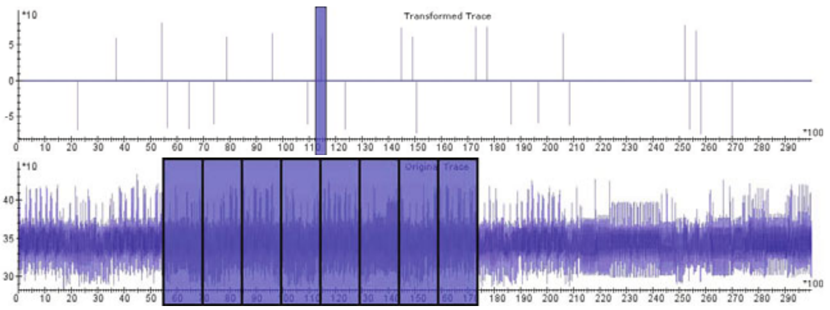


Fig. 3. A selected POI (top) and 8 sections which are used for the features (bottom)

For each section a number Des_{ss} of Haar wavelet responses are calculated (using the summed trace) at constant distance from each other with a wavelet scale of Des_{hw} times the scale of the POI. In order to focus on the area directly around the POI, a Gaussian curve centered at the POI and with a standard deviation of Des_{ga} is used to normalize the Haar wavelet response.

To include information about the polarity of the section all the wavelet responses are summed, and to include information about the intensity of the section all the absolute wavelet responses are summed. The number of sections is based on the feature count Des_{fc} (2 features per section). The constants Des_{hw}, Des_{ss}, Des_{fc}, Des_{ga} are tunable parameters of the algorithm.

4.3 Matcher

The Matcher (Algorithm 4) creates a mapping between the POI set of the reference trace and the POI set of the target trace based on the feature vectors. The Matcher has to be robust to the fact that not every POI appears in both trace and that descriptions of different operations may be similar.

For every POI from the reference trace a distance is calculated to every POI from the target trace with the same scale. To allow comparison against a threshold the normalized Euclidean distance is used, which is a special case of the Mahalanobis distance and is defined in equation (7).

$$d(\overrightarrow{x}, \overrightarrow{y}) = \sqrt{\sum_{i=1}^{N} \frac{(x_i - y_i)^2}{\sigma_i^2}} \tag{7}$$

where σ_i is the standard deviation of the x_i over the sample set. In our case σ_i is calculated using every possible POI with the same scale. To do this the all the POIs have to be calculated before the matching starts (note that this is not shown in the algorithms), this will also speed up the algorithm. In some cases, especially those with large scales, there are not enough POIs to give a proper estimate for σ_i, but we find in practice this does not cause problems.

Algorithm 3. The main loop of the Descriptor

```
procedure DESCRIBE($S$:sumtrace, $pois$:POIlist)
    for All POI $p$ from $pois$ do
        $v = ZeroFilledVector(Des_{fc})$
        $da = Des_{as} * p.scale$                                  ▷ size of the description area
        $sc = Des_{fc}/2$                                          ▷ section count
        $ss = da/sc$                                               ▷ section size
        $o = p.pos - sc/2 * ss$                                    ▷ offset, most left sample point
        $step = max(1, p.scale/Des_{ss})$;

        for $s = 0; s < sc; s++$ do                                ▷ For each section
            for $i = 0; i < ss; i+= step$ do                       ▷ For each sample in the section
                $t = $ HAAR($S, o + s * ss + i, Des_{hw} * p.scale$)     ▷ Calc Haar
                $t* = $ GAUSSIAN($o + s * ss + i - p.pos, Des_{ga}$)     ▷ Normalize
            end for
            $v[2 * s] += t$                                        ▷ Add response for section
            $v[2 * s + 1] += abs(t)$                               ▷ Add absolute response for section
        end for
        SETFEATUREVECTOR($p, v$)
    end for
end procedure
```

Each POI from the reference trace is matched with the POI with the smallest distance in the target trace. For efficiency, if the distance is greater than the threshold Mat_{md} the match is removed. Testing shows that this removes most of the mismatches as can be seen in Fig. 4.

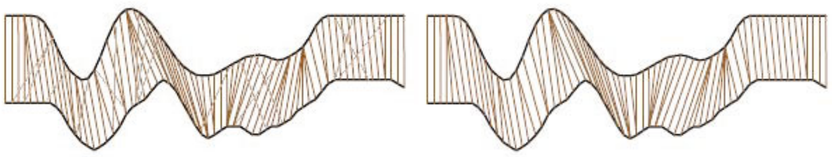


Fig. 4. Two matched POI sets, with $Mat_{md} = 1000$ (left) and $Mat_{md} = 4$(right)

In the remaining matches there can still occur cross matches. By this we mean that sample point ref_n of the reference trace is matched against sample point tar_n of the target trace while at the same time sample ref_{n+p} is matched against sample tar_{n+q} with $p \cdot q < 0$. This is not allowed because it would violate the temporal behavior of the trace.

To resolve these cross matches the confidence of a match and a penalty function are used. The confidence of a match is defined by the distance of the best match divided by the distance for the second best match. The penalty function halves the confidence of a match for every other match it crosses. The conflicting match with the lowest confidence will than be removed from the set. This is repeated until all cross matches are resolved.

Algorithm 4. The main loop of the Matcher

```
procedure MATCH($ref$:POIlist, $tar$:POIlist)
    $matches = emptylist$

    for All POI $p$ from $ref$ do                                              ▷ Match POIs
        $q1 = argmin_q(\text{DIST}(p.feats, q.feats)) \mid q \in tar)$                  ▷ Closest
        $q2 = argmin_q(\text{DIST}(p.feats, q.feats)) \mid q \in tar \wedge q \neq q1)$    ▷ Second closest
        $d1 = \text{DIST}(p.feats, q1.feats)$

        $conf = 1 - d1/\text{DIST}(p.feats, q2.feats)$
        if $d1 < Mat_{md}$ then
            ADDMATCH($matches, \{p, q1, conf\}$)
        end if
    end for

    while crossmatches exist in $matches$ do                          ▷ Remove crossmatches
        $worst = \text{argmin}_m\ (m.conf * 0.5^{\text{NUMCROSSES}(m,matches)} \mid m \in matches)$
        REMOVEMATCH($matches, worst$)
    end while
    return $matches$
end procedure
```

4.4 Warper

The Warper (Algorithm 5) takes a list of matched POIs from two traces and shrinks and stretches sections of the target trace so it will be aligned with the reference trace. In the result trace the values of the samples that are included in the match list are set to the values of the target trace. Values in between the matched points are interpolated. Various interpolation schemes have been tried such as Nearest Neighbor, Linear, Cosine, Cubic and Hermite [4]. Hermite

Algorithm 5. The main loop of the Warper

```
procedure WARP($tar$:trace, $mat$:Matchlist)
    $t = TraceOfSize(tar.size)$
    for $m = 1; m < mat.size; m++$ do
        $ddest = mat[m].ref.pos - mat[m-1].ref.pos$          ▷ delta in reference trace
        $dsrc = mat[m].tar.pos - mat[m-1].tar.pos$              ▷ delta in target trace
        $r = dsrc/ddest$                                          ▷ stretch/shrink factor

        for $i = 0; i < ddest; i++$ do
            $t[mat[m].ref.pos + i] = \text{INTERPOLATEDPOINT}(tar, mat[m].tar.pos + i * r)$
        end for
    end for
    t[t.size-1] = tar[tar.size-1]                                    ▷ Copy last sample
    return t
end procedure
```

interpolation is similar to cubic but has tension and biasing parameters. The tension controls tightens the curve at known points whereas bias twists the curve towards one of the two points. Testing shows that differences were minimal but slightly in favor of the Cubic interpolation scheme. The other schemes have not been researched further.

The start and the end of the traces are not necessarily aligned. The samples before the first matched sample are interpolated with the same parameters as the samples between the first and the second matched samples. The same is done at the end of the trace. We fill this with the values of the nearest sample.

5 Experiments and Results

In our experiments we intend to compare RAM to other well-known methods for dealing with misalignment: Static Alignment [9], SW-DPA [6] and Elastic Alignment [10]. We compare first-order success rates of CPA after performing the different alignment techniques applied to measurements taken from a software DES implementation.

The hardware used for our measurements consists of the standard side channel equipment: a smart card reader, a LeCroy $104Xi$ oscilloscope and our acquisition and analysis software. We also employed a 48 MHz analog lowpass filter. The traces were acquired from a programmable smart card, which contains a software implementation of DES, including software countermeasures that can be turned on and off. For these measurements we enabled the random delays countermeasure, which introduces random wait states during execution of the DES algorithm. The card runs at an external clock of 4 MHz.

Timing measurements for the comparison of different alignment methods are performed on a computer with an E6750 2.66 GHz processor and 2 Gb of RAM and running Windows 7. We sampled the traces at 250 megasamples per second. We chose for a much higher frequency than the clock speed to get a more accurate reading per clock cycle. The acquired traces were compressed right after acquisition by averaging samples until we effectively obtained one sample per clock. In total we acquired 8248 consecutive clock cycles per DES execution, providing us with several rounds of DES. This was performed with random inputs for a total of 500k traces.

A typical trace from the resulting set is shown in Fig. 5. Note that the y-axis shows negative mVolts. This is due to an offset introduced during acquisition and has no consequences for our research.

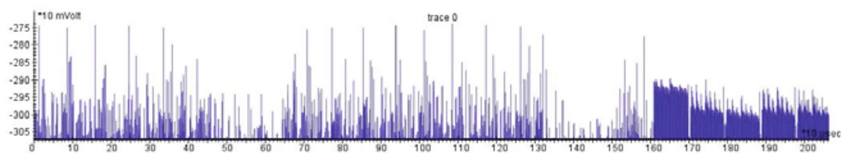


Fig. 5. A typical trace from the data set, showing the last round of the encryption

5.1 Settings

Here we elaborate on different settings used for alignment algorithms we compare our algorithm with.

Static alignment settings: For static alignment we selected a trace fragment of 600 samples at the end of the encryption. The allowed shift for this fragment was set to 250 samples.

Sliding window settings: We preprocessed the traces for a SW-DPA attack [6], by averaging the samples as described previously. The number of samples per clock cycle was set to 1 and the size of the sliding window that we used was 50, 100 and 200.

Elastic alignment settings: We also compare our proposed algorithm to Elastic Alignment. We set the radius in which FastDTW (as in [10]) will search for the optimal warppath. A radius of 70 provided a good trade-off between alignment quality and speed.

RAM align: For the new algorithm the initial values of the constants were chosen close to the values of U-SURF [2] whenever possible. The other baselines for the constants were tuned on the basis of previous experimenting. Using the correlations of the target traces with the reference trace it was decided whether increasing or decreasing the constant yields better results. Various values for the constants that we tested are given in Table 1.

Table 1. Different values for the constants that were used for tuning the algorithms

Constant	Notation	Values tested	Baseline value
Minimum Wavelet Size	Det_{mw}	3, 6, 12	6
Response Threshold	Det_{th}	2, 2.5, 3	3
Area Size	Des_{as}	12, 16, 20	20
Feature Count	Des_{fc}	16, 24, 32	32
Haar Wavelet Size	Des_{hw}	1.5, 2.5	-
Standard Deviation Gaussian	Des_{ga}	2.8, 3.3, 3.8	-
Samples Per Section	Des_{ss}	10, 20	-
Maximum Distance	Mat_{md}	2.3, 2.5, 2.7	2.5

Some additional explanations for constants and their values are given below.

Minimum Wavelet Size (Det_{mw}) This constant specifies the stopping criterion for the detector. The detector starts with large wavelets and decreases them after every iteration. If this wavelet size is reached then the detector is finished. In addition, lowering this value increases computation time, as usually there are much more possible POIs at low wavelet sizes, which possibly complicates correct matching. Lowering this value provides more precision for alignment and could increase the overall score.

Response Threshold (Det_{th}) The detector will only select samples to become a POI if the wavelet response at that point is greater than Det_{th} times the standard deviation of the trace. A higher value input less points to the matcher, but may result in too few POIs to do proper alignment. A lower value also increases the running time of the algorithm. The matching algorithm is quadratic in the number of POIs detected, which makes this very computationally demanding.

Area Size (Des_{as}) This specifies the size of the area around the POI which will be used for the generation of the feature vector for this POI. The area of this POI will be Des_{as} times the scale of the POI. Low values mean that POIs are related to each other based on the samples close to the POI (which could have patterns that exist multiple times in the trace). High values take samples further away into account, but could cause confusion that due to the fact that areas of POIs overlap too much.

Feature Count (Des_{fc}) This constant is related to the area size. It specifies how many features should be calculated in the specified area. More features means longer computation time but more accuracy for relating POIs.

Haar Wavelet Size (Des_{hw}) The descriptor uses Haar wavelets of size Des_{hw} times the scale of the POI to generate the feature vectors. Higher values means searching for patterns at lower signal frequencies. The wavelet becomes bigger and is less sensitive for higher frequencies.

Standard Deviation Gaussian (Des_{ga}) The descriptor weighs the Haar wavelet responses with a Gaussian with an standard deviation of Des_{ga} times the scale of the POI. Lowering the value of this constant means that the description is focused on samples closer to the POI.

Samples Per Section (Des_{ss}) This is the number of samples used per section. The more samples used, the more accurate the descriptor can describe that section. If significantly less samples are used it speeds up the algorithm.

Maximum Distance (Mat_{md}) This constant specifies the maximum allowed distance between two matched POIs. If a match has a distance of more than Mat_{md} it will be discarded. Lowering this value will prevent cross matching, but lowering it too much will discard usable matches.

Due to the fact that some of the constants influence each other, changing the setting on one constant may change the optimal settings for the others. More detailed graphical representation of the constants tuning is given in the appendix. After tuning the best results were obtained with the following values for the constants: $Det_{mw} = 3, Det_{th} = 2.5, Des_{as} = 16, Des_{fc} = 32, Des_{hw} = 2, Des_{ga} = 3.3, Des_{ss} = 10, Mat_{md} = 2.3$.

Result Analysis. The final results are the calculated first order success rates. After alignment, the trace set is split into subsets. For each subset a number of traces N_s is selected and a CPA [3] attack is performed on these traces. The module counts the number of successful attacks S_+ and the number of unsuccessful attacks S_- and calculates the first order success rate R_s:

$$R_s = \frac{S_+}{S_+ + S_-}$$

.

5.2 Comparison Results

The running times of the different algorithms are listed in table 2. The proposed algorithm clearly outperforms Elastic Alignment in computation time. Although Static Alignment and SW-DPA are faster they are not able to align the traces in such a way that a successful DPA attack can be performed as can be seen in Fig. 6. When comparing the success rate to the number of traces used the proposed algorithm performs similar to Elastic Alignment. However, when comparing the success rate to the time it took to align the traces, the proposed algorithm outperforms Elastic Alignment with an order of magnitude.

Table 2. Timing results for the various alignment methods. The time listed for SW-DPA is the additional time it took to perform the DPA attack.

Algorithm	Run Time	Time Per Trace
Static Alignment	12 minutes	1.44 ms
SW-DPA	18 minutes	2.16 ms
RAM	76 minutes	9.1 ms
Elastic Alignment	3115 minutes	373.8 ms

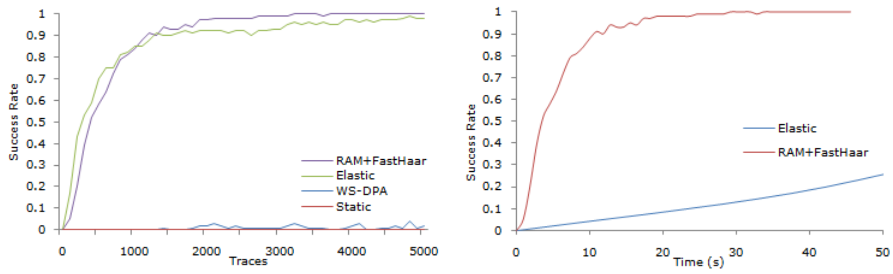


Fig. 6. Success To Trace Ratio (Left) and Success to Time Ratio (Right). Note that data points in the right graph are not measurements but translated and interpolated points from other experiments.

6 Conclusions and Future Work

Although several algorithms to align the measurements exist today they are all limited in either success-to-number-of-traces-ratio or computation time. In this work we introduce a new algorithm that obtains the best performances of previous works in terms of both the success rate and computation time. The proposed algorithm consists of four components. Each of these components can be replaced so that new approaches can easily be tested. This provides an easy to use framework for alignment algorithms.

We illustrate our results by experiments on a smartcard with a software implementation of triple DES to measure the performance of the algorithm. We used several experiments to tune the parameters of the algorithm. All of the parameters are dependent on properties of the traces to be aligned. This results in an algorithm which is easy to use. However, it is possible that further experimenting (with other implementations and platforms) could result in other values for the parameters.

We compared our proposed algorithm with Static Alignment, Sliding Window DPA and Elastic Alignment. While Static Alignment and Sliding Window DPA are not capable of properly aligning the used trace set, Elastic Alignment showed excellent performance but was relatively slow. It took almost 52 hours to process the 500 000 traces from our data set. Our proposed algorithm performed similarly in terms of success ratio compared to Elastic Alignment but took only 76 minutes to process the data set.

Acknowledgments. We would like to thank Riscure for providing an environment for fruitful discussion during the research, and for providing the side-channel analysis platform that was used for this work (Inspector). This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State and by the European Commission under contract number ICT-2007-216676 ECRYPT NoE phase II and by the K.U.Leuven-BOF (OT/06/40).

References

1. Ballard, D.: Generalizing the Hough transform to detect arbitrary shapes. Pattern Recognition 13(2), 111–122 (1981)
2. Bay, H., Ess, A., Tuytelaars, T., Van Gool, L.: Speeded-Up Robust Features (SURF). Comput. Vis. Image Underst. 110, 346–359 (2008)
3. Brier, E., Clavier, C., Olivier, F.: Correlation Power Analysis with a Leakage Model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)
4. Burden, R., Faires, J.: Numerical analysis, Paciific Grove, California, United States (2004)
5. Charvet, X., Pelletier, H.: Improving the DPA attack using Wavelet, transform. In: NIST Physical Security Testing Workshop (2005)
6. Clavier, C., Coron, J.-S., Dabbous, N.: Differential Power Analysis in the Presence of Hardware Countermeasures. In: Koç, Ç.K., Paar, C. (eds.) CHES 2000. LNCS, vol. 1965, pp. 252–263. Springer, Heidelberg (2000)
7. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
8. Lowe, D.G.: Object recognition from local scale-invariant features. In: Proceedings of the International Conference on Computer Vision, ICCV 1999, vol. 2. IEEE Computer Society, Washington, DC, USA (1999)
9. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security). Springer-Verlag New York, Inc., Secaucus (2007)
10. van Woudenberg, J.G.J., Witteman, M.F., Bakker, B.: Improving Differential Power Analysis by Elastic Alignment. In: Kiayias, A. (ed.) CT-RSA 2011. LNCS, vol. 6558, pp. 104–119. Springer, Heidelberg (2011)

Appendix

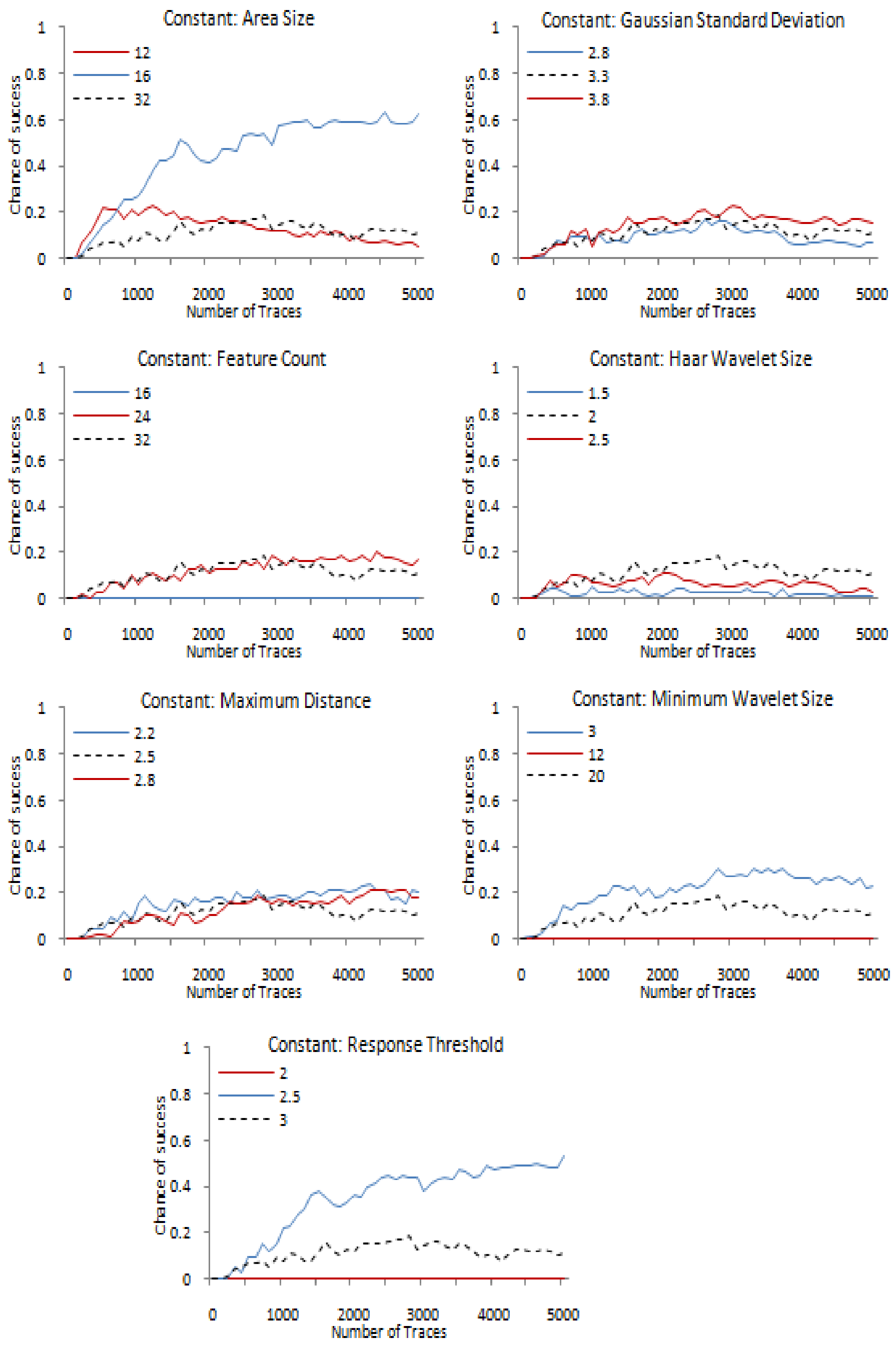


Fig. 7. The results for tuning the constants

Combined Software and Hardware Attacks on the Java Card Control Flow

Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet

Smart Secure Devices (SSD) Team – XLIM Labs, Université de Limoges
83 Rue d'Isle, 87000 Limoges, France
{guillaume.bouffard,julien.cartigny,jean-louis.lanet}@xlim.fr

Abstract. The Java Card uses two components to ensure the security of its model. On the one hand, the byte code verifier (BCV) checks, during an applet installation, if the Java Card security model is ensured. This mechanism may not be present in the card. On the other hand, the firewall dynamically checks if there is no illegal access. This paper describes two attacks to modify the Java Card control flow and to execute our own malicious byte code. In the first attack, we use a card without embedded security verifier and we show how it is simple to change the return address of a current function. In the second attack, we consider the hypothesis that the card embeds a partial implementation of a BCV. With the help of a laser beam, we are able to change the execution flow.

Keywords: Java Card, control flow, laser, Java Card Stack, attack.

1 Introduction

Java Card is a kind of smart card that implements one of the two editions, "*Classic Edition*" or "*Connected Edition*", of the Java Card 3.0 standard [8]. Such smart cards embed a virtual machine (VM) which interprets codes already romized with the operating system or downloaded after issuance[1]. In fact, Java Card is an open platform for smart cards, *i.e.* able of loading and executing new applications after issuance. Thus, different applications from different providers run in the same smart card. Thanks to type verification, byte codes delivered by the Java compiler and the converter (in charge of giving a compact representation of class files) are safe, *i.e.* the loaded application is not hostile to other applications in the Java Card. Furthermore, the Java Card firewall checks application permissions and access in the card, enforcing isolation between them.

Java Cards have shown improved robustness compared to native applications regarding many attacks. They are designed to resist to numerous attacks using both physical and logical techniques. Currently, the most powerful attacks are hardware based attacks and particularly fault attacks. A fault attack modifies

[1] Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [3]. This protocol ensures that the owner of the code has the necessary authorization to perform the action.

parts of memory content or a signal on internal bus, which can lead to deviant behavior exploitable by an attacker. A comprehensive consequence of such attacks can be found in [6]. Although fault attacks have been mainly used in the literature from a cryptanalytic point of view (see [1,4,9]), they can be applied to every code layers embedded in a device. For instance, while choosing the exact byte of a program the attacker can bypass counter-measures or logical tests. We called *mutant* such modified application.

2 State of the Art

2.1 Java Card Security

The Java Card platform is a multi-application environment where critical data of an applet must be protected against malicious access from another applet. To enforce protection between applets, classical Java technology uses the type verification, class loader and security managers to create private namespaces for applets. In a smart card, complying with the traditional enforcement process is not possible. On the one hand, the type verification is executed outside the card due to memory constraints. On the other hand, the class loader and security managers are replaced by the Java Card firewall.

Allowing code to be loaded into the card after post-issuance raises the same issues as the web applets. An applet not built by a compiler (handmade byte code) or modified after the compilation step may break the Java sandbox model. Thus, the client must check that the Java-language typing rules are preserved at the byte code level. Java is a strongly typed language where each variable and expression has a type determined at compile-time, so that if a type mismatch from the source code, an error is thrown. The Java byte code is also typed. Moreover, local and stack variables of the VM have fixed types even in the scope of a method execution but no type mismatches are detected at run time, and it is possible to make malicious applets exploiting this issue. For example, pointers are not supported by the Java programming language although they are extensively used by the Java VM (JVM) where object referenced from the source code are relative to a pointer. Thus, the absence of pointers reduces the number of programming errors. But it does not stop attempts to break security protections with unfair use of pointers.

The BCV is an essential security component in the Java sandbox model: byte code alteration contained in an ill-typed applet may induce a security flaw. The byte code verification is a complex process involving elaborate program analyses using a very costly algorithm in time consumption and memory usage. For these reasons, lot of cards do not implement this kind of component and rely on the responsibility of the organization which signs the code of the applet to ensure that they are well-typed.

The separation of different applets is enforced by the firewall which is based on the package structure of Java Card and the notion of context. When an applet is created, the Java Card Runtime Environment (JCRE) uses an unique Applet IDentifier (AID) from which it is possible to retrieve the name of the package in which the applet is defined. If two applets are an instance of classes of the same Java Card package, they are considered in the same context. There is also a super user context, called the JCRE context. Applets associated with this context can access to objects from any other context on the card.

Each object is assigned to a unique owner context which is the context of the created applet. An object method is executed in the owner object context. This context provides information allowing, or not, to access to another object. The firewall prevents a method executed in a context from accessing to any attribute or method of objects to another context.

2.2 The CAP File

The CAP (for Convert APplet) file format is based on the notion of components. It is specified by Oracle [8] as consisting of ten standard components: `Header`, `Directory`, `Import`, `Applet`, `Class`, `Method`, `Static Field`, `Export`, `Constant Pool` and `Reference Location` and one optional: `Descriptor`. Moreover, the targeted Java Card VM (JCVM) may support user `custom` components. We except the `Debug` component because it is only used on the debugging step and it is not sent to the card.

Each component has a dedicated role and is linked to each others. A modification, volunteer or not, of a component is difficult and may provide meaningless file. An invalid file is often detected during the installation step by the target JCVM.

2.3 Logical Attacks

The Hubbers and Poll's Attack Erik Hubbers *et al.* made a presentation at CARDIS 2008 about attacks on smart card. In their paper [5], they present a quick overview of the classical attacks available and gave some counter-measures. They described four methods:

1. CAP file manipulation,
2. Fault injection,
3. Shareable interfaces mechanisms abuse and
4. Transaction Mechanisms abuse

The goal of (1) is to modify the CAP file after the building step to bypass the BCV. The problem is that, like explained before, an on-card BCV is an efficient system to block this attack. Using the fault injection in (2), the authors succeed to bypass the BCV. Even if there is not particular physical protection, this attack is efficient but quiet difficult to perform and expensive.

The idea of (3) abusing shareable interfaces is really interesting and can lead to trick the VM. The main goal is to obtain a type confusion without the need to modify the CAP files. To do that, the authors create two applets which communicate using the shareable interface mechanism. To create a type confusion, each applet uses a different type of array to exchange data. During compilation or on loading, there is no way for the BCV to detect a problem. But it seems that every card tried, with an on-card BCV, refused to allow applets using shareable interface. As it is impossible for an on-card BCV to detect this kind of anomaly, Hubbers *et al.* emitted the hypothesis that any use of shareable interface on card can be forbidden with an on-board BCV.

The last option left is the transaction mechanism ([4]). The purpose of transaction is to make a group of atomic operations. Of course, it is a widely used concept, for instance in databases, but still complex to implement. By definition, the rollback mechanism should also deallocate any objects allocated during an aborted transaction and reset references to such objects to null. However, Hubbers *et al.* found some cases where the card keeps the reference to objects allocated during transaction even after a rollback.

Moreover, the authors described the easiest way to make and exploit a type confusion to gain illegal access to otherwise protected memory. A first example is to get two arrays with different types, a byte and a short array. If a byte array of 10 bytes is declared and it exists a reference to a short array, it is possible to read 10 shorts, so 20 bytes. With this method they can read the 10 bytes stored after the array. If Hubbers *et al.* increase the size of the array, they will be able to read as much memory as they want. The main problem is more *how to read memory before the array?*

The other used confusion is between an array of bytes and an object. If Hubbers *et al.* put a byte as first object attribute, it is bound to the array length. Then it is really easy to change the length of the array using the reference to the object. With this attack, the problem becomes *how to give a reference to an object for another object type?*

Barbu *et al.*'s Attack: Combined Physical & Logical Attack. At CARDIS 2010, Barbu *et al.* described a new kind of attack in their paper [2]. This attack is based on the use of a laser beam which modifies a runtime type check (the `checkcast` instruction) while running. This applet was checked by the on-card BCV, considered as valid, and installed on the card. The goal is to cause a type confusion to forge a reference of an object and its content. We consider three classes `A`, `B` and `C`. They are declared in the listing 1.1.

```
public class A {              | public class B { | public class C {
    byte b00, ..., bFF        |     short addr   |     A a;
}                             | }                | }
```

Listing 1.1. Classes used to create a type confusion

The cast mechanism is explained in the JCRE specification [8]. When casting an object to another, the JCRE dynamically verifies if both types are compatible, with a `checkcast` instruction. Moreover, an object reference depends on the card architecture. The following example can be used:

```
T1 t1;                  aload @t1
T2 t2 = (T2) t1; ⟺ checkcast T2
                        astore @t2
```

The authors want to cast an object `b` to an object `c`. If `b.addr` is modified to a specific value, and if this object is cast to a `C` instance, you may change the referenced address by `c.a`. But the `checkcast` instruction prevents from this illegal cast.

Barbu *et al.* use in his `AttackExtApp` applet (listing 1.2) an illegal cast at line 9.

```
 1 public class AttackExtApp extends Applet {
 2     B b; C c; boolean classFound;
 3     ... // Constructor, install method
 4     public void process(APDU apdu) {
 5       ...
 6       switch (buffer[ISO7816.OFFSET_INS]) {
 7         case INS_ILLEGAL_CAST:
 8           try {
 9             c = (C) ( (Object) b );
10             return; // Success, return SW 0x9000
11           } catch (ClassCastException e) {
12             /* Failure, return SW 0x6F00 */
13           }
14           ... // more later defined instructions
15 }    } }
```

Listing 1.2. `checkcast` type confusion

This cast instruction throws a `ClassCastException` exception. With specific material (oscilloscope, *etc.*), the thrown exception is visible in the consumption curves. With a time-precision attack, the authors prevent the `checkcast` from being thrown with the injection of laser based fault. When the cast is done, the references of `c.a` and `b.addr` link the same value. Thus, the `c.a` reference may be changed dynamically by `b.addr`. This trick offers a read/write access on smart card memory within the fake `A` reference. Thanks to this kind of attack, Barbu *et al.* can apply their combined attack to inject ill-formed code and modify any application on Java Card 3.0, such as EMAN1 [6].

3 EMAN2: A Stack Underflow in the Java Card

3.1 Genesis

The aim of this attack is to modify the register which contains the method return address by the address of an array which contains our malicious byte code. To succeed, the target smart card has no BCV and we know its loading keys. For this work, we used two tools developed in the Java-language. The first one, the CFM [11] (for CAP File Manipulator) provides a friendly way to parse and full-modify the CAP files. The other one is the Java library OPAL [10] used to communicate with the card. So, to perform this attack, we must:

1. find the array address which contains the malicious byte code;
2. find where is located, in the Java Card stack, the address of the return function;
3. change this address by the address of the byte code contained in our malicious array.

We will explain each step in the next subsections.

3.2 How to Obtain the Address of Our Malicious Array?

In a previous work [6], we explained how to execute auto-modifiable code in a Java Card. This malicious byte code was stored in a byte-array and called by an ill-formed applet. We also have to remember the way to obtain the array address.

```
1 public short getMyAddressByteArray (byte[] array) {
2   short foo=(byte)0x55AA;
3   array[0] = (byte)0xFF;
4   return foo;
5 }
```

Listing 1.3. Method to retrieve the address of an array

In order to retrieve the address of an array, we implemented the method `getMyAddressByteArray` described in the listing 1.3. In its unmodified version, it returns the value contained in `foo`. The instruction in line 3 uses an array given in the function parameter. As seen in listing 1.4, the JCVM needs first to push a reference to the array `tab`[2]. Finally the function returns the previously pushed value of `foo`.

If an event changed our byte code like described in the listing 1.5, our function directly returns the reference of the array given as parameter. To make this modification, we use the CFM to "`nop`" each instruction between *push the array reference* and *return the short pushed value*. These instructions are written in a bold font in the listing 1.5. Using a card without BCV, an applet containing this function provides address of each array given in its parameter. The returned address is locate in the EEPROM area.

[2] In our tested card, all references are performed in a `short` type.

```
public short
getMyAddressByteArray
                  (byte[] array) {
03 // flags: 0 max_stack : 3
21 // nargs: 2 max_locals: 1
10 AA       bspush      -86
31          sstore_2
19          aload_1
03          sconst_0
02          sconst_m1
39          sastore
1E          sload_2
78          sreturn
}
```

Listing 1.4. The Java byte code corresponding to the function 1.3

```
public short
getMyAddressByteArray
                  (byte[] array) {
03 // flags: 0 max_stack : 3
21 // nargs: 2 max_locals: 1
10 AA       bspush      -86
31          sstore_2
19          aload_1
00          nop
00          nop
00          nop
00          nop
78          sreturn
}
```

Listing 1.5. The function 1.3 with the modified return

On the targeted JCVM, the address returned by the malicious function `getMyAddressByteArray` does not refer to the array data. It is a pointer on the array header which is structured by 6 bytes that include the type and the number of contained elements. If the array is transient, the RAM array address follows the header. Else, the array data is stored after the 6-byte header.

3.3 Java Card Stack

To perform this attack, we should understand the Java Card stack. In fact, a Java Card contains two stacks, the native and the JCVM stack. The first one is used by the smart card operating system. The second one, is used by the JCVM to execute some Java applets value pushed in the Java Card stack.

To characterize the Java Card stack, We used the method `ModifyStack`, listed in 1.6. This method has three parameters: `apduBuffer`, a reference to a byte array; `apdu`, a reference to an instance of the `APDU` class; and `a`, a short value. The figure 1(b) represents the Java Card stack where each method parameter, variable and a reference to the class instance (`this`) are stored in the local variables area. Next, the information present in the frame header (in L_6 and L_7) are important data which hold the method return address. Finally, the stack contains data pushed while the method run[3].

The BCV must checks several points. In particular: it should prevent any violations of the memory management (illegal reference access), stack underflow or overflow. This means these checks are potentially not verified during runtime and thus can lead to vulnerabilities. The Java frame is a non persistent data structure implemented in different ways and the specification gives no designed direction for it.

[3] The maximum number of values to push is defined in the field `MAX_STACK` included in each Java Card method header.

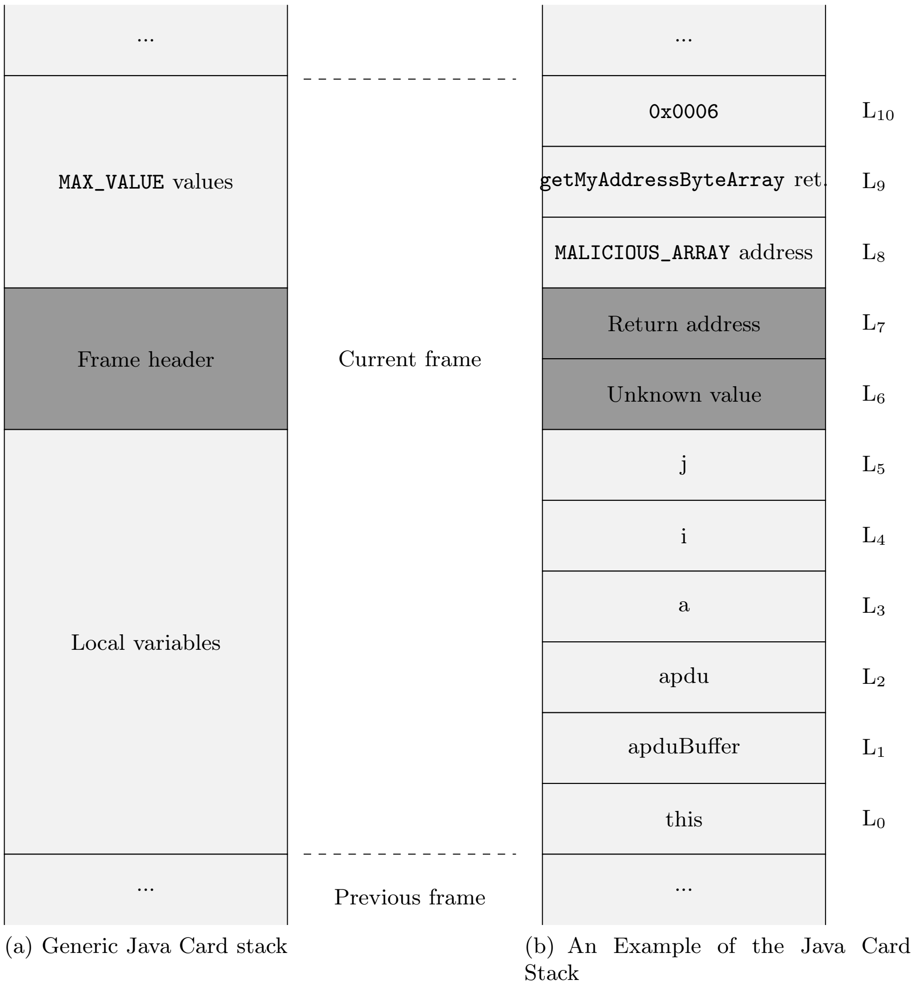


Fig. 1. Java Card stack characterization

3.4 Our Attack

Our attack aims to change the index of a local variable[4]. We propose to use two instructions: `sload` and `sstore`. As described in the JCVM specification [8], these instructions are normally used in order to load a short value from a local variable and to store a short value in a local variable. The CFM allows us to modify the CAP file in order to access the system data and the previous frame. As example, the code in the listing 1.6, line 4, stores the value returned by

[4] The specification says that the maximum number of variables that may be used in a method is 255. It includes local variables, method parameters, and in case of an instance method invocation, a reference to the object on which the instance method is being invoked.

```
1 public void ModifyStack(byte[] apduBuffer, APDU apdu, short
      a)
2 {
3   short i=(short) 0xCAFE;
4   short j=(short) (getMyAddressByteArray(MALICIOUS_ARRAY)
      +6);
5   i = j ;
6 }
```

Listing 1.6. Function to modify the Java Card stack

`getMyAdressByteArray()` and adds 6 into variable `j`. Then, it loads the value of `j`, and stores it into variable `i` (line 5).

So, if we change the operand of `sload` (`sload 5`, at the offset `0x11` of the listing 1.7) we store information from a non-authorized area into the local 5. Then, this information is sent out using an APDU. We tried this attack using a +2 offset and we retrieved the short value `0x8AFA` which was the address of the caller. Thus, we were able to read without difficulty in the stack after our local variables. Furthermore, we can write anywhere into the stack below: there is no counter-measures. The targeted smart card implements an interpreter that relies entirely on the byte code verification process.

Next, we modified the CAP file to change the return address by our malicious array address, this step was explained in the section 3.2. When this modification is performed, the exception `0x1712` is throw. So, we proved within this applet that we can redirect the control flow of such a JCVM.

```
        public void ModifyStack
                (byte[] apduBuffer, APDU apdu, short a) {
0x00:     02 // flags: 0    max_stack:  2
0x01:     42 // nargs: 4    max_locals: 2
0x02:     11 CA FE    sspush          0xCAFE
0x05:     29 04       sstore          4
0x07:     18          aload_0
0x08:     7B 00       getstatic_a     0
0x0A:     8B 01       invokevirtual   1
0x0C:     10 06       bspush          6
0x0E:     41          sadd
0x0F:     29 05       sstore          5
0x11:     16 05       sload           5
0x13:     29 04       sstore          4
0x15:     7A          return
        }
```

Listing 1.7. Malicious byte code applet of the function 1.6

3.5 Counter-Measure

As we said, no important knowledge are needed in Java Card security and the simple modifications of a CAP file, with the tool [11], may perform these attacks.

The purpose of the stack underflow is to get access to memory area normally used by the system to the previous frame. A simple counter-measure would consist in checking the number of locals and arguments provided in the header of the method. With this simple check one cannot gain access to the system area where the JPC (previous Java Program Counter) and SPC (previous Stack Pointer) are stored. In order to avoid parsing the previous frame, the implementation can use the linked frame approach like in the simple RTJ VM references. This approach implies to create a new frame and to copy the argument of the current frame into the new one, instead of the implemented method which uses the current stack as the beginning of the new frame. Desynchronizing frames will avoid simply a stack underflow attack.

4 EMAN4: Modifying the Execution Flow with a Laser Beam

4.1 Description of Our Attack

In the section 3, we supposed that there is no BCV. This hypothesis allowed us to modify the CAP file before loading it on the card. For the following, the targeted card has an improved security system based on a partial implementation of a BCV. This component statically checks the byte code during the loading step and dynamic byte code checks are done during the runtime.

To perform this attack, we provide an external modification, such as the Barbu *et al.*'s attack, with a laser beam to change the control flow to execute our own malicious byte code. Furthermore, we have the smart card loading keys.

In order to modify the execution flow, we will use the `for` loop properties. Next, after the understanding of how this kind of loop works, we modify it to change the control flow.

4.2 How Re-loop a for Loop

The `for` loop is probably the most widely used loop in the imperative programming languages. A classic `for` loop, such as in the listing 1.8, may be split in three parts. The first one is the declaration of the loop with the preamble (the initialization of the loop), followed by the stop condition and a function executed at each iteration. Next, the loop body contains the executed instructions for each iteration. Finally, a jump-like instruction re-loop to the next iteration if the stop condition is not satisfied.

According to the amount of instructions contained in the loop body, the re-loop instruction has relative offset on 1 or 2-byte (± 127 or ± 255 bytes). In the Java Card byte code, the re-loop instruction may be a `goto` or `goto_w`. For our attack, we are focused on the `goto_w` statement at the offset `0xEB` (listing 1.9).

```
for (short i=0 ; i<n ; ++i){
 foo = (byte) 0xBA;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 // Few instructions have
 // been hidden for a
 // better meaning.
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
}
```

Listing 1.8. A `for` loop

```
0x00: sconst_0
0x01: sstore_1
0x02: sload_1
0x03: sconst_1
0x04: if_scmpge_w        00 7C
0x07: aload_0
0x08: bspush             BA
0x0A: putfield_b         0
0x0C: aload_0
0x0D: getfield_b_this    0
0x0F: putfield_b         1
// Few instructions have
// been hidden for a
// better meaning.
0xE3: aload_0
0xE4: getfield_b_this    1
0xE6: putfield_b         0
0xE8: sinc               1 1
0xEB: goto_w             FF17
```

Listing 1.9. Associated byte codes of the loop 1.8

4.3 Our Attack

To begin, we install into a Java Card an applet which contains the loop `for` described in the listing 1.8. The function which contains this loop is compliant with each security rule of Java Card and the embedded smart card BCV allows its loading.

An external modification based on a laser beam against the `goto_w` instruction, at the offset `0xEB` in the listing 1.9, may change the control flow of the applet. We would like to redirect this flow in the array `MALICIOUS_ARRAY` to execute our malicious byte code. Thus, changing the `goto_w` parameter `0xFF17` to `0x0017` involves a relative jump to the 17^{th} byte after this instruction. To success attack, our array must locate after the modified function in the EEPROM area.

Smart Card Memory Management. The main difficulty regarding this attack is the memory management. Indeed, the static array `MALICIOUS_ARRAY` must physically be put after our malicious function. For that, we analyzed how our targeted smart card stores its data. In order to understand the algorithm used by the card to organize its memory, we did the following method:

1. first, few chosen applets are installed on the card within a careful dump of the EEPROM memory between each install,
2. next, the card is stressed by installing and deleting different applets size. A dump is done at each step.

For each analyzed smart card, we obtained the same algorithm used to manage the memory. These Java Cards have a *first fit* algorithm where the applet data are stored after its byte code. If the smart card managed few applets without causing fragmentation, it is likely that the applet data is stored before the corresponding applet byte code.

In our case, there were no installed applet before we installed our. The dump obtained is listed in 1.10.

```
0x0A7F0:  18AE 0188 0018 AE00 8801 18AE 0188 0018
0x0A800:  AE00 8801 18AE 0188 0018 ae00 8801 18ae
0x0A810:  0188 0059 0101 A8FF 177A 008A 43C0 6C88
0x0A820:  abcd ef00 0000 0000 0000 0000 0000 0000
0x0A830:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A840:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A850:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A860:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A870:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A880:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A890:  0000 0000 0000 0000 1117 1200 0000 8D6F
0x0A8A0:  C000 0000 0000 00FE DCBA
```

Listing 1.10. Memory organization of our installed applet

As may be seen in the dump 1.10, the function to fault precedes the array `MALICIOUS_ARRAY` in light-gray. This dump is a linked byte code contrary to the byte code listed in 1.9.

The Goto Redirection. Before injecting our fault, the function returns `0x9000` (status without error).

After precisely targeted the high-byte parameter of the `goto_w` instruction located at `0xA817` in the listing 1.10, a laser beam attack swaps `0xFF17` to `0x0017`. This fault allows to redirect the execution flow. Indeed, the `goto_w` jumps forward to go into the array `MALICIOUS_ARRAY`. A landing area of `nop` catches up the instruction pointer which will execute our malicious code, here an exception throws the value `0x1712`. This result proves that we succeeded to change the control flow of our applet.

Moreover, even if the memory is encrypted, this kind of attack has fifty percent to change the `goto_w` instruction statement to redirect towards the front.

4.4 Counter-Measures

Creating a mutant application uses the same way than changing an applet after its loading. To protect the JCVM against this attack, voluntary or not, we developed some counter-measures described in [7]. We are going to present a brief resume of these counter-measures.

The XOR Detection Mechanism. This protection is based on basic blocks. It allows code integrity and application control flow checking. A basic block is a sequence of instructions with a single entry point and a single exit point[5]. For each basic bloc, a checksum is computed by using the XOR operation on all the bytes composing a basic block. Then this table is stored in the CAP file as a Java Card custom component. The interpreter has to be modified to exploit and verify the checksum information. During runtime, the interpreter computes again the checksum and compares it with the stored values.

The Field of Bit Detection Mechanism. This counter-measure checks the nature of the element stored in the byte array of the CAP file. A tag (bit) is associated to each byte of the bytecode. The tag has the value 0 if the bytecode is an opcode, and it has the value 1 if the byte code is a value (a parameter of an *opcode*). During an attack, the following situations can appear:

1. An increase of operands number for the instruction, it is the case when `add` (no operand) is replaced by `icmpeq` (one operand).
2. A decrease of operands number for the instruction, it is the case when `aload` (one operand) is replaced by `athrow` (no operand).
3. No change on operand number: it is the case when an `iload` (one operand) is replaced by a `return` (one operand).

This method can detect when the changing `1` and `2` happen. During the compilation, a field of bit is generated representing the type of each element contained in the method byte array. It is stored also as a Java Card custom component in the CAP file. The interpreter checks before executing an *opcode* that its byte was scheduled to be executed or not.

The Path Check Mechanism. This method computes the control flow graph of the method by extracting the basic blocks from the code. The list of paths from the beginning vertex is computed for each vertex of the control flow graph. This computed paths are encoded using the following convention:

1. Each path begins with the tags `0` and `1` to avoid a physical attack that changes it to `0x00` or to `0xFF`.
2. If the instruction that ends the current block is an unconditional or conditional branch instruction when jumping to the target of this instruction, then the tag `0` is used.
3. If the execution continues to the instruction that immediately follows the final instruction of the current block then the tag `1` is used.

If the final instruction of the current basic block is a switch instruction, the path is made by any number of bits that are necessary to encode all the targets. When interpreting the byte code, the VM computes the path followed by the program using the same convention; for example, when jumping to the target of a branch instruction it saves the tag `0`. Then prior to the execution of a basic block, the

[5] The execution of a basic block starts only at an entry point, and leaves a basic block only at an exit point.

VM checks that the followed path is an authorized path, *i.e.* a path that belongs to the list of path computed for this basic block. In the case of a loop (backward jump) the interpreter checks the path for the loop, the number of references and the number of values on the operand stack before and after the loop, to be sure that for each round the path remains the same.

5 Conclusion

In this paper we described two ways to change the execution flow of an application after loading it into a Java Card. The first method, EMAN2, provides a way to change the return address of the current method contained in its frame stack. This attack is possible because there is no check during the stack operations. The second method, EMAN4, uses a laser beam to modify a well-formed applet loaded and installed on the card to become mutant, even with the on-board BCV.

These two attacks allow to execute malicious code in the JCVM without being detected by the firewall component. In the case of EMAN2, we proposed two counter-measures. *A contratrio*, EMAN4 needs a good knowledge of the targeted JCVM and to find the faulted area with the laser beam.

References

1. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.: Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 260–275. Springer, Heidelberg (2003)
2. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg (2010)
3. Global Platform: Card Specification v2.2 (2006)
4. Hemme, L.: A Differential Fault Attack Against Early Rounds of (Triple-) DES. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 254–267. Springer, Heidelberg (2004)
5. Hubbers, E., Poll, E.: Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours. Dept. of Computer Science NIII-R0438, Radboud University Nijmegen (2004)
6. Iguchi-Cartigny, J., Lanet, J.: Developing a Trojan applet in a Smart Card. Journal in Computer Virology (2010)
7. Lanet, J.L., Bouffard, G., Machemie, J.B., Poichotte, J.Y., Wary, J.P.: Evaluation of the Ability to Transform SIM Application into Hostile Applications. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 1–17. Springer, Heidelberg (2011)
8. Oracle: Java Card Platform Specification
9. Piret, G., Quisquater, J.-J.: A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 77–88. Springer, Heidelberg (2003)
10. Smart Secure Devices (SSD) Team – XLIM, Université de Limoges: OPAL: An Open Platform Access Library, http://secinfo.msi.unilim.fr/
11. Smart Secure Devices (SSD) Team – XLIM, Université de Limoges: The CAP file manipulator, http://secinfo.msi.unilim.fr/

Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures

Guillaume Barbu[1,2], Guillaume Duc[2], and Philippe Hoogvorst[2]

[1] Oberthur Technologies, Innovation Group,
Parc Scientifique Unitec 1 - Porte 2,
4 allée du Doyen George Brus, 33600 Pessac, France
[2] Institut Télécom / Télécom ParisTech, CNRS LTCI,
Département COMELEC,
46 rue Barrault, 75634 Paris Cedex 13, France

Abstract. Until 2009, Java Cards have been mainly threatened by Logical Attacks based on ill-formed applications. The publication of the Java Card 3.0 Connected Edition specifications and their mandatory on-card byte code verification may have then lead to the end of software-based attacks against such platforms. However, the introduction in the Java Card field of Fault Attacks, well-known from the cryptologist community, has proven this conclusion wrong. Actually, the idea of combining Fault Attacks and Logical Attacks to tamper with Java Cards appears as an even more dangerous threat. Although the operand stack is a fundamental element of all Java Card Virtual Machines, the potential consequences of a physical perturbation of this element has never been studied so far. In this article, we explore this path by presenting both Fault Attacks and Combined Attacks taking advantage of an alteration of the operand stack. In addition, we provide experimental results proving the practical feasibility of these attacks and illustrating their efficiency. Finally, we describe different approaches to protect the operand stack's integrity and compare their cost with a particular interest on the time factor.

Keywords: Java Card, Fault Attack, Logical Attacks, Combined Attack, Countermeasures.

1 Introduction

Java Card systems are generally considered as intrinsically safer than native ones due to the security brought by the Java Card Runtime Environment (JCRE). Indeed the strongly-typed Java language and the abstraction layer provided by the Java Card Virtual Machine (JCVM) thwarts many Logical Attacks, such as stack overflow for instance. Therefore numerous attacks against Java Cards consist in corrupting the binary representation of Java Card applications in order to bypass the inherent security of the platform[1–3]. Nevertheless, the recent Java Card 3.0 Connected Edition has rendered such Logical Attacks (LA) unpracticable by making on-card bytecode verification mandatory. That is to say, it should not be possible to load an application that is not conform to the Java Card specifications [4–7].

However, as every embedded system, Java Cards are sensitive to attacks based on physical phenomena, amongst which fault-injection-based attacks. The principle of a fault injection on a smartcard is to modify the physical environment of the card in order to provoke an abnormal behavior of the component. It can target either the processor, the data/address bus or even the memory cells [8]. Since their publication in 1996, Fault Attacks (FA) have been mainly tackled in the literature with regards to embedded cryptographic implementations [9–11]. However these attacks can potentially target any function of an embedded system [12].

Two years ago, the idea of combining FA with LA has emerged [13]. Such attacks, called *Combined Attacks* (CA), use a fault injection to allow a malicious application to bypass the security mechanisms of the system. CA have turned out to be very efficient against improperly secured platform [14, 15]. This highlights the need to neglect none of the components of an embedded system when dealing with fault detection, and with security in general.

In Java-based systems, the *operand stack* appears as a central element. However its behaviour when targeted by fault injection has never been studied in the literature. In this paper we investigate this path and describe both FA and CA against Java Cards by disturbing the operand stack. We present practical results and detail two case-studies leading to the corruption of an application execution flow and an unduly granted authentication. These case studies prove the necessity of carefully ensuring the integrity of the operand stack. To reach this goal, we present and compare the efficiency of three different countermeasures.

The rest of this paper is organized as follow. In Section 2, we relate the uttermost importance of the operand stack in a Java Card environment. We also introduce the notions relative to FA and CA and present the fault model we consider in this work. Section 3 describes several Fault Attacks targeting the operand stack and leading to abuse Java Card applications. Section 4 presents how an attacker can threaten the platform and other applications with Combined Attacks focused on the operand stack. Finally, Section 5 describes different countermeasures against such attacks and compares their respective costs.

2 Basics of Operand Stack, Fault and Combined Attacks

In this section we give an overview of the operand stack in a Java-based environment. Then we detail the principles of FA and CA. Finally, we define the fault model we have chosen in the context of our work and discuss this choice.

2.1 The Operand Stack, a Central Element of the JCVM

The JCVM, and more generally, Java Virtual Machines (JVMs) are known as stack-based machines, in opposition to register-based machines. Actually, several stacks are described in the JVM specification [16]. We focus our interest on one kind of these: operand stacks.

A Java *frame* is created on each Java method *invoke* to store temporary VM-specific data. The operand stack is the part of this frame in charge of holding the operands and results of the VM instruction. Most of these instructions consist in popping a certain number of operands, executing a specific process and pushing a returned value. For instance, the execution of an `iadd` (adding two integer values: *value1* and *value2*) is specified as follows:

QUOTE. *"Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is value1 + value2. The result is pushed onto the operand stack."* [16]

The integrity of the values passing through the operand stack appears then crucial. In this paper, we focus our attention on this central element of the JCRE and study its robustness with regards to fault injections.

2.2 Fault and Combined Attacks

In this section we intend to introduce the notions relative to FA and CA.

FA and the Notion of Fault Model. Embedded systems are subject to the laws of physics. The impact of physical phenomena on such systems has been widely studied by the scientific community, with a particular interest on secure systems. This interest has led to the conclusion that without particular protections, sensitive information can be retrieved from the so-called *side channel leakages* such as execution timing, power consumption or electromagnetic radiation. But another conclusion that has been drawn is that by modifying the physical environment of the system, one can alter its behaviour. Such physical perturbation can be caused by various tools such as a laser beam or a glitch generator. This is the basis of the *perturbation* or *fault-injection attacks.*

In order to evaluate the possible consequences of a fault injection, it is then necessary to provide a model of the possible errors induced by the perturbation. Different fault models are commonly considered in the literature which mainly depends on:

- the impact of the fault:
  - whether it corrupts a bit, a byte, a data-word.
  - whether the value is:
    - set to a random value.
    - *stuck-at* all-0 or all-1.
- the precision of the fault.

Combined Attacks. Until 2009, the majority of the literature dealing with Java Card security remained focused on the effects of Logical Attacks (LA) [1, 2, 17, 18]. These attacks are generally based on the corruption of the binary representation of a Java Card application (*.cap* or *.class* file) into a so-called ill-formed application before it is loaded on-card. Such modifications aim at circumventing certain controls enforced by the JCVM. But in most cases, they also make the application illegal with regards to the Java Card specifications. Therefore the modified application should not be able to pass static analysis tools such as the Java bytecode verifier. The bytecode verification being a costly process, it is generally executed off-card on Java Card 2.2.2 and earlier, as a part of the application development tool chain. The usual philosophy of LA is then to skip this step and to directly load unverified applications on platforms that allow it.

The recently released Java Card 3.0 Connected Edition specifications, has made mandatory the on-card execution of the bytecode verification. Therefore loading ill-formed application is not possible anymore. This statement has given a push to the introduction of the combination of LA with FA into the Java Card field and practical applications have been published over the last two years. In these works FA are used to bypass certain security mechanisms in order to allow a LA. The so-called Combined Attacks allow then to take the benefits of both FA and LA. Indeed, they are more realistic than LA since they do not rely on an unverified application loading and potentially more powerful than FA since the malicious application can make permanent changes and act like a trojan inside the card.

2.3 The Selected Fault Model

In the scope of this work, we only consider FA targeting a JCVM. This has led us to define a fault model allowing the attacker to modify the value pushed onto the operand stack into a predetermined value or even to a chosen value, with some limitations. Indeed we consider two different fault models: the common *stuck-at* fault model and a model taking into account the value previously pushed onto the stack. This model is detailed below.

In the constrained context of single-threaded Java Cards, optimization may lead to use a single global operand stack. However, according to the specifications, an operand stack is allocated within a Java frame, on a method *invoke*. In both cases, this allocation is most likely done in RAM. Therefore, pushing an operand on the stack consists in writing this operand at a given address that only depends on the number of elements already on the stack.

In our fault model, the fault injection targets the execution of the JCVM. More precisely, we assume that the perturbation allows to prevent (at least partially) the updating of the operand stack during a push operation. The resulting erroneous value would then be either all-0, all-1 or a value resulting from an incomplete writing. As a consequence, and assuming the attacker knows the values previously pushed onto the operand stack, we can conclude that she is able to predetermine the erroneous value. Furthermore, assuming she can run and attack her own application on the platform, she can choose the value previously pushed onto the stack and therefore control the resulting erroneous value. The experimental validation of this fault model is shown in Appendix A.

The following sections details possible exploitation of fault injections following this fault model through both FA and CA.

3 Fault Attacks on the Operand Stack

In this section we explore some potential consequences of a successful fault injection on an integral value pushed onto the operand stack. In the rest of this section, we assume that the attacker can only execute applications already loaded on-card. We start with a brief description of an attack on the instruction byte of the APDU (Application Protocol Data Unit) buffer. Then we raise the issue of boolean values in a Java environment with regards to fault injection and put into practice FA on a conditional branching instruction of the VM.

3.1 Taking Advantage of Erroneous Integral Values

As any other smartcard, a Java Card follows the ISO 7816 specifications [19]. Particulary, Java Card applets receive their command through APDUs, according to the specified format (Fig. 1).

CLA	INS	P1	P2	LC	DATA	LE

Fig. 1. Format of an APDU command

The Java Card API provides a class representing APDUs and a virtual method to access the data sent within this APDU through a byte array. This byte array defines then the behaviour of the applet. For instance to select a specific instruction in the process method, an applet execute typically the following lines:

```
byte ins = apduBuf[ISO7816.OFFSET_INS];
switch (ins) {        // push ins on the stack and execute the
                      // appropriate switch instruction.

  case INS_A: processInstructionA(apdu); break;
  case INS_B: processInstructionB(apdu); break;
  ...
  default: ISOException.throwIt(ISO7816.SW_INS_UNKNOWN);
}
```

The value of `ins` is pushed onto the operand stack before executing the `switch` instruction. Therefore, a successful fault injection during the pushing of the value is likely to totally change the behaviour of the applet. However, the consequences are then dependent on the applet itself, but considering an e-wallet applet, turning a payment into a credit operation is definitely interesting from the attacker's point of view. The chances of success are nevertheless quite low regarding our fault model since the previous value on the operand stack (`apduBuf`'s reference in this case) is not known and there is no reason why it should be in relation with the different instructions.

3.2 The Case of Boolean Values

In this section, we discuss the particular cases of the boolean type and of conditional branching instructions. Then we describe FA on such instructions and give experimental results proving the efficiency of our fault model.

Booleans and Conditional Branching in Java Card. Amongst the basic types of the Java language, we find different types of integral values differing by their size or sign (byte, char, short, ...). But we also find a specific boolean type, which supports only two values: `true` and `false`. Indeed, the Java language forbid the use of any other type than boolean in `if` statement, unlike C language for instance.

Nevertheless, there is no such thing as a boolean type at the bytecode level and the Java compiler produces only bytecodes manipulating values of type `int` when processing operations on boolean variables. Finally, and most importantly with regards to the remainder of this section, the conditional branching instructions produced by the compilation of a simple if statement: `ifeq` and `ifne`, only compare the top of stack value (*i.e.* the previously pushed operand) with 0 and branch or not depending on the result of this comparison (branch if the comparison succeeds in the case of an `ifeq`, branch if the comparison fails in the case of an `ifne`). That is to say, the specification imposes that any other value than 0 will be interpreted as `true` by the JCVM.

One may note that this statement is true for any Java-based system.

FA against Conditional Branching Instructions. Several choices are offered to an attacker in order to corrupt a conditional branching instruction on a Java Card. In this section we give the details of a FA against an `ifeq` instruction evaluating a positive (true) condition by setting the previously pushed operand to 0. We consider the following code (application Java source code on the left and the corresponding bytecode on the right):

```
1.  boolean b = dummyTrue();          |       1. aload_0
                                      |       2. invokevirtual #96
                                      |       5. istore 6
2.  if (b) {                          |       7. iload 6
                                      |       9. ifeq 12
3.    Util.setShort(buffer,           |      12. aload_2
        (short)0,(short)0x1111);      |      13. iconst_0
                                      |      14. sipush 0x1111
                                      |      17. invokestatic #84
                                      |      20. pop
4.  }                                 |
5.  else {                            |
6.    Util.setShort(buffer,           |      21. aload_2
        (short)0,(short)0x2222);      |      22. iconst_0
                                      |      23. sipush 0x2222
                                      |      26. invokestatic #84
                                      |      29. pop
7.  }                                 |      ...
```

```
8.  Util.setShort(buffer,            |
         (short)2, proof);           |
```

The `dummyTrue()` method initializes the instance field `proof` (used at line 8) proving the method has been executed and return a boolean value (`true`).

The target of our attack is this value pushing, before the `dummyTrue()` returns. The goal of the attack is then to force this value to 0. In case of success, the `ifeq` instruction will result in a jump at line 21 in the bytecode sequence and the returned value will be 0x2222 instead of 0x1111.

Experimental Results. We put this attack into practice on a recent smartcard embedding a Java Card 2.2.2 VM. The fault injection is achieved with a laser beam applied on the rear-side of the component.

After empirically searching the fault injection parameters (timing, impact location, intensity) that "maximize" the number of successful FA, we reach a success rate of 78.25%, out of 10,000 disturbed executions of the application. We then adapt the test application to attack an `ifne` instruction by changing line 2 of the Java source code into `if (!b)`. Once the fault injection parameters adjusted, we reach this time a success rate of 70.92%, also out of 10,000 disturbed executions.

The results of similar attacks on a `false` condition evaluation are expected to be at least as good as the results obtained above. Indeed, since any value other than 0 is interpreted as `true`, any alteration of the pushed operand would lead to a successful attack.

These FA definitely raise the security issue caused by the specifications of the `ifeq` and `ifne` instructions, and to a certain extent by the lack of a real boolean type at the Java bytecode level.

4 Combined Attack through Faulty Object References

In this section we consider the combination of a fault injection in the operand stack and a malicious application. This implicitly assumes that the attacker has the opportunity to load and execute her own application on the platform. This privilege is far from obvious on released products. However such attacks must be considered in the context of platforms allowing post-issuance application loading like Java Cards. In the following, we describe two CA taking advantage of a faulty object reference on the operand stack in slightly different ways: type confusion and instance confusion.

4.1 Yet Another Way to Type Confusion

Type safety is a fundamental element of Java-based systems in general and of Java Cards in particular. Consequently this property has been largely studied and is used in many of the published attacks against Java Cards.

We do not provide another particular type-confusion-based attack in this section, but we describe how a fault injection in the operand stack can lead to break the type safety property with a good probability.

As previously stated, we consider that the attacker can load her own application on-card. Nevertheless, we assume that the application has to pass a bytecode verifier to be loaded. This bytecode verifier can be either on-card, as specified in the latest Java Card specifications or off-card. Provided the bytecode verifier is sound, the malicious application has then to be well-formed. As a consequence, the well-known *.cap* file or *.class* file manipulation to cause a type confusion is not an option.

Our strategy to break type safety is basically the same as the one proposed in [20]. That is to say, the attacker creates in her application several instances of a given class C and counts on an error to modify the Java reference of a given instance of another class $C*$ into that of one of the several instances of C. The main difference with the work presented in [20] is that our fault model allows us to predetermine the error. Therefore the success rate of the attack should not depend on the number of instances of class C that have been created although a sufficient number of instances of class C can be necessary in practice.

4.2 Instance Confusion: The Case Study of Security Role Impersonation

In this section, we introduce the concept of instance confusion and present the case study of an attack using this concept.

Instance Confusion. By analogy to the concept of type confusion, where an instance of a class C is used as if it were an instance of another class $C*$, we introduce the concept of *instance confusion*. An instance confusion consists in using an instance i of a given class (or of a class implementing a given interface) as if it were another instance $i*$ of the same class (or of a class implementing the same interface).

Obviously, instance confusions within the bounds of the attacker's application may not represent a threat. Furthermore, to take advantage of an instance confusion outside the bounds of her application, the attacker should have to circumvent the Java Card application firewall. In the remainder of this section, we show that an appropriate instance confusion can allow the attacker to unduely gain privileges in another application on a Java Card 3.0 through the use of an authentication service.

JC3.0 User Authentication. The Java Card 3.0 specifications provide an authentication facility through a dedicated set of service interfaces. These interfaces are organized as illustrated in Fig. 2.

To allow user authentication, these shared services are mapped to specific Unified Resource Identifiers (URI) as any other Shareable Interface Object (SIO). These SIOs are first registered into the service registry by the application providing the service through the `register` method of the `ServiceRegistry` class of the Java Card API. They can then be retrieved in the service registry using their URI by calling the `lookup` method of the same class.

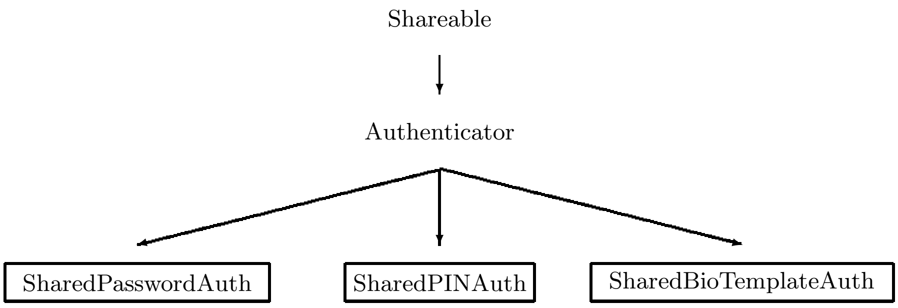


Fig. 2. Java Card 3 authenticator classes and interfaces hierarchy

Each of these authentication service interfaces expose methods allowing to:

– authenticate a user with provided credentials (`check`),
– check wether or not a user is authenticated (`isValidated`),
– reset the authentication status of an authenticated user (`reset`).

These methods are typically called either by the application, or by the web container to restrict access to specific services or content, as detailed in the JCRE specifications (§6.4.4 and 6.4.5 of [4]). The important point to notice is that these methods are exposed in shareable interfaces. That is to say they are accessible across the application firewall. They are then likely to be abused by an attacker through an instance confusion.

Setting Up and Exploiting Instance Confusion. Let us assume the attacked application uses the service offered by the `SharedPasswordAuth` interface. The first step for the attacker is then to create and load an application with several instances of a class implementing this interface. This class would typically have `check` and `isValidated` methods always returning `true` and `reset` method doing nothing, to bypass access control.

When the targeted application is about to get the authenticator instance (*i.e.* when the `lookup` method pushes its Java reference onto the operand stack), the attacker can then try to corrupt it. If she manage to provoke an instance confusion between the legitimate authenticator and one of her own, any call to the `check` or `isValidated` method would return `true` and the targeted client application would have all the reasons to consider her as an authenticated user.

The attacker may then access critical services within the attacked application. It is important to notice that even if redundant checks are performed to verify the authentication, one successful fault on the authenticator's reference is sufficient. This attack appears then more powerful than the FA on conditional branching of Sect. 3.2, since in this case each additional check would require an additional perturbation of the component.

Experimental Results. Our experiments are done on a Java Card 2.2.2. Therefore, we cannot use the API's authenticator interfaces. However, we experiment our attack on an applet holding one instance a of a class A implementing an interface I and 256 instances of a class B, also implementing I. We intend then to prove that an attacker should be able to provoke an instance confusion on specific objects. To match the previously described attack, the application obtains object a through a virtual method `getA()` and stores it in a local variable of type I, the interface implemented by A and B. The attack consists then in injecting a fault while pushing a onto the stack and check if the resulting operand is an instance of B. The test application is then the following:

```
byte[] buffer = apdu.getBuffer();
buffer[0] = 0x7F;
I a = getA(); // attacked method

if (a instanceof Object) {
   if (a instanceof B)
      buffer[0] = 0x01;  // SUCCESS
   else if (a instanceof A)
      buffer[0] = 0x02;
}
```

Out of 10,000 attacked executions of our application, we obtain the following results:[1]

– 8.74% of success: the operand popped from the stack is an instance of B.
– 25.42% of attack failure : the operand popped from the stack is an instance of A, i.e. a itself, the fault injection had no effect.
– 65.86% of unknown error : the execution of the application did not complete, *i.e.* an exception was thrown or the fault injection caused a card failure.

With regards to a theoretic security of about 2^{40} if we consider an 8-character password, an attack on a password-based authenticator with a success rate of about 10% is quite outstanding.

This section has shown that a CA taking advantage of an erroneous value on the operand stack can be even more dangerous than FA. Finally, the two previous sections have proven the need to ensure the integrity of the operand stack.

5 Countermeasures

Within this section we present different approaches to design a software countermeasure against the attacks previously described. The aim of these countermeasures is then restrained to protecting the system against a faulty value on the operand stack. Also, we focus here on dynamic checks in the context of a defensive VM and do not consider static defenses such as detecting potential dangerous mutation within an application [21, 22].

[1] As expected with regards to our fault model, increasing the number of instances of class B up to 1024 did not really enhance the success rate of the attack (almost 10%).

5.1 When to Check for Faults?

A foundation of countermeasure designing lies in the definition of the assets to protect. A good comprehension of the threats is necessary to achieve this work. This section aims at analysing the attacks previously described in order to determine the operations that are sensitive and thus worth protecting. As the attacks target the JCRE, we will consider operations at the Java bytecode level.

In the context of Java Cards, we want to prevent:

- Data from being unduely sent out of the card,
- Applications from ill-behaving.

Indeed, these identified assets summarize the assets defined in the Java Card Protection Profile (§3.2 of [23]).

We can then restrict fault detection to the bytecode instructions related to:

- Field manipulation (get/putfield, get/putstatic).
- Control-flow breaks (invokes, returns, conditions, exceptions).

Other instructions are arithmetic or logic operations, or operate on local variables. Apart from operation on static class fields, those are the operations protected by the Java Card application firewall.

5.2 Software Fault Detection

In this section we detail different approaches to detect faults within the scope of the JCVM.

The Basic Approach: Redundant Checks. The most straightforward implementation of a fault detection mechanism on the stack would be to check the coherency between the value pushed onto the stack and the top of stack value after the push operation. Likewise, with regards to the pop operation, we will check the coherency between the value that has been popped and the former top of stack value. That is to say:

```
push(expected);
if (get_tos() != expected)
   handle_fault();
```

for the push operation.

```
expected = pop();
if (get_prev_tos() != expected)
   handle_fault();
```

for the pop operation.

We implemented this countermeasure on a Java Card Virtual Machine. The additional costs on various bytecode instructions are presented in Table 1 of Sect. 5.3.

1st Refined Approach: Propagating Errors to Ensure Fault Detection. Our approach to reduce the cost of redundant check is to propagate a potential error to another component of the JCVM. This is then only valid if the standard JCVM behaviour is to check this other component.

The Java Card application firewall aims at ensuring a strict isolation between the different applications and the JCRE. A typical implementation of this mechanism we found on numerous cards and simulation tools is to assign a context identifier to each application. This identifier is also assigned to each object instance created within the scope of an application. The context isolation is then enforced by comparing an object context identifier and the current application identifier, according to the JCRE specification [4]. We choose to propagate the operand stack errors to this value. The implementation of the countermeasure is then:

```
push(expected);
fw_context_id |=
   (get_tos() ^ expected);
```

for the push operation.

```
expected = pop();
fw_context_id |=
   (get_prev_tos() ^ expected);
```

for the pop operation.

Consequently if an error occurs on the pushed value, the current context of ownership is modified. Therefore an attacker is no longer able to retrieve data from the attacked application since she would have to either call virtual or interface methods to send data out of the card or eventually use instance class fields. In both ways, she would have to pass through the application firewall and the firewall will not allow it. Similarly, if the fault aims at corrupting a conditional branch, the subsequent execution will be interrupted as soon as a firewall check occurs. An additional check is only necessary on access to static fields that are not protected by the application firewall.

The fact that few additional checks need to be inserted (only for access to static fields) is clearly an advantage regarding the computational cost of this method. The major drawback of this method is that corrupting the `fw_context_id` value, it is possible (although we consider the chances as low) that we fix it to the value identifying another application installed on the card. In such a case, our countermeasure would eventually open a breach in the application firewall. Table 1 of Sect. 5.3 presents the experimental cost of this refined countermeasure.

2^{nd} Refined Approach: Introduction of a Stack Invariant. A second approach to detect faults in the operand stack consist in adding in the Java frame structure a variable σ that allows to exhibit an invariant property.

Definition 1. *σ is the sum, considering the XOR operation, of all the values pushed on and popped from the operand stack.*

We can then exhibit the following invariant property:

Property 1. Let $\mathcal{S}_T$ be the set of all the values contained by the operand stack at a given time T. Then at any given time T,

$$\sigma \oplus \Sigma \mathcal{S}_T = 0$$

Proving this property and its invariance is straightforward. Indeed since σ is by definition the sum of the values pushed onto and popped of the stack, all the values that have been popped have been eliminated from the XOR sum. Therefore, only the values that are still on the stack at a given time T are components of σ.

The implementation of the countermeasure is then:

```
push(expected);
sigma ^= expected;
```

for the push operation.

```
expected = pop();
sigma ^= expected;
```

for the pop operation.

As previously stated, we can check the invariant property on firewall checking and access to static fields and methods by XORing all the values on the stack to σ.

This approach requires then to add a routine in charge of checking the invariant property. Also it requires to add one word in each Java frame created. Table 1 in Sect. 5.3 presents the experimental cost of this countermeasure.

5.3 Costs Comparison

In this section, we present in Table 1 the cost (in time) of the different countermeasures introduced in the previous sections. Then we discuss the result of this comparison and the different benefits and drawbacks of the different approaches.

Table 1. Countermeasures impact on bytecode instructions execution time. (% referenced to an initial implementation with no countermeasures.)

Instructions	Basic	Propagation	Invariant
`aload+astore`	39.09%	21.98%	12.29%
`aload+getfield+astore`	19.83%	12.39%	11.75%
`aload+aload+putfield`	27.93%	18.77%	17.59%
`aload+invokevirtual+return`	7.53%	1.69%	1.77%
`aload+invokevirtual+areturn+astore`	8.82%	3.26%	2.38%
`aload+putstatic`	18.60%	11.58%	8.89%
`getstatic+astore`	19.18%	10.76%	10.21%

As the costs for the different countermeasures are given in percentage, it is important to bare in mind that the different instructions have very different complexity (which explains the large difference between the results for the sequences `aload+astore` and `aload+invokevirtual+return` for instance).

The Redundant Approach. The performance degradation caused by this straightforward countermeasure turns out to be not acceptable.

The Propagation Approach. This countermeasure is definitely more efficient than the basic one. To fix the potential issue of context identifier manipulation, an option could be to force legitimate identifiers to even values and propagated errors to odd values. The detection of an invalid identifier would then be straightforward.

The Invariant Approach. The invariant method is also more efficient than the basic one. Its performance is even a little better than that of the propagation countermeasure. Another advantage of this approach is that it does not present the drawback of a potential breach opening.

As expected, the invariant and propagation approaches turn out to be more efficient than the basic one and are relatively close in terms of performance. Nevertheless, the propagation method requires no additional data and only few additional checks on access to static fields. However, as we implemented it, the propagation method potentially opens a security breach in the application firewall. The use of another variable that would be frequently checked may be recommended. Such variables are typically implementation-dependent and we could not exhibit another quasi-standard one. If such variable should not be found in a particular implementation, the invariant approach has proven to be slightly better than the propagation one in terms of execution time. It should easily be implemented at the cost of an additional data-word per Java frame.

6 Conclusion

As stated in the introduction of this work, Java Cards are safer than native devices with regards to Logical Attacks by nature. However, we have raised in this article the issue of the possible alteration of an operand stack and demonstrated how such attacks can eventually compromise both the Java Card platform and the applications loaded on-card. Indeed, we have described and put into practice both Fault and Combined Attacks against a Java Card by disturbing the operand stack.

In particular, this work has permitted to highlight the weakness that represents the lack of a boolean type at the VM-instruction level. Furthermore, we have exhibited new means to combine fault injection with malicious applications to cause a type confusion despite the bytecode verification, and to abuse authentication services through instance confusion.

Finally, we have detailed and compared different countermeasures. Amongst them, both the propagation and invariant approaches bring a good security level without impacting too much the performances of on-card applications. However, future works intending to refine the identification of the moment when an integrity check should be performed may allow to reduce the cost of these countermeasure.

Acknowledgement. The authors would like to thank Nicolas Morin, for his helping hand during the fault injection campaign, and Christophe Giraud for his fruitful review(s).

References

1. Witteman, M.: Java Card Security. In: Information Security Bulletin, vol. 8, pp. 291–298 (2003)
2. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008)

3. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan Applet in a Smart Card. Journale on Computers and Virology 6, 343–351 (2010)
4. Sun Microsystems Inc.: Runtime Environment Specification, Java Card Platform Version 3.0.1 Connected Edition (2009)
5. Sun Microsystems Inc.: Virtual Machine Specification, Java Card Platform Version 3.0.1 Connected Edition (2009)
6. Sun Microsystems Inc.: Application Programming Interface, Java Card Platform Version 3.0.1 Connected Edition (2009)
7. Sun Microsystems Inc.: Java Servlet Specification, Java Card Platform Version 3.0.1 Connected Edition (2009)
8. Giraud, C., Thiebeauld, H.: A Survey on Fault Attacks. In: Smart Card Research and Advanced Applications, CARDIS 2004 (2004)
9. Anderson, R., Kuhn, M.: Tamper Resistance: a Cautionary Note. In: Proceedings of the Second USENIX Workshop on Electronic Commerce, vol. 2, p. 1 (1996)
10. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)
11. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997)
12. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE 94, 370–382 (2006)
13. Barbu, G.: Fault Attacks on Java Card 3 Virtual Machine. In: e-Smart 2009 (2009), http://www.strategiestm-net.com/proceedings/e-smart/OBERTHURGuillaumeBarby-FaultAttacksonJavaCard3VirtualMachine.pdf
14. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg (2010)
15. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 133–147. Springer, Heidelberg (2010)
16. Lindholm, T., Yellin, F.: Java Virtual Machine Specification, 2nd edn. Addison-Wesley, Inc. (1999)
17. Hyppönen, K.: Use of Cryptographic Codes for Bytecode Verification in Smartcard Environment. Master's thesis, University of Kuopio (2003)
18. Hogenboom, J., Mostowski, W.: Full Memory Attack on a Java Card. In: Proceedings of 4th Benelux Workshop on Information and System Security, Louvain-la-Neuve, Belgium (2009), http://www.dice.ucl.ac.be/crypto/wissec2009/static/13.pdf
19. ISO/IEC: 7816-3: Identification Cards – Integrated Circuit Cards – Part 3: Cards with Contacts – Electrical Interface and Transmission Protocols (2006)
20. Govindavajhala, S., Appel, A.W.: Using Memory Errors to Attack a Virtual Machine. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy, SP 2003. IEEE Computer Society (2003)
21. Sere, A.A., Iguchi-Cartigny, J., Lanet, J.L.: Automatic Detection of Fault Attack and Countermeasures. In: Proceedings of the 4th Workshop on Embedded Systems Security, WESS 2009, pp. 1–7 (2009)
22. Séré, A.A.K., Iguchi-Cartigny, J., Lanet, J.L.: Checking the Paths to Identify Mutant Application on Embedded Systems. In: FGIT, pp. 459–468 (2010)
23. Sun Microsystems Inc.: Java Card Protection Profile Collection, version 1.1. Technical report (2006)

A Practical Validation of the Fault Model

The experimental results we present here have been obtained on a recent ARM-based smartcard device. A Java Card 2.2.2 Virtual Machine is running on the device. The fault injection was achieved with a laser equipment.

The Test Application. We intend to provide an experimental validation of the previously introduced fault model. We then develop a Java Card applet, the `process` method of which is exposed below.

```
1. public void process(APDU apdu) {
2.   [...]
3.   ref = Util.getShort(buffer, OFFSET_CDATA);
4.   target = Util.getShort(buffer, (short) (OFFSET_CDATA+2));
5.   ref = ref;     // push and pop ref : sload #ref
                                         sstore #ref
6.   res = target;  // push and pop target : sload #target
                                             sstore #res
7.   Util.setShort(buffer, OFFSET_CDATA, res);
8.   [...]
9. }
```

The first step of our applet consists then in pushing a reference value onto the operand stack and popping it. Therefore we know which value was written in the operand stack before we proceed.

Then we push a second value onto the stack. This second push is the target of our fault injection. The subsequent popped value which is stored in variable `res` is then expected to be an erroneous value. The last step of the applet process is then to send this value out of the card.

Fig. 3 illustrates the evolution of the operand stack along the execution of lines 5 and 6.

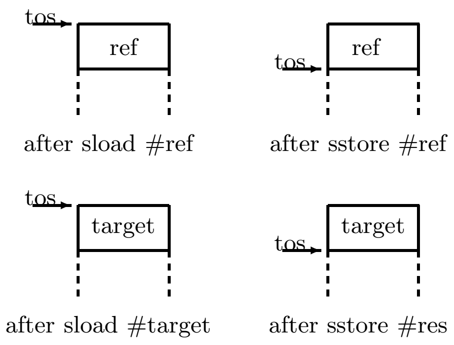


Fig. 3. Evolution of the operand stack content and of the top-of-stack (tos) along execution of lines 5 and 6 of the test applet

Experimental Results. To evaluate the validity of our fault model, we perform several fault attacks with different parameters (namely, the time and space parameters of the attack as well as the width and intensity of the laser beam) and different input parameters.

Table 2 sums up the different results obtained. In this table we only present the results regarding a given couple of input : (`ref`, `target`) = (`0xAABB`, `0xCCEE`). The cells highlighted in grey within Table 2 denote the results that were dependent on the inputs. We also highlighted the results 0x0000 and 0xFFFF which correspond to the two *stuck-at* fault models. Note that we only present in this table the results that were reproducible.

Table 2. Results (`res` values) of the fault attacks with various fault injection parameters and fixed `ref` and `target` values

0x00F1	0x0000	0x00F2	0x00CC	0x149C	0x0121	0x0D88	0xFF19
0x0006	0x0129	0x149E	0xEA00	0x00BB	0x27FF	0x168F	0x000D
0x1490	0x4778	0x0011	0xD203	0xC14A	0x00D9	0xAABC	0x1200
0x2B2B	0x0012	0x5576	0xBB00	0x6000	0x7600	0xFFFF	0xAA0B
0xAA8B	0x2AAF	0xAAEC	0xAAEB	0xABB0	0x2AAE	0xAB6B	0xEEA0

Conclusions. It is difficult to deduce the exact perturbation caused by the laser beam in all cases, especially when the results are not correlated with the inputs. Such results may be correlated with internal values contained in the registers of the processor at the time the laser is activated.

On the other hand, some results can be easily interpreted since they are compound of different chunks of either the reference or the target value and 0s (for instance, `0x0000`, `0x00CC`, `0xBB00`).

To conclude, the results we obtained only partially validate our fault model since `res` is either all-0, all-1, truncated `ref`, truncated `target` or other unknown values.

However this proves that an adversary can manage to disturb the push operation and may have a certain control on the erroneous operand eventually pushed.

Formal Analysis of CWA 14890-1

Ashar Javed

Hamburg University of Technology (TUHH)
Hamburg, Germany
ashar.javed@tu-harburg.de

Abstract. Formal analysis is of importance in order to increase confidence that the protocol satisfies its security requirements. In particular, the results obtained from the formal analysis of the smart card security protocols when smart cards are used as a specific type of Secure Signature Creation Devices (SSCDs) are presented. SSCDs are developed to support the EU-directive on electronic signatures. In this paper, we focus on security properties, called the *authentication* and *secrecy*. The device authentication protocols mentioned in CWA 14890-1 are modeled using the high-level protocol specification language HLPSL and verified with the help of AVISPA tool. Our formal analysis does not reveal any weaknesses of the CWA 14890-1 protocol suite.

Keywords: smart cards, CWA 14890-1 authentication protocols, formal analysis, SSCDs, AVISPA.

1 Introduction

CEN Workshop Agreement CWA 14890-1 [1] describes the European standardization activities and solutions for smart cards as a specific type of a Secure Signature Creation Device (SSCD). SSCDs means configured software or hardware which is used to manipulate the Signature Creation Data (SCD) [2]. SCD is unique data, such as codes or private cryptographic keys, which are used by the signatory to create an electronic signature. SSCDs are developed to support the EU-directive on electronic signatures. It builds on ISO/IEC 7816-4 [3]. The key issue of CWA 14890-1 is to enable interoperability, so that smart cards from different manufacturers can interact with different kind of signature creation applications [1]. CWA 14890-1 describes the following device authentication protocols:

– An Asymmetric Session Key Agreement Protocol with Privacy Protection
– An Asymmetric Session Key Transport Protocol based on RSA
– Symmetric Authentication Protocol

Formal analysis is of importance in order to increase the assurance that the protocol satisfies its security requirements. In this sense, the Automated Validation of Internet Security Protocols and Applications (AVISPA) and the Security

Protocol Animator for AVISPA (SPAN) [6] tools have been used to validate device authentication protocols mentioned in CWA 14890-1.

Contribution. Our main contribution is to provide the first comprehensive formal analysis of CWA 14890-1. We explain formalization of the protocols in AVISPA's high-level protocol specification language HLPSL [7], and describe approach to verifying that the device authentication protocols in CWA 14890-1 does indeed satisfy the security requirements and are safe w.r.t. finite number of sessions in our analysis runs. Even though the CWA 14890-1 specs amount to over 150 pages, our formal models are 16 pages of HLPSL code. Of course, our model abstracts from some of the low-level details that are in the 150-odd pages CWA 14890-1 specs, e.g., bit-level selection of data but such abstraction seems crucial to keep an overview and understand the CWA 14890-1 standard as a whole.

Synopsis. The paper is organised as follows. In Section 2 a brief overview of AVISPA tool. Section 3 comments on smart cards. Section 4 is devoted to the description and specifications of an asymmetric session key agreement protocol. In Section 5, we describe the asymmetric key transport protocol. Section 6 comments on symmetric authentication protocol. Finally in Section 7 we give our conclusions.

2 The AVISPA Tool

The AVISPA tool is used for the Automated Verification of Internet Security Protocols and Applications [4,5]. High-Level Protocol Specification Language (HLPSL) is used to interact with the AVISPA tool. In HLPSL, protocol designer specify a security protocol along with the security requirements that should be met. HLPSL is role based formal specification language. The AVISPA tool automatically translates (via HLPSL2IF Translator) a security protocol into an equivalent description written in the rewriting-based formalism known as Intermediate Format IF [8].

The current version of the tool integrates four back-ends: the On-the-fly Model-Checker OFMC [9], the Constraint-Logic-based Attack Searcher CL-AtSe [10], the SAT-based Model-Checker SATMC [11], and the Tree Automata tool based on Automatic Approximations for the Analysis of Security Protocols analyzer TA4SP [12]. All the back-ends of the tool analyze protocols under the assumptions of perfect cryptography and that the protocol messages are exchanged over a network that is under the control of a Dolev-Yao intruder [13].

OFMC and CL-AtSe are dedicated to the *refutation* of protocols and focus on a finite number of sessions. The analysis with the SATMC and TA4SP back-ends were always 'Inconclusive' in our experiments. Both back-ends do not support `modulus` and `xor` operators, thats why analysis was always 'Inconclusive'. The device authentication protocol mentioned in CWA 14890-1 make use of `modulus` and `xor` operators.

Space does not permit discussion on back-ends. We refer to the AVISPA's user manual (google avispa-project) for deeper concerns about AVISPA. *From* *now on we will only consider OFMC and CL-AtSe back-ends for the analysis of protocols with AVISPA tool.*

3 Smart Card Commands

Following smart card commands specified in ISO/IEC 7816-4 [3] suffice to cover all smart cards related authentication protocols.

1. *Manage Security Environment (MSE)* command informs the smart card about the encryption algorithms, signature algorithms, hash algorithms and keys to be used in the subsequent command sequence.
2. The *Get Challenge* command requests challenge (e.g., random number) from the smart card.
3. The *Get Data / Read Binary* command is used for the retrieval of data object(s) e.g., serial number.
4. *Perform Security Operation (PSO)* command is used to compute hash and digital signature. The data to be hashed or the final digest respectivly are sent to the card with the *PSO*.
5. The *Internal Authenticate* is used when a smart card has to authenticate itself, the terminal sends an *Internal Authenticate* Command.
6. When a terminal has to authenticate itself, the smart card expects an *External Authenticate* command containing an authentication token.
7. Finally the *Mutual Authenticate* command combines *Internal Authenticate* and *External Authenticate* into one command.

4 An Asymmetric Session Key Agreement Protocol with Privacy Protection

Key agreement is the process of establishing a shared secret key between two entities A and B in such a way that neither of them can predetermine the value of the shared secret key. [1].

In an asymmetric session key agreement protocol with privacy protection, to avoid the card disclosing private information, such as identity, a secure channel session is established before any other operation. To do so, the protocol starts with an *unauthenticated Diffie-Hellman key exchange* [15] and then authenticates the Interface Device (IFD) before the Integrated Circuit Card (ICC). In device authentication protocol card reader (i.e., IFD) can authenticate itself to smart card (i.e., ICC) without having to know smart card's identity. The following section shows the general flow of device authentication protocol.

4.1 Authentication Steps

Step 1. The reader starts the protocol by sending the *Read Binary* command. With the help of *Read Binary* command IFD reads the public key quantities from the file. For instance, in a Diffie Hellman Key exchange scheme the public key quantities would be the public parameters p, q and g. The public key quantities reveal information about the authentication mechanism. As long as these quantities are used in many ICCs, the identity of an ICC cannot be determined from this information.

$$\textbf{Card Reader} \xrightarrow{\texttt{Read Binary Command}} \textbf{Card}$$

$$\textbf{Card Reader} \xleftarrow{\texttt{p, q, g}} \textbf{Card}$$

Step 2. After receiving the public parameters, card reader chooses random number `a` with $1 \leq$ `a` $\leq$ `q - 1`, computes a key token $K_{IFD} = g^a$ mod `p` and sends the key token to smart card with the help of *Manage Security Environment* command in order to establish a secure session. 'OK' as response from smart card after receiving the key token i.e., K_{IFD}.

$$\textbf{Card Reader} \xrightarrow{\substack{\texttt{Manage Security Environment Command} \\ K_{IFD}}} \textbf{Card}$$

$$\textbf{Card Reader} \xleftarrow{\texttt{ok}} \textbf{Card}$$

Step 3. The opposite key token/portion K_{ICC} is returned in the response of a *Get Data* command. Upon receiving the *Get Data* command smart card computes $K_{ICC} = g^b$ mod `p` and transmits key token K_{ICC} to card reader.

$$\textbf{Card Reader} \xrightarrow{\texttt{Get Data Command}} \textbf{Card}$$

$$\textbf{Card Reader} \xleftarrow{K_{ICC}} \textbf{Card}$$

Step 4. At this point, neither card reader nor smart card has revealed his identity. The reader and the card have completed simple unauthenticated Diffie-Hellman key agreement. Neither side has been authenticated yet, but by using the common secret i.e., K_{IFDICC}, they now derive an encryption key `Kenc` and a MAC key `Kmac` that will be used to protect the remainder of the authentication protocol from casual eavesdroppers. Both keys are 112-bit 3DES keys. Note that there could still be a man-in-the-middle at this point in the protocol. If we model authentication (as an experiment) with the help of `witness` and `request` goal facts (see section 4.5) at this point then AVISPA also finds man-in-the-middle. It does not make sense to model authentication at this point of the protocol. The reason is that man in the middle attacker could have made contact with the IFD by himself – he did not even need to be in the middle as the protocol involves an anonymous DH-Key Exchange. Both the reader and the card calculate:

$$\text{HASH1} = \text{HMAC}[K_{IFDICC}]\ (1)$$
$$\text{HASH2} = \text{HMAC}[K_{IFDICC}]\ (\text{HASH1} \mid\mid 2)$$

112 bits are selected from HASH1 to produce `Kenc`, and 112 bits are selected from HASH2 to produce `Kmac`. After generation of the encryption and MAC keys reader sends the *Manage Security Environment* command. The MSE command sets the key reference of the public key of trusted certification authority to be used for the verification of the IFD's authentication certificate. 'OK' as response from the smart card because we assume that key of trusted certification authority is present in card.

`Manage Security Environment Command`

Card Reader $\longrightarrow$ **Card**

`ok`

Card Reader $\longleftarrow$ **Card**

Step 5. Upon receiving 'OK', reader now sends its certificate to the card by encrypting it with `Kenc`. The MAC of the encrypted certifcate is also send to the card with the help of *Perform Security Operation* command. The card will verify the certificate with the help of public key of certification authority. After verification card will send 'OK' as response.

`Perform Security Operation Command`
`({Reader.PKifd}_Kenc).Hash(Kmac, ({Reader.PKifd}_Kenc))`

Card Reader $\longrightarrow$ **Card**

`ok`

Card Reader $\longleftarrow$ **Card**

Step 6. Upon receiving 'OK' from the smart card, the reader requests a 64 bits (8 bytes) random number (`RND_ICC`) from the smart card with a *Get Challenge* command in order to prove its authenticity dynamically. Upon receiving *Get Challenge*, the smart card generates a 64 bits random number used as nonce. The smart card stores random number in its internal memory and replies with random number to the reader.

`Get Challenge Command`

Card Reader $\longrightarrow$ **Card**

`RND_ICC`

Card Reader $\longleftarrow$ **Card**

Step 7. The IFD computes a signature on the concatenation of the challenge (PRND2) with its own key token (K_{IFD}) information. The signature is a signature with message recovery, so all parameters in the signature can be considered to be recoverable. The Diffie-Hellman key parameters (DH.P) are part of the signature in order to provide authenticity of the parameters. The MAC of the encrypted signature is also transmitted to the card with the help of *External Authenticate* command. Card now verifies the MAC, decrypts, and verifies the signature using private key generated in step 4. At the conclusion of this step, card has authenticated reader and knows that K_{IFD} and K_{ICC} are fresh and authentic. 6A and BC are hexadecimal numbers. After verification card sends 'OK' as response.

```
External Authenticate Command
SIG = 3DESEncrypt Key (6A || PRND2 || hash[PRND2 || KIFD || SN_IFD || RND_ICC
|| KICC || DH.P] || BC)
```

Card Reader $\longrightarrow$ **Card**

```
ok
```

Card Reader $\longleftarrow$ **Card**

Step 8. Upon receiving 'OK', reader sends *Read Binary* Command in order to get the ICC's encrypted certificate plus MAC of encrypted certificate. However at this point, while card knows there is no man-in-the-middle because card checked the signature from reader, reader does not know whom he is talking to, and hence is unsure if there may be a man-in-the-middle attack, thats why he demanded the certificate.

```
Read Binary Command
```

Card Reader $\longrightarrow$ **Card**

```
({Card.PKicc}_Kenc).Hash(Kmac, ({Card.PKicc}_Kenc))
```

Card Reader $\longleftarrow$ **Card**

Step 9. Before processing the *Internal Authenticate* command the ICC's private authentication key must be set by the *Manage Security Environment* command. The MSE command updates the current security environment.

```
Manage Security Environment Command
```

Card Reader $\longrightarrow$ **Card**

```
ok
```

Card Reader $\longleftarrow$ **Card**

Step 10. The IFD performs an *Internal Authenticate* command. The challenge (`PRND`) sent to the ICC with this command is `RND_IFD`. The ICC then computes the signature over the challenge and the key token K_{IFD}, K_{ICC} and returns it to the IFD encrypted with secure messaging. The signature is a signature with message recovery, so all parameters in the signature can be considered to be recoverable. Reader can now verify the MAC and decrypt signature. The IFD verifies the certificate using the public key of the trusted certification authority.

$$\begin{tikzcd}[column sep=huge] \textbf{Card Reader} \arrow[r, "\substack{\texttt{Internal Authenticate Command}\\ \texttt{RND\_IFD}}"] & \textbf{Card} \end{tikzcd}$$

```
SIG = 3DESEncrypt Key ( 6A || PRND || hash(KICC || SN_ICC || RND_IFD ||
                          KIFD || DH.P) || BC )
```

$$\begin{tikzcd}[column sep=huge] \textbf{Card Reader} & \textbf{Card} \arrow[l] \end{tikzcd}$$

4.2 HLPSL Specifications

In this section we show the fragments of HLPSL specifications of protocol. Our intended target is to model the protocol as close as possible to the protocol description given in 4.1. The protocol behavior could be modeled, except next issues / limitations. The detailed HLPSL model can be found in [18]. We refer to the AVISPA's user manual (google avispa-project) for deeper concerns about HLPSL.

Restrictions Applied:

– Our HLPSL model abstracts from some of the low-level details that are in the protocol specifications, e.g., bit-level selection of data. Such abstraction seems crucial to keep an overview and understand the protocol as a whole. During the generation of encryption and MAC keys, protocol make use of bit-level selection of data. This could not be mapped in HLPSL specifications.
– Protocol includes provisions for the optional exchange of chain of public-key certificates. This is not included in the model.
– We assume that appropriate public key of the trusted certification authority is present in IFD and in ICC.

4.3 Roles

In order to describe the protocol we should specify the actions of each kind of participant, i.e., the basic roles. Roles are independent processes: they have a name, receive information by parameters and contain local declarations. Basic roles are played by an agent whose name is received as parameter. The actions of a basic role are transitions, describing changes in their state depending on events or facts. To describe protocol in HLPSL we introduce two basic roles: ifd and icc. We now present the declaration of basic roles and their (typed) parameters in HLPSL:

```
role ifd (  Reader, Card: agent, PKifd : public_key, Hash, SHA1 : hash_func,
            SND, RCV: channel (dy) ) played_by Reader def=
  local
        State, C1, C2      : nat,
        PKicc : public_key,
        KICC, KIFD : symmetric_key,
        ...
role icc (  Reader, Card: agent, PKicc : public_key, Hash, SHA1 : hash_func,
            SND, RCV: channel (dy)) played_by Card def=
  local
        State, C1, C2 : nat,
        PKifd : public_key,
        KICC, KIFD : symmetric_key,
        ...
```

Here we have: `Reader`, `Card` are agents playing roles ifd and icc respectively. `Hash`, `SHA1` are hash functions used in calculating MAC and in session key generation respectively. `PKifd`, `PKicc` are public keys of Reader and Card respectively. `SND`, `RCV` are channels for sending and receiving messages. In HLPSL variable names start with capital letters; constants, keywords[1] and data types[2] start with lower-case letters.

4.4 Composed Roles

Composed roles instantiate one or more basic roles, "**gluing**" them together so that they execute together, usually in parallel (with interleaving semantics) [7]. Composed roles have no transition section rather a composition section in which the basic roles are instantiated. The composition is presented in HLPSL.

```
role session( Reader, Card: agent,
              Hash, SHA1 : hash_func,
              PKifd, PKicc : public_key )
def=
  local SIFD, RIFD, SICC, RICC: channel (dy)
  composition
        ifd(Reader,Card,PKifd,Hash,SHA1,SIFD,RIFD)
     /\ icc(Reader,Card,PKicc,Hash,SHA1,SICC,RICC)
end role
```

Symbol $\wedge$ denotes here a parallel execution. A transition is a rule that can be fired if the left-hand side is satisfied i.e., before symbol `=|>`.

[1] Role, played_by and def = are keywords in HLPSL.

[2] Agent, public_key, hash_func and channel are data types in HLPSL. dy is the attribute of channel type and it represents Dolev-Yao.

4.5 Role IFD

For reasons of space we show the transitions of the role ifd that are more important as compared to the other one because these transitions contain the sending of the key token of the reader i.e., K_{IFD}, signed certificate along with the *External Authenticate* Command.

```
2.  State   = 2 /\ RCV(G'.P'.Q') =|>
%% Manage Security Environment Command. The detailed syntax of MSE command
%%can be found in CWA 14890-1. A is a random number that lies between 1 and q-1
     State' := 4 /\ A' := new()
%% KIFD' is a Key Token of Reader used in generation of Mutual Key i.e., KIFDICC
    /\ KIFD' := exp(G',A')
    /\ SND(two_two.four_one.a_6.1_a_6.KIFD')
8.  State   = 8 /\ RCV(ok) =|>
%% Perform Security Operation Command. The detailed syntax of MSE command
%%can be found in CWA 14890-1. {Reader.PKifd}_Kenc is a certificate encrypted
%%with encryption key i.e., Kenc & contains the agent name
%%of reader & its public key. Reader now transmits it together with its MAC to Card.
State' := 10 /\
    SND (two_a.zero_zero.a_e.(({Reader.PKifd}_Kenc). Hash(Kmac,({Reader.PKifd}_Kenc))))
12. State   = 12  /\ RCV(RND_ICC') =|>
    %% EXTERNAL AUTHENTICATE COMMAND
State' :=  14 /\ PRND' := new()
%% SIG = 3DESEncrypt Key (6A || PRND2 || h[PRND2 || KIFD
    %% || SN.IFD || RND.ICC || KICC || DH.P || BC)
/\     SND(eight_two.zero_zero.zero_zero.{({six_a.PRND'.KIFD.sn_ifd.
       RND_ICC'.KICC.(G.P.Q).b_c}_inv(PKifd))}_Kenc.
       Hash(Kmac,{({six_a.PRND'.KIFD.sn_ifd.
       RND_ICC'.KICC.(G.P.Q).b_c}_inv(PKifd))}_Kenc ))
/\ witness(Reader,Card,ifd_icc_run_id_for_authentication_of_ifd,RND_ICC')
```

This first transition in the above code fragment is called `'2.'`, though the names of the transitions serve merely to distinguish them from one another. It specifies that if the value of State is equal to 2 and a message (Diffie-Hellman Public Quantities) is received on channel `RCV`, then a transition fires which sets the new value of State to 4 and sends the *Manage Security Environment Command* on channel `SND`. HLPSL uses dot operator to denote the concatenation of messages as you see in the `RCV` channel above. Comments in HLPSL begin with the % symbol and continue to the end of the line. In any transition, the old value and the new value of a variable are syntactically distinguished: the prime symbol ' has to be attached to the name of a variable for considering its new value. Prime notation stems from the temporal logic TLA [17], upon which HLPSL is based. It is important to realise that the value of the variable will not be changed until the current transition is complete. So, the right-hand-side tells us that the value of the State variable, after transition `'2.'` fires, will be 4.

The second transition in HLPSL model above states that: if the value of variable State equals to 8 and we receive on channel `RCV ok` then reader sends the *Perform Security Operation Command* in which reader sends its certificate (Identity plus public key) encrypted with encryption key i.e., `Kenc`. The MAC of the encrypted certificate is also sent so that card can check the integrity with the help of MAC key (i.e., `Kmac`) also available at the card side". HLPSL uses the same notation for the encryption and decryption i.e., `{Message}_Key`.

Another transition is: The IFD computes a signature on the concatenation of the challenge with its own key token (K_{IFD}) information. The signature is a signature with message recovery, so all parameters in the signature can be considered to be recoverable. The Diffie-Hellman key parameters are part of the signature in order to provide authenticity of the parameters. The MAC of the encrypted signature is also transmitted to the card with the help of *External Authenticate Command*. Note that for the analysis tool as well as for the modeling in the tool's language, a signature is equivalent to an encryption with a private key i.e., `{Message}_inv(PublicKey)`.

4.6 Modelling Authentication in HLPSL

Authentication, security property, is modelled by means of several goal predicates in HLPSL: witness(agent,agent,protocol_id,message), request(agent,agent,protoc ol_id,message) and wrequest(agent,agent,protocol_id,message). The `witness` and `request` are goal facts related to authentication. The last line of the transition in above section 4.5 named `'12.'` is an authentication related event `witness`. Here it should be read as follows: means that honest agent Reader wants to execute the protocol with agent Card by using (RND_ICC') as value for the authentication identifier ifd_icc_run_id_for_authentication_of_ifd. Goal facts `witness` and `request` are used to check that a principal is right in believing that its intended peer is present in the current session, has reached a certain state, and agrees on a certain value, which typically is fresh. They have identical third parameter and it should be declared as a constant of type `protocol_id` in the top-level role. The label ifd_icc_run_id_for_authentication_of_ifd (of type `protocol_id`) is used to identify the goal. The third parameter is used to associate the `witness` and `request` predicates with each other and to refer to them in the goal section. There is also `wrequest` which corresponds to weak authentication (also called non-injective agreement according to Lowe's paper [16]). No replay protection is imposed if one uses `wrequest`.

Goal fact `request` means that agent Card accepts the value (TEMP_RND_ICC') and now relies on the guarantee that agent Reader exists and agrees with him on the value (TEMP_RND_ICC') for the authentication identifier ifd_icc_run_id_for_a uthentication_of_ifd. Goal fact `request` is used for strong authentication (also called injective agreement according to Lowe's paper [16]). In general, the authenticated issues a `witness` fact as soon as possible in his protocol execution, i.e., as soon as he has known the name of the authenticator and the payload (the data that shall be agreed upon) message. The authenticator issues in his last rule a `request` fact, i.e., when he has executed the protocol to the end during the session from his point of view. Also authentication goals have a direction, namely from an **authenticator to an authenticated**, in the sense that the authenticator convinces himself of the identity of the authenticated.

4.7 Role ICC

Now we show role icc's transition responsible for the receiving of the key token and signature from the card reader:

```
3.  State  = 3 /\ RCV(two_two.four_one.a_6.1_a_6.KIFD')  =|>
    State':= 5 /\ SND(ok)
13. State  = 13  /\
        RCV(eight_two.zero_zero.zero_zero.{({six_a.PRND'.KIFD'.
    TEMP_SN_IFD'.TEMP_RND_ICC'.KICC'.
    (G'.P'.Q').b_c}_PKifd)}_Kenc.Hash(Kmac,{({six_a.PRND'.
    KIFD'.TEMP_SN_IFD'.TEMP_RND_ICC'.KICC'.
    (G'.P'.Q').b_c}_PKifd)}_Kenc ))
%%  Verify the random number
        /\ RND_ICC = TEMP_RND_ICC'
%% Card has now authenticated Reader and knows that KIFD and KICC are fresh and authentic.
        /\ KIFD = KIFD'
        /\ KICC = KICC'
=|>
    State':= 15  /\ SND (ok)
    /\ request(Card,Reader,ifd_icc_run_id_for_authentication_of_ifd,TEMP_RND_ICC')
```

The first transition in the above code fragment of role icc shows the reception of the key token of the IFD i.e., K_{IFD} used in session key generation. After reception of the key token card sends ok.

The second transition shows the reception of the signature. It also shows that at this point of time agent card is verifying the random number that he generated earlier against the random number received as a part of signature. $=$ is comparison operator in HLPSL. Card also checks the authenticity of the key tokens. After checking the authenticity of key tokens, card issued authentication related event `request` described in section 4.6.

4.8 Role Environment

A top-level role is always defined. This role contains global constants and a composition of one or more sessions, where the intruder may play some roles as a legitimate user. Here we also define an initial intruder knowledge set using `intruder_knowledge` token. Initially the intruder knows all agents' names along with their public keys (`Pkifd` and `Pkicc`), hash functions (`h` and `sha1`) and his public and private keys i.e., `ki` and `inv(ki)` respectively. The constant `i` is used to refer to the intruder. One should introduce a goal section to define security goals and to look for an attack. The authentication properties to be checked are listed in the goal section. The `authentication_on` keyword specify authentication goal with replay protection i.e., freshness of the agreement (or session) between the two, and directly corresponds to Lowe's injective agreement [16].

```
goal
authentication_on ifd_icc_run_id_for_authentication_of_ifd
authentication_on icc_ifd_run_id_for_authentication_of_icc
end goal
environment()
```

4.9 Automatic Analysis of the Protocol

The analysis with the AVISPA tool is performed on the following parallel sessions scenarios of the protocol.

Man-in-the-Middle Attack Scenario

In our first experiment, we consider a configuration: One session between agents reader and card and one session between card and reader in order to check for *man-in-the-middle* attack.

```
   session(reader,card,h,sha1,pkifd,pkicc)
 ∧ session(card,reader,h,sha1,pkicc,pkifd)
```

Replay Attack Scenario

In our second experiment, we consider a configuration: One normal session between agents reader and card and in order to check for *replay attacks* we repeat the normal session.

```
   session(reader,card,h,sha1,pkifd,pkicc)
 ∧ session(reader,card,h,sha1,pkifd,pkicc)
```

Impersonating Attack Scenarios

In our third experiment, the analysis is also performed on the set of configurations where an intruder (represented by `i`) is playing the role of legitimate agent(s) in order to attack the protocol. In AVISPA we can define an explicit intruder knowledge set using `intruder_knowledge` token. In the following analysis scenario intruder is *impersonating* `card`.

```
   session(reader,card,h,sha1,pkifd,pkicc)
    ∧ session(reader,i,h,sha1,pkifd,ki)
```

In our forth experiment, the analysis is performed on parallel sessions where intruder is *impersonating* `reader` in order to attack the protocol.

```
   session(reader,card,h,sha1,pkifd,pkicc)
     ∧ session(i,card,h,sha1,ki,pkicc)
```

Experiments

From now on we tested different parallel session scenarios until the results were exhaustive. In our fifth experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. One session between reader and intruder.

```
   session(reader,card,h,sha1,pkifd,pkicc)
 ∧ session(card,reader,h,sha1,pkicc,pkifd)
    ∧ session(reader,i,h,sha1,pkifd,ki)
```

In our sixth experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. One session between intruder and card.

```
    session(reader,card,h,sha1,pkifd,pkicc)
  ∧ session(card,reader,h,sha1,pkicc,pkifd)
      ∧ session(i,card,h,sha1,ki,pkicc)
```

In our seventh experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. One session between reader and intruder. One session between intruder and card.

```
    session(reader,card,h,sha1,pkifd,pkicc)
  ∧ session(card,reader,h,sha1,pkicc,pkifd)
     ∧ session(reader,i,h,sha1,pkifd,ki)
      ∧ session(i,card,h,sha1,ki,pkicc)
```

In our seventh experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. Two sessions between reader and intruder. One session between intruder and card.

```
    session(reader,card,h,sha1,pkifd,pkicc)
  ∧ session(card,reader,h,sha1,pkicc,pkifd)
     ∧ session(reader,i,h,sha1,pkifd,ki)
     ∧ session(reader,i,h,sha1,pkifd,ki)
      ∧ session(i,card,h,sha1,ki,pkicc)
```

In our eight experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. Session between reader and intruder. Two sessions between intruder and card.

```
    session(reader,card,h,sha1,pkifd,pkicc)
  ∧ session(card,reader,h,sha1,pkicc,pkifd)
     ∧ session(reader,i,h,sha1,pkifd,ki)
      ∧ session(i,card,h,sha1,ki,pkicc)
      ∧ session(i,card,h,sha1,ki,pkicc)
```

In our ninth experiment, we consider a configuration: One normal session between agents reader and card. One session between card and reader. Two sessions between reader and intruder. Two sessions between intruder and card.

```
    session(reader,card,h,sha1,pkifd,pkicc)
  ∧ session(card,reader,h,sha1,pkicc,pkifd)
     ∧ session(reader,i,h,sha1,pkifd,ki)
     ∧ session(reader,i,h,sha1,pkifd,ki)
      ∧ session(i,card,h,sha1,ki,pkicc)
      ∧ session(i,card,h,sha1,ki,pkicc)
```

In our last experiment, we consider a configuration: Two parallel sessions between agents reader and card. One session between card and reader. Two sessions between reader and intruder. Two sessions between intruder and card.

```
     session(reader,card,h,sha1,pkifd,pkicc)
   ∧ session(reader,card,h,sha1,pkifd,pkicc)
   ∧ session(card,reader,h,sha1,pkicc,pkifd)
      ∧ session(reader,i,h,sha1,pkifd,ki)
      ∧ session(reader,i,h,sha1,pkifd,ki)
       ∧ session(i,card,h,sha1,ki,pkicc)
       ∧ session(i,card,h,sha1,ki,pkicc)
```

This is actually the limit of the analysis tool (i.e., AVISPA's OFMC and CL-AtSe) for this protocol: with more parallel sessions, no answer comes. While it may be interesting to use parallel computing to raise this limit, we think that the analyzed scenarios are the most relevant ones for the smart card protocol. Our analyzed scenarios fall into the following categories: Man in the middle attacks, replay attacks, impersonation and parallel session attacks attacks. In all experiments, the security properties under test are the correspondence properties explained above in section 4.6. **For all analyzed scenario, no attacks were found on the protocol in the presence of DY model.** According to [14],

> *"Finding an attack for a protocol with a fixed number of sessions is a NP-complete*[3] *problem with respect to a Dolev-Yao model [13] of intruders. Results does not assume a limit on the size of messages intruder can generate."*

Table 1 shows the summary of validation results using AVISPA.

Table 1. AVISPA Validation Results

AVISPA Tool Summary	
Backend Tool	**Result**
OFMC	Safe
CL-AtSe	Safe

5 Specification and Analysis of Asymmetric Key Transport Scheme Based on RSA

Key Transport is the process of transferring a secret key, chosen by one entity (or a trusted center), to another entity, suitably protected by asymmetric techniques. Key transport requires encryption of the key part, this is typically done with the public key of the counterpart [1].

The protocol mentioned is favorable for situations where the ICC should be authenticated prior to the IFD. It is appropriate to application fields where privacy is not a key issue but simplicity is required and/or the ICC is to be authenticated first. The Key Transport Protocol also specifies that smart card

[3] The complexity class NP (Non-deterministic Polynomial time) is the set of all decision problems solvable in polynomial time on a non-deterministic Turing machine.

and reader agree on two session keys to protect subsequent communications. In smart card applications, one key is used for encryption/decryption operations while the second key is used to compute message authentication codes (MAC). CWA 14890-1 specifies two key triple DES (2KTDES) in cipher block chaining mode with fixed initialization vector 0 as encryption method. The length of challenges is specified as 64 bits. Additionally, the procedure to establish session keys is defined and the length of the key derivation data is set to 256 bits.

5.1 Authentication Steps

Step 1. The reader starts the protocol with a `Manage Security Environment` command informing the smart card which key reference will be used for subsequent messages. For simplicity we assume that the key reference is for the public key of the trusted certification authority. The public key is used for the verification of the IFD's certificate. 'OK' will be the ICC's response if referenced public key of the certification authority is present in card and in this case it is there.

$$\textbf{Card Reader} \xrightarrow{\texttt{Manage Security Environment Command}} \textbf{Card}$$

$$\textbf{Card Reader} \xleftarrow{\texttt{ok}} \textbf{Card}$$

Step 2. The reader next sends the `Perform Security Operation` command in which reader sends the certificate to the smart card. The certificate is encrypted / signed[4] with the private key of the reader and it has the identity of the sender along with its public key . The card verifies the certificate using the public key whose reference was received in the previous step. After verification card sends 'OK' as response. At this point of time the public key of the IFD is now known by the ICC, and can be trusted.

$$\textbf{Card Reader} \xrightarrow{\texttt{Perform Security Operation Command}} \textbf{Card}$$

$$\textbf{Card Reader} \xleftarrow{\texttt{ok}} \textbf{Card}$$

Step 3. Next, the reader fetches the serial number of the smart card along with the card certificate that contains the identity of the card(i.e., `Card`) along with its public key (i.e., `PKicc`) encrypted / signed by the help of card's private key (`inv(PKicc)`) by sending the `Read Binary` command. The IFD verifies the certificate using the public key of the trusted certification authority. The public key of ICC is now known by the IFD, and can be trusted.

[4] Note that for the analysis tool as well as for the modeling in the tool's language, a signature is equivalent to an encryption with a private key i.e., `{Message}_inv(PublicKey)`.

`Read Binary Command`

$$\textbf{Card Reader} \xrightarrow{\hspace{6cm}} \textbf{Card}$$

`{Card.PKicc.sn_icc}_inv(PKicc)`

$$\textbf{Card Reader} \xleftarrow{\hspace{6cm}} \textbf{Card}$$

Step 4. Next the `Manage Security Environment` command updates the current security environment by setting the ICC's private authentication key. Furthermore, the IFD's public key needs to be selected for encryption in order to transport the ICC's key contribution data i.e., `KICC`. `KICC` is a 32 byte random number generated by the ICC. Keys are set by sending the key references (private key of card plus public key of the reader) to the card. Card will send 'OK' as response.

`Manage Security Environment Command`

$$\textbf{Card Reader} \xrightarrow{\hspace{6cm}} \textbf{Card}$$

`ok`

$$\textbf{Card Reader} \xleftarrow{\hspace{6cm}} \textbf{Card}$$

Step 5. Upon receiving 'OK' from smart card, the reader generates its own 64 bits random number and sends random number along with its serial number in the `Internal Authenticate Command`.

```
Internal Authenticate Command
     RND_IFD || SN_IFD
```

$$\textbf{Card Reader} \xrightarrow{\hspace{6cm}} \textbf{Card}$$

Upon receiving the `Internal Authentication` command, the smart card then computes the digital signature. Generate the padding random number (`PRND1`) and concatenates with the random number generated for the key derivation (`KICC`) and takes the hash of padding random number, random number, reader's serial (`SN_IFD`) and random number (`RND_IFD`), encrypts the concatenated data using its private key and sends the encrypted data to the reader. The reader decrypts the data with the public key and checks whether the second random number equals the one stored in its internal memory and whether the second serial number matches its own. After verification the reader authenticated the card.

```
SIG = DS[Private Key.ICC.AUT](6A || PRND1 || KICC || hash(PRND1 ||
                          KICC || C) || BC)
                      where C = SN_IFD || RND_IFD
```

$$\textbf{Card Reader} \xleftarrow{\hspace{6cm}} \textbf{Card}$$

Step 6. Then, the reader requests a 64 bits (8 bytes) random number (`RND_ICC`) from the smart card with a `Get Challenge` command in order to prove its authenticity dynamically. Upon receiving `Get Challenge`, the smart card generates a 64 bits random number used as nonce. The smart card stores random number in its internal memory and replies with random number to the reader.

$$\begin{tikzcd}\textbf{Card Reader} \arrow[rrr, "\texttt{Get Challenge Command}"] & & & \textbf{Card}\end{tikzcd}$$

$$\begin{tikzcd}\textbf{Card Reader} & & & \textbf{Card} \arrow[lll, "\texttt{RND\_ICC}"']\end{tikzcd}$$

Step 7. Upon receiving `Get Challenge`, the reader computes the signature. Generates a padding random number (`PRND`) and concatenates with `KIFD` i.e., a 32 byte random number generated by the IFD for key derivation / key token and takes the hash of padding random number, random number, card's serial (`SN_ICC`) and random number (`RND_ICC`), encrypts the concatenated data using its private key and sends the encrypted data to the card with the help of `External Authenticate` command. `External Authenticate` command delivers the digital signature of the IFD to the card.

`External Authenticate Command`
`where SIG = DS [Private Key.IFD.AUT] (6A||PRND||KIFD||hash(PRND.||KIFD||`
`RND_ICC||SN_ICC)||BC)`

$$\begin{tikzcd}\textbf{Card Reader} \arrow[rrr] & & & \textbf{Card}\end{tikzcd}$$

Upon receiving the `External Authentication` command, smart card decrypts the data with the public key of the counterpart and checks whether the second random number equals the one stored in its internal memory and whether the second serial number matches its own. After verification the card now authenticated the reader.

Once smart card and reader have stored both their own and received key derivation data, they can generate the session keys. CWA 14890-1 specifies two key triple DES (2KTDES) as the algorithm to be used for encryption, decryption, and integrity protection. Therefore, four 8 byte keys have to be generated. First, both protocol participants XOR the key derivation data of reader and card resulting in a value SK (session key). Then, two 32 bit counters are appended to SK, resulting in SK1 and SK2. The value of the first counter is 1, the value of the second is 2. Each protocol participant then calculates hash values of SK1 and SK2. CWA 14890-1 stipulates SHA-1 as hash algorithm. The first 8 bytes of SHA-1(SK1) are used as the first encryption key; the second 8 bytes are used for the second encryption key, while the last four bytes of the hash value are not used. The value of SHA-1(SK2) is used similarly. The first 16 bytes are used for both integrity keys and the last four bytes are not used. These keys are stored in the internal state of both communication partners. From now on, these keys can be used to secure further communications.

5.2 HLPSL Specifications

HLPSL specifications of Asymmetric Session Key Transport Protocol have been omitted for lack of space. Our intended target is to model the protocol as close as possible to the protocol description given in 5.1. The protocol behavior could be modeled, except issues / limitations as described in section 4.2. The detailed HLPSL model can be found in [18].

5.3 Modeling Secrecy in HLPSL

Secrecy of a message means that the specified set of agents can see the message. HLPSL supports secrecy goal with the help of predefined predicate `secret`. In key transport protocol and in symmetric authentication protocol (see section 6), both roles i.e., `ifd` and `icc` creates session key (`SK`), and so we augment transitions of both roles in both protocols, with the following `secret` facts where the primes is required there to refer to the new values of `SK'`:

```
secret(SK',sec_ifd_session_key,{Reader,Card}
secret(SK',sec_icc_session_key,{Card,Reader}
```

The labels `sec_ifd_session_key` and `sec_icc_session_key` are of type `protocol_id`. They must be declared in the `const` section of the `environment` role. In the case of secrecy facts, *protocol ids* serve merely to distinguish different secrecy goals.

5.4 Security Analysis

For security analysis our methodology for experiments 1-10 were the same as we did in section 4.9. **For all analyzed scenario, no attacks were found on the key transport protocol in the presence of DY model.** In all experiments, the security properties under test are the correspondence properties explained in section 4.6 and secrecy properties explained above in section 5.3 for this protocol.

6 Specification and Analysis of Symmetric Authentication Scheme

The symmetrical authentication protocol may be used to construct a *secure channel* between an application and a signature creation device providing either only integrity or both integrity and confidentiality. The long term shared secret keys i.e., `Kenc` (encryption key) and `Kmac` (message authentication code/integrity key) are supposed to be already present in the IFD (reader) and ICC (smart card) and are used in authentication protocol. The long term shared secret keys are replaced as soon as a fresh session keys are available.

6.1 Authentication Steps

Step 1. The IFD requires the ICC's identity serial number (`SN_ICC`) for the mutual authentication token. The card reader starts the protocol with the help of `Get Data Command` by fetching the serial number (`SN_ICC`) of the smart card and stores it for later use.

`Get Data Command`

$$\textbf{Card Reader} \longrightarrow \textbf{Card}$$

`SN_ICC`

$$\textbf{Card Reader} \longleftarrow \textbf{Card}$$

Step 2. Then, the reader requests a 64 bits (8 bytes) random number (`RND_ICC`) from the smart card with a `Get Challenge` command in order to prove its authenticity dynamically. Upon receiving `Get Challenge`, the smart card generates a 64 bits random number used as nonce. The smart card stores random number in its internal memory and replies with random number to the reader.

`Get Challenge Command`

$$\textbf{Card Reader} \longrightarrow \textbf{Card}$$

`RND_ICC`

$$\textbf{Card Reader} \longleftarrow \textbf{Card}$$

Step 3. Once the reader receives random number of the smart card, it generates its own 64 bits random number. Further 256 random bits are selected for use as key derivation data. The reader stores key derivation data in its internal memory. Next, the reader generates the command data for a `Mutual Authenticate` command. The `Mutual Authenticate` command authenticates ICC and IFD in one command. It concatenates random and serial numbers of reader with random and serial numbers of card and key derivation data of reader, and encrypts the resulting string under the encryption key selected in the initial `Manage Security Environment` command. Last, the reader generates a MAC of the encrypted data using the selected integrity key. Then, the reader sends the `Mutual Authentication` command with the command data just generated to the smart card.

```
Mutual Authenticate Command
E[Kenc](S) || MAC[Kmac]( E[Kenc](S) )
where S = RND_IFD || SN_IFD || RND_ICC || SN_ICC || KIFD
```

$$\textbf{Card Reader} \longrightarrow \textbf{Card}$$

Upon receiving the `Mutual Authentication` command, the smart card first checks the integrity of the message. If it can confirm command data integrity, the command data are decrypted. Next, the smart card checks whether the second random number has the same value as the random number stored in the card's internal memory, and whether the second serial number equals its own serial number. If these two tests are successful, the card stores the key derivation data in its internal memory. Now, the smart card selects its own 256 bits key derivation data and stores it in its internal memory, concatenates random and serial numbers of card with random and serial numbers of reader and key derivation data of the card, encrypts the concatenated data using the previously selected encryption key, and calculates a MAC over the encrypted data using the integrity key. The smart card then sends the encrypted data and MAC back to the reader. After verifying the MAC value, the reader decrypts the response and checks whether the second random number equals the one stored in its internal memory and whether the second serial number matches its own.

```
        E[Kenc](R) || MAC[Kmac](E[Kenc](R))
R = RND_ICC || SN_ICC ||RND_IFD || SN_IFD || KICC
```

$$\textbf{Card Reader} \longleftarrow\!\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\!- \textbf{Card}$$

Once smart card and reader have stored both their own and received key derivation data, they can generate the session keys. The procedure to generate session keys is same as mentioned in section 5.1.

6.2 HLPSL Specifications

HLPSL specifications of symmetric authentication protocol have been omitted for lack of space. Our intended target is to model the protocol as close as possible to the protocol description given in section 6.1. The protocol steps could be modeled, except next issue / limitation. The detailed HLPSL model can be found in [18].

Restriction Applied
During the generation of session keys symmetric authentication protocol make use of bit-level selection of data [5]. This could not be modeled in HLPSL specifications while all the protocol steps are modeled in HLPSL.

6.3 Automatic Analysis of the Protocol

For security analysis our methodology for experiments 1-10 were the same as we did in section 4.9. In all experiments, the security properties under test are the correspondence properties explained in section 4.6. **For all analyzed scenario, no attacks were found on the protocol in the presence of DY model.**

[5] The first 64 bits of SHA-1(SK1) are used as the first encryption key, the second 64 bits are used for the second encryption key, while the last 32 bits of the hash value are not used. The value of SHA-1(SK2) is used similarly. The first 128 bits are used for both integrity keys and the last 32 bits are not used.

7 Conclusion

In this paper device authentication protocols mentioned in CWA 14890-1, have been presented and analyzed. We model core security properties as correspondence properties and use the AVISPA tool to automate our security analysis. We have found that device authentication protocols mentioned in CWA 14890-1 are safe w.r.t. given finite number of sessions. Since we have carefully reviewed our formalizations to validate that they faithfully describe the protocols mentioned in CWA 14890-1, and since the tool used is mature enough, we can be confident that in the device authentication protocols there are no design flaws that can lead to attacks on *authentication* and *secrecy*. Of course, vulnerabilities at the cryptographic or implementation level cannot be excluded with this approach. In summary, we found our analysis has provided a greater degree of confidence in the correctness of device authentication protocols mentioned in CWA 14890-1.

Acknowledgments. This work was supported by Master Scholarship grant from Higher Education Commission (HEC), Pakistan and DAAD, Germany. We would like to thank Dieter Gollmann, Sebastien Canard, Jan Meier, Helmut Scherzer and the anonymous reviewers for helpful feedback and suggestions for improvements.

References

1. CEN Workshop Agreement, ftp://ftp.cenorm.be/PUBLIC/CWAs/e-Europe/eSign/cwa14890-01-2004-Mar.pdf
2. Signature Creation Smart Cards, http://www.security-technologynews.com/article/signature-creation-smart-cards.html
3. ISO/IEC 7816-4 Identification cards, Integrated circuit cards Part 4: Organization, security and commands for interchange (November 2004), http://www.ttfn.net/techno/smartcards/iso7816_4.html
4. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Hankes Drielsma, P., Heám, P.C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., Vigneron, L.: The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)
5. AVISPA: Automated Validation of Internet Security Protocols and Applications. FET Open Project IST-2001-39252 (2003), http://www.avispa-project.org/
6. A Security Protocol ANimator for AVISPA, http://www.irisa.fr/celtique/genet/span/
7. Deliverable D2.1: The High Level Protocol Specification Language, http://www.avispa-project.org/delivs/2.1/d2-1.pdf
8. Deliverable D2.3: The Intermediate Format, http://www.avispa-project.org/delivs/2.3/d2-3.pdf
9. Basin, D., Mödersheim, S., Vigano, L.: OFMC: A Symbolic Model-Checker for Security Protocols. International Journal of Information Security (2004)
10. Turuani, M.: The CL-Atse Protocol Analyser. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 277–286. Springer, Heidelberg (2006)

11. Armando, A., Compagna, L.: SATMC: A SAT-Based Model Checker for Security Protocols. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 730–733. Springer, Heidelberg (2004)
12. Boichut, Y., Heam, P.-C., Kouchnarenko, O., Oehl, F.: Improvements on the Genet and Klay Technique to Automatically Verify Security Protocols. In: Proc. of Automated Verification of Infinite States Systems (AVIS 2004). ENTCS, pp. 1–11 (2004)
13. Dolev, D., Yao, A.: On the Security of Public-Key Protocols. IEEE Transactions on Information Theory 2(29) (1983)
14. Rusinowitch, M., Turuani, M.: Protocol Insecurity with Finite Number of Sessions is NP-complete. In: 14th IEEE Computer Security Foundations Workshop, pp. 174–187. IEEE Computer Society (2001)
15. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory IT-22(6), 644–654 (1976)
16. Lowe, G.: A Hierarchy of Authentication Specifications. In: Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW 1997), pp. 31–43. IEEE Computer Society Press (1997)
17. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16(3), 872–923 (1994)
18. Javed. A. HLPSL Models of CWA 14890-1 Protocols, http://ashar-javed.blogspot.com/2011/05/formal-analysis-of-cwa-14890-1-protocol.html

Author Index