UCL
Université
catholique
de Louvain

École polytechnique de Louvain
ICTEAM Institute

Towards Fair Evaluation and Intellectual Property Protection of Cryptographic Implementations

Stéphanie Kerckhof

Thèse soutenue en vue de l'obtention du grade de
docteur en sciences de l'ingénieur

Membres du jury:

Prof. François-Xavier Standaert	UCL, Belgique - Promoteur
Prof. Olivier Pereira	UCL, Belgique - Secrétaire
Dr. Gael Rouvroy	intoPIX s.a., Belgique
Prof. Lilian Bossuet	Université de Lyon, France
Prof. Tim Güneysu	Ruhr Universität Bochum, Allemagne
Prof. Christophe De Vleeschouwer	UCL, Belgium - Président

Louvain-la-Neuve, Belgique
June 2014

ACKNOWLEDGMENTS

First of all, I would like to thank Prof. François-Xavier Standaert for being such a good supervisor. His door was always open for questions and his guidance helped me going through this thesis. Thanks a lot for continuously believing in me and for pushing me forward in periods of doubts (and also for being so sympathetic regarding my emotional crisis).

I would also like to thank Prof. Olivier Pereira and Dr. Gaël Rouvroy for their advices as part of my guidance committee. I am also grateful to Lilian Bossuet and Tim Güneysu for having accepted to be members of the jury, and to Christophe De Vleeschouwer for having been the president of the jury. Thanks to all of them for the interesting questions and ideas they raised during the private defense.

I also really enjoyed being part of the Crypto Group. This gave me the opportunity to work (and make friend) with very nice people. I therefore would like to express my gratitude to all my current and former colleagues with whom I have enjoyed lunch times, coffee breaks and formal as well as very informal discussions. In particular, thanks a lot to Sylvie Baudine for helping us all year round with our administrative tasks and for her patient proofreading of this text. Similarly, I am grateful to the ELEN administrative and technical staff for their continuous help: Anne Adant, Brigitte Dupont, Isabelle Dargent, Viviane Sauvage and Pascal Simon.

Next, my thanks go to my co-authors for their valuable help on our published papers. More specifically, I would like to thank François Durvaux who has been the principal co-author on most of my publications. I really appreciated working with you. I would also like to deeply thank my office mates Marcel Medwed (the best office mate ever) and more recently Antony Journault. Thanks a lot for your support, for making me laugh when I was sad and for cheering me up in all times.

I would also like to express my thanks to the intoPIX R&D team for the great time I spent with them before I started my thesis. I learned a lot thanks to you. More specifically, thank you Pascal Pellegrin for your availability and for your answers to all my hardware questions.

I am grateful to the Fonds National de la Recherche Scientifique of Belgium for funding this research.

To conclude, I would like to thank my family for their moral support during these four (sometimes difficult) years. Thank you Jean-Luc, Martine, Cindy, Maureen and Quentin for always being there for me. I am also very grateful

to have such amazing friends! Thanks to all of you for taking my mind off of my work in so many ways when it is needed.

ABSTRACT

With the emergence of the internet of things, the research surrounding the design of cryptographic algorithms suited for constrained environments has rapidly gained ground. This fast development of new security primitives and their implementation on electronic devices has led to consider several concerns in terms of fair evaluation, physical security and protection of their *intellectual property* (IP). These can be summarized with the following questions:

1. How can we fairly compare the performances of algorithms? How do lightweight algorithms compare to the standard ones? What is the impact of design choices on the implementation results?

2. To what extent do countermeasures against side-channel attacks improve security and at which cost? What is the impact of new technologies on the physical security and efficiency of cryptographic implementations?

3. How to protect IPs in a way that is both flexible and secure?

To answer these questions, we first propose different evaluation frameworks which allow us to compare the efficiency of cryptographic algorithms. From the obtained results, we extract some general design guidelines and show that normalized metrics (such as energy) are generally more illustrative. We then come up with more efficient implementations of the shuffling countermeasure which is an important solution to improve security against side-channel attacks. We also present the first implementation of the randomized look up table countermeasure. Finally, we propose a new flexible IP protection infrastructure exploiting the implementation power consumption as a signature.

CONTENTS

ACRONYMS

AES	Advanced encryption standard
ALM	Adaptive logic module
ALU	Arithmetic logic unit
ASIC	Application specific integrated circuit
CLB	Configurable logic block
CMOS	Complementary metal oxide semiconductor
DES	Data encryption standard
DPA	Differential power analysis
DRAM	Dynamic random access memory
DSP	Digital signal processing
EEPROM	Electrically erasable programmable random access memory
FFT	Fast fourier transform
FPGA	Field-programmable gate array
FRAM	Ferroelectric random access memory
HDL	Hardware description language
I/O	Input/output
IC	Integrated circuit
IoT	Internet of things
IP	Intellectual property
IV	Initial value
LAB	Logic array block
LE	Logic element
LFSR	Linear feedback shift register
LSB	Least significant bit
LUT	Look up table

MDS	Maximum distance separable
MOSFET	Metal oxide semiconductor field effect transistors
MSB	Most significant bit
NBS	US National Bureau of Standards
NIST	National institute of standards and technology
NSA	United States national security agency
POI	Point of interest
PUF	Physically unclonable function
RAM	Random access memory
RFID	Radio frequency identification
RISC	Reduced instruction set computing
RLUT	Randomized look up table
ROM	Read only memory
RP	Random permutation
RSI	Random start index
RTL	Register transfer level
S-Box	Substitution box
SHA	Secure hash algorithm
SNR	Signal to noise ratio
SPA	Simple power analysis
SPH	Soft physical hash function
SPN	Subtitution-permutation-network
SRAM	Static random access memory
VHDL	VHSIC hardware description language
VHSIC	Very high speed integrated circuit
XOR	Exclusive OR

AUTHOR'S PUBLICATION LIST

[1] Stéphanie Kerckhof, Baudoin Collard, François-Xavier Standaert. FPGA implementation of a statistical saturation attack against PRESENT. In Abderrahmane Nitaj and David Pointcheval, editors, *AFRICACRYPT*, volume 6737 of *Lecture Notes in Computer Science*, pages 100-116. Springer, 2011.

[2] Stéphanie Kerckhof, François Durvaux, Nicolas Veyrat-Charvillon, Francesco Regazzoni, Guerric Meurice de Dormale, François-Xavier Standaert. Compact FPGA implementations of the five SHA-3 finalists. In Emmanuel Prouff, editor, *CARDIS*, volume 7079 of *Lecture Notes in Computer Science*, pages 217-233. Springer, 2011.

[3] François Durvaux, Benoît Gérard, Stéphanie Kerckhof, François Koeune, François-Xavier Standaert. Intellectual property protection for integrated systems using soft physical hash functions. In Dong Hoon Lee and Moti Yung, editors, *WISA*, volume 7690 of *Lecture Notes in Computer Science*, pages 208-225. Springer, 2012.

[4] Stéphanie Kerckhof, François Durvaux, Cédric Hocquet, David Bol, François-Xavier Standaert. Towards green cryptography: A comparison of lightweight ciphers from the energy viewpoint. In Prouff and Schaumont, editors, *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 390-407. Springer, 2012.

[5] J. Balasch, B. Ege, T. Eisenbarth, B. Gérard, Z. Gong, T. Güneysu, S. Heyse, S. Kerckhof, F. Koeune, T. Plos, T. Pöppelmann, F. Regazzoni, F.-X. Standaert, G. Van Assche, R. Van Keer, L. van Oldeneel tot Oldenzeel, I. von Maurich. Compact implementation and performance evaluation of hash functions in ATtiny devices. In Stefan Mangard, editor, *CARDIS*, volume 7771 of *Lecture Notes in Computer Science*, pages 158-172. Springer, 2012.

[6] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, François-Xavier Standaert. Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note. ASIACRYPT 2012. 740-757

[7] T. Eisenbarth, Z. Gong, T. Güneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Regazzoni, F.-X. Standaert, L. van Oldeneel tot Oldenzeel. Compact implementation and performance evaluation of block ciphers in ATtiny devices. In Aikaterini Mitrokotsa and Serge Vaudenay, editors, *AFRICACRYPT*, volume 7374 of *Lecture Notes in Computer Science*, pages 172-187. Springer, 2012.

[8] Stéphanie Kerckhof, François Durvaux, François-Xavier Standaert, Benoît Gérard. Intellectual property protection for FPGA designs with soft physical hash functions: First experimental results. In *HOST*, pages 7-12. *IEEE*, 2013.

[9] Stéphanie Kerckhof, François-Xavier Standaert, Eric Peeters. From New Technologies to New Solutions: Exploiting FRAM Memories to Enhance Physical Security. In Aurélien Francillon and Pankaj Rohatgi, editors, *CARDIS*, *Lecture Notes in Computer Science*. Springer, 2013. To appear.

[10] Ludovic-Henry Gustin, François Durvaux, Stéphanie Kerckhof, François-Xavier Standaert, Michel Verleysen. Support vector machines for improved IP detection with soft physical hash functions. In Emmanuel Prouff, editor, *COSADE*, *Lecture Notes in Computer Science*. Springer, 2014. To appear.

[11] François Durvaux, Stéphanie Kerckhof, Francesco Regazzoni, François-Xavier Standaert. A survey of recent results in FPGA security and intellectual property protection. In Konstantinos Markantonakis and Keith Mayes, editors, *Secure Smart Embedded Devices, Platforms and Applications*. Pages 201-224. Springer, 2014

[12] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, Kerem Varici, François Durvaux, Lubos Gaspar, Stéphanie Kerckhof. SCREAM & iSCREAM: Side-channel resistant authenticated encryption with masking. Submission to the CAESAR competition.

INTRODUCTION

I.1 SCOPE AND MOTIVATION

Over the last 50 years, the constant decrease in size and cost of electronic devices, known as the technology scaling trend, has enabled their integration inside everyday applications. A new concept, named the *internet of things* (IoT) has consequently rapidly gained ground in the scenario of modern wireless telecommunications. The basic idea behind IoT is the pervasive presence around us of a variety of things or objects – such as *radio frequency identification* (RFID) tags, sensors, actuators, mobile phones, etc. – which are able, through unique addressing schemes, to interact with each other and cooperate with their neighbors to reach common goals [10]. Since sensitive information may transit from one node of the network composing the IoT to the other, this model raises privacy and security concerns. This is why cryptographic algorithms are often an integral part of the IoT's nodes.

In this context, the need for small size, weight, energy and cost implied by the IoT scenario has promoted the research in the direction of lightweight cryptographic algorithms. In general, symmetric cryptographic primitives, such as block ciphers and hash functions are good candidates for low-cost implementations and many new designs going in that direction have been proposed over the last decade (e.g. ICEBERG, LED, PRESENT, PRINCE, etc. for block ciphers and Quark, SPONGENT, PHOTON, etc. for hash functions). Even if these algorithms are useful and inventive in many ways, determining which one to use in a particular context with good confidence may be difficult. Furthermore, the notion of low-cost can be misleading as different authors may have different definitions of low-cost in mind. For example, does a low-cost implementation correspond to a small implementation size, a low power consumption, a low latency or a low energy consumption? Besides, an algorithm may have low-cost properties for a particular implementation technology but become more costly when implemented on an other type of technology. Finally, no general guidelines helping the design of lightweight cryptographic algorithms are available. For all those reasons fair evaluation of cryptographic implementations is an important research topic.

Next to these concerns, while cryptographic algorithms are usually designed to be theoretically secure under some assumptions regarding the adversary's resources, the electronic devices on which they are implemented may not be secure. This leads to new issues regarding the practical security of the primi-

tives. Indeed, an adversary may have access to the physical properties of the electronic device – such as electromagnetic emanations, power consumption and temperature dissipation – as well as to ways to alter the behavior of the cryptosystem and thereby compromise the secret information that is manipulated by the device. Fault attacks are an example of physical attacks for which the cryptosystem behavior is altered by provoking faults (e.g. under the form of a clock glitch, a power spike or a bit flip using a laser) at specific points of time. Another type of attack consists in removing the electronic device package and directly probing the relevant wires in order to read the secret information. Next to that, non-invasive attacks have also been proposed in the literature. Those attacks rely on the fact that electronic devices produce physical leakages that are related to the intermediate values they are manipulating.

In order to prevent those attacks, specific countermeasures were developed. More specifically, looking at the countermeasures proposed for side-channel attacks, we mainly find two categories of countermeasures : the countermeasures hiding the sensitive information (e.g. addition of random delays, shuffling of the order of independent operations, etc.) and the masking countermeasures (i.e. the ones that conceal each intermediate value processed by the algorithm using random values called the masks). Even if these countermeasures are useful against side-channel attacks, they all require additional resources and execution time. This raises thus an important question: to what extend do the countermeasures improve security and at which cost? Besides, some recent countermeasures could highly benefit from secure pre-computations. This need is highly related to the problem of fast and efficient non-volatile storage. The question is then: are there existing technologies allowing efficient secure pre-computations?

A last important issue related to electronic devices, and more specifically to the algorithms implemented inside them, is the protection of their *intellectual property* (IP) against theft or illegal copies. Indeed, the increasing cost and complexity of electronic device manufacturing processes as well as the increasing complexity of the current hardware/software codesigns has motivated the designers to opt for a new business model: software and hardware implementations are now often sold as IP cores that can be integrated as part of a larger system by a third-party. This re-use of IP cores presents enormous gains in development cost and time. Unauthorized exploitation is however easier to achieve with IP cores than with a completely manufactured system. Various solutions have thus been introduced to mitigate those risks. But as usual in security-related issues, the protection mechanisms have to deal with contradictory goals. On the one hand, IP owners primarily aim at preventing the piracy of their designs. On the other hand, users want to easily integrate these designs in their development flows. The first goal suggests integrating security mechanisms at low abstraction levels (e.g. in circuit layouts). The second goal rather suggests integrating these mechanisms at high abstraction levels (e.g. in the description codes of the designs). In other words, solutions

to prevent the counterfeiting of electronic systems can be seen as a trade-off between security and flexibility.

In this thesis, we will investigate those three topics and try to find suitable answers to the various questions they are raising. In particular, we first propose a general evaluation of different cryptographic algorithms on several electronic devices. This allows us to give some general intuitions about good practices to follow while designing new algorithms. Besides, we are also able to evaluate the point up to which lightweight algorithms are relevant with respect to standard ones.

Secondly, we come up with two new implementations of the shuffling countermeasure that improve previously existing propositions. From these implementations, we are able to propose the first comprehensive evaluation of the shuffling countermeasure. In addition to that, we investigate to which extent a newly available memory technology (*ferroelectric random access memory* (FRAM)) can improve the performances and security of protected implementations. Specifically, we first consider the implementation of the shuffling countermeasure with this memory. Then, we present the first working implementation of the *randomized look up table* (RLUT) countermeasure [179] which can be viewed as a generic masking scheme that provides unconditional security against side-channel attacks of all orders. This implementation helps us to demonstrate that using this kind of countermeasure combined with FRAM allows higher security levels than higher-order masking but at similar cost.

Finally, we propose a new IP protection mechanism that applies to a large variety of IPs. This mechanism takes advantage of the idea of *soft physical hash function* (SPH) and exploits the physical features of the IP to protect. However, unlike for watermarking-based approaches, our mechanism is able to detect counterfeited designs without any addition to the original implementation. In this thesis, we provide a general framework for our approach as well as the definitions and properties related to it. We eventually experiment our IP protection mechanism on some case studies and show some promising results.

I.2 THESIS OUTLINE

Part I. We start this work by providing the background information needed to better apprehend this thesis. This part is composed of the following two chapters:

Chapter 1. As a preliminary discussion, we describe the different cryptographic algorithms that we will be using in the remainder of this work. We first provide general considerations regarding the construction and properties of block ciphers and hash functions. For these two categories of algorithms, we then explain the construction of the previous and actual standards in the field. We finally give further details concerning other block ciphers and hash functions that will be used throughout this thesis.

Chapter 2. In order for these algorithms to be a part of real life applications, they need to be implemented on some electronic devices. Therefore, we describe in this chapter two main categories of devices: the hardware devices and the devices containing embedded software (which we will often be calling software devices). For each of these categories of electronic devices, we introduce the basic concepts related to them as well as their structures. We then provide information regarding their design flows. Finally, we introduce some metrics and explain their importance in implementation evaluations.

Part II. From the information presented in Part I, we may now start implementing cryptographic algorithms. In the next chapters, we first provide general comparison for block ciphers and hash functions on software and hardware devices. We then evaluate the security and efficiency of two software countermeasures.

Chapter 3. Our first set of implementations concerns the block ciphers. In this chapter, we first investigate the software implementations of 12 block ciphers for a constrained device (8-bit microcontroller with reduced set of instructions). Then, we propose to evaluate the ASIC implementations of 6 block ciphers. As each of these works has been conducted in contribution with other authors, we provided common methodologies for each of them. The results related to software evaluation have been presented at AFRICACRYPT 2012 [62], while the hardware ones were proposed at CHES 2012 [103].

Chapter 4. We extend the evaluation work presented in Chapter 3 for block ciphers to the case of hash functions. Again, we provide results in two implementation scenarios (i.e. implementations for software and hardware devices). Regarding the software implementations, we adapted the methodology used for block ciphers and kept the same device. The hardware implementations, on their side, were carried out on FPGAs instead of ASICs. These two evaluations are again joint works involving several authors. The first work was presented at CARDIS 2012 [16], while the second one was proposed in the context of the SHA-3 competition and presented at [105].

Chapter 5. In this chapter, we tackle the concerns related to the practical security of cryptographic implementations. First, we propose two new implementation approaches for the shuffling countermeasure. Then we describe an evaluation framework adapted to the context of shuffling in which we propose the use of the permutation leakage. From that framework, we are then able to provide a security evaluation of the countermeasure. In a second part, we analyze the impact of FRAM memories on the efficiency of existing countermeasures. To do so, we propose software implementations of the shuffling and the RLUT countermeasures. These works were proposed in collaboration with other authors to the conferences ASIACRYPT 2012 [192] and CARDIS 2103 [106].

Part III. In this part, we finally consider the IP protection problem.

Chapter 6. In this chapter, we provide a solution against unlicensed *intellectual property* (IP) usage that uses both the leakage of the device in which the IP is embedded and the concept of SPH. We provide general definitions for our approach and give some experimental results for software and hardware IPs. The work included in this chapter was presented at WISA 2012 [60] and HOST 2013 [104].

Conclusion and Perspectives. To conclude, we provide a summary of the contributions proposed in this work. We also give some potential future research subject for each of the investigated topics. Finally, we briefly discuss the problematic of automation of the evaluation process versus the confidence and optimality of the results.

PART I

PRELIMINARIES

CHAPTER 1

CRYPTOGRAPHIC ALGORITHMS

Cryptography covers a wide range of applications such as secret communication, authentication and digital signatures and can be defined as *the scientific study of techniques for securing digital information, transactions, and distributed computations* [102]. The cryptography field of research is often divided into two categories: the cryptographic algorithms and the cryptographic protocols. Cryptographic algorithms can be viewed as the building blocks necessary to implement the cryptographic protocols.

In this thesis, we will only focus on the cryptographic algorithms which may further be divided according to the way their key is used. First, the *keyless algorithms* encompass all the cryptographic algorithms that do not need a key to be executed. Then, the *symmetric-key algorithms* are algorithms for which the sender and receiver of a secret message share the same key. Lastly, in the case of asymmetric-key algorithms or *public-key algorithms*, the receiver of a secret message needs two different keys: a public key, that can be shared with everyone and with which the message will be encrypted, and a matching secret key, only known by the receiver, and with which the secret message can be decrypted.

In the next chapters, we concentrate mainly on lightweight embedded applications that require low energy or power consumption. We will consequently focus our discussion on two types of cryptographic algorithms that are usually well suited in this case: the block ciphers which are part of the symmetric-key algorithms, and the hash functions for the keyless algorithms.

1.1 BLOCK CIPHERS

1.1.1 General description

A block cipher is a symmetric-key algorithm that operates on fixed length blocks of data. During the encryption, the message or *plaintext* p is transformed, using a secret *key* k, in a *ciphertext* c of the same length. An inverse

function using the same key as the encryption decrypts the ciphertext. Those processes are illustrated in Figure 1.1 and may be denoted by $c = \texttt{ENC}_k(p)$ for encryption and $p = \texttt{DEC}_k(c)$ for decryption.

Fig. 1.1.: Illustration of the encryption/decryption process

A block cipher can be seen as a family of pseudo-random permutations, each key k defining one permutation inside the family. Standard notions of security state that a permutation is pseudo-random if it is indistinguishable from a random permutation. That is, if we randomly chose between the output of a random permutation and the output of a block cipher (generated using a randomly chosen key), an adversary should not be able to guess with a probability much more than $1/2$ which of the permutations was used to generate the output he has access to [102]. Besides, Shannon stated in [172] that "in a strongly ideal cipher, all statistics of the cryptogram are independent of the particular key used". In order to tend to this concept of ideal cipher, he proposed two methods called the *confusion* and the *diffusion* methods. The purpose of the confusion method is to make the relation between the key and the ciphertext more complex. While the goal of the diffusion method is to hide all possible redundancy existing in the plaintexts. In other words, a change of one bit in the plaintext should result in average in the modification of half of the bits in the ciphertext.

As illustrated on Figure 1.2, block ciphers are often composed of simple functions, called *rounds*, which are iterated several times. The most common type of existing block ciphers are *subtitution-permutation-networks* (SPN) for which the rounds are composed as follows:

- *The key-mixing layer* combines a round key to the actual state. Usually, an XOR operation is used to achieve this. Round keys are derived from the master key using a *key scheduling* algorithm. Those round keys may either be precomputed and stored in memory or computed on-the-fly alongside the message encryption. The choice of one or the other alternative will be guided by the design requirements as one option requires less time to execute but more memory than the other.

- *The non-linear substitution layer* is used to add confusion to the cipher. This substitution is generally achieved by *substitution boxes* also called S-Boxes in the remainder of the text. An S-Box is an a-bit to b-bit non linear transformation that maps every possible a-bit value to a b-bit value. Typically, S-Boxes are stored in memory under the form of *look up*

tables (LUTs). In order to limit memory usage, the input and output sizes are usually limited to a few bits and the substitution layer is composed of several S-Boxes applied in parallel.

- The diffusion layer or *permutation layer* can be implemented using either simple permutations, that can be applied on different level such as the bit, nibble or byte level, or *maximum distance separable* (MDS) codes. In the case of simple permutations, the diffusion layer is usually resource free when applied in hardware as it only consists in wire crossing. It can however become extremely time consuming in software when applied on the bit level.

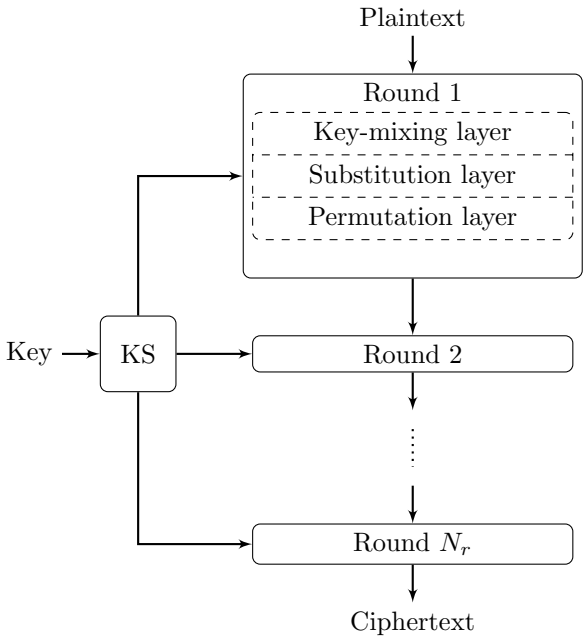

Fig. 1.2.: Illustration of the different parts of a block cipher

In the next sections, we describe existing block ciphers that will be used throughout this thesis. In section 1.1.2, we concentrate on the details of two standardized block ciphers. Then, section 1.1.3 gives an overview of several block ciphers designed with a low cost goal in mind.

1.1.2 Standard block ciphers

Data encryption has been used for ages. It is however only in 1973 that the US National Bureau of Standards (NBS) (which is now called the National

Institute of Standards and Technology or NIST) launched its first call for candidates suitable for protecting confidential governmental information. The *data encryption standard* (DES) was first published in 1975 and was accepted as a standard in 1976 [141]. Over the years, with the increase of the available computing power, exhaustive key search on DES became possible, which led the NIST to issue a call for proposals to replace DES. In 2001, the block cipher named Rijndael, proposed by Joan Daemen and Vincent Rijmen, became the *advanced encryption standard* (AES) [52].

Data encryption standard

DES is based on a block cipher construction called a *Feistel network*. In a Feistel network, the round inputs X are separated in two halves, $X_i = (L_i||R_i)$ (with $||$ the concatenation operator, L representing the left part and R the right part). The right half is passed through a round function f and then XORed with L_i. The two parts are then swapped and the input to the next round is $X_{i+1} = (L_{i+1}||R_{i+1}) = (R_i||(L_i \oplus f(R_i)))$. The round function f may contain the same kind of building blocks as SPN rounds (S-Boxes, permutations, ...). However, unlike SPN, Feistel Networks allow the use of non invertible round functions.

DES is a 16-round Feistel network that uses 56-bit keys and operates on 64-bit blocks. It is depicted in Figure 1.3. The key schedule is simply composed of permutations and generates 48-bit subkeys. Since the round function only works on half of the block size at a time, the input and output lengths of DES round function is 32 bits. The input of f is first expanded to 48 bits before being XORed with the 48-bit subkey. Next, the state is passed through S-Boxes that take inputs of 6 bits and produce outputs of 4 bits. The last building block of f is a bit-wise permutation. Bit-wise permutations on the complete state are also present at the beginning and the end of the encryption process (`IP` for initial permutation and $\texttt{IP}^{-1}$ which is the inverse of `IP`).

Thanks to the Feistel network structure, the decryption algorithm is the same as the encryption one with the subkeys taken in reverse order.

Advanced encryption standard

AES is an SPN operating on blocks of 128 bits and for which the key size can either be 128, 192 or 256 bits. It is composed of n rounds (n = 10, 12 or 14 for AES-128, AES-192 or AES-256 respectively). Each round is composed of four transformations: SubBytes, ShiftRows, MixColumns, and finally AddRoundKey as depicted in Figure 1.4. All those transformations operate on bytes and the AES state is often defined as 4 by 4 array of bytes. SubBytes substitutes each byte of the state array by another one using an 8-bit S-Box. In ShiftRows, the bytes in each row are circularly shifted by a given number of bytes. More precisely, the bytes of *ith* row are shifted by i positions for $0 \leq i \leq 3$. MixColumns constitutes, with ShiftRows, the linear diffusion layer. The bytes inside each column of the state are mixed together using a multiplication on polynomial

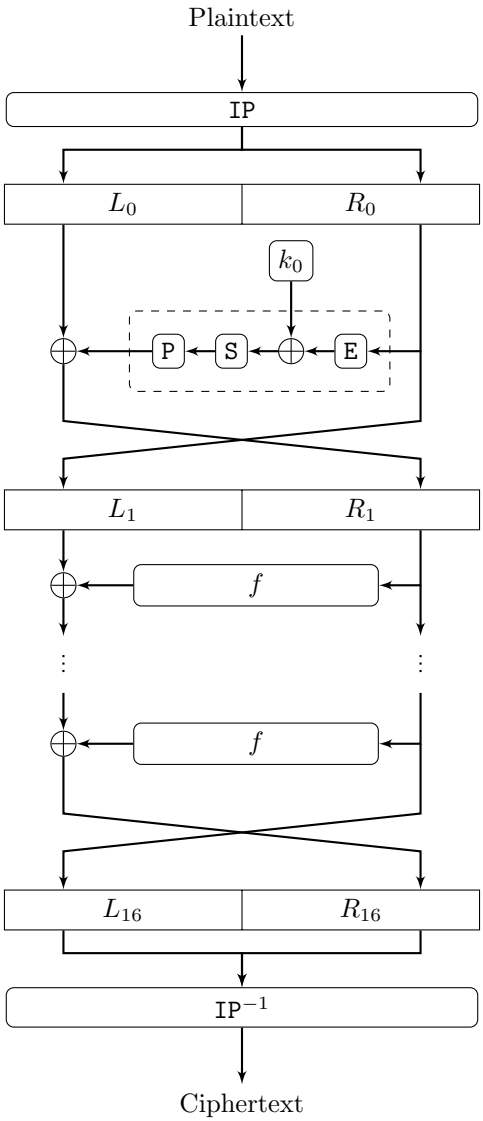

Fig. 1.3.: Illustration of the DES block cipher

having coefficients in $GF(2^8)$. The last operation, AddRoundKey, combines each byte of the state array with the corresponding byte of the round key array using an XOR operation.

In the AES key schedule, the first round key is the masterkey itself. Each following round key is then derived from the previous one using 32-bit words shifts and XOR operations. One of the 32-bit words composing the subkey is also transformed using an 8-bit S-Box, an 8-bit XOR and byte permutations.

Unlike DES, AES is not involutional (i.e. self-inverse). The decryption algorithm will thus not be the same as the encryption one. Each of the layers present in the encryption process will however stay the same but with different parameters (e.g. a different S-Box for the substitution layer). It is also worth noticing that, in the case of on-the-fly generation of the subkeys, the complete encryption key schedule will have to be performed in order to get the first decryption round key.

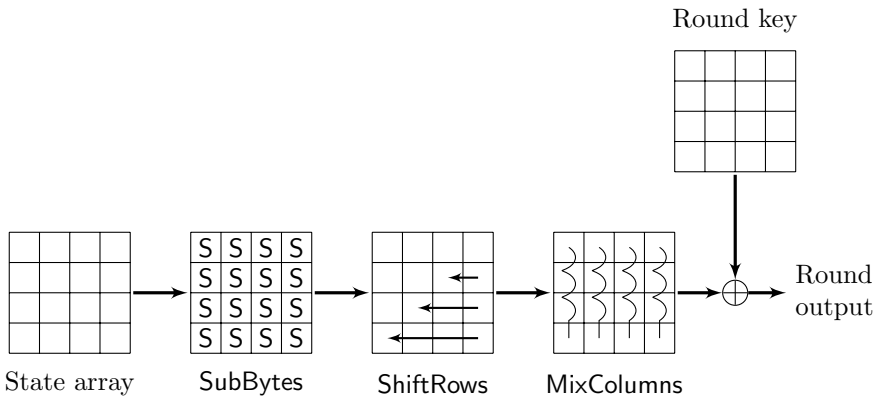

Fig. 1.4.: Illustration of one round of AES

Along with the versions which became AES-128, AES-192 and AES-256, other variants of the *Rijndael* cipher were proposed [51]. Among those variants, Rijndael-256/256 will be dealt with in later sections. This version processes 256-bit key and state. Rijndael-256/256 encryption iterates a round function 14 times. This round is composed of the same four transformations as the AES.

1.1.3 Lightweight block ciphers

Small embedded devices (including smart cards, RFIDs, sensor nodes) are now deployed in many applications. They are usually characterized by strong cost constraints. Yet, as they may manipulate sensitive data, they also require cryptographic protection. As a result, many lightweight ciphers have been

proposed in order to allow strong security guarantees at a lower cost than standard solutions. Here are succinct descriptions of all the lightweight block ciphers, ordered by date of introduction, that will be addressed in this thesis.

IDEA, 1990 [112] is a patented cipher whose patent expired in May 2011 (in all countries with a 20 year term of patent filing). Its underlying Lai-Massey construction does not involve an S-Box or a permutation network such as in other Feistel or common SPN ciphers. Instead, it interleaves mathematical operations from three different groups to establish security, such as addition modulo 2^{16}, multiplication modulo $2^{16}+1$ and addition in $\mathrm{GF}(2^{16})$ (XOR). IDEA has a 128-bit key and 64-bit inputs and outputs. A major drawback of its construction is the complex key schedule involved that also requires the extended Euclidean algorithm for the decryption operation. For efficient implementation, this complex key schedule needs to be precomputed and stored in memory.

TEA, 1994 [199] is a 64-bit block cipher using 128-bit keys (although equivalent keys effectively reduce the key space to 2^{126}). TEA stands for tiny encryption algorithm and, as the name says, this algorithm was built with simplicity and ease of implementation in mind. A C implementation of the algorithm corresponds to about 20 lines of code and involves no S-Box. TEA has a 64-round Feistel structure, each round being based on XOR, 32-bit addition and rotation. The key schedule is also very simple, alternating the two halves of the key at each round. TEA is sensitive to related-key attacks using 2^{23} chosen plaintexts and one related-key query, with a time complexity of 2^{32}.

KASUMI, 1999 [3] is a block cipher derived from MISTY1 [125]. It is used as a keystream generator in the UMTS, GSM, and GPRS mobile communications systems. It has a 128-bit key and 64-bit inputs and outputs. The core of KASUMI is an eight-round Feistel network. The round functions in the main Feistel network are irreversible Feistel-like network transformations. The key scheduling is done by bit-wise rotating the 16-bit subkeys or XORing them with a constant. There are two types of S-Boxes: 7-bit S-Boxes and 9-bit S-Boxes.

NOEKEON, 2000 [50] is a block cipher with a key length and a block size of 128 bits. The block cipher consists of a simple round function that bases only on bit-wise Boolean operations and cyclic shifts. The round function is iterated 16 times for both encryption and decryption. Within each round, a working key is XORed with the data. The working key is fixed during all rounds and is either the cipher key itself (direct mode) or the cipher key encrypted with a null string. The self-inverse structure of NOEKEON allows efficiently combining the implementation of encryption and decryption operation with only little overhead.

ICEBERG, 2004 [181] is an involutional block cipher optimized for hardware implementations. It uses 128-bit keys and blocks of 64 bits. A first non linear layer consists of two 4-bit S-Box layers separated and followed by bit-wise permutations applied on 8-bit blocks of the state. The second layer is composed

of successive permutations combined with a binary matrix multiplication and a key addition layer. The two types of permutations used here are bit-wise permutations, one applied on 64-bit blocks of data and the other on blocks of 4 bits. The multiplication uses a binary involutional matrix that is applied on blocks of 4 bits. ICEBERG has a key scheduling process as complex as one round of the cipher. First, a key expansion (composed of substitution, byte shifts and bit permutations) is performed. Then, a key selection process reduces the key to 64 bits.

mCrypton, 2005 [116] is a block cipher specifically designed for resource-constrained devices such as RFID tags and sensors. It uses a block length of 64 bits and a variable key length of 64, 96 and 128 bits. In this work, we implemented the variant with a 96-bit key. mCrypton consists of an AES-like round transformation (12 rounds) and a key schedule. The round transformation operates on a 4×4 nibble array and consists of a nibble-wise non-linear substitution, a column-wise bit permutation, a transposition and a key-addition step. The substitution step uses four 4-bit S-Boxes. Encryption and decryption have almost the same form. The key scheduling algorithm generates round keys using non-linear S-Box transformations, word-wise rotations, bit-wise rotations and a round constant. The same S-Boxes are used for the round transformation and key scheduling.

HIGHT, 2006 [91] is a hardware-oriented block cipher designed for low-cost and low-power applications. It uses 64-bit blocks and 128-bit keys. HIGHT is a variant of the generalized Feistel network and is only composed of simple operations: XOR, mod 2^8 additions and byte-wise rotations. Its key schedule consists of two algorithms: one generating whitening key bytes for initial and final transformations; the other one for generating subkeys for the 32 rounds. Each subkey byte is the result of a mod 2^8 addition between a master key byte and a constant generated using a linear feedback shift register.

SEA, 2006 [180] is a scalable family of encryption algorithms, defined for low-cost embedded devices, with variable bus sizes and block/key lengths. In this work, we implemented $\text{SEA}_{96,8}$, i.e. a version of the cipher with 96-bit blocks and keys. SEA is a Feistel cipher that exploits rounds with 3-bit S-Boxes, a diffusion layer made of bit and word rotations and a mod 2^n key addition. Its key scheduling is based on rounds similar to the encryption ones and is designed so that keys can be derived "on-the-fly" both in encryption and decryption.

DESL, DESX, and DESXL, 2007 [113] are lightweight variants of the DES cipher. For the L-variant, all eight DES S-Boxes are replaced by a single S-Box with well chosen characteristics to resist known attacks against DES. Additionally the initial permutation (IP) and its inverse (IP^{-1}) are omitted, because they do not provide additional cryptographic strength. The X-variant includes an additional key whitening of the form: $\text{DESX}_{k,k1,k2}(x) = k2 \oplus \text{DES}_k(k1 \oplus x)$. DESXL is the combination of both variants. The main goal of the developer

was a low gate count in hardware implementations similar to the original DES proposal.

PRESENT, 2007 [30] is a hardware-oriented lightweight block cipher designed to meet tight area and power restrictions. It features a 64-bit block size and 80-bit key size. PRESENT implements a substitution-permutation network in 32 rounds. The permutation layer consists only of bit permutations. Together with the tiny 4-bit S-Box, the design enables minimalistic hardware implementations. The key scheduling consists of a single S-Box lookup, a counter addition and a rotation.

KATAN and KTANTAN, 2009 [38] are two families of hardware-oriented block ciphers. They have 80-bit keys and a block size of either 32, 48 or 64 bits. The cipher structure resembles that of a stream cipher, consisting of shift registers and non-linear feedback functions. The difference between KATAN and KTANTAN lies in the key schedule. KTANTAN is intended to be used with a single key per device, which can then be burnt into the device. This allows KTANTAN to achieve a smaller footprint in a hardware implementation. In the following, we considered the implementation of KATAN with 64-bit block size.

KLEIN, 2011 [78] is a family of lightweight software-oriented block ciphers with 64-bit plaintexts and variable key length (64, 80 or 96 bits - we will focus on the 80-bit version later in this work). It is primarily designed for software implementations in resource-constrained devices such as wireless sensors and RFID tags, but its hardware implementation can be compact as well. The structure of KLEIN is a typical SPN with 12/16/20 rounds for KLEIN-64/80/96, respectively. One round transformation consists of four operations AddRoundKey, SubNibbles (4-bit involutive S-Box), RotateNibbles and MixNibbles (borrowed from AES MixColumns). The key schedule of KLEIN has a Feistel-like structure. It is agile even if keys are frequently changed and is designed to avoid potential related-key attacks.

1.2 HASH FUNCTIONS

1.2.1 General description

Definitions

Cryptographic *hash functions* are functions that take messages of arbitrary-length as input and compress them to short fixed-length output strings. This output, also called *digest* or *hash value*, can be seen as a *fingerprint* of the message. Hash functions are useful in many cryptographic and computer security applications. For example, the authenticity of a large document transmitted over an insecure channel may be guaranteed by sending the hash value of the file over a secure channel. This allows authentication with better performances than sending the complete document over the secure channel. The use of hash functions as one-way functions for the storage of passwords is an

other example of application. Here, only the hash value of a user's password is stored. When the user tries to log in with his password, the corresponding hash is computed and if it matches the stored hash value, access is granted. That way, passwords are never stored in the clear and cannot be retrieved in case of attacks.

Depending on the application in which a hash function is used, some properties might be needed regarding the resistance to attacks. The most commonly used properties are:

- *Preimage resistance* or *one-wayness*: Given a hash value y, it must be computationally infeasible for an adversary to find an input message x such that $h(x) = y$. If the hash function does not present any weakness, finding a preimage is supposed to be as complex as an exhaustive search. The complexity of an attack is thus 2^n with n the output size of the hash function.
- *Second preimage resistance* or *weak collision resistance*: Given a message x, and its hash value $h(x)$, it must be computationally infeasible for an adversary to find a message $x' \neq x$ such that $h(x') = h(x)$. Once again, the complexity of a second preimage attack is supposed to be 2^n.
- *Collision resistance* : It must be computationally impossible to find two distinct input messages, x and x', that map to the same hash value, $h(x) = h(x')$. From the birthday paradox [102], which refers to the probability that in a set of n persons at least two of them will have the same birthday, it is expected that the complexity of finding a collision is approximately $2^{n/2}$. It is often assumed that performing 2^{80} operations is beyond actual computational power [49]. Thus, hash functions will require a hash size of at least 160 bits in order to be collision resistant.

Hash function constructions

As for block ciphers, hash functions generally use a repetition of a simple function having a fixed input length or block length. The message M is first padded to a multiple of the block length and is divided into t blocks M_i. Those message blocks are then processed one after the other by the hash function. There are mainly two types of iterated hash function constructions: the *Merkle-Damgård* and the *sponge* constructions.

The first construction was made popular by Merkle and Damgård who independently stated that an iterated hash function using a collision resistant compression function will also be collision resistant [54] [130]. In this construction, the hash value is computed as illustrated on Figure 1.5 where f is the *compression function*, H_i the intermediate state value or *chaining value*, IV the *initial value* and g an optional *output transformation*.

The second construction was recently proposed by Bertoni et al. [26]. The sponge functions are characterized by two parameters: a bitrate r and a ca-

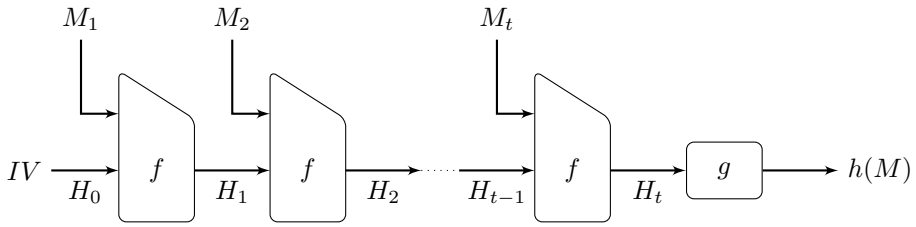

Fig. 1.5.: The Merkle-Damgård construction

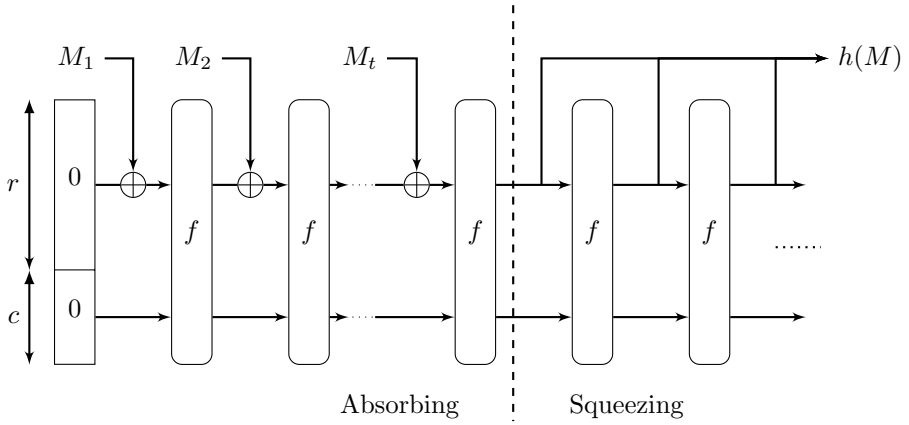

Fig. 1.6.: The sponge construction

pacity c. They are based on a fixed-length permutation f that takes $r + c$ bits of state as input. In a first phase, called the *absorbing phase*, the state is successively XORed with r bits of the message and updated using the permutation. Once every part of the message is "absorbed", the second phase, called the *squeezing phase* can start. During this phase, r bits of state are output after each application of the permutation. It is from those output bits that the sponge digest is formed. This process is illustrated in Figure 1.6.

Iterated hash functions may also be categorized according to the building blocks used to construct them. In this case, we distinguish two types of hash functions: the *dedicated hash functions* (algorithms that have been specifically designed for hashing operations) and the *block cipher-based hash functions* for which existing block ciphers are used as main component of the algorithm. In the next paragraphs, we only discuss those two types of hash functions in the case of the Merkle-Damgård compression functions. Similar constructions are however also possible when building sponge function permutations.

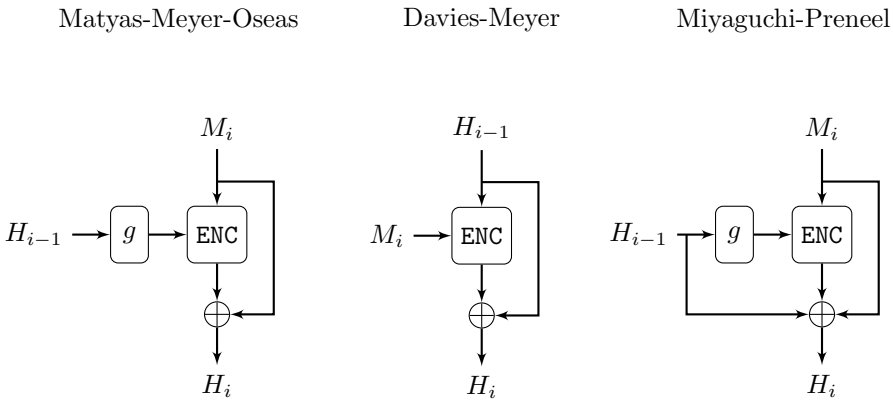

Fig. 1.7.: Composition of three popular block cipher-based compression functions

Block cipher-based compression functions

Block cipher-based compression functions often consists of an encryption block combined with a XOR operation. This can be written as $f(H_{i-1}, M_i) = \texttt{ENC}_a(b) \oplus c$ where a, b anc c can each take one of the following values : the previous state H_{i-1}, the message block M_i, a combination of the two $H_{i-1} \oplus M_i$ or a constant value k. From the 64 possible combinations, that were studied in [153], twelve were judged to be secure. Among those combinations, the three most popular ones are the *Matyas-Meyer-Oseas*, *Davies-Meyer* and *Miyaguchi-Preneel* constructions [108]. As illustrated in Figure 1.7, Matyas-Meyer-Oseas construction uses chaining value as encryption key and the message block as block cipher input. In the case where the block cipher has different key and block sizes, a function g is needed to map the chaining value to the appropriate number of bits. In the Davies-Meyer construction, the chaining value is encrypted using the message block as a key. Lastly, the Miyaguchi-Preneel scheme is the same as the Matyias-Meyer-Oseas one, except that here, instead of the message block M_i, it is the value $H_{i-1} \oplus M_i$ that is XORed with the block cipher output.

The drawback of those kinds of constructions is that many block ciphers have a block size of at most 128 bits, while collision resistant hash function require hash values of at least 160 bits. An other type of construction, called *double block length construction* was therefore proposed. These constructions are also based on bloc ciphers and output hash values that are twice as large as the block size. The *Hirose* construction [88] is an example of double block length hash functions and is illustrated in Figure 1.8 where c is a constant. Here, the chaining value is composed of two parts, G_{i-1} and H_{i-1}, and one

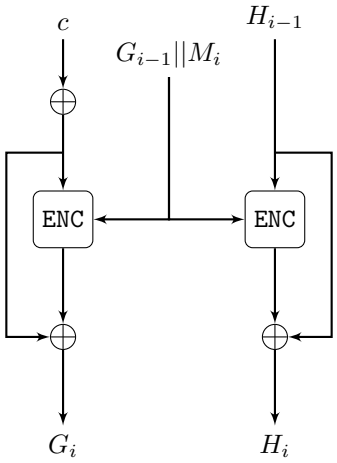

Fig. 1.8.: Composition of the Hirose double block length compression function

half of the chaining value is used with the message block as the key of the two encryptions.

An example of dedicated Hash Function: MD4

Concerning the dedicated hash functions, the *MDx-family* contains some of the most popular hash algorithms. In particular, MD4 was proposed in 1990 by Ronald Rivest [165]. MD4 is now obsolete, as several attacks have been mounted against it. It is however still relevant to mention it since it has influenced the design of other important hash functions such as SHA-1 and SHA-2. MD4 hashes 512-bit message blocks to 128-bit chaining variables using software-oriented 32-bit operations. MD4 is composed of 3 rounds with each round composed of 16 steps. One of these steps is illustrated in Figure 1.9 for which the blocks `A` to `D` represent 32-bit words composing the chaining variable, H_i is a 32-bit message word and K_i a constant. `F` is a non-linear function that compresses three 32-bit words into a single one, $\boxplus$ are additions modulo 2^{32} and $\lll$ a bit-wise circular rotation.

The next sections describe in details the hash functions that will be studied in this thesis. They are divided in four categories: the standard hash functions (section 1.2.2), the SHA-3 competition finalists (section 1.2.3), the lightweight hash functions (section 1.2.4) and the block cipher-based hash functions(section 1.2.5).

1.2.2 Standard hash functions

The SHA-family (which stands for *Secure Hash Algorithm*) is a set of hash functions standardized by the NIST over the last two decades. SHA-1 was de-

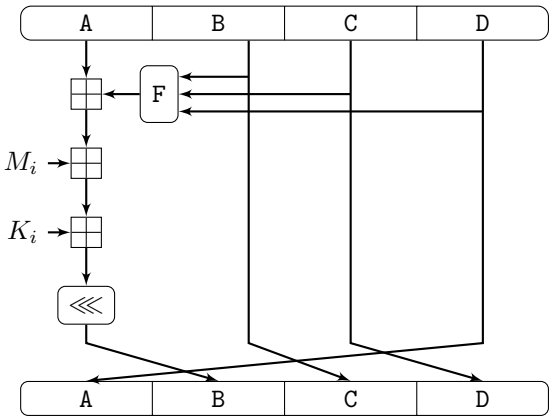

Fig. 1.9.: Composition of a step in MD4

signed by the United States national security agency (NSA) and was published as a standard in 1993. It is an iterated design which outputs a 160-bit hash value from the compression of 512-bit message blocks. In 2001, a new set of cryptographic hash functions, collectively named SHA-2, was published by the NIST as a standard [140]. Due to some flows discovered in the SHA-1 hash function, the NIST launched in 2007 a competition aiming at developing a new standard. In 2013, Keccak was announced as the winner of the competition and is now referred to as SHA-3.

SHA-2

The SHA-2 family is composed of four hash functions named after their digest lenght SHA-224, SHA-256, SHA-384 and SHA-512. SHA-256 and SHA-224 process 512-bit message blocks and use 256-bit chaining variables, while in SHA-512 and SHA-384, message blocks of 1024 bits are compressed to 512-bit intermediate states. One step of the compression function of the SHA-2 family is illustrated on Figure 1.10. Each intermediate state is divided into eight working registers (denoted by the letters `A` to `H`). The functions Σ_0 and Σ_1 are composed of circular rotations and XOR operations. Ch and Ma are non linear functions using the logical operations XOR, NOT and AND. M_i denotes a part of the message and K_i a constant.

SHA-3 (Keccak)

Keccak [24] is a family of sponge functions using a permutation called Keccak-$f[b]$ with b the size of the state ($b = r + c$). The SHA-3 version of Keccak proposes to use exclusively Keccak-$f[1600]$ for all output lengths/security levels, whereas lightweight alternatives can use for instance Keccak-$f[200]$ or Keccak-$f[400]$, leaving Keccak-$f[800]$ as an intermediate choice [25]. Inside

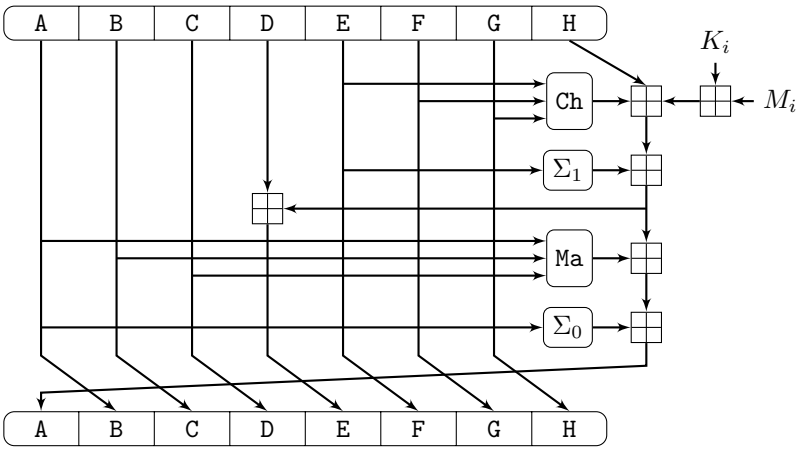

Fig. 1.10.: Composition of a step in the SHA-2 family compression function

Keccak-$f[b]$, the state to process is organized in 5×5 *lanes* of $b/25$ bits each, or alternatively as $b/25$ *slices* of 25 bits each. The round function processes the state using a non-linear layer (χ), a linear diffusion operation (θ), inter- and intra-slice dispersion steps (ρ, π) and the addition of round constants (ι).

1.2.3 The SHA-3 competition finalists

The SHA-3 competition was held from 2008 to 2013 and was composed of 3 rounds. The NIST selected 51 candidates among all the submissions they received for Round 1. In 2009, 14 candidates were accepted for Round 2, only five of them succesively reached the last round. In this section, we describe the four finalists which were part of Round 3 along with Keccak.

BLAKE [13] 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 [28]. It provides resistance to long-message second preimage attacks, and explicitly handles hashing with a salt (a random number used as additional input to the hash function) and a "number of bits hashed so far" counter. The internal structure is the local wide-pipe[1], which was already used within the LAKE hash function [14]. The compression algorithm is a modified version of Bernstein's stream cipher ChaCha [22], 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 oper-

[1]The wide-pipe construction is similar to the Merkle-Damgård one except that the internal state is twice as large as the output hash value

ations 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 [75] 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 SPN which borrows components from the AES, 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 2011.

JH [206] 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 the 1024-bit chaining value H_{i-1} is XORed with a 512-bit message block 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 is then once again XORed with M_i. This work 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.

Skein [69] is built out of a tweakable block cipher [118] 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).

1.2.4 Lightweight hash functions

S-Quark and D-Quark are hardware-oriented hash functions with respective digests of length 256 and 176 bits [11]. This family is based on the sponge construction with respective widths 256 and 176 bits and rates 32 and 16 bits. The absorbing phase consists in XORing the message with a part of the state and applying a permutation P. The digest is obtained by squeezing 32/16 bits of the state, then applying the permutation P, and so on until the correct digest size is obtained. The permutation P is composed of 1024/704 iterations of an update function that essentially consists in performing linear retro-action on the state (the polynomials used slightly differ from a version to another). Note that the capacity of S-Quark is only $c = 224$, leading to 112-bit security level (instead of 128 for other lightweight hash functions in our evaluations).

PHOTON is a sponge-based, lightweight, and hardware-oriented hash function family introduced in 2011 [83]. The internal state is represented as a matrix with 4- or 8-bit entries depending on which of the five PHOTON flavors has been selected. Each PHOTON round consists in XORing the message into the state (absorbing) and applying a permutation P twelve times. The first step when applying P is AddConstants where round constants are XORed to the first column of the state. Then, in the SubCells layer, the PRESENT S-Box (PHOTON-160/36/36) or AES S-Box (PHOTON-256/32/32) is applied to every entry of the state. During ShiftRows the rows of the state matrix are rotated. In the MixColumnsSerial layer a flavor-specific MixColumns matrix is applied to every column of the state several times. This strategy results in a much more compact hardware implementation compared to, e.g. the AES MixColumns layer. When all blocks of the message have been absorbed, a flavor-specific number of squeezing iterations are performed. In each iteration, a small amount of the state is extracted as part of the new hash value and the permutation P is applied twelve times.

SPONGENT is a family of lightweight hash functions based on a wide PRESENT-type permutation [29]. It relies on a sponge construction: a simple iterated design that takes a variable-length input and can produce an output of an arbitrary length n based on a permutation π_b operating on a state of a fixed number b of bits. The different variants are referred to as SPONGENT-$n/c/r$ for different hash sizes n, capacities c, and rates r. Out of the 13 proposed variants, we implemented SPONGENT-160/160/80 and SPONGENT-256/256/128.

Keccak also proposes lightweight alternatives, different from the SHA-3 submission, as explained in Section 1.2.2.

1.2.5 Block cipher-based constructions

Rogaway-Steinberger LP/lp362 is a hash function construction based on a fixed-key block cipher. The construction principle was developed by Rogaway and Steinberger in 2008 [166] and defines a so-called linearly-determined, permutation based (LP) compression function. An $\mathrm{LP}^{\mathtt{A}}_{mkr}$ compression function operates on a matrix A (satisfying a special independence criterion) and k permutations, turning an input block of mn bits into an output block of rn bits. The parameter n gives the bit width of the block cipher. We used the block cipher NOEKEON [50] to carry out the fixed-key permutations, for which n equals 128 bits. An LP362 compression function converts a 384-bit input block into a 256-bit output block by using six fixed-key permutations. In case of the LP362 scheme, every permutation uses a different key, whereas in the lp362 scheme, all permutations use the same key (according to [166] both have similar security bounds). XOR operations and finite-field multiplications are used to combine the output of the individual permutations into a single value. The compression function is turned into a hash function using the Merkle-Damgård construction.

Hirose double block length construction was implemented in this work using AES-256 as block cipher. The construction has thus an output size of 256 bits and the compression function takes 128 bits of input per iteration.

Davies-Meyer. Two instantiations of that construction, using the ciphers Rijndael-256/256 and SEA which were described in sections 1.1.2 and 1.1.3, have been studied.

Shrimpton-Stam construction. Based on the proof that compression function constructions relying on pseudorandom permutations (PRPs) cannot reach optimal security using less than 3 permutations, Shrimpton and Stam proposed a construction relying on 3 different PRPs [173] where the chaining value H_i is updated according to a message block M in the following way $H_{i+1} = f_3\left(f_1(M) \oplus f_2(H_i)\right) \oplus f_1(M)$.They mention that these permutations can be instantiated with a cipher (using three different keys) but only in plaintext feedback mode (that is, the same as in the Davies-Meyer construction). A single instantiation based on Rijndael-256/256 has been considered here.

CHAPTER 2

IMPLEMENTATION TECHNOLOGIES

In Chapter 1, we detailed the construction of several cryptographic algorithms. In order for them to be used as part of an application, some design and implementation steps need to be carried out. These steps will differ depending on the type of devices that are used for the implementation. This chapter aims at describing the implementation technologies we will use throughout the thesis. Typically, those technologies may be divided into two general categories: the hardware and software devices. In a first section, we will present two types of hardware devices, namely the *application specific integrated circuits* (ASICs) and the *field-programmable gate arrays* (FPGAs), together with the design flow and usual evaluation metrics they are related to. Then, in a second section, we will introduce similar concepts in the case of software devices, and more specifically for the case of *microcontrollers*.

2.1 HARDWARE DEVICES

2.1.1 Application specific integrated circuit

Integrated circuits (ICs) are small electronic devices made of semiconductor material. In 1958, Jack Kilby, from Texas Instruments, was the first to integrate a small electronic circuit (a flip flop composed of two transistors) onto a single slice of Germanium [107]. This was the advent of the IC industry which now produces circuits containing several billions of transistors. At first, ICs were built using bipolar transistors, which rely on the junction of two types of semiconductors. In the 1960s, *metal oxide semiconductor field effect transistors* (MOSFET) were developed and more commonly used in circuits. Due to their power dissipation at idle, MOSFET transistors were later replaced by the *complementary metal oxide semiconductor* (CMOS) using a combination of both n-type and p-type MOSFETs.

What is an ASIC?

ICs are often divided into two categories: the standard ICs whose functionality is solely determined by the IC vendor (e.g. memories, microcontrollers and standard logic ICs), and the ASICs which are dedicated to a specific user need [190]. This second category is the one that we will consider in the remainder of this work since our implementations are intended for end-user applications. ASICs (and ICs in general) are fabricated on thin silicon wafers that serve as *substrate*. The transistors and the wires connecting them are composed of several layers which are defined using masks through *photolitography*.

Depending on the degree of customization of the layers, ASICs may be classified as follows [175, 190]:

- *The full-custom ASICs* are ASICs for which some or all the logic cells, layout and wiring are handcrafted specifically for one given customer application. Typically, full-custom designs will be used when existing logic cells are not fast enough, consume too much power or are not small enough. This approach, however, comes at the cost of high manufacturing and design time.

- *The cell-based custom ASICs* use predesigned standard cells (e.g. AND and OR gates, flip-flops and multiplexers), macro cells and mega cells (e.g. RAM, ROM, multipliers and micro-processors). In this case, the circuit is usually described using *hardware description language* (HDL) and the layout is left to automated design tools, with the possibility to manually optimize any performance-limiting aspect of the design. This approach results in performances and flexibility that are close to the full-custom ones but with reduced design times and risks.

- *The semi-custom ASICs* are ICs for which some masks but not all are unique for a customer application. In this type of design, all the layers defining the transistors and other active components have been preprocessed leaving the customization of the contact and metal layers (i.e. the layers defining the connections between the transistors) to the customer. Semi-custom ASICs benefit of very low design cost and manufacturing times but implies reduced flexibility and performances (e.g. the size of the IC is fixed by the preprocessed layer and it is often difficult to fill 100% of the area with a particular design). Gate arrays are examples of semi-custom designs. They tend to disappear however as they are being replaced by FPGAs, which will be discussed in section 2.1.2.

In this thesis, we will only focus on cell-based custom ASICs. In particular, we will use commercial 65-nanometer CMOS low power libraries of standard cells for our implementations.

Moore's law and technology scaling

In 1965, Gordon E. Moore, co-founder of Intel, observed the growth of the number of transistors embedded in ICs and predicted that it would roughly double every two years [134]. As illustrated on Figure 2.1, this prediction occurred to be more or less accurate over the years and was mainly enabled by the technology scaling. Naturally, in order to double the number of transistors per chip, and thereby increase the number of possible functionalities of each IC, the size of the transistors and the wires connecting them each other needed to be reduced. This size is set by the mask dimensions which are limited by the resolution of the manufacturing process. Researches in the domain therefore tended to develop processes with still better resolutions. This trend in technology scaling is illustrated on Figure 2.2 which shows that the transistor size has been reduced by a factor 1000 over the last 45 years.

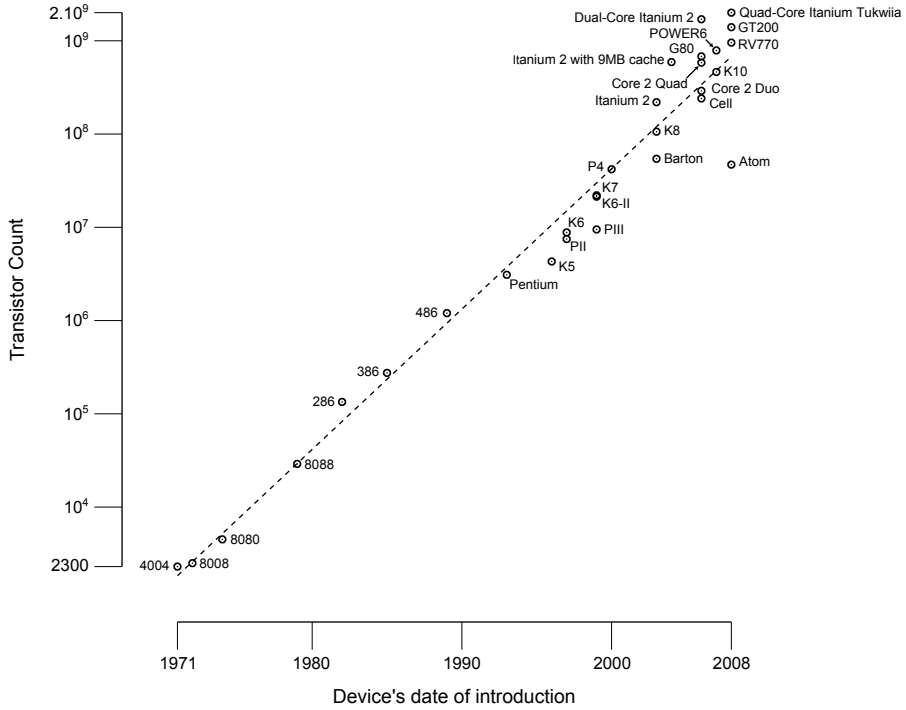

Fig. 2.1.: Number of transistors embedded in CPUs from 1971 to 2008 versus Moore's law (represented by the dashed line). Adapted from [201].

Smaller transistors are also faster (i.e. they allow higher working frequencies), as electrons have smaller distances to travel, and are more energy efficient, as less electrons are needed to charge the gate. Despite this reduction of energy per operation, the total power consumption per chip dramatically suffered from the exponential growth of the number of transistors in ICs and from

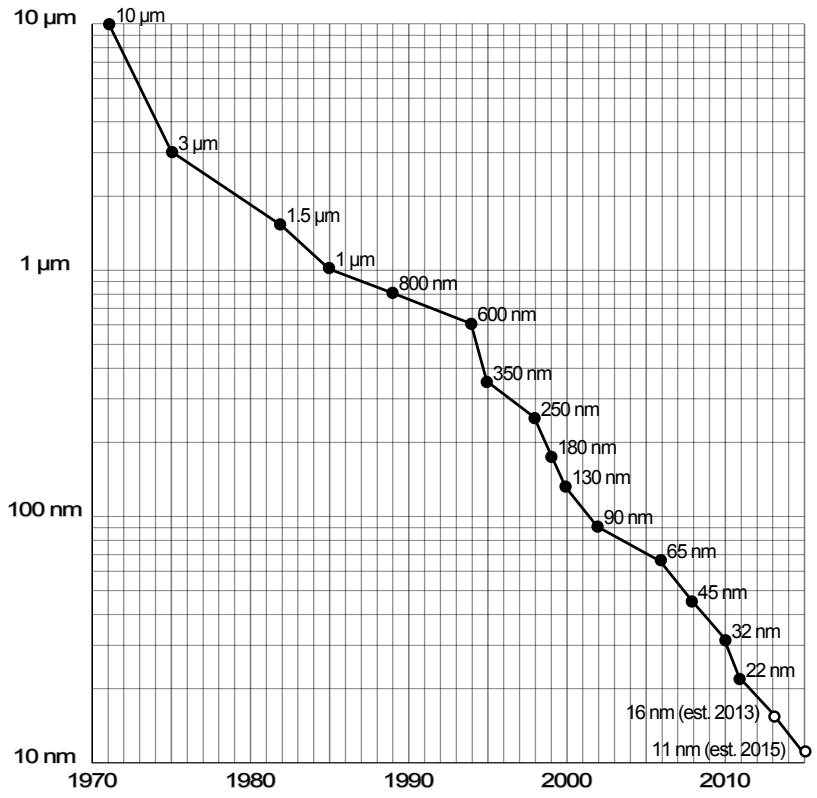

Fig. 2.2.: Evolution of the size of semiconductor manufacturing process nodes over the years. Adapted from [202].

the clock frequency increases [31]. At that time, the dominating component of power dissipation was the dynamic power consumption of the gates and more specifically the switching activity which can be expressed as $P_{sw} = \alpha C_L V_{dd}^2 f$ with α the activity factor of the gate, C_L the load capacitance, V_{dd} the supply voltage and f the clock frequency. Since voltage has a quadratic effect on power, the IC industry tends to lower the supply voltage in order to reduce the power consumption.

In the late 90's, as technology scaling was reaching the nanometer era, two detrimental effects started to rise: the increase in leakage current [168] and the process variability [34]. Prior to the 90 nm node, leakage power was only a concern during sleep mode, being negligible compared to dynamic power during switching activity. In nanometer processes, with low threshold voltages (which is a direct consequence of the decrease in supply voltage) and thin gate oxide, leakage can account for as much as a third of the total active power [198]. Performance variability on its side is a result of random dopant fluctuations, short-channel side effects and loss of channel control by

the gate which are also consequences of the constant decreasing of the gate size combined to the constraints put on supply voltage, threshold voltage and oxide thickness.

Frequency/voltage scaling

All applications do not have equal time performances requirements. For some applications having lower computational loads, the operating frequency may be reduced to the minimum sufficient to complete the task on schedule. Besides, since the gate delay, and thereby the inverse of the maximum operating frequency, is proportional to $C_L V_{dd}/I$, with I the MOSFET drain current when in on state, supply voltage can also be lowered to the minimum necessary to operate at a given frequency. This simultaneous scaling in frequency and voltage allows saving a lot on power consumptions. The voltage lowering can be pushed to the limit where V_{dd} is lower than the threshold voltage V_t. The MOSFETs then operate in subthreshold or weak-inversion regime, as was proposed by Swanson and Mendl in 1972 [183]. Under this condition, the circuit thus use subthreshold leakages as the active drain current, which depends exponentially on the threshold voltage and the MOSFET bias voltages.

2.1.2 Field-programmable gate arrays

An FPGA is an IC designed to be configured by a customer or a designer after manufacturing. As opposed to ASICs, where the device is customized for a particular design, FPGAs can be reconfigured for a specific application even after the product has been installed in the field, hence the name "field-programmable"[1] [5, 208]. In the remainder of this section, we will first present the general architecture of FPGAs. Then, we will briefly introduce the different FPGA manufacturers and the type of FPGA they are selling. Finally, we will give more details about the composing blocks of the FPGAs we will be using in this thesis.

FPGA architecture

The most common FPGA architecture consists, as illustrated in Figure 2.3, of an array of programmable logic blocks (called *configurable logic block* (CLB) or *logic array block* (LAB) depending on the vendor), programmable *input/output* (I/O) pads and a network of programmable interconnections intended to connect the blocks to each others [203, 212]. Next to this minimal configuration, some other resources like memories, multipliers and processors may also be present inside the FPGA.

The CLBs are usually divided into several logic cells (called *slices*, *adaptive logic modules* (ALMs) or *logic elements* (LEs)). These cells are composed of LUTs that are used to create combinatorial function, flip-flops which can be

[1] Although one-time programmable FPGAs are available, the dominant type of FPGAs are SRAM-based and can be reprogrammed as the design evolves.

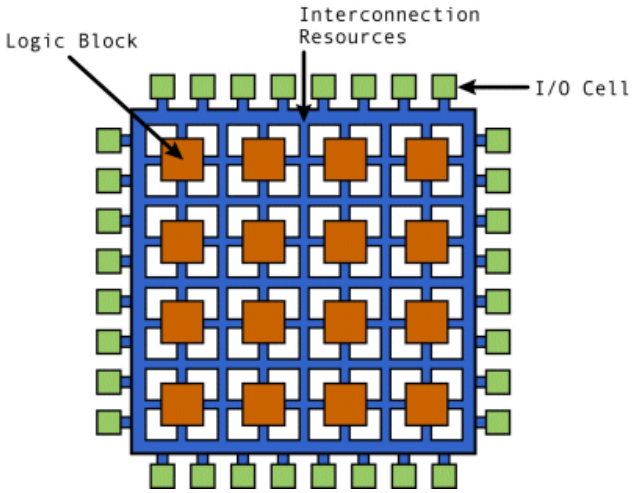

Fig. 2.3.: Generic FPGA architecture. Illustration from [212].

seen as clocked storage elements and multiplexers which route the logic within the logic cell and with the external resources. An example of slice structure is presented in Figure 2.4.

Programmable interconnections come in two flavors: the short lines designed to interconnect close CLBs to each other and the long lines which are spanning horizontally and vertically through the device and are intended to connect critical CLBs that are physically far from each other without including much delay. As shown on Figure 2.5, switch matrices connect the long and short lines together in flexible combinations. Special long lines, called global clock lines, are also available and are specifically designed for low-skew routing. These lines are connected to the clock buffers and to each clocked element inside the CLBs.

FPGA process technologies

Several competing process technologies[2] are available for programming FPGAs. First, SRAM-based FPGAs are the most popular ones as they use standard fabrication processes. Since SRAM is a reprogrammable type of memory, SRAM-based FPGAs may be reprogrammed any number of times. The main disadvantage of SRAM is that it is volatile, thereby requiring external boot devices for reprogramming at power up.

A second family of FPGAs use antifuses as building technology. An antifuse has the opposite function of a fuse, i.e. an antifuse is designed to have high resistance by default and can be modified to become a permanent conducting

[2]The different technologies described in this subsection are mostly the same as the ones used for building microcontrollers memories. The way they are working is explained in more details in section 2.2.1

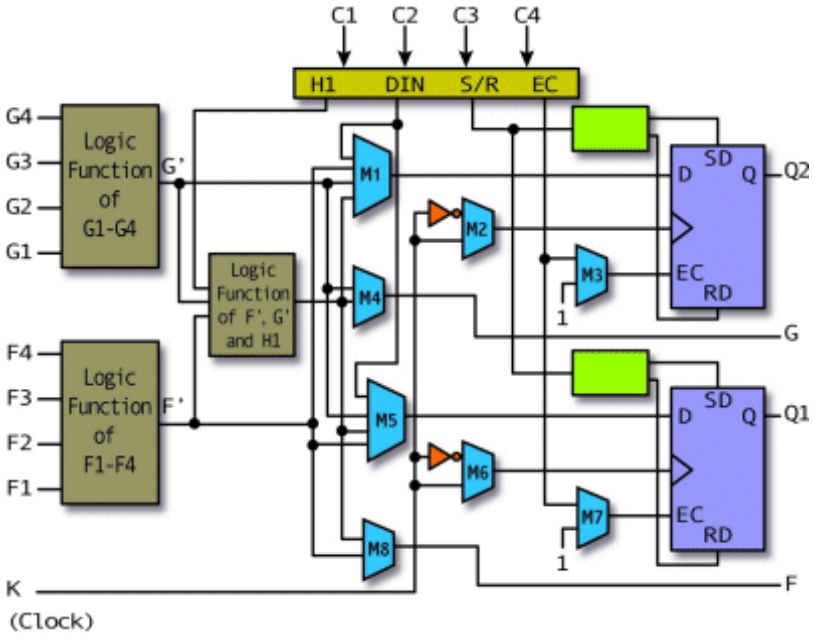

Fig. 2.4.: Example of slice structure. Illustration from [212].

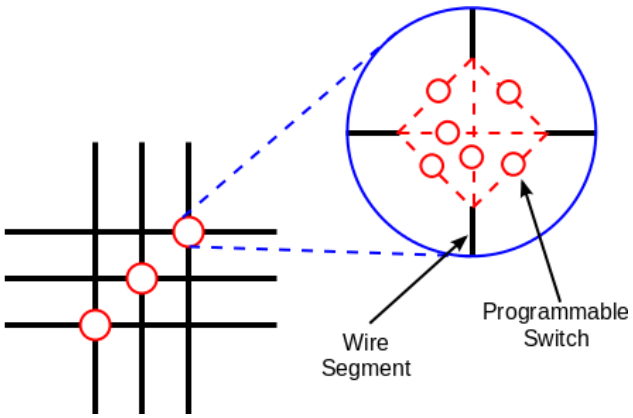

Fig. 2.5.: FPGA's switch matrix topology. Illustration from [203].

electrical path as soon as a high level voltage is applied to it. Antifuses have the benefit to be non-volatile and induce very small routing delays. On the other hand, they require a complex fabrication process and once they are programmed, they cannot be changed.

Finally, flash-based FPGAs combines the advantages of both previous technologies. Flash memories are non-volatile and can be electrically repro-

grammed. They use a standard fabrication process and they have low power consumptions. Currently, however, only one vendor sells flash-based FPGAs.

FPGA manufacturers and families

Currently, five companies are present on the FPGA market: Xilinx, Altera, Lattice Semiconductor, Microsemi and QuickLogic. Xilinx and Altera control together more than 80 percent of this market [93], making them the current leaders. Xilinx proposes SRAM-based FPGAs designed for a great range of applications. Their products were initially divided into two categories: the Spartan family intended for the low-to-mid-end market and the Virtex family covering the high-end market. Recently, two new families have been introduced: the Artix family optimized for low cost and low power applications and the Kintex family optimized for best price/performance ratio. The most recent Xilinx's FPGAs, the UltraScale devices, are built using a 20nm process technology. Altera, on its side, manufactures three families of SRAM-based FPGAs: the Cyclone, Aria and Stratix families which are respectively intended for low-cost, midrange and high-end markets. Their latest manufactured FPGAs use 14nm process technology. The last three vendors are less known, but are gradually winning market shares by targeting specific applications and submarket. Lattice Semiconductor tackles the low-power and low-cost FPGA market. Microsemi designs low-power and mixed-signal FPGAs and is the only manufacturer proposing flash-based and antifuse FPGAs. Finally, QuickLogic proposes customizable semiconductors for mobile devices.

Xilinx FPGAs

In this work, we will only focus on the Xilinx families of FPGAs and in particular to the Virtex-II Pro, Virtex-6 and Spartan-6 families. The main features of these FPGAs will be briefly described here.

The Virtex-II Pro FPGAs are part of the second generation of FPGAs proposed by Xilinx. They were introduced in 2002 and built using a 130-nm process technology. Each Virtex-II CLB element is composed of four slices which are themselves composed of two 4-input function generators, carry logic, arithmetic logic gates, multiplexers (that can combine function generators in order to form logic functions having up to 8 inputs) and two storage elements (which can be configured as registers or latches) [210]. As shown on Figure 2.6, the 4-input generator can be programmed either as a 4-input LUT, a 16-bit shift register or 16 bits of distributed memory. Virtex-II Pro FPGAs can have up to 44096 slices and also dispose of multipliers and 18 Kbits block RAMs.

Spartan-6 and Virtex-6 FPGAs were introduced in 2009 and built using 45-nm and 40-nm process technologies respectively. They basically have the same kind of building blocks and differ mostly by their performances and number of available resources. Their CLBs are composed of two slices, each containing four 6-input LUTs as well as eight storage elements [209, 211]. Similarly to the Virtex-II Pro FPGAs, the function generators of some of the Virtex-6 and

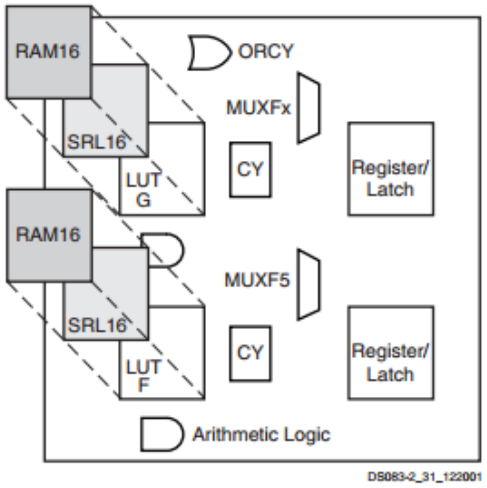

Fig. 2.6.: Virtex-II Pro slice configuration. Illustration from [210].

Spartan-6 slices may be programmed as 64-bit distributed RAMs or as variable-length shift registers (with a maximum length of 32 bits). Again, besides the CLBs, some other components are available, e.g. *digital signal processing* (DSP) slices and 18 Kb or 36 Kb block RAMs. The number of available slices in Virtex-6 FPGAs range from 11640 to 88560, while for Spartan-6 FPGAs, their number varies from 4800 to 23038.

2.1.3 Hardware design flow

The hardware design flow, illustrated Figure 2.7, is essentially made of five successive steps for which a succinct description is given next.

1. *Description.* This step consists in describing the behavior of an algorithm using a *register transfer level* (RTL) abstraction. This term originates from the fact that most systems may be seen as a collection of registers (that store binary data) and logic operations (which are applied to those binary data). The RTL description is often specified through HDLs, e.g. VHDL or Verilog. Such languages determine the number and type of operations that have to be implemented on chip, as well as their scheduling. The result of this step is called the hardware source code.

2. *Synthesis.* Here, the hardware source code is converted into an idealized electronic circuit. The synthesis tools automatically map a high-level description into a lower-level one using a library of cells or logic gates. It is functionally equivalent to the hardware source code and corresponds to the target platform (e.g. ASIC or FPGA). The result of this step is called a netlist. Netlists are idealized in the sense that they do not cor-

respond to a physical description of the chip. Yet, the result of synthesis is highly dependent on the tool options (e.g. space- or time-oriented optimizations).

3. *Place and route.* At this point of the design flow, the automated tools build a physical model for all the logic and memory elements of the netlist. They also simulate their mapping on the circuit surface, taking the position of each cell, the width and length of connections, ... into account. The result of this step is called the layout for ASICs and the bitstream for FPGAs. This output is again dependent on the tool options.

Note that for reconfigurable devices such as FPGA, the design flow essentially stops here (i.e. moves to final Step 5). Namely, the device can be programmed using the bitstream obtained at Step 3 and will next act as a dedicated circuit with the intended functionality. By contrast for ASIC, an additional Step 4 has to be carried out:

4. *Manufacturing.* Once the design of the chip is finished, its layout is sent to a manufacturer in order to be physically built. Manufacturing usually includes a packaging step that consists in protecting the chip with a plastic coating.

5. *Integration.* Eventually, and optionally, the chip can be integrated into a larger system, combining different electronic pieces together.

2.1.4 Evaluation metrics for hardware implementations

Evaluating hardware implementations is a challenging task. In this section, we introduce different metrics that can be used for this purpose, together with possible shortcomings with respect to their relevance for comparing algorithms. Namely, we will consider the area, power consumption, throughput and energy cost, as summarized in Figure 2.8. This selection was motivated by the fact that these metrics are generally reflective of the application constraints that may be encountered in practice. In general, the most revealing units for these metrics are physical (i.e. μm^2, Watts, bit/sec and Joules). However, as these physical units can only be obtained at the very end of an implementation process, convenient first-order estimates are obtained with the gate count, switching activity (i.e. number of bit transitions per clock cycle), number of cycles per algorithm execution and block size.

The first important observation regarding these metrics is that they are always *relative*, meaning that it is usually possible to optimize a single metric quite arbitrarily, if the other ones can be degraded. Hence, in order to make any comparison relevant, it is necessary to agree on some application objectives. For example, the area and power can be relative to time or energy constraints, while the throughput and energy can be relative to area or power

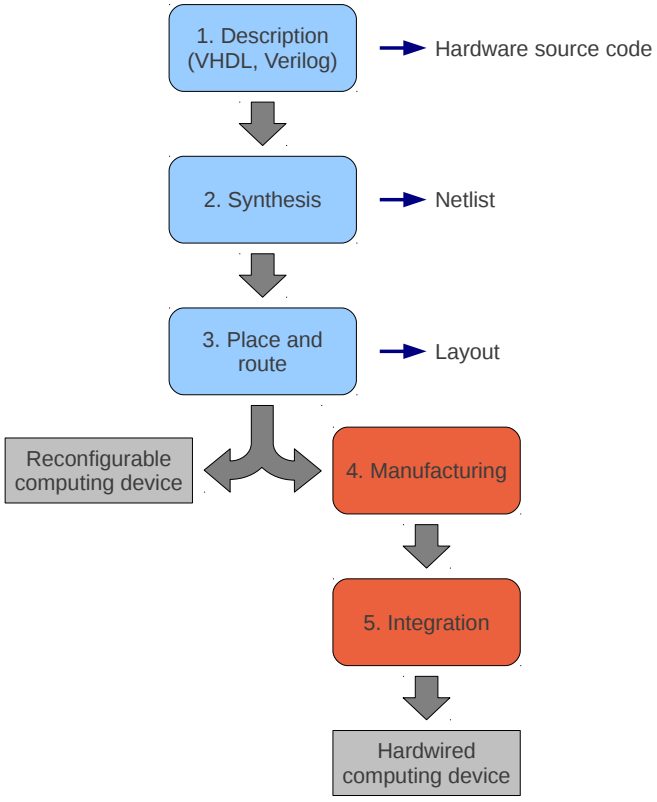

Fig. 2.7.: Hardware design flow.

constraints. As a result, optimization goals can be roughly separated as "design for low area or power" and "design for high throughput or low energy". In the first case, designers will typically share the resources (to decrease the area cost) and reduce the datapath (to limit the switching activity). Such optimizations are illustrated for the case of a block cipher S-Box layer in the left part of Figure 2.9. Quite naturally, re-using the same component also implies that the relative cost of the S-Box compared to the control logic and memory decreases, which implies a lower efficiency. Therefore, in the second case, the designer will rather unroll the S-Boxes (i.e. implement them all on chip) and parallelize their computation (i.e. perform them in a single clock cycle), as illustrated in the right part of the figure.

Besides, the performances of these implementations will also depend on their clock frequency, for which the maximum value f_{max} is inversely proportional to the longest combinatorial path (aka critical path) between two registers. If the clock frequency is not sufficient, inner pipelining can be used

application constraints	physical units	relative to	pre-layout units	HW design goals	algorithmic design goals	relevance w.r.t. algorithms
AREA	um²	time or energy constraints	#gates	share resources	reduce components cost & versatility	weakly discriminant ***
INST. POWER (dynamic)	W (J/sec)	time or energy constraints	switching activity	reduce datapath	reduce components cost & versatility	somewhat arbitrary **
THROUGHPUT	bit/sec	area or power constraints	#cycles (& block size)	unroll, parallelize & pipeline	minimize the total combinatorial cost	very arbitrary *
ENERGY	J/enc, J/bit	area or power constraints	#cycles X POWER	unroll	minimize the total combinatorial cost	somewhat discriminant ***

Fig. 2.8.: Summary of evaluation metrics for hardware implementations.

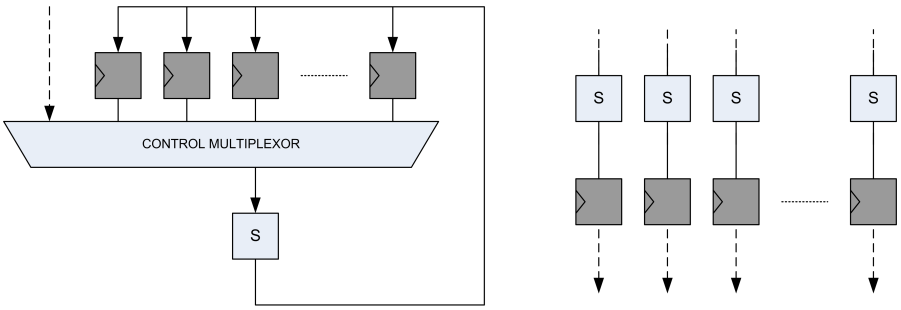

Fig. 2.9.: Left: serial design with resource sharing. Right: unrolled & parallelized design.

in order to cut the critical path by the addition of registers, as illustrated in Figure 2.10.

Having roughly described these optimization techniques allows us to come back on the relativity of the metrics when comparing different algorithms. For example, the throughput is very arbitrary, as it can be straightforwardly improved by multiplying the circuit size. The same observation holds (to a smaller extent) for the instantaneous power consumption, as a designer could theoretically reduce his datapath to a single bit, independently of the algorithm to implement. In general, a lack of instantaneous power can also be overcome by decreasing the clock frequency and relying on decoupling capacitances. Hence, applications where this metric really matters are quite limited (RFID being the most frequent example). The area cost becomes slightly more discriminant, since increasing the sharing of resources generally implies a cost penalty in the control part. Finally, the energy per encryption is more discriminant, as it corresponds to an integral over time and is not compressible beyond what is allowed by the total combinatorial cost of an algorithm. Quite naturally, many combined metrics can also be derived, e.g. the “throughput

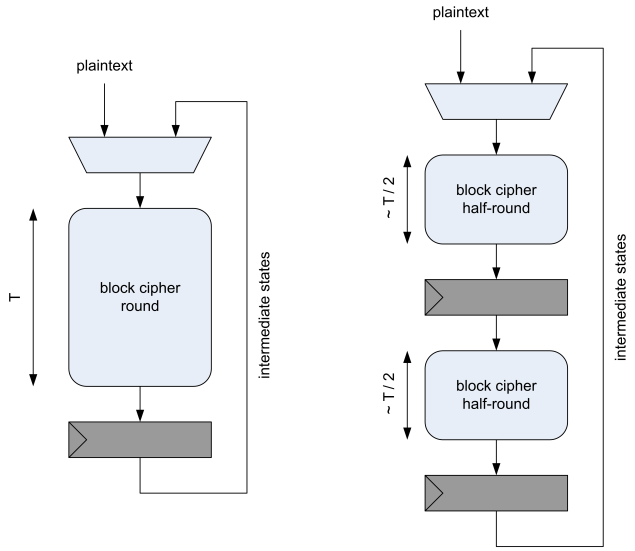

Fig. 2.10.: Inner pipelining of a block cipher round.

over area ratio" is one of the most popular tool to express the performance efficiency of a given hardware implementation.

The main consequence of this relativity is that the fair comparison of hardware implementations is always specialized to a set of constraints. For this purpose, it is always necessary to carefully specify the methodology that was used to perform a particular evaluation. Next to that, some general comments concerning implementation comparisons are worth being mentioned.

(1) Present hardware design flows make intensive use of automated tools, of which the options highly influence the final performance. For example, in the case of ASICs, imposing stronger constraints on the clock frequency can be automated in this way, at the cost of area increases. In such cases, it is useful to agree on the maximum tolerated penalty (compared to the area obtained without frequency constraints). In the same way, the choice of synthesis and place and route options for the FPGA design flow has a great impact on the results [59]. It is thus recommended to keep the same set of options for all the evaluated implementations.

(2) Once all design choices have been made, it is possible to further tune the performances of an implementation, e.g. by taking advantage of frequency / voltage scaling for ASIC.

(3) As technologies are shrinking to the nanometer scale, a part of their power consumption may become static (i.e. happen independent of the switching ac-

tivity)[3]. As leakage currents are essentially dependent on the circuit size, it implies that the optimization goals for area and power become closer in this case. Significant leakage currents also have an impact on the energy performances.

(4) In general, comparisons are only meaningful for algorithms with the same properties, e.g. same block and key size for block ciphers. Yet, different block sizes can sometimes be reflected in the metrics (e.g. by computing the energy per bit rather than per block).

2.2 SOFTWARE DEVICES

Manufacturing hardware devices is a long and difficult task. Therefore, for some applications that do not require the large range of possibilities provided by hardware designs, some general-purpose devices may be sufficient. In this context, a particular type of ICs designed to control the operations in embedded systems has been developed. They are called *microcontrollers* and allow a limited set of predefined operations which are then controlled by a user-defined software code. In this section, we will first introduce the concepts related to microcontrollers (Section 2.2.1). Then, we will describe the steps composing the software design flow (Section 2.2.2). Finally, we will give some information about the useful metrics for evaluating software implementations (Section 2.2.3).

2.2.1 Microcontrollers

Microcontrollers are electronic devices capable of manipulating information in a way specified by a sequence of instructions. This sequence of instructions is defined by the user and hence microcontroller are programmable. In this section, we will first show the different elements composing a microcontroller. Then, we will discuss the existing microcontroller architectures. We will also give a brief introduction regarding the memories usually available in microcontrollers. Finally, we will describe the three microcontrollers that will be used for implementations in later chapters.

Microcontroller composition

As illustrated on Figure 2.11, the usual components present in microcontrollers are the following:

- *The CPU* is the part of the microcontroller that executes the software program (stored in memory as a sequence of instructions). The CPU is typically composed of an *arithmetic logic unit* (ALU) which performs

[3]Note that this effect can be mitigated by exploiting low-leakage libraries.

the arithmetic and logic operations, and a *control unit* which fetches the instructions in memory and executes them using the ALU when necessary. The CPU also has access to several *registers*, i.e. internal memory cells that can be accessed more quickly than the microcontroller main memory. Different types of register are available in CPUs. First, during the execution of instructions, data registers may be used to store the operands and results of arithmetic and logic operations. Then, the CPU may indirectly access the memory by reading the value stored in an address register[4]. Besides, some special purpose registers are used to hold the program state (e.g. the program counter giving the memory address of the next instruction to execute and the status registers holding informations about instruction's results). Those are the most commonly encountered types of register, other types of register may be available depending on the microcontroller.

- *The memory* stores the information required for the execution of the program. Most of the time, it is divided into two parts which are called the *program* and *data memories*. The program memory contains the instructions to be executed and the program parameters (e.g. constants and tables). It requires a non-volatile type of memory. The data memory, on the other hand, may be volatile as it usually only contains the intermediate data on which the instructions are performed. More details about the technologies usually used to build microcontroller memories will be given later in this section.

- *The peripherals* are components that are connected to the bus system in order to exchange information with the CPU. Typically, they ensure a link between the CPU and the external world (e.g. analog-to-digital converters and I/O controllers), but they may also simply perform internal actions needed by the CPU (e.g. timers).

- *The bus system* is composed of several electrical interconnections allowing the exchange of information between all the components of the microcontroller. This bus system is usually composed of three parts: a data bus carries data from the CPU to the memory and peripherals, an address bus is used to indicate which memory location needs to be written or read, and a control bus transports the signals that control and synchronize the exchange of information. A microcontroller is often designated by the size of its data bus (i.e. an n-bit microcontroller is able to process n-bit wide data words in a single operation).

[4]Note that in some microcontroller the registers can store data as well as addresses, they are usually called general purpose registers in that case.

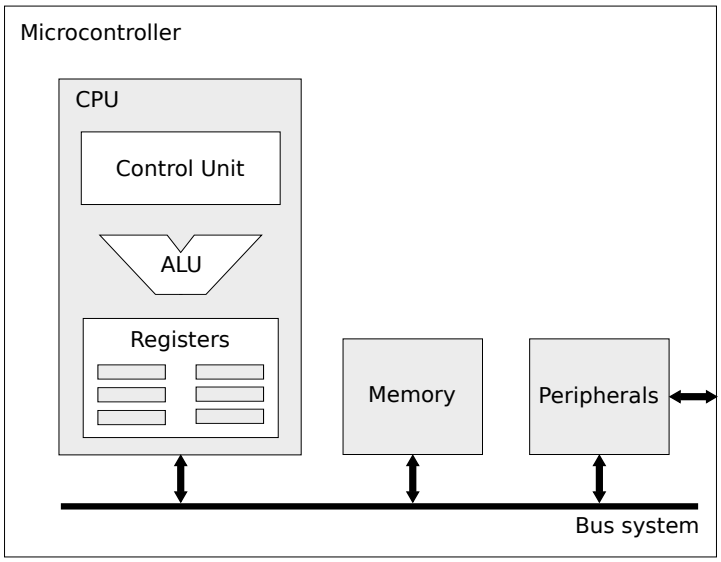

Fig. 2.11.: Microcontroller basic composition (Von Neumann architecture).

Microcontroller architectures

The architecture of a microcontroller is defined by the way in which its elements are interconnected and exchange data with each other. In 1945, John Von Neumann described a first structure of "very high speed automatic digital computing system" that could be programmed by codes residing in memory [193]. In this structure, the composing parts of the microcontroller are all connected to each other using a unique bus system. By contrast, a second structure, called the Harvard architecture and illustrated in Figure 2.12, uses two sets of bus systems in order to exchange information: one for the program and the other one for the data. The Harvard architecture has the benefit that the CPU can simultaneously read an instruction from the program memory and write or read the data memory. Besides, this type of architecture prevents the user from accidentally writing data inside the program memory, thereby corrupting the software code. The Harvard architecture is however less flexible as two distinct memory banks are required. Since the size of these resources are fixed, the Von Neumann architecture will be more suited for applications having varying needs in program and data sizes [148].

Microcontroller memories

Memories used inside microcontrollers may be divided into two main categories: the *read only memories* (ROMs) and the *random access memories* (RAMs). Typically, ROMs are considered as memories that can only be read by the CPU. Yet, certain types of ROMs may be modified, but this is at the

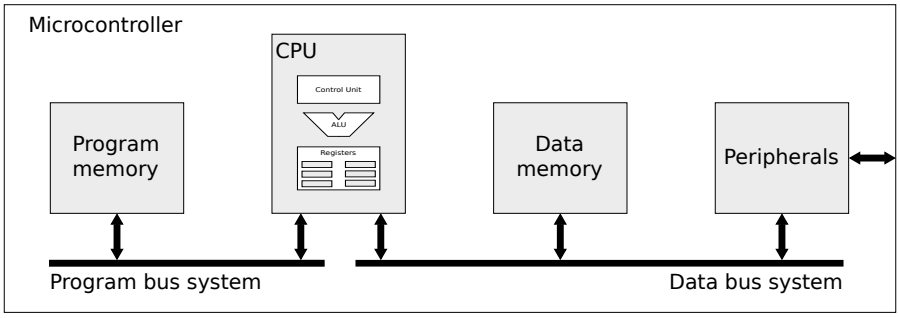

Fig. 2.12.: Harvard architecture.

cost of a slow and difficult process. RAMs are memories that can both be read and modified by the CPU. Even if RAMs offer the benefit of being easily and quickly modifiable, they are generally volatile and thus loose all stored data once the microcontroller is powered down.

The most commonly used types of ROMs in microcontrollers are:

- The *electrically erasable programmable random access memories* (EEPROMs) which are composed of arrays of floating-gate transistors [99] that can be electrically programmed. In these transistors, the Fowler-Nordheim tunneling [70] or hot-carrier injection [171] mechanisms are used to modify the amount of charge stored in the floating gate. This type of memory has a limited life since it can only be reprogrammed a limited number of times (e.g. up to 100 000 write/erase cycles for the EEPROM embedded in the Atmel microcontrollers [46]). Furthermore, even if EEPROMs are electrically reprogrammable, erasing and writing a byte of data is a slow process (e.g. 1,8 ms per byte again for Atmel microcontrollers)

- The *flash memories* which were developed based on the EEPROMs technology. While EEPROMs need two to three transistors per stored bit, flash memories only need one. This makes them less expensive to produce than EEPROMs. On the other hand, unlike EEPROMs which may be erased and written one byte at a time, flash modifications can only be made at the page level (e.g. ATmega644P are arranged in pages of 256 bytes for which up to 4,5 ms are need for a write or erase operation). Again, the life time of such memories is limited (e.g. 10 000 write/erase cycles for the flash memory embedded in the Atmel microcontrollers [46]).

Regarding RAMs, the following types of technologies are the most encountered ones in microcontrollers:

- The *static random access memories* (SRAMs) use memory cells with internal feedback that retain their values as long as power is applied. The word static differentiates SRAMs from *dynamic random access memories* (DRAMs) for which the stored information needs to be periodically refreshed. This type of memory allows fast read and write accesses. It also has relatively low power consumption. It has however a complex structure (six transistors are needed per bit) which makes it expensive to produce in comparison to other types of RAMs.

- The *ferroelectric random access memories* (FRAMs) was recently proposed as a promising alternative to EEPROMs and flash memories. It combines the advantages of non-volatile memories with much faster write speed (e.g. 125 ns per 64-bit word for the 130-nm TI FRAM technology exploited in MSP430FR devices), less power (82 uA/MHz active power in the same technology) and infinite (10^{15}) write-erase cycle performances (FRAM properties are summarized in Table 2.1 and compared to the flash ones). These characteristics allow FRAMs to store any kind of data (i.e. there is no distinction between code and data memory). FRAM stores information through the use of a stable electric dipole found in ferroelectric crystals (insensitive to the magnetic field). The polarization-voltage hysteresis loops for such materials are very similar to the B-H curve of magnetic materials. Exploiting this fact, an FRAM bit cell structure consists of a ferroelectric capacitor containing the crystal. The capacitor is connected to a plate line, bit lines, and a transistor switch to access it. This is also referred to as an 1T-1C memory cell mode (Figure 2.13, left). By contrast, 2T-2C memory cell modes (Figure 2.13, right) would store the data as 2 opposite values in each 1T-1C cell of its structure (similar to what is found in EEPROM for instance). Reading the data from FRAM is performed by placing a voltage on the plate line. The idea is that for every read operation, one tries to set the cell to a 0 state. If the voltage causes the dipoles inside the capacitor to flip

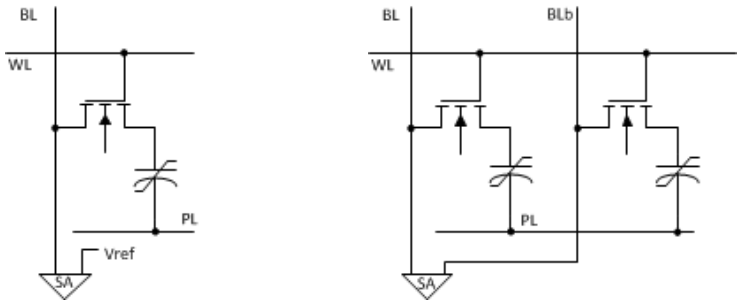

Fig. 2.13.: FRAM bit cell modes: left: 1T-1C structure, right: 2T-2C structure.

Table 2.1.: Comparison of flash memories and FRAMs properties

Flash	FRAM
Program memory only	Unified memory
10^5 reprogramming	10^{15} reprogramming
1 page (256 bytes) at a time	1 byte at a time
4.5 ms per page write or erase	A few clock cycles per byte
Active power: 250-300 μA/MHz	Active power: 82 μA/MHz

its orientation, then a large charge Q is generated on the bit line. On the contrary, if the orientation of the dipole is already negative prior to applying the voltage to the plate line in a read cycle, then the dipole direction does not flip, and only a small charge Q is induced on the bit line. The difference can be measured by a sense amplifier. An important consequence of this description is that FRAM reads are destructive, and therefore require a refresh process. Nevertheless, this process is automatically completed by the controller and therefore transparent to the user.

Description of used microcontrollers

In the next chapters we will mainly use three types of microcontrollers:

- The ATtiny45 [47] is an 8-bit *reduced instruction set computing* (RISC) microcontroller from ATMEL AVR series. The microcontroller uses a Harvard architecture with separate instruction and data memory. Instructions are stored in a 4 kB flash memory (2048×16 bits). Data memory involves the 256-byte SRAM, a register file with 32 8-bit general-purpose registers, and special I/O memory for peripherals like timer, analog-to-digital converter or serial interface. Different direct and indirect addressing methods are available to access data in RAM. Especially indirect addressing allows accessing data in RAM with very compact code size. Moreover, the ATtiny45 has an integrated 256-bytes EEPROM for non-volatile data storage. The instruction-set of the microcontroller contains 120 instructions which are typically 16-bits wide. The instructions are processed within a two-stage pipeline with a pre-fetch and an execute phase. Most instructions are executed within a single clock cycle, leading to a good instructions-per-cycle ratio. Compared to other microcontrollers from ATMEL's AVR series such as the ATmega devices, the ATtiny45 has a reduced instruction set (e.g. no multiply instruction), smaller memories (flash, RAM, EEPROM), no in-system debug capability, and less peripherals. The ATtiny45 also has lower power consumption and is cheaper.

- The ATmega644P [46] is also an 8-bit RISC microcontroller from ATMEL AVR series. This microcontroller has the same properties as the

ATtiny45. The differences mainly reside in the memory sizes (64 kB of flash memory, 4 kB of SRAM and 2 kB of EEPROM), the higher power consumption and the available peripherals. Next to that, the ATmega644P microcontroller supports both signed/unsigned multiplications and fractional format.

- The MSP430FR5739 [92] is a Texas Instruments microcontroller that provides an ultra-low-power 16-bit RISC CPU, and a set of instructions performing operations on either 8 or 16 bits of data. The available non-volatile FRAM memory has a size of 16 kB (other available microcontrollers from the same family have FRAM sizes reaching up to 64 kB, with 128-and 256-kB capabilities already announced).

2.2.2 Software design flow

In general, most systems used in current applications combine (hardwired or reconfigurable) computing devices with software programming. Software is used either to control the dedicated hardware, or to perform some general purpose computations for which standard computing platforms (microcontrollers, micro-processors) provide sufficient performance. The software design flow, illustrated in Figure 2.14, is essentially made of three steps that we describe next.

1. *Description.* As for the hardware flow, a high-level description language is used (e.g. C, C++, Java, ...). It specifies a sequence of (sometimes data-dependent) operations to execute on some fixed hardware in order to perform an algorithmic task. The result of this step is called the software source code.

2. *Conversion.* This step consists in converting the high-level software source code into a lower-level description, denoted as the machine code. As for the hardware flow, it exploits automated tools (i.e. interpreters or compilers) the options of which may significantly affect the final performances.

3. *Loading.* This last step consists in writing the machine source code into the computing device memory, from which instructions are executed. As a result, a fully functional system is obtained and ready to use in actual applications.

2.2.3 Evaluation metrics for software implementations

Similarly to section 2.1.4 where we described common evaluation metrics used for hardware implementations, we will here consider the software evaluation metric and the possible design goals related to them. The main two evaluation

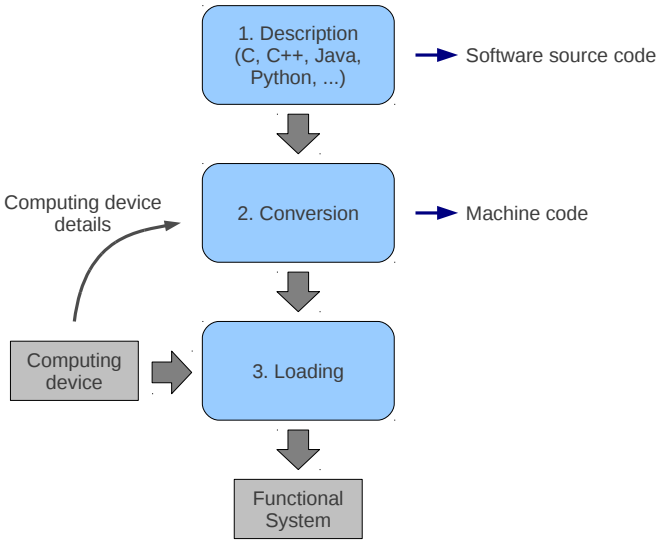

Fig. 2.14.: Software design flow.

metrics we will consider for software implementations are the cycle count and the code size. The first metric represents the number of cycles needed to execute one iteration of the algorithm, while the second one corresponds to the number of bytes that are required in memory for storing the instructions composing the implementation and the intermediate data. The code size is often divided into two parts: the program size (i.e. the part of the code that is stored in a non-volatile memory) and the data size (i.e. the space in volatile memory that is needed while executing the program).

Evaluation of software implementations is a complex task. Once again, but to a smaller extent than for hardware, the metrics used for comparison are relative to each other, which means that it is often necessary to fix the implementation goals beforehand. As an example, the cycle count may be reduced by placing often requested data inside the general purpose registers. Executing instructions from those registers is indeed faster than from the microcontroller memory. This, however, comes at the expense of code size since, unlike operations performed on data stored in memory, operations on registers cannot be looped. As a consequence of the relativity of the metrics, we will usually have to choose between designs having low cycle count and designs having small code size.

Regarding the energy, since the power consumption is more or less fixed for each given microcontroller and not linked to the implementation size, the energy consumed by one iteration of the algorithm is proportional to the cy-

cle count. It is therefore usually sufficient to compare implementations using their cycle counts (or cycle counts divided by output size if we are interested in the energy per bit) when energy consumption is an important factor. Some experimental measurements will however be given in chapter 3 for information purpose. Note that combined metrics such as "code size times cycle count" may also be derived to express the performances of a given software implementation.

To conclude, it is worth mentioning that the implementations that will be discussed in chapters 3 and 4 were made by different designers. Therefore and in order to have comparable results, some common guidelines were provided to all of them. Some of the guidelines could however not always be followed, because of the cipher specifications making them less relevant, e.g. implementing 4-bit S-Boxes using 8-bit tables which is not in line with a small code size goal but allows a more straight foreward implementation on 8-bit microcontrollers. Those guidelines and their exceptions will be thoroughly detailed in the next chapters.

2.3 SIDE-CHANNEL ATTACKS AND COUNTERMEASURES

All the devices we discussed in the previous sections have physical properties such as power consumption, electromagnetic emanations and temperature dissipation. Those properties may be measured and are related to the operations performed inside the device as well as to the data on which the operations are performed. Whenever a microcontroller, FPGA or ASIC is used as a cryptographic device those physical leakages may compromise secret information that is manipulated inside the device. Using the physical emanations of a cryptographic device is known as side-channel analysis or side-channel attacks. In this section, we will first briefly discuss the details concerning side-channel attacks theory. Then, we will describe some usual countermeasures used inside electronic devices in order to protect cryptographic implementations from those attacks.

2.3.1 Side-channel attacks

The goal of side-channel attacks is to recover secret information from a leaking implementation. In this thesis we will only consider side-channel attacks against block ciphers. In this context, an adversary will usually use a divide-and-conquer strategy in order to recover the complete cipher key, i.e. the adversary will target small pieces of the round key (or subkey) one after the other. As illustrated in Figure 2.15 and described in more details in [17] and [122], side-channel attacks follow three main steps:

1. For different subkey candidates j and different plaintexts P, the adversary predicts some intermediate values in the target implementation, e.g. $V_{j,P} = \mathsf{S}(P \oplus j)$.

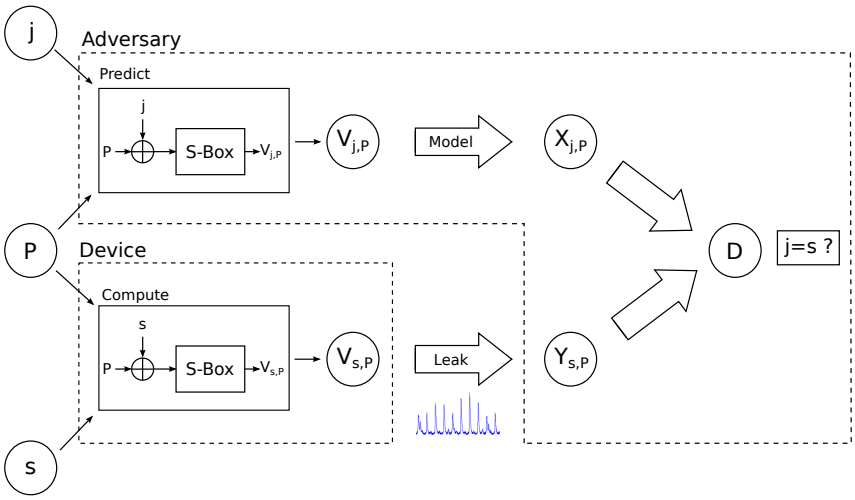

Fig. 2.15.: Illustration of a side-channel attack procedure.

2. For each of those predicted values, the adversary builds a model intended to represent the leakage ($X_{j,P}$). These models can either be built based on hypotheses made on the type of device under attack (e.g. Hamming weight of the predicted value is a frequently used model to represent CMOS-based microcontrollers), or using a profiling phase on the device (i.e. the adversary has the opportunity to build templates that caracterize the power consumption of the device).

3. For each subkey candidate j, the adversary compares the modeled leakage to the leakage measurement ($Y_{s,P}$) that was produced with the same plaintext P and the secret subkey s. The comparison is performed using a distinguisher D. If the attack is successful, the best comparison result should be obtained for the key candidate $j = s$. Several statistical distinguishers have been proposed over the last decades. Among others, the Pearson's correlation coefficient [37], the Bayesian distinguisher (as used in the template attacks [41]) and the mutual information [76] are examples of such distinguishers.

2.3.2 Protection against side-channel attacks

In order to prevent side-channel attacks or at least to make them more difficult to execute, various countermeasures have been introduced over the years. A strict classification is hard. However two frequently encountered classes of countermeasures are hiding and masking, which will be discussed in the following paragraphs. The goal of these countermeasures is to make the power consumption of a cryptographic device independent of the intermediate values

that are processed. This is usually achieved using randomness and by acting either on the leaking signal or on the data that is processed.

Another possible category, which will not be considered here, includes countermeasures such as leakage resilient cryptography and re-keying [61, 111, 132, 177, 178]. These countermeasures are based on the idea of preventing side-channel attacks at the protocol level by for example updating the key. Note finaly that, in this thesis, we mainly focus on countermeasures that can be implemented at a software level. There are however a lot more existing propositions of countermeasures that can be applied either to the software or to the hardware level. For more information on that topic, some references such as [121] are available in the literature.

Hiding

In the case of hiding, the cryptographic device is protected by making it difficult for an attacker to find exploitable information in power traces. The algorithm is usually executed in the same way as for an unprotected device but the operations are hidden either by randomizing the power consumption of the device (e.g. by performing the operations at different moments in time during each execution or by adding dummy operations at random points in the execution [188]) or by making the power consumption constant over the whole execution (e.g. by using hardware logic cells for which the power consumption is independent of the processed data and performed operations [151, 186]). Here, we will only be concerned by the first group of hiding scenarios and, in particular, we will evaluate the shuffling countermeasure in more details in Chapter 5. The core idea behind shuffling [87] is to randomize the power consumption by executing independent operations in a different order at each execution. That way, the points of interests are spread over several cycles,

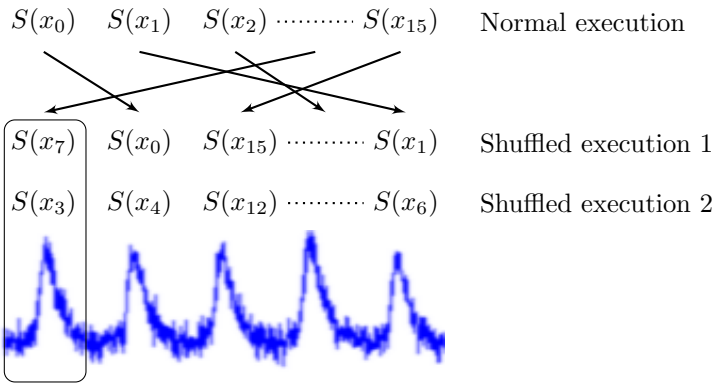

Fig. 2.16.: Representation of the shuffling countermeasure applied to the S-Box operations.

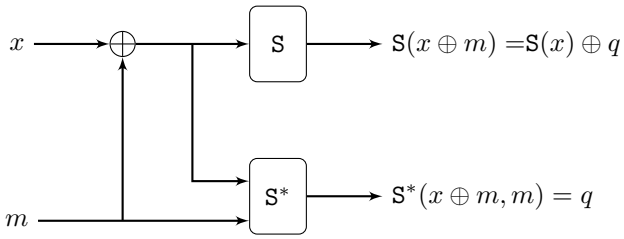

Fig. 2.17.: Representation of a 1st-order masking scheme applied to an S-Box

which forces the adversary to combine several points in order to recover useful information. This is illustrated in Figure 2.16 for the case of the S-Box operations.

Masking

The masking countermeasure makes the power consumption independent of the processed data by randomizing them. That is, for a dth-order masking scheme, every sensitive variable x processed during the computation is randomly split into $d+1$ shares, e.g. $x = x_0 \oplus x_1 \oplus \ldots \oplus x_d$ [163]. With masking countermeasure, the execution of the algorithm needs to be slightly modified: all the linear operations must be applied to the $d+1$ shares and non-linear operations must be adapted in a way that the masks can be removed at the end. The case of an S-Box is illustrated in Figure 2.17 where a correction function $\mathtt{S}^*$ is used in order to produce an output mask q such that $\mathtt{S}(x \oplus m) = \mathtt{S}(x) \oplus q$.

PART II

LIGHTWEIGHT IMPLEMENTATIONS AND COUNTERMEASURES

CHAPTER 3

LIGHTWEIGHT IMPLEMENTATIONS OF BLOCK CIPHERS

3.1 INTRODUCTION

Lightweight cryptography is an active research direction, as witnessed by the number of algorithms aiming at "low-cost" implementations designed over the last years. Looking at block ciphers, the list includes (but is not limited to) DESXL, HIGHT, ICEBERG, KATAN, KLEIN, LED, mCrypton, NOEKEON, Piccolo, PRESENT, SEA and TEA. Although these algorithms are useful and inventive in many ways, determining which one to use in which application with good confidence can be difficult. One first reason for this is that the very definition of low-cost is hard to capture, as it is highly dependent on the target platform. As an illustration, operations that are cheap in hardware (e.g. wire crossings) may turn out to be annoyingly expensive in software. In fact, even for a given technology, various criteria could be considered to evaluate the low-cost nature of different algorithms. The implementation size (measured in gates, program memory, ...) generally comes in the first place, but power or energy can be more reflective in certain application scenarios. Besides, lightweight cryptography has mainly been developed through several independent initiatives, over an already long time period. This is in contrast with the design of standard algorithms for which the selection was the result of an open competition. One outcome of the AES and SHA-3 competitions is the publication of well motivated comparative studies. Taking the example of the AES, several works can be mentioned for both hardware and software implementations [66, 72, 169, 196]. By contrast, only a few evaluations of lightweight algorithms are available in the literature. For example, in the hardware case, the companion paper of the KATAN algorithm includes gate counts and throughput estimations for several ciphers [38], but they consider different technologies. Regarding software implementations, notable exceptions include B. Poettering's open-source codes for AES [150],the XBX frameworks [197] and an interesting survey of lightweight cryptography implementations [64]. Un-

fortunately, these references are still limited by the number of ciphers under investigation and the fact that in some cases the source code is not available for evaluation.

In this chapter, we evaluate the performances of several block ciphers with different block and key sizes. The software implementations of 12 ciphers are investigated in Section 3.2, while the hardware implementations of 6 block ciphers are compared in Section 3.3. Finally, we conclude with a summary of the important observations made during evaluation process and with general remarks about cipher design in Section 3.4.

3.2 SOFTWARE EVALUATION OF LIGHTWEIGHT BLOCK CIPHERS

In this section, we provide performance evaluations for low-cost block ciphers, and investigate their implementation on an ATMEL AVR ATtiny45 device [9]. The goal of this work is to extend the benchmarking of 12 lightweight and standard block ciphers, namely AES, DESXL, HIGHT, IDEA, KASUMI, KATAN, KLEIN, mCrypton, NOEKEON, PRESENT, SEA, TEA, and make their implementation available under an open-source license. To the best of our knowledge, four of these algorithms (KASUMI, KLEIN, mCrypton and KATAN) are implemented for the first time on an 8-bit platform. The ciphers were selected according to three criteria: all selected candidates should (a) give no indication of flawed security, (b) be freely usable without patent restrictions, and (c) likely result in lightweight implementations with a footprint of less than 256 bytes of RAM and 4 KB of code size for a combined encryption and decryption function.

In order to make comparisons as meaningful as possible, the guidelines for evaluations of hardware implementations proposed in [73] are adapted to our software context. Yet, as the project involves 12 different designers, we also acknowledge that some biases can appear due to slightly different implementation choices. Hence, as usual for performance evaluations, looking at the source codes is essential in order to properly understand the reasons of different performance figures. Overall, we hope that this initiative can be used as a first step in better analyzing the performances of block ciphers in a specific but meaningful class of devices. We also hope that it can be used as a starting point to further develop cryptographic libraries for embedded platforms and, in the long run, add security against physical attacks (e.g. based on faults or side-channel leakage) as another evaluation criteria.

This section is structured as follows. First, Section 3.2.1 establishes our evaluation methodology and the used setup. Section 3.2.2 provides succinct descriptions and motivation of the implementation choices made by the designers. Finally, performance evaluations are given in Section 3.2.3. A webpage containing all the open-source codes is available at [62].

3.2.1 Methodology and setup

In order to be able to compare the performances of the different ciphers in terms of speed, memory space and energy, the developers were asked to respect a list of common constraints, detailed hereunder.

1. The code has to be written in assembly, in a single file. It has to be commented and easily readable, for example, giving the functions the name they have in their original specifications.
2. The cipher has to be implemented in a low-cost way, minimizing the code size and the data-memory use.
3. Both encryption and decryption routines have to be implemented.
4. Whenever possible, and in order to minimize the data-memory use, the key schedule has to be computed "on-the-fly". The computation of the key schedule is always included in the algorithm evaluations.
5. The encryption process should start with plaintext and key in data memory. The ciphertext should overwrite the plaintext at the end of this process (and vice versa for decryption).
6. The target device is the 8-bit microcontroller ATtiny45 from ATMEL's AVR device family. It has a reduced instruction set and does not provide a hardware multiplier.
7. The encryption and decryption routines have to be called by a common interface.

The SEA reference code was sent as an example to all designers, together with the common interface (also provided at [62]).

The basic metrics considered for evaluation are code size, RAM size, cycle count in en- and decryption, and energy consumption. From these basic metrics, a combined metric is extracted (see Section 3.2.3). For the energy-consumption evaluations, each cipher is programmed and executed on an ATtiny45 mounted on a power-measurement board. A 22-Ohm shunt resistor is inserted between the V_{dd} pin and the 5V́ power supply, in order to measure the current consumed by the controller while encrypting. The common interface generates a trigger at the beginning and end of each encryption. The power traces are measured between those two triggers using an oscilloscope that is equipped with a differential probe. We average one hundred encryption traces for each energy evaluation using randomly generated plaintexts and keys for each encryption. The average energy consumed by an encryption is deduced by integrating the measured current.

Finally note that the 12 ciphers are implemented by 12 different designers, with slightly different interpretations of low-cost optimizations. As a result, some of the guidelines could not always be followed, because of the cipher specifications making them less relevant. In particular, the following exceptions deserve to be mentioned.

(1) Due to its high complexity (see Section 1.1.3 for explaination), the key scheduling of IDEA is not computed "on-the-fly" but precomputed.
(2) The key in KATAN has to be restored externally for subsequent invocations.
(3) In order to get a good time/memory trade-off, the designers of KLEIN and PRESENT decided to implement their 4-bit S-Boxes as 8-bit LUTs. Accessing those tables therefore gives the output of two consecutive S-Boxes in one cycle. The designer of mCrypton, on his side, preferred to implement his 4-bit S-Box as a 16-byte table, leaving the four MSBs of each byte unused.

3.2.2 Implementation details

AES. The code is written following the standard specifications and operates on a state matrix of 16 bytes. In order to improve performances, the state is stored in 16 registers, while the key is stored in RAM. In addition, five temporary registers are used to implement the MixColumn step. The S-Box and the round constants were implemented as simple LUTs. The multiplication operation needed by the MixColums layer is computed with shift and XOR instructions.

DESXL. In order to keep code size small, a function which can compute all permutations and expansions depending on the calling parameters is used. This function is also capable of writing 6-bit outputs for direct usage as S-Box input. Because of the bit-oriented structure of the permutations which are slow in software, this function is the performance bottleneck of the implementation. The rest of the code is a straight forward application of the specifications. Besides the memory requirement for plain-/ciphertext and the keys $k, k1, k2$, 16 bytes of additional SRAM are required for the round key and the state. The S-Box and all permutation and expansion tables are stored in flash memory and processed directly from there.

HIGHT. The implementation choices are oriented in order to limit the code size. First, the intermediate states are stored in RAM at each round, and only two bytes of text and one byte of key are loaded in registers at a time. This way, it is possible to re-use the same code fragment four times per round. Next, the byte rotation at the output of the round function is integrated in the memory accesses of the surrounding functions, thus minimizing temporary storage and gaining cycles. Eight subkey bytes are generated once every two rounds, and are stored in RAM. Finally, except for the mod 2^8 additions that are replaced by mod 2^8 subtractions and some other minor changes, the same functions as in encryption are used in decryption.

IDEA. This cipher is implemented including a precomputed key schedule performed by separate functions for encryption and decryption, prior to the actual cipher operation. During cipher execution the precomputed key (104 bytes) is then read byte by byte from the RAM. The plaintext/ciphertext and

the internal state are kept completely in 16 registers and 9 additional registers are used for temporary computations and counters. IDEA requires a 16-bit modular multiplication as basic operation. However, in the AVR device used in this work, no dedicated hardware multiplier unit is available. Multiplication was therefore emulated in software resulting in a data-dependent execution time of the cipher operation and an increased cycle count (about a factor of 4) compared to an implementation made for devices having dedicated hardware multipliers.

KASUMI. The code is written following the functions described in the cipher specifications. During the execution, the 16-byte key remains stored in the RAM, as well as the 8-byte running state. This allows using only 12 registers and 24 bytes of RAM. Some rearrangement are done to skip unnecessary moves between registers. The 9-bit S-Box is implemented using an 8-bit LUT, with the MSBs concatenated in a secondary 8-bit table. The 7-bit S-Box is implemented using an 8-bit LUT, leaving the MSBs of this table unused. The round keys are derived "on-the-fly". Decryption is very similar to encryption, as usual for a Feistel structure.

KATAN-64[1]**.** The main optimization goal is to limit the code size. The entire state of the cipher is kept in registers during operation. In order to avoid excessive register pressure, the inputs and outputs are stored in RAM, and this RAM space is used to backup the register contents during operation. Only three additional registers need to be stored on the stack. The fact that three rounds of KATAN can be run in parallel is not used in this implementation. Doing so would require more complicated shifting and masking to extract bits from the state, and thus significantly increase the code size, for little or no performance gain. As the KATAN key schedule is computed "on-the-fly", the key in RAM is clobbered and needs to be restored externally for subsequent invocations. Keeping the master key in RAM would require 10 additional words (note that KTANTAN key schedule does not modify the key, so it does not have this limitation). In order to implement the non-linear functions efficiently, addition instructions were used to compute several logical ANDs and XORs in parallel through carefully positioning the input bits and using masking to avoid undesired carry propagation.

KLEIN-80. Despite the goal of small footprint, the 4-bit involutive S-Box is stored as an 8-bit LUT to save clock cycles. As it can be used in both encryption and decryption, this corresponds to a natural trade-off between code size and processing speed (a similar choice is made for mCrypton and PRESENT, see next paragraphs). In order to save memory usage during processing, the MixNibbles step (borrowed from AES MixColumns) is implemented by a single function without using LUTs. Overall, 29 registers are used during the computations. Among them, 8 registers correspond to the intermediate state, 10

[1] All six variants of the KATAN/KTANTAN family are supported via conditional assembly. As previously mentioned, our performance evaluations focus on the 64-bit version of KATAN.

to the key scheduling, 2 to the round counter and 9 are used as temporary storage.

mCrypton. The reference code directly follows the cipher specification. The implementation aims for a limited code size. Therefore, as much code as possible is reused for decryption and encryption. In addition, up to 20 registers are used during the computations to reduce the cycle count. 12 registers are used to compute the intermediate state and the key scheduling, 6 registers for temporary storage, one for the current key scheduling constant and one for the round counter. After each round the modified state and key scheduling state are stored in RAM. The round key is derived from the key scheduling state and is temporarily stored in RAM. The four 4-bit S-Boxes are stored in four 16-byte tables, wasting the 4 most significant bits of each entry, but saving cycle counts. The round constants used in the key scheduling algorithm are stored in an 8-bit table.

NOEKEON. The implementation aims to minimize the code size and the number of utilized registers. During execution of the block cipher, input data and cipher key are stored in the RAM (32 bytes are required). In that way, only 4 registers are used for the running state, one register for the round counter, and three registers for temporary computations. One of the address registers is used for indirect addressing of the data in the RAM. Similar to the implementation of SEA (detailed below), using more registers for the running state will decrease the cycle count, but will also increase the code size because of a less generic programming. For decrypting data, the execution sequence of the computation functions is changed, which leads to a very small increase in code size.

PRESENT. The implementation is optimized in order to limit the code size with throughput as secondary criteria. State and round key are stored in the registers to minimize accesses to RAM. The S-Boxes are stored as two 256-byte tables, one for encryption and one for decryption. This allows for two S-Box lookups in parallel. However, code size can easily be reduced if only encryption or decryption is performed. Besides, using a single 16-byte table for implementing the two S-Boxes could halve the overall code size, but would significantly impact encryption times. The code for permutation, which is the true performance bottleneck, can be used for both encryption and decryption.

SEA. The reference code is written following directly the cipher specifications. During its execution, plaintexts and keys are stored in RAM (accounting for a total of 24 bytes), limiting the register consumption to 6 registers for the running state, one register for the round counter and three registers of temporary storage. Note that higher register consumption would allow decreasing the cycle count at the cost of a less generic programming. The S-Box is implemented using its bitslice representation. Decryption uses exactly the same code as encryption, with "on-the-fly" key derivation in both cases.

TEA. Implementing TEA is almost straightforward due to the simplicity of the algorithm. The implementation is optimized to limit the RAM usage and

code size. As far as RAM is concerned, we only use the 24 bytes for plaintext and key storage, with the ciphertext overwriting the plaintext in RAM at the end of the process. The only notable issue regarding implementing TEA concerns rotations. TEA is optimized for a 32-bit architecture and the fact that only 1-position shift and rotations are available on the ATtiny, plus the need to propagate carries, made these operations slightly more complex. In particular, 5-position shifts are optimized by replacing them by a 3-position shift in the opposite direction and recovering boundary carries. Nonetheless, TEA proves to be very easy to implement, resulting in a compact code of 648 bytes.

3.2.3 Performance evaluation

We considered 6 different metrics: code size (in bytes), RAM use (in bytes), cycle count in encryption and decryption, energy consumption (in μJ) and a combined metric, namely the product of code size with cycle count normalized by the block size. The results for our different implementations are given in Table 3.1 and are compared in Figures 3.1 to 3.6 for which a hatched bar means that the value is out of scale. We detail a few meaningful observations below.

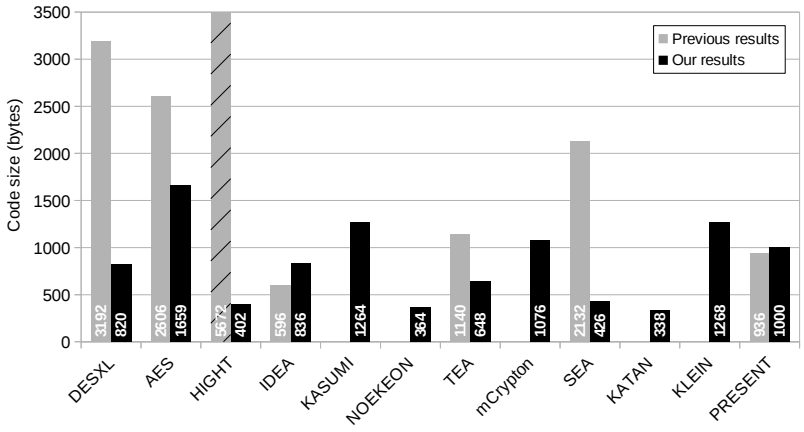

Fig. 3.1.: Code size: comparison with previous work [64].

First, as our primary goal was to consider compact implementations, we compared our code sizes with the ones listed in [64]. As illustrated in Figure 3.1, we reduced the memory footprint for most investigated ciphers, with specially strong improvements for DESXL, HIGHT and SEA. The code sizes of our implementations may also be compared using the same figure. The frontrunners are HIGHT, NOEKEON, SEA and KATAN (all take less than 500 bytes of ROM). One can notice the relatively poor performances of mCrypton,

Table 3.1.: Performance evaluation of our implementations on the AVR ATtiny45 microcontroller. Results obtained in this work are given in **bold** face.

Cipher	Block Size [bits]	Key Size [bits]	Code Size [bytes]	RAM [bytes]	Cycles (enc+key)	Cycles (dec+key)	Energy [μJ]
AES	128	128	1659	33	4557	7015	19,2
AES [64]	128	128	2606	0	6637	7429	-
DESXL	64	184	820	48	84602	84602	348,9
DESXL [64]	64	184	3192	0	8531	7961	-
HIGHT	64	128	402	32	19503	20159	79,8
HIGHT [64]	64	128	5672	0	2964	2964	-
IDEA	64	128	836	232	~8250	~22729	34,3
IDEA [64]	64	128	596	0	2700	15393	-
KASUMI	64	128	1264	24	11939	11939	47,6
KATAN	64	80	338	18	72063	88525	289,2
KLEIN	64	80	1268	18	6095	7658	25,1
mCrypton	64	96	1076	28	16457	22656	68
NOEKEON	128	128	364	32	23517	23502	95,9
PRESENT	64	80	1000	18	11342	13599	45,3
PRESENT [64]	64	80	936	0	10723	11239	-
SEA	96	96	426	24	41604	40860	173,7
SEA [64]	96	96	2132	0	9654	9654	-
TEA	64	128	648	24	7408	7539	30,3
TEA [64]	64	128	1140	0	6271	6299	-

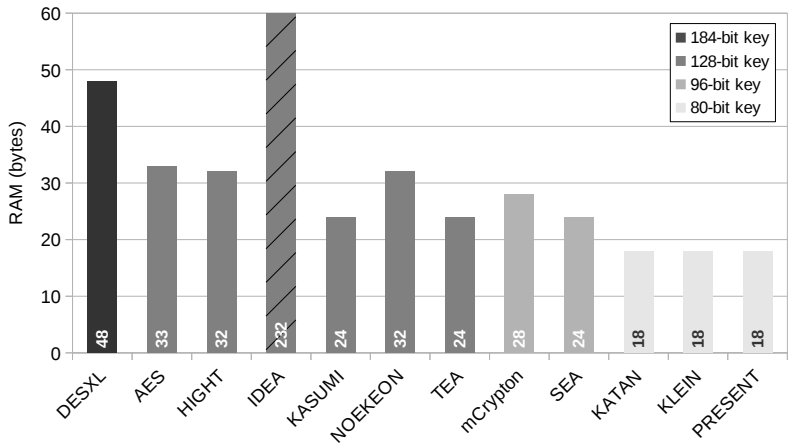

Fig. 3.2.: Performance evaluation: RAM use.

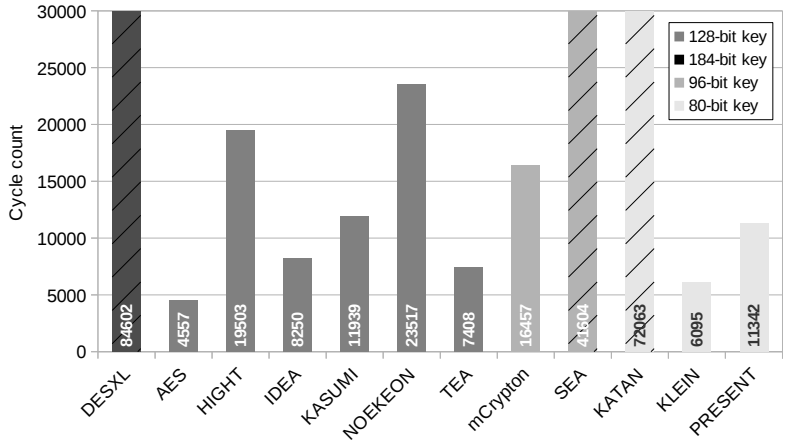

Fig. 3.3.: Performance evaluation: cycle count (encryption).

PRESENT and KLEIN. This can be explained in part by the more hardware-oriented flavor of these ciphers (e.g. the use of bit permutations or manipulations of 4-bit nibbles is not optimal when using 8-bit microcontrollers). As expected, standard ciphers such as the AES and KASUMI are more expensive, but only up to a limited extent (both are implemented in less than 2000 bytes of ROM).

The RAM use in Figure 3.2 first exhibits the large needs of IDEA regarding this metric (232 words) that are essentially due to the need to store a precomputed key schedule for this cipher. Besides, and following our design guidelines, the RAM use essentially reflects the size of the intermediate state

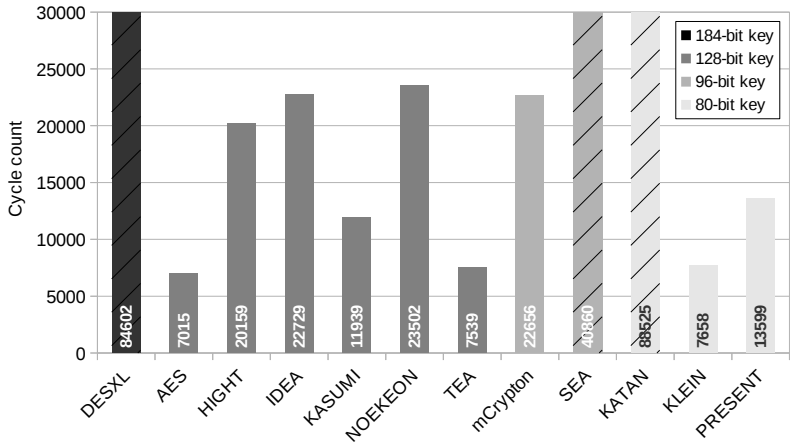

Fig. 3.4.: Performance evaluation: cycle count (decryption).

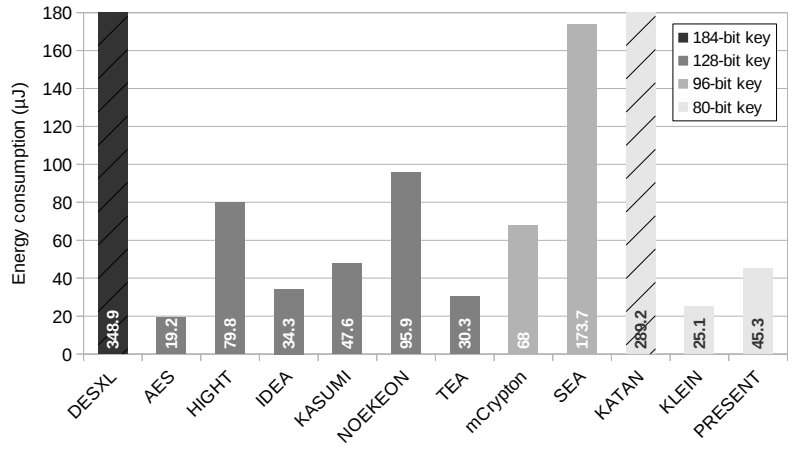

Fig. 3.5.: Performance evaluation: energy consumption (encryption).

that has to be stored during the execution of the algorithms. Note that for the AES, this contrasts with the "Furious" implementation [150] for which the subkeys are all stored in RAM, increasing its memory usage to 192 bytes. This also explains our slightly reduced performances in terms of cycle count for this cipher.

The cycle count in Figure 3.3 clearly illustrates the performance loss that is implied by the use of simple round functions in most lightweight ciphers. This loss is critical for DESXL and KATAN where the large number of round iterations leads to cycle counts beyond 50,000 cycles. It is also large for SEA, NOEKEON and HIGHT. By contrast, these metrics show the excellent efficiency

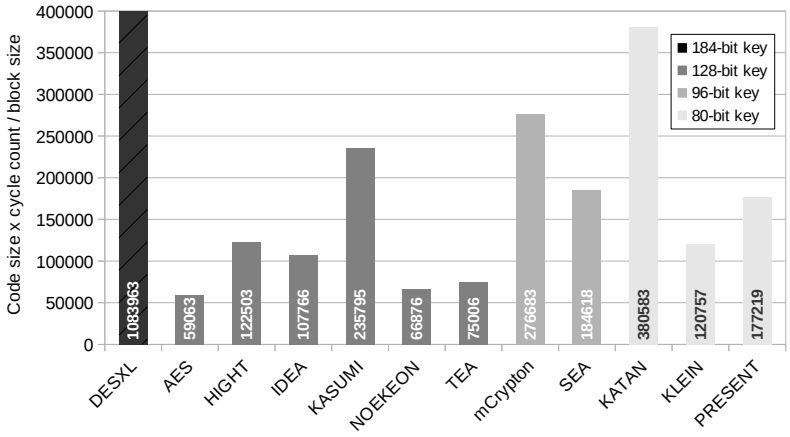

Fig. 3.6.: Performance evaluation: combined metric (encryption).

of the AES. Cycle count for decryption (Figure 3.4) shows similar results, with noticeable changes. Most visibly, IDEA decryption is much less efficient than its encryption. The AES also shows non-negligible overhead to decrypt. In contrast, a number of ciphers behave identically in encryption and decryption, e.g. SEA where the two routines are almost identical.

As expected, the energy consumption of all the implemented ciphers (Figure 3.5) is strongly correlated with the cycle count, confirming the experimental results in [56]. However, slight code dependencies can be noticed. It is an interesting scope for research to investigate whether (and to what extent) different coding styles can further impact the energy consumption.

Lastly, the combined metric in Figure 3.6 first shows the excellent size vs. performance trade-off offered by the AES. Among the low-cost ciphers, NOEKEON and TEA exhibit excellent figures as well, probably due to their very simple key scheduling. This comes at the cost of possible security concerns regarding related key attacks. HIGHT and KLEIN seems to provide a good trade-off between code size and cycle count. A similar comment applies to SEA, where parts of the overhead comes from a complex key scheduling algorithm (key rounds are as complex as the rounds for this cipher). Despite their hardware-oriented nature, PRESENT and mCrypton offer decent performance in 8-bit devices as well. KATAN falls a bit behind, mainly because of its very large cycle count. Only DESXL appears not suitable for such an implementation context.

3.3 HARDWARE EVALUATION OF LIGHTWEIGHT BLOCK CIPHERS

In this section, we compare the hardware performances of 6 block ciphers, with different block and key sizes. Namely, we considered the AES and NOEKEON for 128-bit blocks and keys, HIGHT and ICEBERG for 64-bit blocks and 128-bit keys, and KATAN and PRESENT for 64-bit blocks and 80-bit keys. This choice of algorithms was motivated by having different block and key sizes, together with different styles of key scheduling and decryption. The section is organized as follows. First, we analyze hardware design choices and describe different architectures for encryption, decryption and encryption/decryption, with and without round unrolling and parallelization. This allows us to quantify the combinatorial cost and delays of the different ciphers, and to analyze their respective implementations. We then evaluate in Section 3.3.2 different figures of merits for these 6 candidates, with a particular focus on the energy efficiency. Next, in Section 3.3.3, we study the tuning of physical parameters, and evaluate the impact of frequency/voltage scaling on our comparisons. Doing so, we investigate the relevance of the energy per bit as a comparison criteria for lightweight block ciphers, i.e. its independence with respect to hardware design choices and frequency/voltage scaling. In other words, we question the extent to which such a metric reflects algorithmic design choices and discuss its possible biases. We answer positively and argue that it nicely summarizes the "energy efficiency" of an algorithm. We also show that the informativeness of this metric is further improved if correlated with the "performance efficiency" (usually measured with a throughput over area ratio). The results of those experiments will help us try to extract useful suggestions for new lightweight cryptographic algorithms. These will be discussed, together with conclusions about software implementations, in Section 3.4.

3.3.1 Methodology

In order to make our performance evaluations as relevant as possible, we defined a strict methodology for all our implementations. It defines requirements on the target architectures, their interface and the implementation flow.

Regarding architectures, and for all the investigated ciphers, we considered encryption, decryption and encryption/decryption designs. The reference point of our evaluations is a standard looped implementation of the AES, performing one encryption in 12 cycles, taking advantage of the efficient S-Box representation of Mentens et al. [129]. This choice was mainly motivated by our low-energy consumption goal. Further reduction of the area (e.g. with 8-bit or 32-bit architectures) would lead to less energy-efficient designs. We also thought that the throughputs of these AES implementations (of a few Gbps) were large enough for a wide range of applications. Next, for all the investigated lightweight ciphers, we analyzed the generic unrolled architecture depicted in Figure 3.7, where N_r rounds are executed per clock cycle. Having

at least one full round implemented was again motivated by our low-energy consumption objective. We used this generic architecture in order to determine the number of lightweight cipher rounds that are needed to consume the same area, or that require the same delay as an AES round. Besides, they are also interesting architectures for very low-latency implementations. As unrolling without adding pipeline is generally a suboptimal choice regarding the critical path, we further considered two implementation scenarios. In the first one, we assumed a clock frequency of 100MHz (determined by the system): it corresponds to a context where such an unrolling is indeed motivated by external constraints. Next, we estimated the maximum clock frequency. In this second case, we further investigated the impact of parallel (or pipelined) architectures. In order to exhaustively analyze our large design space (various N_r values, encryption vs. decryption vs. encryption/decryption, $f = 100$MHz vs. f_{max}), we heavily relied on generic VHDL/Verilog programming. We additionally used a common (generic as well) interface for all the ciphers, with plaintext/ciphertext (resp. key) port width corresponding to the block (resp. key) size, and a simple handshaking mechanism to control the flow.

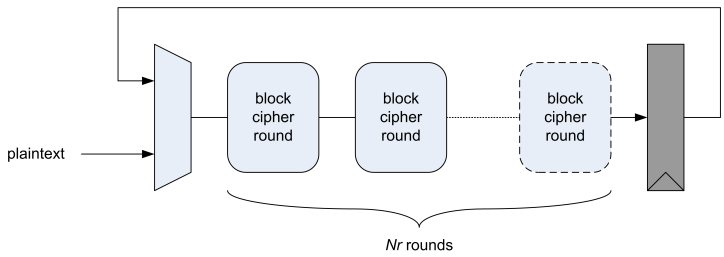

Fig. 3.7.: Unrolled architectures with various number of rounds.

Regarding the synthesis environment, we operated in two steps. In the first place, and in order to study the impact of architectural choices, we investigated the previously defined operating frequencies (i.e. $f = 100$MHz and f_{max}) at 1.2V, i.e. the nominal supply voltage for the technology used (see Section 3.3.2). As previously mentioned, the maximum frequency depends on the synthesis and place-and-route, since the CAD tool can further optimize a design to reach a target timing constraint at the cost of an area increase. Thus, we precisely defined the maximum frequency as the frequency obtained when the area of the design has increased by 10% compared to the unconstrained design. This step allowed us to identify the most efficient architectures for each lightweight cipher. Next, for this reduced set of architectures, we analyzed the possibilities of frequency/voltage scaling by carefully tuning the supply voltage (in Section 3.3.3).

All the ciphers were implemented following a classical ASIC flow. We used a commercial 65-nanometer CMOS low-power technology. Synthesis was performed using the Synopsys tools suite. We used the switching activity annota-

tion, by means of behavioral simulation, in order to extract realistic power and energy figures. Finally, the supply voltage exploration was performed using the standard cells library that was re-characterized at different V_{dd}'s.

3.3.2 Implementation results at fixed V_{dd}=1.2V

Using the previously defined methodology, we first reported the selected performance metrics of our different syntheses at 1.2V supply voltage in Figures 3.8 to 3.17 where each point in the curves corresponds to a different unrolling parameter N_r. These figures allow us to evaluate the efficiency of the different ciphers implemented. In this section, we report on a number of useful observations regarding both hardware design and algorithmic design issues.

As a starting point, we looked at the area curves (given for $f = 100$MHz in Figure 3.8). In general, one would expect the circuit size to increase linearly with the number of rounds unrolled. However, in the case of lightweight ciphers, we observed that a number of rounds may be needed before such a linear dependency appears. This fact is in direct relation with the limited combinatorial cost of the rounds in certain ciphers (most visibly, KATAN). That is, if the cost of a round is small in comparison with the state registers and control logic, doubling the number of rounds unrolled will not double the consumed area. A similar behavior is observed for the critical path in Figure 3.9. If the rounds are simple enough for this critical path to be in the control logic, then doubling the amount of rounds unrolled will not result in cutting the maximum frequency by two. Again, this effect is amplified for KATAN, as its round computations only affect a few bits, the other ones being routed from the state register to itself. Overall, these figures recall that the definition of a round is arbitrary: several rounds of a lightweight ciphers are generally needed to reach the cost and delay of an AES round.

Looking at the throughput curves first confirms that in general, unrolling an implementation without pipelining mainly makes sense if the clock frequency is fixed below the maximum one (see Figure 3.10), e.g. because of system constraints. Yet, we also remark that for some ciphers, unrolling a few rounds without pipeline improves the throughput at maximum frequency too (see Figure 3.11). This is a consequence of "simple rounds" and the previously mentioned non-linear increase of the critical path for low N_r values. Note that even for KATAN, the throughput starts to decrease beyond $N_r = 2^6$ (this data is not included in the figures, for visibility reasons). Besides, increasing N_r at maximum frequency does not lead to the expected constant curves. This is explained by a detrimental side-effect related to the overhead cycles required to charge/discharge the plaintext and master key in their registers. Namely, this overhead becomes more significant with the number of implemented rounds (i.e. when the number of clock cycles per encryption becomes small). Note that the impact of this interfacing drawback is stronger for NOEKEON and

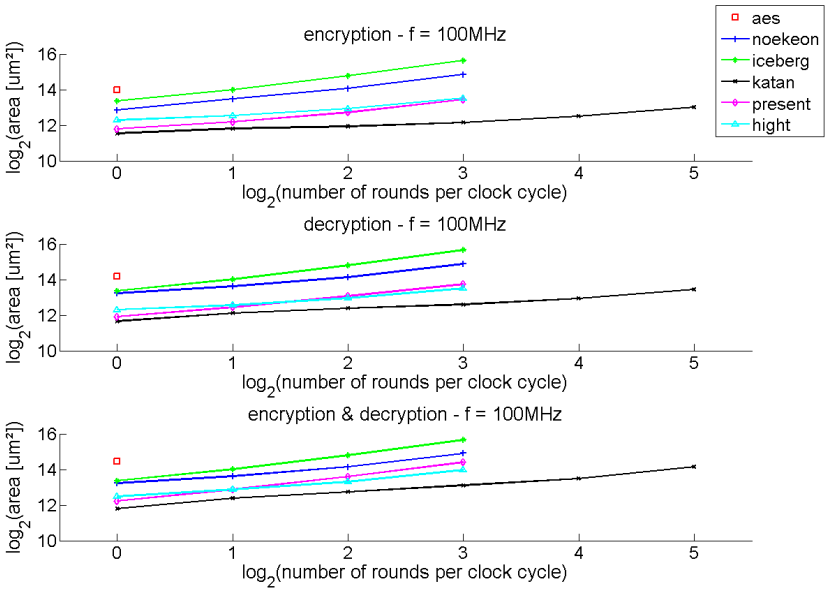

Fig. 3.8.: $V_{dd} = 1.2$V, $f = 100$MHz: area.

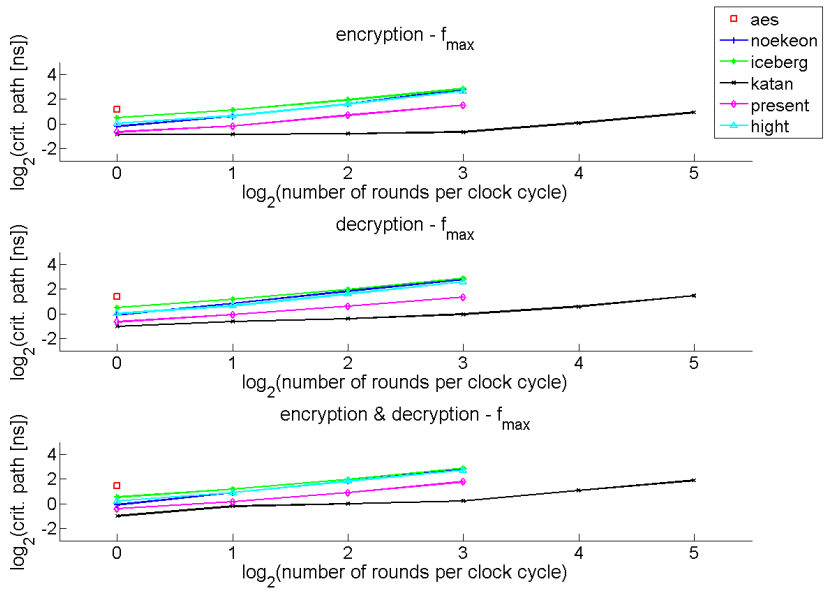

Fig. 3.9.: $V_{dd} = 1.2$V, f_{max}: critical path.

HIGHT, as they respectively account for 2 and 3 cycles for these ciphers (one for loading the data, one per initial/final transformation).

The average power implementation results also exhibit different conclusions for $f = 100$MHz and f_{max} (see Figures 3.12 and 3.13). In the first case, they are dominated by the switching activity in the circuit, that increases with N_r. Hence, the power is correlated with the circuit size in this case. By contrast at maximum frequency, unrolling is either neutral or implies a reduction of the average power, when the maximum frequency decreases with N_r faster than the area (e.g. for HIGHT).

Interestingly, the energy per bit (given for f_{max} in Figure 3.14) is remarkably similar in our two frequency contexts, because it is dominated by the switching activity in the selected low-leakage technology. This confirms that it is a reasonably discriminant metric for algorithmic comparisons. It is also quite correlated with the throughput over area metric at 100MHz in Figure 3.15, i.e. when unrolling the algorithms affects the throughput and area in opposite directions, with close to equivalent impact. Quite naturally, this correlation vanishes at maximum frequency, due to the inefficiency of unrolling without pipeline. Finally, the previously mentioned side-effects (such as overhead cycles and unbalanced use of logic and memory) are naturally reflected in these curves as well.

A summary of our implementation results regarding algorithm efficiency (both in terms of performances and energy) is depicted in Figure 3.16, where the energy per bit is represented in function of the throughput over area ratio, for our different architectures. Such figures naturally require a few cautionary remarks. First, they have to be interpreted with care, as they only provide a big picture of the implementation efficiency. Practical case studies may focus specifically on different combinations of metrics. Second, even if assuming that efficiency is indeed the design goal, comparing algorithms is difficult as their performances are sometimes close, and depend on the architectures (e.g. encryption-only, decryption-only and encryption/decryption designs lead to different ratings). Yet, we believe that a few interesting conclusions can be extracted that we now detail.

Starting with encryption designs, the comparison roughly suggests NOEKEON $\geq$ PRESENT $\approx$ KATAN $\geq$ HIGHT $\approx$ AES $\geq$ ICEBERG. This ordering is explained by different factors. Maybe the most important one is the significant differences in the key scheduling. At the extremes, NOEKEON does not have any, while for ICEBERG, the key scheduling is as complex as the encryption rounds. Next, the respective block and key sizes strongly matter as well. Having larger key sizes naturally implies lower efficiency in general (but theoretically provides improved security). Less obviously (and less significantly), smaller block sizes are also negative for efficiency. For example, working on 128-bit blocks instead of 64-bit ones doubles the datapath size, but it rarely implies doubling the overall cost (thanks to the strong diffusion layers in modern ciphers). Considering the decryption architectures allows putting

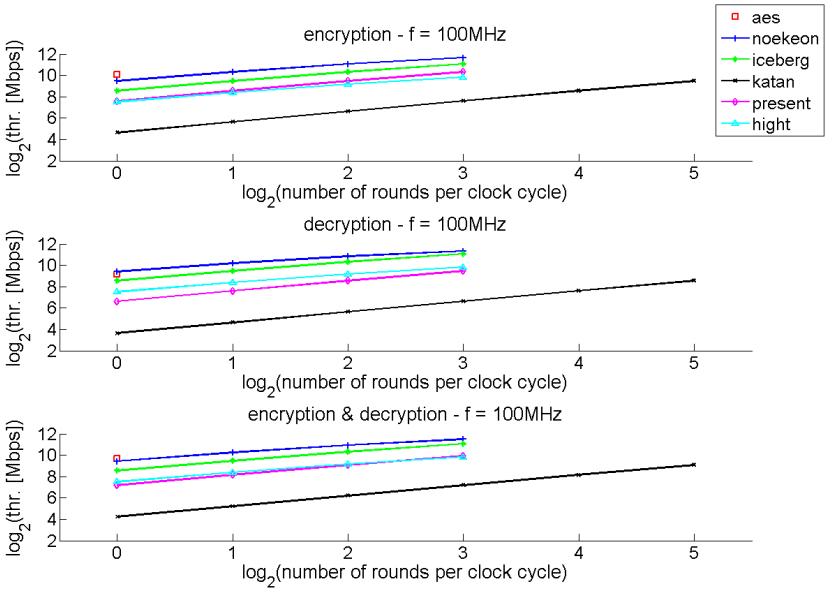

Fig. 3.10.: $V_{dd} = 1.2\text{V}$, $f = 100\text{MHz}$: throughput.

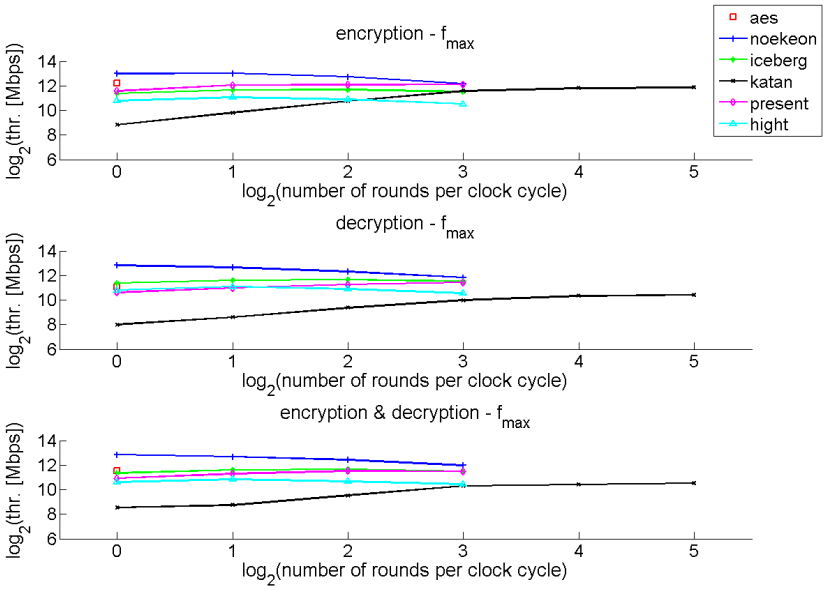

Fig. 3.11.: $V_{dd} = 1.2\text{V}$, f_{max}: throughput.

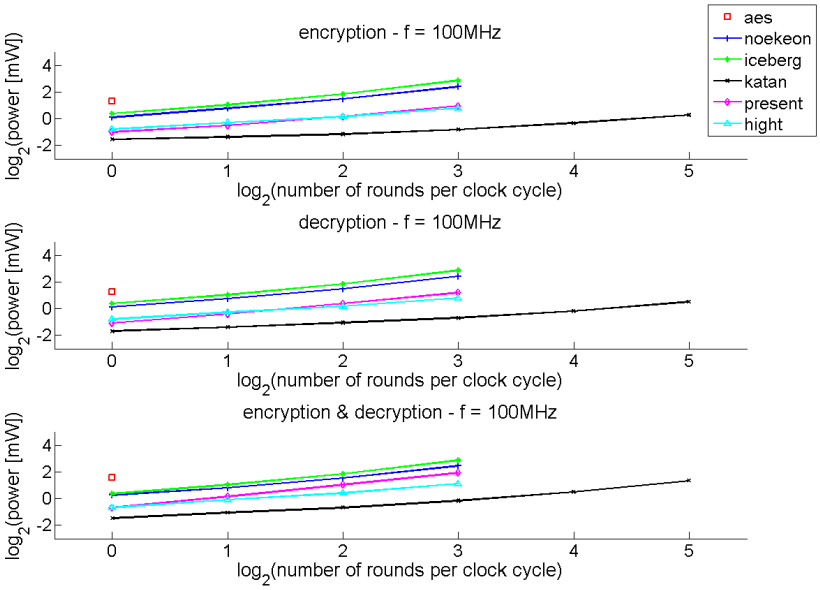

Fig. 3.12.: $V_{dd} = 1.2\text{V}$, $f = 100\text{MHz}$: power.

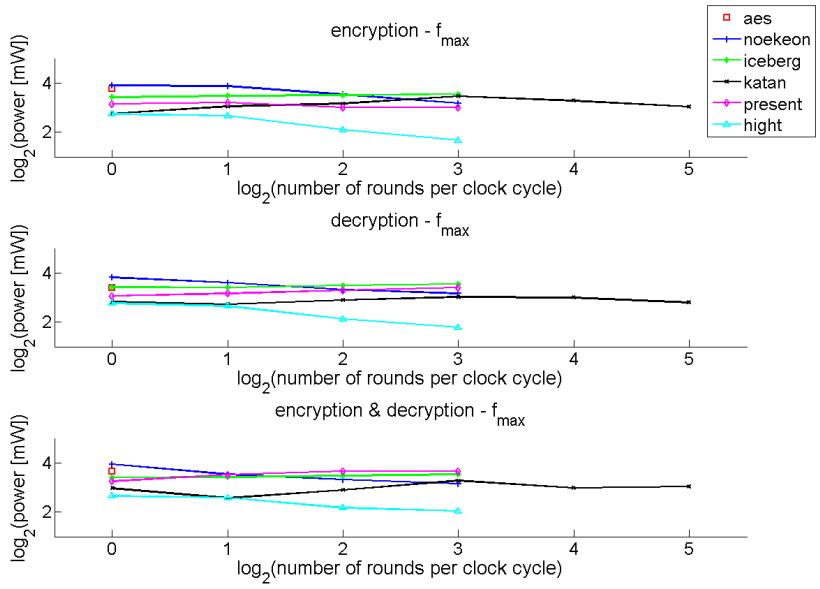

Fig. 3.13.: $V_{dd} = 1.2\text{V}$, f_{max}: power.

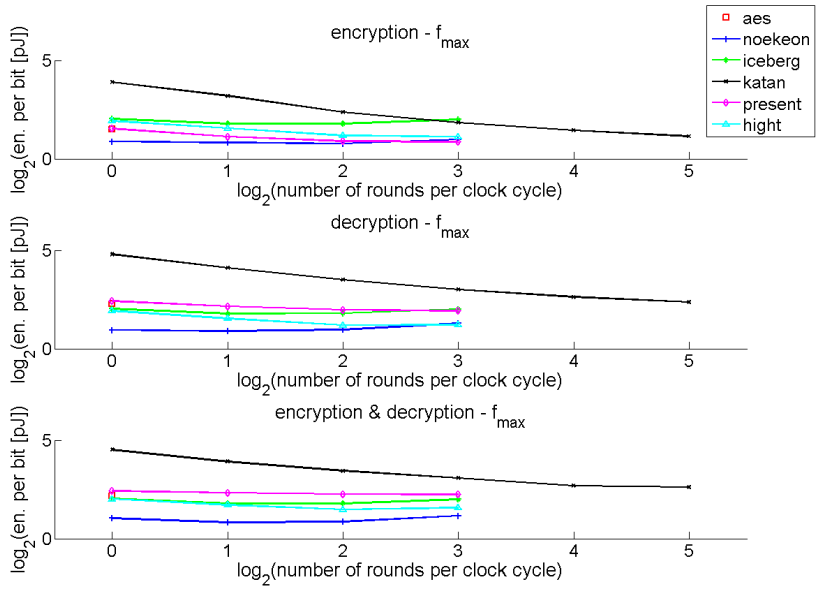

Fig. 3.14.: $V_{dd} = 1.2$V, f_{max}: energy per bit.

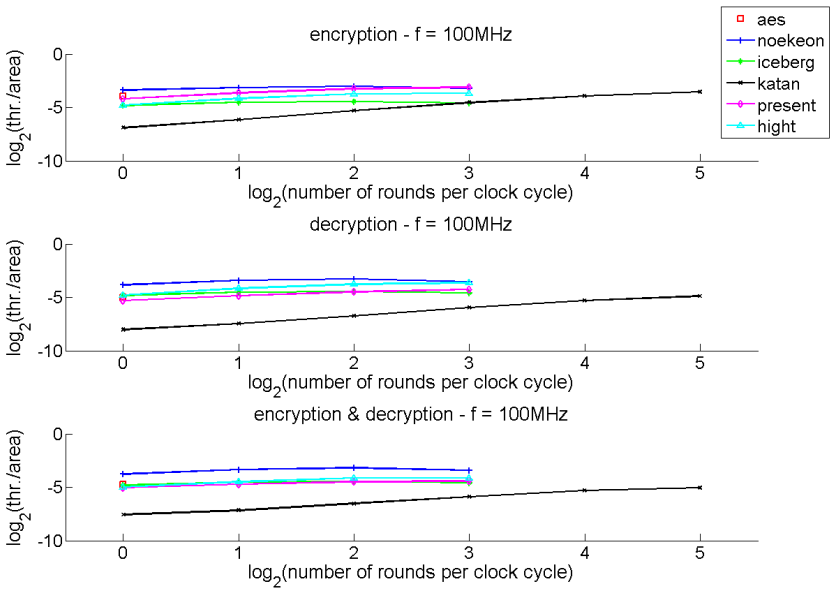

Fig. 3.15.: $V_{dd} = 1.2$V, $f = 100$MHz: throughput over area ratio.

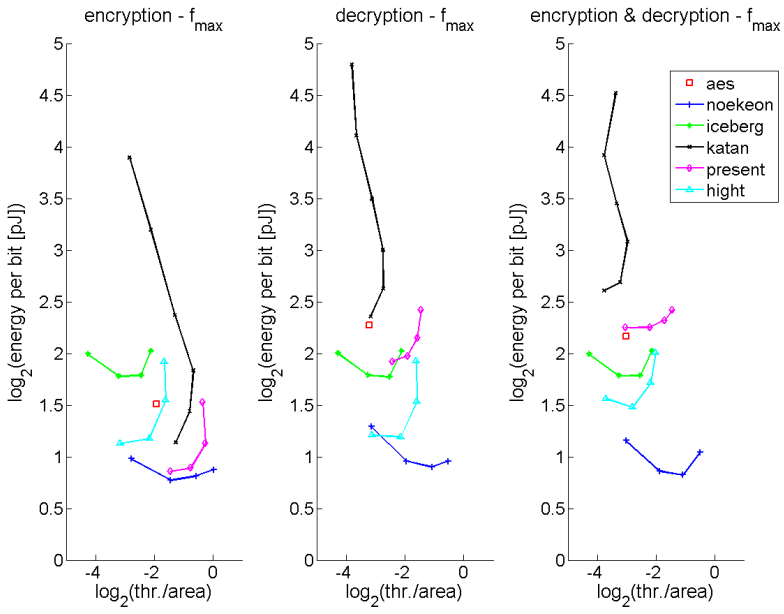

Fig. 3.16.: $V_{dd} = 1.2$V: throughput over area ratio vs. energy per bit.

forward one more design issue, namely the need to perform the key scheduling "on-the-fly" in forward direction before doing it in backward direction[2] for ciphers such as AES, KATAN and PRESENT. It implies that the comparison is modified into NOEKEON $\geq$ HIGHT $\geq$ ICEBERG $\approx$ PRESENT $\geq$ AES $\approx$ KATAN. Finally, the encryption/decryption designs further indicate the possibility to efficiently share the resources between the cipher and its inverse. Here, involutional ciphers such as ICEBERG gain a particular advantage, leading to a rating: NOEKEON $\geq$ HIGHT $\approx$ ICEBERG $\geq$ AES $\approx$ PRESENT $\geq$ KATAN.

An alternative view of the performance and energy efficiency of the different algorithms is given in Figure 3.17 (for f_{max}), where we plot the ratio between the throughput and the product of the area and the energy per bit. Again, such a figure requires a careful interpretation as they only provide one "global efficiency" metric. Yet, it is interesting to note that for all ciphers, the architecture providing the best such global efficiency has approximately the same latency. Intuitively, this suggests that the computational security of cryptographic algorithms imposes to iterate Boolean functions with a minimum complexity that is somewhat comparable for all ciphers. As an illustration, we provide the complete synthesis results for these most efficient architectures for

[2]This choice is natural in hardware implementations as storing a fully precomputed expanded key in registers would generally require too large memory overheads.

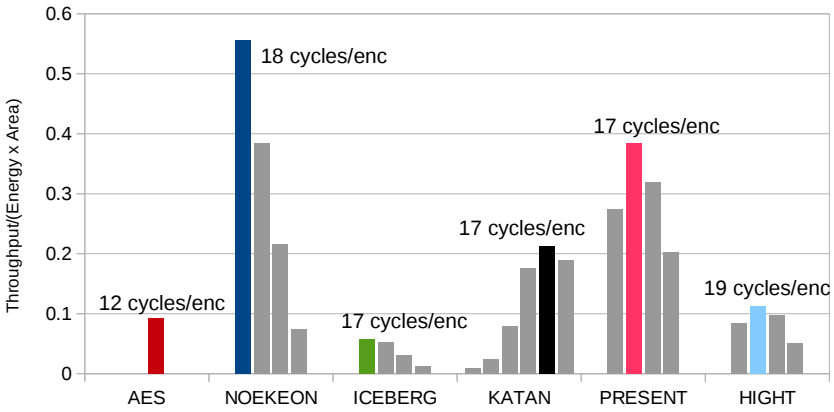

Fig. 3.17.: $V_{dd} = 1.2$V, f_{max}, encryption only: combined metric - efficiency. The latency of the most efficient architecture (colored bar) is given next to it. Each bar in a cipher's column corresponds to a different N_r.

all ciphers in Table 3.2, where the approximate symbol means that we provide an average for ED figures.

Impact of parallelism/pipeline

As mentioned in the previous section, unrolling our architectures without parallelizing or pipelining becomes suboptimal at maximum frequency. In cases where the (already high) throughputs obtained in Table 3.2 are not sufficient for a given application, it is possible to further increase them with parallelization and pipelining. For this purpose, it is natural to start from the efficient architectures with N_r determined as in Table 3.2. In the first case, we just multiply several circuits as depicted in Figure 3.18. Since only the control part can be shared between the multiple instances, we essentially double the throughput at the cost of a doubled area (in particular, the control part is small in our implementations and this "doubling rule" was precise up to a few percents). In the case of outer pipelining, it is additionally possible to spare a few multiplexers. Yet, the trends observed for all metrics and all ciphers are essentially the same as well. In particular for the throughput over area ratio and the energy per encrypted bit (i.e. the two metrics we mainly focus on in this work), we observed similar conclusions for all the investigated ciphers. This behavior is again due to the limited cost of the control part compared to the state registers and the datapath in our block cipher implementations. As doubling the parallelism or pipeline essentially comes at the cost of a doubled area, the throughput over area ratio remains close to constant. Since the same comment applies to the energy per bit (i.e. doubling the throughput doubles the power consumption), we conclude that parallelization and pipeline do not

Table 3.2.: Implementation results for most "globally efficient" architectures.

Cipher	Mode E,D,ED	Area [μm^2]	f_{max} [MHz]	Latency [cycles]	Throughput [Mbps]	Power [mW]	Energy [pJ per bit]
AES	E	17921	444	12	4740	13,5	2,9
$N_r = 1$	D	20292	377	22	2195	10,6	4,8
	ED	24272	363	≈17	≈2997	≈12,6	≈4,4
NOEKEON	E	8011	1149	18	8173	15,0	1,8
$N_r = 1$	D	10431	1075	19	7243	14,1	1,9
	ED	10483	1075	≈18,5	≈7445	≈15,35	≈2,1
HIGHT	E	6524	641	19	2159	6,3	2,9
$N_r = 2$	D	6524	645	19	2173	6,3	2,9
	ED	8217	540	19	1820	6,1	3,3
ICEBERG	E	11377	699	17	2632	10,7	4,0
$N_r = 1$	D	11359	699	17	2632	10,7	4,0
	ED	11408	689	17	2596	10,6	4,0
KATAN	E	6231	952	17	3585	8,1	2,7
$N_r = 16$	D	8616	666	33	1292	9,8	6,1
	ED	12609	473	≈25	≈1347	≈12,7	6,4
PRESENT	E	5024	1123	17	4230	9,3	2,2
$N_r = 2$	D	6060	1041	33	2020	8,9	4,4
	ED	8213	884	≈25	≈2523	≈12,6	4,7

increase the efficiency, nor do they notably affect our comparisons of algorithms.

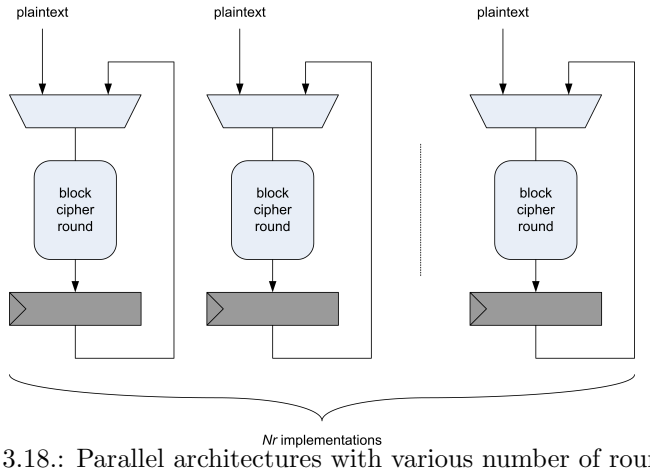

Fig. 3.18.: Parallel architectures with various number of rounds.

3.3.3 Frequency/voltage scaling

The previous sections investigated the impact of architectural choices on the efficiency of different implementations. We used them to compare different lightweight ciphers. A natural extension of this work is to investigate the impact of physical parameters, e.g. in terms of frequency/voltage scaling. Two main questions can be investigated in this setting. First, do the algorithm comparisons remain unchanged with variable supply voltage? Second, are the efficiency differences between different lightweight ciphers significant in front of the differences when tuning a physical parameter. In order to answer these questions, we re-synthesized the "most efficient" implementations of Table 3.2 at different V_{dd}'s. The library we used for this purpose is composed of cells that tolerate supply voltages from 1.2V to 0.4V. Note that, beyond the previously listed remarks about the relative nature of hardware performance comparisons, synthesis results at low supply voltage are particularly sensitive to synthesis options. This confirms the general discussion found in Saar Drimer's PhD dissertation in the context of FPGAs [59]. As a consequence, while we would expect the comparison of algorithms to be fully independent of the frequency/voltage scaling, we observed some curve overlaps in our performance evaluations. Our conclusions are as follows.

From the hardware design point of view and as expected, the critical path increases faster, as the supply voltage decreases (in Figure 3.19). Hence, the maximum frequency and throughput both decrease non-linearly as well, resulting in a reduction of the throughput over area ratio for low V_{dd}'s. More positively, the reason why we lower the supply voltages is to reach lower power

consuming points. This is what we observed in our experiments: the power decreases non-linearly with the supply voltage. However, due to the first observation, this power reduction is also moderated by the critical path increase. As a result, the energy per bit (represented in Figure 3.20) only decreases close to quadratically with the supply voltage, as expected from the energy required to switch the internal capacitances. Note that for all our syntheses, the leakage currents remained negligible (they would become significant below 0.4V in our target technology).

Regarding algorithms, we again plotted a global view of the power and performance efficiency metrics in Figure 3.21. This final picture allows us to answer the two previously listed questions. First, the comparisons of the different algorithms and architectures is essentially similar to the ones for V_{dd} = 1.2V. Yet, and as previously mentioned, some curve overlaps are noticed, due to the increased variability of our synthesis results at low supply voltage. Second, the difference between different algorithms in terms of efficiency is quite limited when compared with the impact of frequency/voltage scaling (this is illustrated on Figure 3.22 where the impact on energy induced by frequency/voltage scaling on the AES is compared to the one induced by choosing another algorithm). For example, the energy per bit can be decreased by an order of magnitude when reducing V_{dd} (at the cost of a throughput decrease). By contrast, the difference between the various block ciphers investigated roughly corresponds to a factor 2 for this metric. Yet, the gains obtained when looking at combined metrics is non-negligible (i.e. at least it is larger than the variability due to synthesis options), in particular when looking at all architectures (i.e. not only the encryption one).

3.4 CONCLUSION

This chapter provides a first comprehensive comparison of lightweight block cipher performances and efficiency in two implementation scenarios (i.e. software and hardware implementations). It confirms that such ciphers do provide interesting figures compared to standard solutions such as the `AES`. However, the gains observed for several of the metrics remain limited and may sometimes not be sufficient to motivate the use of non standard algorithms in actual applications. Furthermore, in the case of hardware implementations and from an energy point of view, frequency/voltage scaling may be an interesting alternative compared to choosing a different cipher if the application context allows it.

Some general conclusion concerning lightweight block cipher design may be extracted from the evaluation results obtained in the previous sections. First, both hardware and software implementation results suggest that using the smallest rounds (e.g. those of KATAN) is not the best strategy to reach energy-efficient implementations (because of the too large number of iterations required to complete each encryption). Besides, the gain in area or code

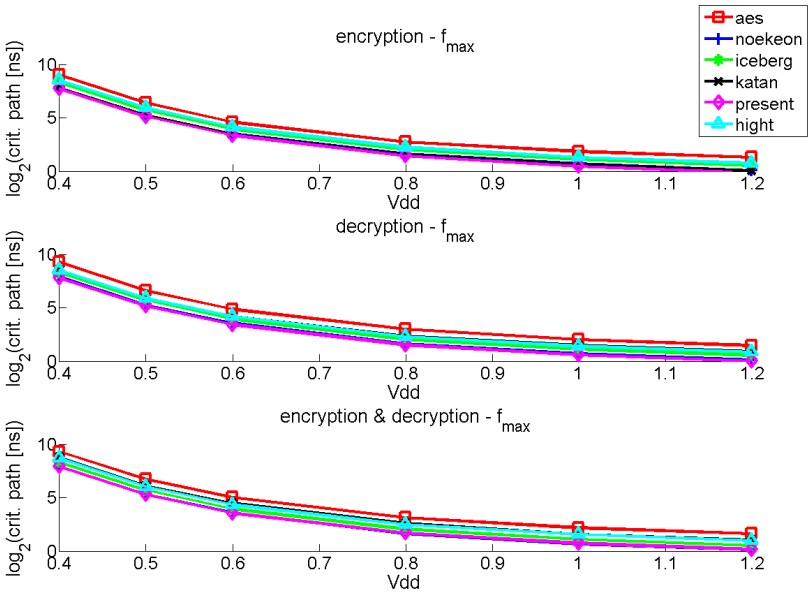

Fig. 3.19.: Voltage scaling: critical path.

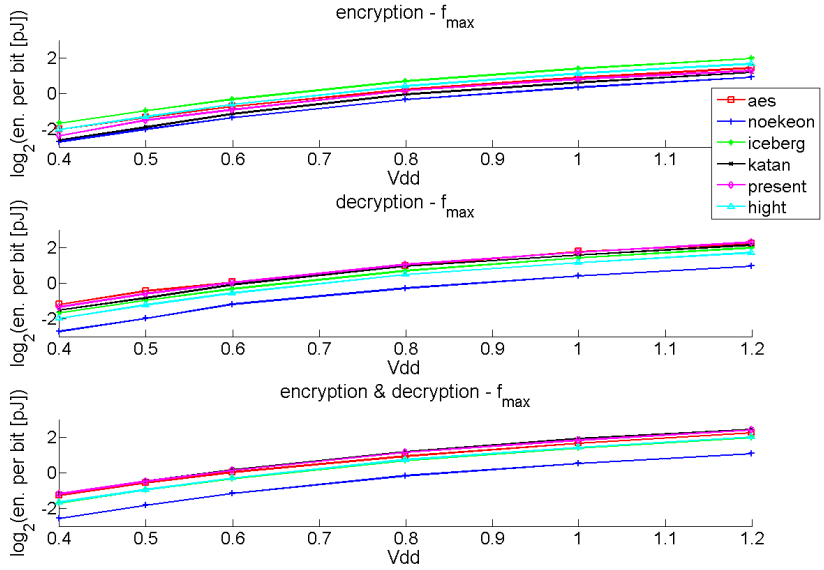

Fig. 3.20.: Voltage scaling: energy per bit.

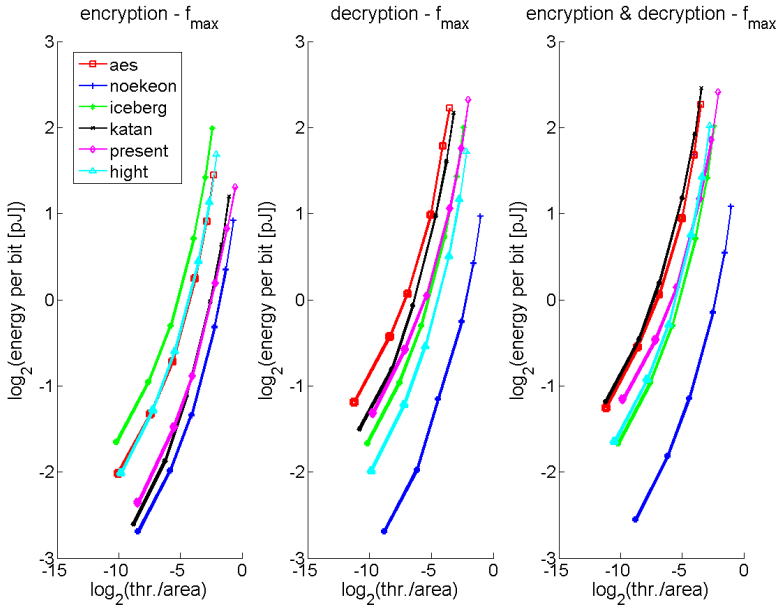

Fig. 3.21.: Voltage scaling: throughput over area ratio vs. energy per bit.

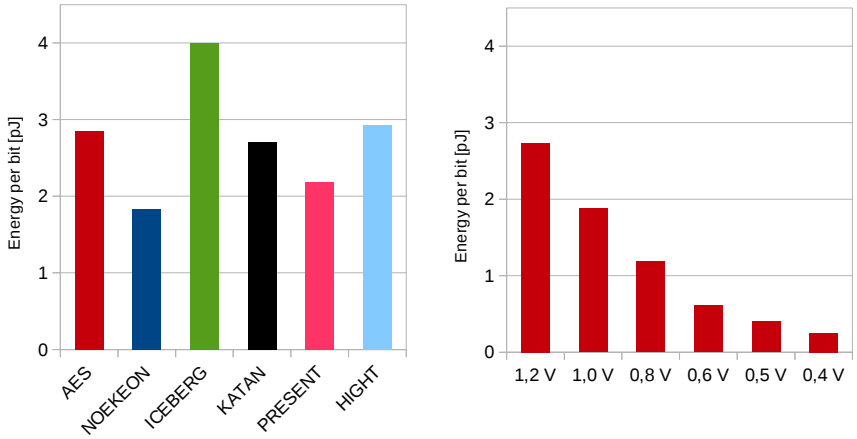

Fig. 3.22.: Voltage scaling: energy per bit. Left: Impact of algorithm choices on energy consumption. Right: Impact of the frequency/voltage scaling on the AES energy consumption.

size implied by these ciphers remains limited compared to other algorithms having better energy figures. We also noticed from our hardware results that the strong similarity in the block cipher design principles (i.e. designs are all based on the wide trail strategy) does lead to remarkably comparable implementation figures.

By contrast, the key-scheduling algorithms greatly differ from one cipher to the other. This has an impact on the efficiency figures (i.e. ciphers with simple key scheduling are generally more efficient than ciphers with a complex one). Similarly, decryption algorithms having involutive key scheduling get better energy results since they do not need a pre-computation step in order to produce the decryption key. Note that these two conclusions only hold for implementations computing their subkeys on the fly.

It is also important to notice that the ciphers that were designed with a hardware-oriented goal in mind tend to be less efficient when implemented in software. Since we think that a good lightweight block cipher should be equally efficient in both implementation scenarios, it could be a good practice to avoid bit- or nibble-wise operations when designing lightweight ciphers. Likewise, avoiding the use of complex operations such as multiplications enables the use of the cipher on a wider variety of plateforms.

From a software point of view, some conclusions may also be drawn regarding the possible goals that can be chosen while implementing a given block cipher. There are mainly two possible lightweight implementations goals: a software implementation can either be optimized for code size or for clock cycles (and thereby for energy). Usually, optimizing a code for energy will correspond to a higher use of the registers in order to store important information (i.e. the implementations that store their complete intermediate state in registers are generally faster than the ones accessing the RAM for their computations). By contrast, a code optimized for code size will make extensive use of loop iterations on small parts of codes and will thereby have to access the RAM more often.

Overall, we believe that these results and the general discussion about performance evaluation raise interesting problems for the design of new block ciphers. Namely, finding how to make the algorithmic choices more discriminant with respect to their implementations is an interesting research direction.

CHAPTER 4

LIGHTWEIGHT IMPLEMENTATIONS OF HASH FUNCTIONS

4.1 INTRODUCTION

We showed in Chapter 3 that evaluating the performances of existing cryptographic algorithms is important in order to better understand which design practices are best when constructing new algorithms. However, and similarly to what we have observed for block ciphers, up to now, the number of existing comparative studies concerning hash functions is quite low. The only exceptions are the studies that were performed in the context of the SHA-3 competition. Besides, these studies do not take into account already existing constructions and often focus on high-speed implementations more than compact ones, e.g. [89, 184] for hardware implementations. Yet, whenever trying to compare different algorithms, compact implementations are an important piece of the puzzle. In particular, they usually reveal a part of the algorithms complexity that does not directly appear in high throughput implementations, e.g. the need to share resources or to minimize memory. Next to that, implementations in small embedded devices such as smart cards, RFIDs and sensor nodes are also motivated by an increasing number of applications. As a result, studying the performances of cryptographic algorithms systematically in this challenging scenario is generally useful.

Following these observations, the goal of this chapter is to compare the performances of several hash functions implemented with a low area goal in mind. More specifically, we will analyze software and hardware implementations of several hash functions in Sections 4.2 and 4.3 respectively.

Again, as this work involves many different developers, we naturally acknowledge possible biases in our performance evaluation results due to slightly different implementation choices and interpretation of the guidelines.

4.2 SOFTWARE EVALUATION OF HASH FUNCTION IMPLEMENTATIONS

In Chapter 3, the implementation of 12 lightweight and standard block ciphers in an ATMEL AVR ATtiny45 has been investigated. We will now extend this study towards hash functions. For this purpose, we considered three main types of algorithms. First, we targeted SHA256 and the SHA-3 finalists. For the latter ones, we only focused on the candidates satisfying the SHA-3 security requirements for the 256-bit output length [143], i.e. providing at least 2^{256} (second) preimage resistance and 2^{128} collision resistance. Second, we selected a number of recently published lightweight hash functions, providing both 2^{80} and 2^{128} "flat" security levels[1] [142]. Eventually, we also implemented several block cipher based constructions, e.g. relying on the AES Rijndael. For all these algorithms, we aimed for the same optimization criteria (namely small source code size and limited memory use) and used a uniform interface (see the details in Section 4.2.1). Resistance against physical (e.g. side-channel, fault) attacks was explicitly excluded from the requirements.

All the source codes were implemented by different developers and are provided on a public webpage [16]. As a result, we hope that this initiative can be used as a first step in better understanding the performances of hash functions in a specific but meaningful class of devices.

The section is structured as follows. First, we discuss the guidelines that had to be followed by all developers and the metrics that are used for the performance evaluation. Then, in Section 4.2.2, we detail the implementation specificities of each evaluated algorithm. Lastly, the implementation results are evaluated in Section 4.2.3.

4.2.1 Methodology and metrics

In order to be able to compare the performances of the different hash functions in terms of speed and memory space, the developers were asked to respect a list of common constraints, detailed hereunder.

(1) The code has to be written in assembly, if possible in a single file. It has to be commented and easily readable, for example, giving the functions the name they have in their original specifications.

(2) The function has to be implemented in a low-cost way, minimizing the code size and the RAM use.

(3) Data does not have to be preserved by the hashing process. This allows direct modification of the data zones in RAM, hence reducing the amount of memory needed.

[1] i.e. the same security is required for collision, preimage and 2nd preimage resistance.

(4) The interface should be made up of 3 functions.

   (a) *init* takes no input and initializes the internal state, which is a dedicated memory zone seen as a black box, and returns no output.

   (b) *update* takes as input a full block of data, updates its internal state by processing that block and returns no output.

   (c) *final* takes as input the (possibly empty) last chunk of data together with its size and processes it before finalizing the hash computation. By convention, the data passed to *final* is necessarily an incomplete block.

(5) Data exchanges are performed with pre-defined memory zones where data has to be put before calling functions, or can be found on their return. For example, the data block to hash has to be put at the pre-defined address SRAM_DATA before a call to *update*, and the final hash can be found at SRAM_STATE on return of *final*. Most input/output values are thus implicitly passed. The only explicitly passed value is the size of the data passed to *final*.

(6) Only the internal state is preserved between calls to these functions. No assumption can be made that other RAM zones (e.g. SRAM_DATA) or registers will stay unchanged.

(7) The target device is an 8-bit microcontroller from the ATMEL AVR device family, more precisely the ATtiny45. It has a reduced set of instructions and, e.g. has no hardware multiplier. A common interface file was provided to all designers (available on [16]).

Note that for some functions (e.g. for block cipher based), the padding was not explicitly defined. In these cases, we appended n null bytes, followed by the length of the message coded as a 64-bit value, where n is chosen to make the global message length a multiple of the block size. The basic metrics considered for evaluation are code size, number of RAM words, and cycle count. Performances were measured on 4 different message lengths: 8, 50, 100 and 500 bytes, ranging from a very small (smaller than one block) to a large message. Note finally that, as already mentioned in Section 2.2.3, all hash functions were implemented by different designers, with slightly different interpretations of the low-cost optimizations. As a result, some of the guidelines were not always followed, because of the hash function specifications making them less relevant (this will be specified when necessary).

4.2.2 Implementation details

SHA-256 and SHA-3 candidates

- **SHA-256.** Like its predecessor SHA-1, SHA-256 is optimized for 32-bit software implementation. Hence, it can be expected to be similarly efficient on 8-bit AVR processors. Implementing the iteration step of its

compression function, a main observation is that six out of eight working registers are just circularly copied. To reduce code and clock cycles for such memory transfer operations, register name reassignment by circular pointer arithmetic is performed instead on the working registers residing in 256 bits of RAM. Circular pointer arithmetic as part of the iteration step is likewise used to update the input word according to the message expansion. Besides 32-bit modular additions, SHA-2 requires 32-bit right rotations by $r = \{2, 6, 7, 11, 13, 17, 18, 19, 22, 25\}$ bits and right shifts by $s = \{3, 10\}$. Rotations and shifts by parameters larger than 8 bits first swap 8-bit register accordingly; then single bit operations on the swapped 32-bit word are performed to correspond to $f = \{r, s\} \bmod 8$. SHA-256 uses up to three 32-bit bit rotations processing the same input in a row so that reordering of rotation and shift operations by ascending f-values improves efficiency.

- **BLAKE-256.** The RAM consumption is mainly due to the storage of the 64-byte input data, the 64-byte state, a 32-byte chain value, an 8-byte salt, and an 8-byte counter. The initialization vectors (32 bytes) and constants (64 bytes) are stored in the flash memory of the microcontroller. We refrained from transferring the constant table into the RAM in order to keep the RAM consumption low. BLAKE's permutation table σ consists of 10×16 entries. However, each entry is only a 4-bit number so we merged two entries in one byte and later selected the upper/lower 4 bits by masking. Thus, the permutation table requires just 80 bytes of ROM instead of 160 bytes. In order to maintain a decent performance while keeping the code size down we incorporated the observation made by Osvik [144] which efficiently loads and stores in-/outputs of the round function $\mathrm{G}_i\,(a, b, c, d)$. Furthermore, we use loops where applicable and move recurring duties such as loading and storing the counter into functions. An exception to this rule is the implementation of the round function. Since it is called 80 times when hashing one message block its runtime heavily impacts the overall performance. Therefore, we decided to unroll critical parts of the round function.

- **Grøstl-256.** Grøstl has a state of 64 bytes. During the update function, we need to keep the state, the input message and the previously computed hash in memory. Thus, we need 192 bytes of RAM. The ShiftBytes is computed by offloading each row, one at a time, from the state into the registers of the microcontroller and then writing it back in the new position. In order to increase the performances and reduce the number of accesses to the memory, the SubBytes is computed together with the ShiftBytes.The MixBytes is computed as proposed by Johannes Feichtner [67, 167], and is carried out one column at a time. Finally, to easily compute the padding, 8 bytes of memory are used to keep track of the numbers of message blocks. These 8 bytes, are copied directly in the appropriate position of the padding block.

- **JH-256.** Even if specifications for a bitsliced implementation of JH are available, they suppose to store 42 256-bit round constants in memory, which is not compliant with our low-cost constraints. Hence, JH was implemented according to the reference specifications. The utilization percentage of the RAM is high as JH needs 128 bytes to store the state, 64 for the input block and 32 for the round constant. In order to improve the performances, the S-Box and linear transformation were combined into two look up tables, of 32 bytes each, as was done in the optimized 8-bit implementation provided by JH author [205]. For the same reason, the initial state was precomputed and stored in program memory. It allows us to save the initialization phase which is equivalent to the processing of one input block. Regarding the permutation, it is performed by reading the states bytes in a different order at the beginning of each round. Finally, state bits are reorganized at the beginning and end of each function E8. This bitwise permutation is time consuming and requires additional memory. Those problems can be partially prevented by reorganizing the input bytes before XORing them with the state.

- **Keccak.** In a first level, we implemented the sponge construction, which comes down to XOR r-bit message blocks into the state, with $r > 0$ the rate, and to call the underlying permutation. In a second level, we implemented the permutations Keccak-$f[b]$ for $b \in \{200, 400, 800, 1600\}$. The sponge construction imposes that the capacity c is twice the security strength level and that $b = r + c$, and our implementation allows any combination of rate and capacity under these constraints. For clarity, the benchmark focuses on three specific instances: the SHA-3 candidate Keccak$[r = 1088, c = 512]$, and the lightweight variants Keccak$[r = 144, c = 256]$ and Keccak$[r = 40, c = 160]$ for the 128-bit and 80-bit security strength levels, respectively. Any pair of instances with $c = 256$ and $c = 160$ would have satisfied the requirements, but our choice aims at minimizing b for a given c and thereby the RAM usage, consistently with a lightweight context. Variants with other RAM usage/code size/cycle count trade-offs can be found in Table 4.4. Inside the implementation, some operations (i.e., the rotations in θ and ρ) are performed on a lane basis, mapping a lane to $b/200$ byte(s). Some other operations, such as χ or the parity computation in θ, are instead slice-oriented, taking advantage of the representation of 8 consecutive slices in 25 bytes [25]. Note that in the specific case of Keccak-$f[200]$, both approaches collide as the state contains exactly 8 slices or 25 lanes, mapped to 25 bytes. RAM usage is composed of $b/8$ bytes for the state and some working memory ($b/40$ bytes, or 0 for Keccak-$f[200]$ as the AVR registers suffice). If the desired output length is greater than the rate (e.g. for lightweight instances), an additional output buffer is needed to perform the squeezing phase.

- **Skein-x-y.** We implemented the SHA-3 finalist Skein-512-256, with an output of 256 bits, limited to the hashing functionality. The internal state is therefore made of eight 64-bit words. To keep the program memory space small and the code readable, some basic 64-bit functions like loading, saving, adding, ... have been coded. The registers are only used temporarily, except the round counter. The message, the state, the key, the key-schedule and the tweak are always in the data space, and modified directly. The three main Threefish functions (addkey, mix and permute) were implemented following the reference specifications. Besides, the modulo 3 and modulo 9 values used in the key schedule were saved in the program memory space. We have also developed Skein-256-256, slightly optimized for the speed and data memory space performances, by leaving most of the time three out of the four state words in the registers.

Lightweight hash functions

- **S-Quark and D-Quark.** The critical point in the implementation of Quark hash functions is the update of the state[2]. This update phase considers the state as two LFSRs that will be updated using three retro-action polynomials[3]. This design is thought for hardware, a context where it is very efficient, but is much more expensive in software. Nevertheless, our choice to implement this step using a bit-slice approach provides rather good performances. The platform is an 8-bit microprocessor and the retro-action polynomials are such that the last 8 bits of each LFSR are not considered. Hence, our implementation performs 8 updates at the same time reducing from 1024/704 to 128/88 polynomial computations. The state is stored in RAM, as it is too large to be kept in registers. Computations are ordered in such a way that the shift of the state is performed on the fly.

- **PHOTON-160/36/36 and PHOTON-256/32/32.** First note that these implementations significantly differ, since PHOTON-160 has a state matrix with 4-bit cells and uses the PRESENT S-Box while PHOTON-256 has 8-bit entries and uses the AES S-Box. This results in different implementation strategies. The state of the implemented PHOTON-160/36/36 variant consists of 7-by-7 4-bit elements which are packed into 25 bytes in order to save memory. This allows an optimal usage of the RAM but naturally also results in additional code in order to extract the correct nibble out of the state. It is a trade-off between code size/speed and RAM

[2]During implementation, a minor inconsistency was discovered between the paper description [11] and the reference code [12], which use different bit ordering conventions. We chose to comply with the description provided in the original article. Compliance with the C code can be obtained by inverting the order of bits in the input message.
[3]An additional third will provide constants for the 1024/704 executions required to apply the permutation P.

usage. As the interface only allows messages that are a multiple of 8 bits while each iteration of a PHOTON-160/36/36 round function absorbs 36 bits, we just process an input block of length 72 bits and call the PHOTON round function internally twice for a full 72-bit block. The largest amount of computational time is spend in the permutation layer for ShiftRows and especially during the MixColumnsSerial step as finite field arithmetic has to be carried out on 4-bit values. The internal state of PHOTON-256/32/32 consists of 36 bytes, arranged as a 6-by-6 matrix, that goes over four different transformations to produce a 32 byte hash digest. Due to their sizes both state and digest have to be stored in SRAM. This generates an inherent implementation overhead, as state bytes need to be fetched from and stored to SRAM once for each transformation. We partially reduce this overhead by merging all row-based transformations, and also by incrementing code size. Due to its use of AES-like permutations, the implementation of the PHOTON-256/32/32 transformations can be carried out quite efficiently on 8-bit controllers. The SubCells transformation is implemented as a memory aligned lookup table resulting in important cycle savings. The MixColumnsSerial transformation, consisting of six consecutive calls to the AES MixColumns transformation, is similarly optimized by implementing the multiplication by '02' as a memory aligned LUT [52].

- **SPONGENT-160/160/80 and -256/256/128**. The SPONGENT-160 state is $160 + 80 = 240$ bits or 30 bytes large. Therefore, the state can be stored in the registers already available on the target device. However, SPONGENT uses a PRESENT-like bit permutation in π_b and therefore every output bit of an S-Box is mapped to a distinct nibble after permutation. If we were to store the state in the available registers, we would only have two registers for additional computations and this would lead to a large code size when implementing the bit permutation. Therefore, the state is stored in SRAM and a three-step iterative approach is used for the bit permutation to achieve a smaller code size. For the permutation, each four consecutive nibbles are permuted and stored in SRAM at the same places. Then, the permuted nibbles are re-ordered to obtain permuted bytes and finally bytes are re-ordered to their appropriate places in the state. Although this approach is code-size efficient, note that it leads to an increase in running time of the overall hashing process. The remaining operations like round constant computation, padding and control logic are implemented in a straightforward manner. SPONGENT-256 has a state of $256 + 128 = 384$ bits or 48 bytes. Since the state does not fit into the available registers, we optimized this variant with respect to code size and the state is kept in SRAM. For the permutation, iteratively four successive bytes are loaded into registers and the permuted byte is constructed from two bits at fixed offsets of each of these four bytes. Afterwards the processed bytes are stored back to SRAM. This method

keeps the code very small but requires a copy of the 48 bytes state and therefore doubles the required memory. Besides the two states, no additional memory is required. The S-Boxes are stored in flash memory and must be aligned to an address which is a multiple of 16 for easier pointer arithmetic. Again, the remaining operations are straightforwardly implemented.

Block cipher based constructions

- **Rogaway-Steinberger LP/lp362.** To realize the Rogaway and Steinberger construction principle, the matrix A suggested by Lee and Park [114] with $\alpha = 2$ has been used. For operations in $\mathbb{F}_{2^{128}}$ (addition and multiplication) we have selected the same irreducible polynomial $x^{128} + x^7 + x^2 + x + 1$ as stated in [166]. The implementation of the block cipher NOEKEON is based on the open source version published in [62], but the decryption functionality has been removed since it is not required for the generation of a permutations. The two implemented variants of Rogaway-Steinberger mainly differ in code size. The lp362 scheme uses a single fixed key for all permutations, the code requests thus 100 bytes less than the LP362 scheme which uses a different fixed key for each of the six permutations. Both variants have similar execution time, consume 92 bytes of RAM, and make use of 8 registers to compute the hash value of a message.

- **Hirose double block length construction** For simplicity we chose an all-zero IV and the additive constant to be 1. One of the advantages of Hirose is that the two parallel AES executions use the same key. However, due to memory restrictions, the key should be computed on-the-fly. Hence, the two encryptions need to be processed in parallel. The AES design is similar to the one presented in [62], with a further optimized ShiftRows operation. Decryption code is not needed and was removed. The key scheduling is performed on-the-fly and processes 32 bits at a time. The full 128-bit state of one encryption block is kept in the registers. Since both encryptions are performed in parallel, the two states have to be swapped in and out of SRAM regularly. Due to the large key size, the swap is performed as little as every 4 rounds, keeping the resulting overhead at a minimum. The implementation needs 82 bytes of RAM. We chose not to overwrite the input to the update function, which results in a need for 16 additional RAM bytes for the input. By overwriting the input these additional 16 bytes can be saved if RAM size is critical.

- **Davies-Meyer construction.** The implementation of the Davies-Meyer construction simply requires making a copy of the message to be XORed with the resulting encryption, resulting in an additional consumption of 32 bytes of RAM.

- **Shrimpton-Stam construction.** The implementation of Shrimpton-Stam construction only requires to take care of remembering inputs of the ciphers to be able to XOR them to the result of the encryptions. We chose simple keys to instantiate the functions f_i so that no extra memory is required to store them. More precisely, we respectively set all key bytes to `0x00`, `0x11` and `0x22` for f_1, f_2 and f_3.
- **Rijndael-256/256.** The operations to be performed during a Rijndael-256/256 encryption are simple and can be made efficient using the well-known techniques for implementing AES on lightweight processors, like the use of a lookup table for the S-Box and the efficient multiplication by '02' for MixColumns [53]. The main issue when working on an ATtiny45 is the state size: whereas AES state can be kept in registers, this is not possible any more for 256-bit blocks. As RAM accesses are time-consuming on the ATtiny, the design of this implementation focuses on minimizing the number of these accesses. This has been done by reorganizing the round loop (without, of course, affecting the behaviour of the cipher) in such a way that the round ends with a ShiftRows operation. Additionally, we used an auxiliary state to perform ShiftRows efficiently. As a result, we can fetch a full column from RAM, immediately perform MixColumns, AddRoundKey and SubBytes, and write the result in the auxiliary RAM state, taking the effect of ShiftRows into account to determine the exact locations in RAM. The next round is then performed similarly, but writing data from the auxiliary state to the initial one, and so on.
- **SEA.** The reference code was written following directly the cipher specifications, and is a natural extension of the 96-bit version designed in [62]. During its execution, plaintexts and keys are stored in RAM (accounting for a total of 48 bytes), limiting the register consumption to 12 registers for the running state, one register for the round counter and some additional temporary storage. The S-Box was implemented using its bitslice representation. The block cipher was then inserted in a Davies-Meyer mode of operation, using a similar code as the version using Rijndael-256/256. Overall, the implementation maintains low code size and RAM use at the cost of a large cycle count, mainly due to the large number of rounds (177) in the 196-bit version of the cipher based on 8-bit words.

4.2.3 Performance evaluation

We first refer to a number of other implementations of hash functions in ATMEL AVR devices [25, 63, 69, 144, 146, 167, 197]. In general, these previous works present benchmarking results in devices from the ATmega family rather than the ATtiny one, hence tolerating larger code sizes and RAM use. As they are hardly comparable with ours, we do not detail them in this section. Overall, we believe they provide a complementary view to ours. In particular, the

pretty complete comparisons of the XBX website certainly sheds another light on the different algorithms [197]. Note also that some of these previous works consider older versions of the SHA-3 candidates. Our following results consider the exact SHA-3 finalists, according to their last updated specifications. We recall that for the functions appearing several times in the tables (e.g. Keccak, Skein, Quark, PHOTON and SPONGENT), the different lines correspond to different specifications and not different implementations of the same algorithm.

Following Section 4.2.1, we evaluated the performances of our different algorithms based on three main metrics, namely the code size (in bytes), RAM use (in bytes) and cycle counts for different message sizes[4]. They are represented in Figures 4.1, 4.2 and 4.3. Besides, we also produced so-called combined metrics that aim to summarize the efficiency of the hash functions in the ATtiny45. We used the product of the code size and cycle count and the product of the RAM use and cycle count for this purpose (Figures 4.4 and 4.5. Eventually, we note that all our results are given numerically in Tables 4.1 to 4.4. As already mentioned, these results have to be interpreted with care, as they both represent the skills of the programmer and the algorithms efficiency. Yet, given this cautionary note, we believe a number of general observations can be extracted.

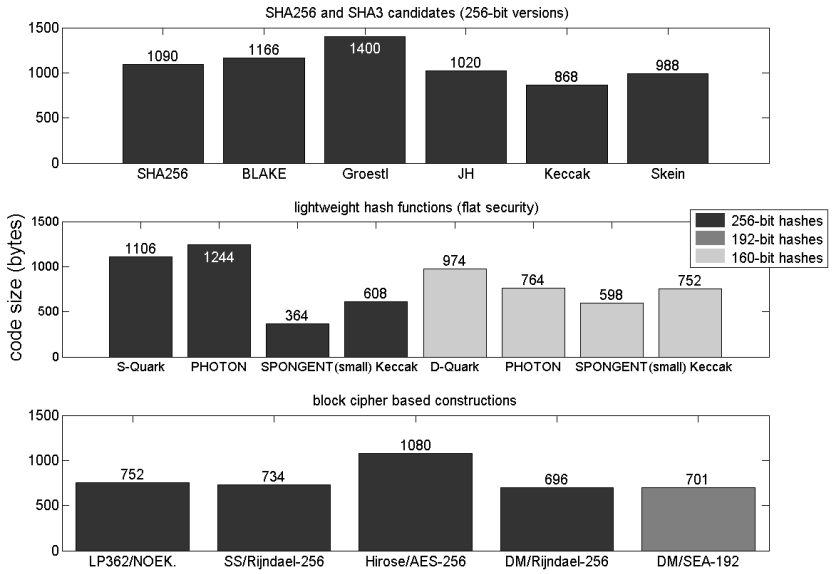

Fig. 4.1.: Performance evaluation: code size (bytes).

[4]Note that for certain (e.g. sponge-based) functions, the data part of the RAM could be arbitrarily reduced by changing the interface. In this case, the RAM use evaluation in the figures excluded the data RAM (reported in gray in the tables).

First, the code size and RAM use illustrate that the implementation constraints were reached for all algorithms. Nevertheless, the cost of the SHA-3 candidates is generally higher than the one of both lightweight hash functions and block cipher based constructions. For some of them, the RAM use is close to the limit of the ATtiny device (i.e. 256). This can be explained by the generally larger states of all SHA-3 candidates. Second, we observe that lightweight algorithms have large cycle counts compared to other hash functions. This implies that their overall efficiency (measured with the combined metrics) is generally low in our implementation context. By contrast, the flexible nature of sponge-based functions (including all lightweight proposals) allows reducing the RAM use quite significantly, which is an interesting feature for hardware and embedded software implementations. Third, it is noticeable that the SHA-3 candidates hardly compete with AES-256 in Hirose construction or Rijndael-256-256 in Davies-Meyer mode. This observation is quite consistently observed for all our metrics. Eventually, and as far as SHA-3 finalists (in the 256-bit versions) are concerned, our investigations suggest that BLAKE offers the best performance figures, followed by Grøstl, Keccak, Skein and JH.

All these results were naturally obtained within a limited time frame. Hence, we encourage the reader to download the codes, available at [16], and possibly improve them with further optimization. Looking at how the AES implementations have evolved since AES was chosen as standard, it is likely that similar improvements can be expected for the hash functions in this work.

4.3 HARDWARE EVALUATION OF HASH FUNCTION IMPLEMENTATIONS

In this section, we consider the hardware performances of the five SHA-3 finalists on recent FPGA devices. More specifically, we compare the finalists on the basis of their compact FPGA implementation. In order to allow the comparison to be as fair as possible, we applied the approach described by Gaj *et al.* at CHES 2010 [73]. 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 4.2.1 for the details). As for their optimization goals, we targeted implementations in the hundreds of slices, 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 extensively discussed in the last years, the evaluation of hardware architectures is inherently difficult, in view of the amount of parameters that may

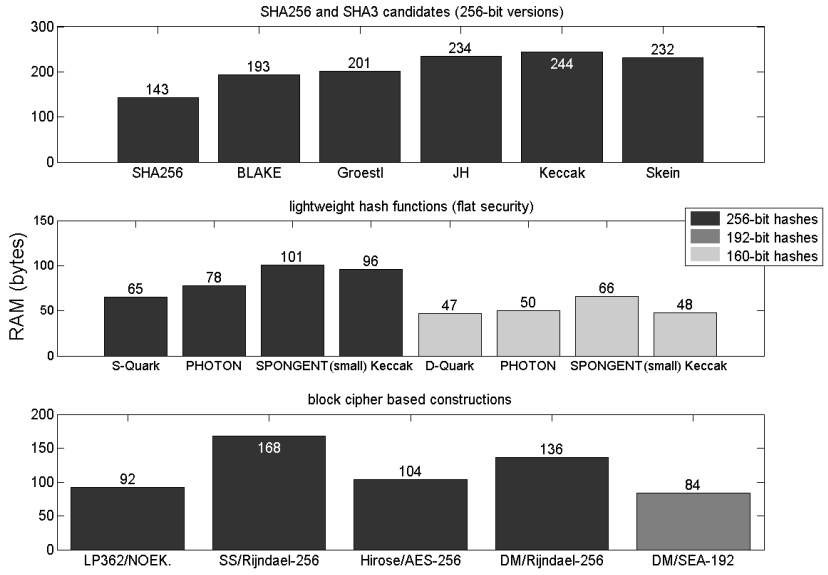

Fig. 4.2.: Performance evaluation: RAM (bytes).

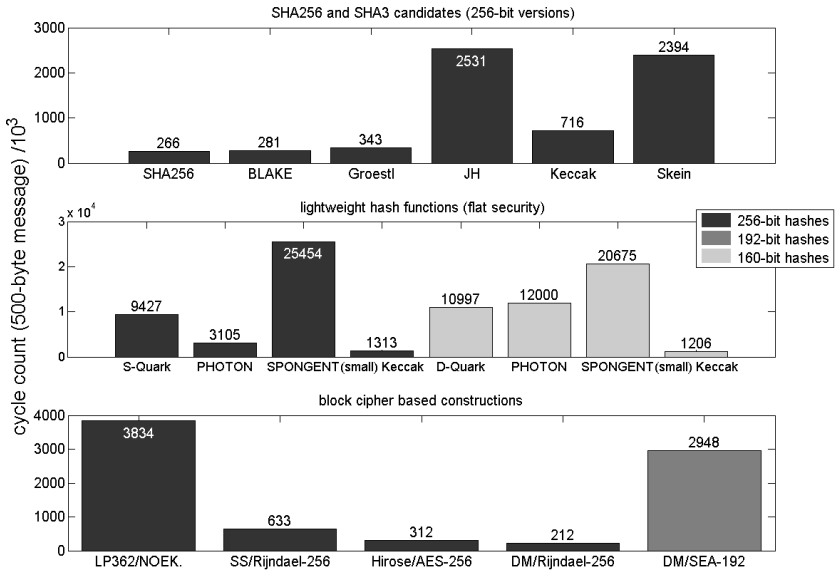

Fig. 4.3.: Performance evaluation: cycle count (500-byte message).

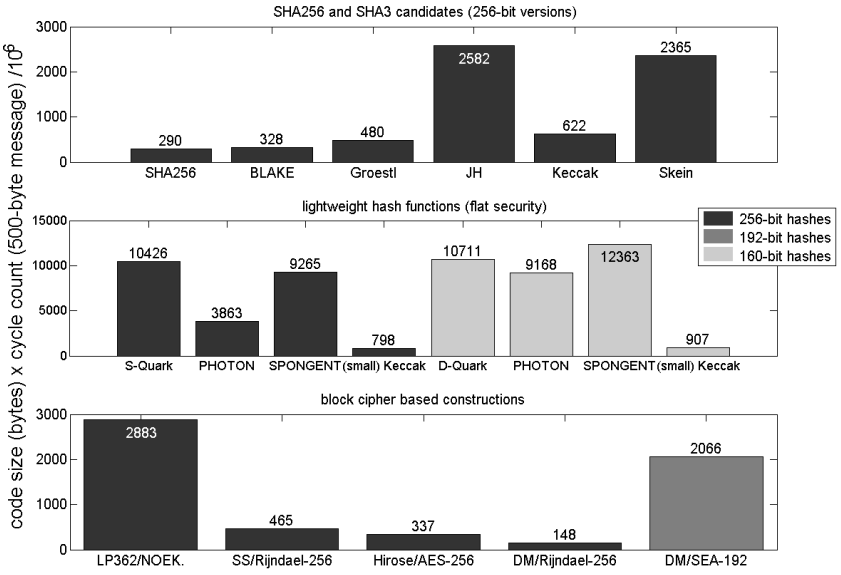

Fig. 4.4.: Performance evaluation: code size (bytes) x cycle count (500-byte message).

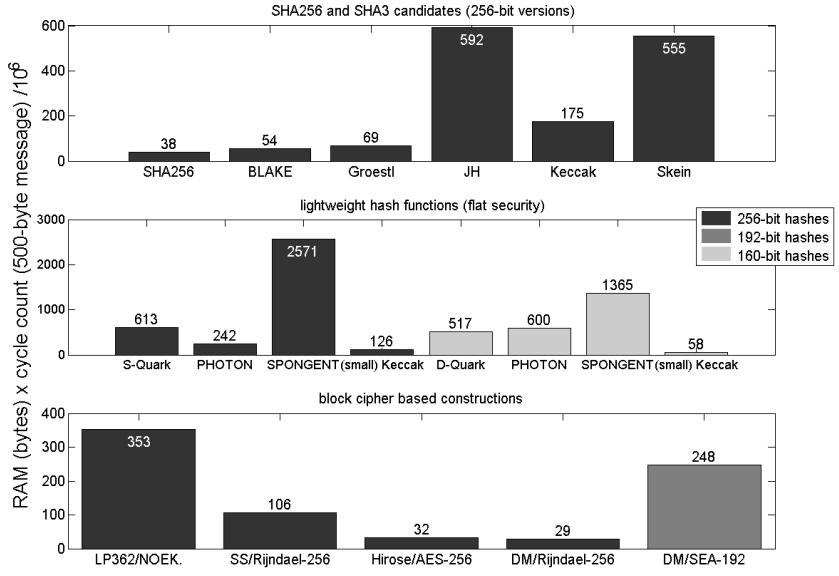

Fig. 4.5.: Performance evaluation: RAM (bytes) x cycle count (500-byte message).

Table 4.1.: SHA256 and SHA3 candidates.

Hash function	**digest size** [bits]	**code size** [bytes]	**RAM** [bytes] **data**	**state**	**stack**	**cycle count (message length)** (8-byte)	(50-byte)	(100-byte)	(500-byte)
SHA-256	256	1090	64	73	6	33 600	33 600	66 815	266 105
BLAKE-256	256	1166	64	120	9	35 714	35 714	70 808	281 372
Groestl-256	256	1400	64	128	9	61 007	61 049	101 279	342 759
JH-256	256	1020	64	162	8	524 602	524 518	785 510	2 531 262
Keccak[r=1088,c=512]*	256	868	*136*	*240*	*4*	178 022	178 022	179 494	716 483
Skein-512-256	256	988	64	160	8	532 346	532 388	798 290	2 393 802

* benchmarked on the AtTiny85 because the read-only input buffer did not fit in the ATtiny45 and our interface did not allow that the message could be directly XORed into the state, hence resulting in an excessive RAM for the AtTiny45.

Table 4.2.: Lightweight hash functions.

Hash function	**digest size** [bits]	**code size** [bytes]	**RAM** [bytes] **data**	**state**	**stack**	**cycle count (message length)** (8-byte)	(50-byte)	(100-byte)	(500-byte)
S-Quark	256	1106	4	60	5	708 783	1 417 611	2 339 023	9 4270 23
PHOTON-256/32/32	256	1244	4	68	10	254 871	486 629	787 896	3 105 396
SPONGENT-256/256/128	256	364	16	96	5	1 542 923	3 856 916	6 170 900	25 454 100
Keccak[r=144,c=256]	256	608	18	92	4	90 824	181 466	317 221	1 313 291
D-Quark	160	974	2	42	5	631 871	1 516 685	2 570 035	10 996 835
PHOTON-160/36/36	160	764	9	39	11	620 921	1 655 364	2 793 265	11 999 914
SPONGENT-160/160/80	160	598	10	60	6	795 294	2 783 241	4 771 186	20 674 746
Keccak[r=40,c=160]	160	752	5	45	3	58 063	162 347	278 269	1 205 627

Table 4.3.: Block cipher based constructions.

Hash function	**digest size** [bits]	**code size** [bytes]	**RAM** [bytes] **data**	**state**	**stack**	**cycle count (message length)** (8-byte)	(50-byte)	(100-byte)	(500-byte)
LP362/NOEK.	256	752	16	66	10	239 828	479 485	838 884	3 833 859
SS/Rijndael-256	256	734	32	130	6	39 738	79 313	158 429	633 225
Hirose/AES-256	256	1080	16	82	6	9918	39 111	68 304	311 579
DM/Rijndael-256	256	696	32	98	6	13 438	26 705	53 205	212 305
SM/SEA-192	192	701	24	50	10	134 121	402 055	669 997	2 947 536

Table 4.4.: Additional results.

Hash function	**digest size** [bits]	**code size** [bytes]	**RAM** [bytes] **data**	**state**	**stack**	**cycle count (message length)** (8-byte)	(50-byte)	(100-byte)	(500-byte)
SHA256 (fast)	256	1242	64	73	6	26 208	26 208	52 031	206 969
Keccak[r=544,c=256]	256	672	68	120	4	93 170	93 842	187 153	748 619
Keccak[r=640,c=160]	160	672	80	120	3	93 170	93 842	187 153	656 108
Keccak[r=240,c=160]	160	570	30	60	3	45 394	91 051	181 821	773 026
Skein-256-256	256	1316	32	80	10	212 872	319 154	531 681	1 806 949
lp362/NOEK.	256	650	16	66	10	239 200	478 233	836 696	3 823 871

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, which generally imply different design choices [73, 86]. In a similar way, comparing FPGA implementations based on different manufacturers can only lead to rough intuition about their respective efficiency. In fact, even comparing different architectures on the same FPGA is difficult, as carefully discussed in [59]. 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 analyze the different metrics they provide (area cost, clock cycles, register use, throughput, ...) in a comprehensive manner.

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.

4.3.1 Related works

Table 4.5 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 was implemented in two different ways. The first one, designed by Aumasson *et al.* [13], 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.* [27], 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 an adder/XOR operator within the same slices. The authors also harness the 18-kbit embedded memory blocks (MB) to implement the register file and store the micro-code of the

Table 4.5.: Existing compact FPGA implementations of third round SHA-3 candidates (* padding included, ** Altera ALUTs).

	Algorithm	Scope	FPGA	Area [slices]	Reg.	MB	Clk cyc.	Freq. [MHz]	Thr. [Mbps]
Aumasson *et al.* [13]	BLAKE-32	CF	V5	390	-	-	-	91	575
Beuchat *et al.* [27]	BLAKE-32	FA	S3	124	-	2	844	190	115
Beuchat *et al.* [27]	BLAKE-32	FA	V5	56	-	2	844	372	225
Aumasson *et al.* [13]	BLAKE-64	CF	V5	939	-	-	-	59	533
Beuchat *et al.* [27]	BLAKE-64	FA	S3	229	-	3	1164	158	138
Beuchat *et al.* [27]	BLAKE-64	FA	V5	108	-	3	1164	358	314
Jungk *et al.* [96]	Grøstl-256	FA*	S3	2486	-	0	-	63	404
Jungk *et al.* [95]	Grøstl-256	FA*	S3	1276	-	0	-	60	192
Jungk *et al.* [95]	Grøstl-512	FA*	S3	2110	-	0	-	63	144
Homsirikamol *et al.* [89]	JH-256	FA	V5	1018	-	-	36	381	5416
Homsirikamol *et al.* [89]	JH-512	FA	V5	1104	-	-	36	395	5610
Bertoni *et al.* [24]	Keccak-256	EM	V5	444	227	-	3870	265	70
Namin *et al.* [139]	Skein-256	CF	AS3	1385**	1858	-	72	574	161

control unit. Table 4.5 shows Spartan-3 (S3) and Virtex-5 (V5) implementation results.

Jungk *et al.* [95][96] 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 presented in previous works are applied. The table only reports the best and most recent results from [95]. 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.* [89] 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.* [25]. 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.* [139] presented a low area implementation of Skein. It provides the core functionality and is evaluated on an Altera Stratix-III (AS3) FPGA.

4.3.2 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.* [73], 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 resource 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 that are based on essentially similar architectures. Next, since we are implementing low area designs, we limited the internal data-path 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 reorganize the message in 64-bit words. This is resource consuming and can be added externally if needed by the user. All our cores are designed to be fully autonomous, which will help us in the comparison of the total resources needed by each candidate.

Drimer presented in [59] that implementation results are subject to great variations, depending on the implementation options. Furthermore, comparing different implementations 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 [73]. 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 [73]). We did not specify the throughput for short message, but the information needed to compute it is present in the result tables of Section 4.3.4.

4.3.3 Implementation details

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-dapatath design. Its architecture is illustrated in Figure 4.6, with an overall organization 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 32 or 64 bits, respectively for the BLAKE-256 and BLAKE-512 implementations. 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 can select one of the following inputs: the message blocks (M), the initialization vector (IV), the ALU result or a zero value used to automatically set the salt if no other value has been specified. The user may initialize the salt (S) and a counter (T) using the message block input of the multiplexer. Loading salt or initializing it to zero takes 4 clock cycles. Loading the IV 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.

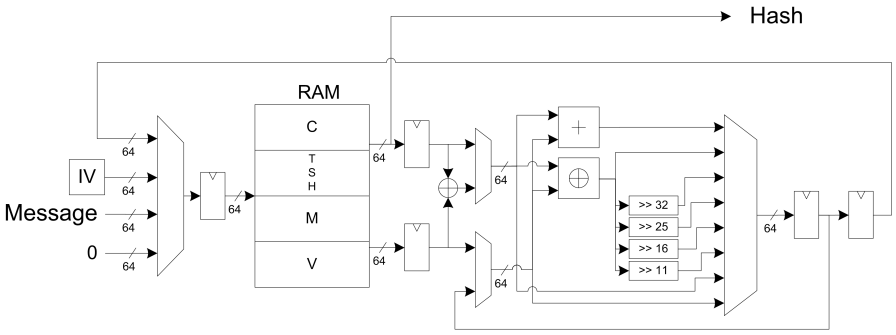

Fig. 4.6.: BLAKE Architecture

The scheduling is made so that, for each call of the round function G (as described in Section 1.2.3), 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.* [27] after adjustement (they considered 14 rounds for the BLAKE-64 version rather than 16), since the overall architectures are very similar.

Grøstl

The 64-bit architecture of Grøstl algorithm is depicted in Figure 4.7. 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 AddRoundConstant 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

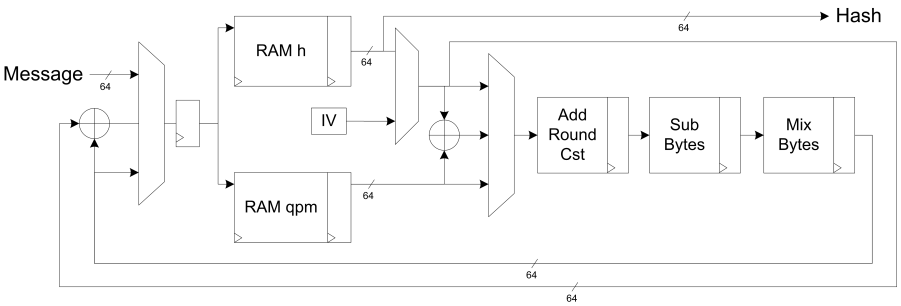

Fig. 4.7.: Grøstl Architecture

(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 textsmallerQ) $\times$ 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 4.8 and is composed as follows. Two 16×32-bit single port distributed RAMs (HASH RAM) are used to store the

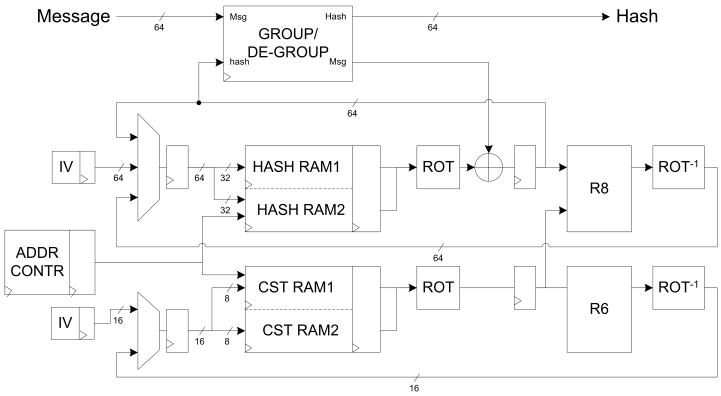

Fig. 4.8.: JH Architecture

intermediate hash values. Those RAMs are first initialized in 16 clock cycles with IV values coming from a 16×64-bit distributed ROM[5] 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. 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

[5]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

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 reorganize 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 4.9, implements the Keccak version proposed as SHA-3 standard. It works on state of 1600 bits organized into 24 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.

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.

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 explicitly instantiating a 64-bit barrel rotator and by storing the rotation offsets into a dedicated look up table. Using a single barrel rotator it is possible to significantly reduce the area requirements. While this negatively affects the performances (since all the 25 words of the state need to be processed by the same component), it allows reaching an overall cost that is comparable to the one of the other algorithms, as will be detailed in the next Section.

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

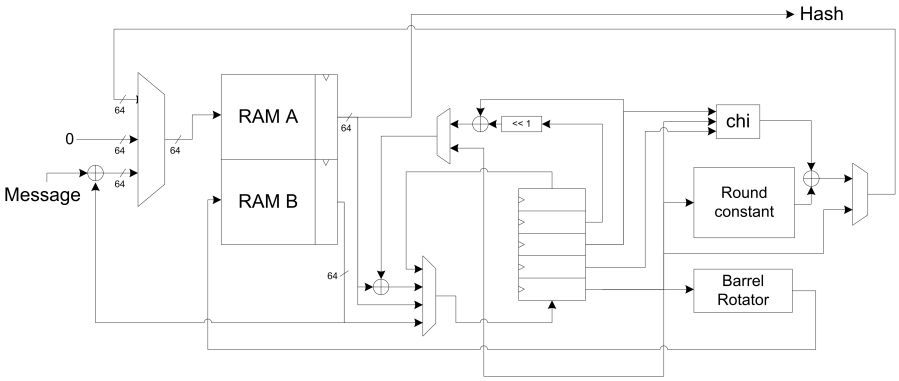

Fig. 4.9.: Keccak Architecture

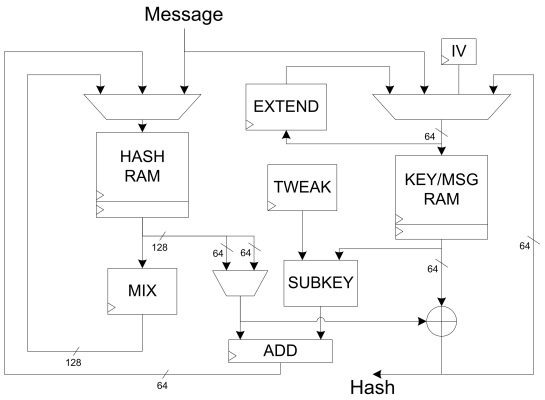

Fig. 4.10.: Skein Architecture

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).

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 4.10. 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.

4.3.4 Implementation results & discussion

The complete implementation results for our different architectures are given in Tables 4.6 and 4.7 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 difference is our exploitation of distributed memories (reported in the slices count) rather than embedded memory blocks. 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 (although not by a big margin). 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.

Table 4.6.: Implementation results for the 5 SHA-3 candidates on Virtex-6 (512-bit digests).

		BLAKE	Grøstl	JH	Keccak	Skein[6]	AES
Properties	Input block message size	1024	1024	512	576	512	128
	Cycles per block	1342	448	688	2137	458	44
	Cycles overhead (pre/post)	12/8	24/354	16/20	9/8	9/466	8/0
Area	Number of LUTs	701	912	789	519	857	658
	Number of Registers	371	556	411	429	350	364
	Number of Slices	192	260	240	144	266	205
	Frequency (MHz)	240	280	288	250	196	222
	Throughput (Mbit/s)	183	640	214	68	219	646
	Efficiency (Mbit/s/slice)	0.95	2.46	0.89	0.47	0.82	3.15
Timing	Number of LUTs	810	966	1034	610	861	845
	Number of Registers	541	571	463	533	357	524
	Number of Slices	215	293	304	188	243	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	0.91	3.08

Table 4.7.: 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
	Cycles per block	1342	448	688	2137	458	44
	Cycles overhead (pre/post)	12/8	24/354	16/20	9/8	9/466	8/0
Area	Number of LUTs	719	912	737	525	1007	685
	Number of Registers	370	574	338	433	441	365
	Number of Slices	230	343	260	193	337	232
	Frequency (MHz)	135	240	113	166	137	125
	Throughput (Mbit/s)	103	548	84	45	153	364
	Efficiency (Mbit/s/slice)	0.47	1.60	0.32	0.23	0.45	1.57
Timing	Number of LUTs	856	766	1106	640	1182	852
	Number of Registers	594	759	646	476	456	529
	Number of Slices	303	281	362	216	344	274
	Frequency (MHz)	150	265	175	166	145	154
	Throughput (Mbit/s)	114	605	130	45	162	448
	Efficiency (Mbit/s/slice)	0.38	2.15	0.36	0.21	0.47	1.64

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

[6]Due to some errors found in our first Skein implementation results, the results presented here slightly differ from those presented in the reference paper [105]. This also holds for Tables 4.7 and 4.8.

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.

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 [73]. 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 [90], where it is stated that Keccak is not straightforwardly suitable for folding. 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 [24].

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 4.8 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 RAMs 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.

Table 4.8.: 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
	Cycles per block	1182	176	688	2137	230	44
	Cycles overhead (pre/post)	12/8	24/122	16/20	17/16	5/234	8/0
Area	Number of LUTs	417	912	789	519	857	658
	Number of Registers	211	556	411	429	350	364
	Number of Slices	117	260	240	144	266	205
	Frequency (MHz)	274	280	288	250	196	222
	Throughput (Mbit/s)	105	815	214	128	218	646
	Efficiency (Mbit/s/slice)	0.90	3.13	0.89	0.89	0.82	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

To conclude this section, 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.

4.3.5 Following works

Some other works concerning hardware implementations of the five SHA-3 finalists have been published subsequently to the time of the realization of our work. In particular, [94] and [100] propose fully autonomous compact implementations of the SHA-3 finalists' variants having 256-bit digests. In [100], Kaps *et al.* come up with two sets of implementations having different goals. The first set is build using only logic elements and with an area constraint set to 800 slices[7], while in the second set, one block RAM is used and the implementation area must be in the range of 400 to 600 slices. Everything was developed for the Xilinx Spartan-3 devices but implementation results are also given for other families of FPGAs. Except for their implementations of BLAKE and Skein, which are respectively more and less efficient than ours, their results are close to ours. Jungk *et al.* [94] propose area-optimized implementations which are initially intended for Virtex-5 devices. However, results are also given for Spartan-3, Spartan-6 and Virtex-6 FPGAs. The main difference in their approach compared to previous ones is that they do not fix

[7]Note that when implemented on Virtex-6 devices, their designs display numbers of slices in the range of 100 to 300 slices.

the datapath to a common size, e.g. Keccak architecture has a 200-bit wide datapath, while BLAKE works on 64 bits at a time. That way, they were able to implement more efficient architectures for Keccak and JH with areas close to 400 slices. Grøstl remains however the most efficient of all the algorithms, and Skein becomes the less efficient one.

4.4 CONCLUSION

This chapter provided compact implementations of hash function algorithms on different types of devices. The goal of those implementations was to try to better understand the impact of hash function constructions on their efficiency once implemented in a real life context.

The main observation that can be extracted from the results presented in the previous sections is that hash function implementations produce results with much larger differences than block cipher implementations. This may be explained by several factors. First, the state size varies greatly from one hash function to the other, e.g. the SHA-3 version of Keccak has a 1600-bit state while block cipher-based and lightweight hash functions have state sizes that can be up to 6 times smaller. Then, unlike block ciphers for which most of the constructions are extremely close to each others (as was shown in Chapter 3), hash function designers have proposed constructions that have great versatility. Finally, some hash functions, such as the sponge ones, propose different tradeoffs in their constructions (based on the choice in rate and capacity), which results in different implementation contexts.

From the software implementation results, we can also conclude that presently, the block-cipher based constructions using AES or Rijndael-256 constructions are the ones giving the best performances (when looking at combined metrics). This is probably related to the fact that these two algorithms have by far been more studied than hash functions in general and are thereby more mature. Another reason could also be that some SHA-3 designers took large security margins in comparison with best known attack complexities whereas AES and Rijndael constructions tend to be so optimized that their security margins are small.

In conclusion, we believe that this comparative study gives interesting results and raises important questions that could be covered in further researches. In particular, finding which is the best approach for designing lightweight hash functions would be an interesting topic. It is also important to insist on the importance of the compact implementation scenario in algorithm comparisons since it seems to be a less trivial context than the high throughput one and helps reveal the complexity of the algorithms. We also believe that with time, several improvements will be proposed to help implementing compact version of the newly standardized SHA-3 algorithm.

CHAPTER 5

IMPLEMENTING COUNTERMEASURES AGAINST SIDE-CHANNEL ATTACKS

5.1 INTRODUCTION

Providing (physical) security against side-channel attacks is a challenging task for cryptographic designers [121]. This is especially true for low-cost embedded devices with strongly constrained resources. Already in the first *differential power analysis* (DPA) paper, Kocher et al. mentioned time randomization as a possible solution to improve security against side-channel attacks [110]. Following, different countermeasures have been proposed to exploit this idea, e.g. relying on the addition of random delays [42, 188], shuffling the execution order of independent operations [87, 164], or more generally, trying to build a non-deterministic processor [19, 126].

In parallel to these advances, some more recent works have tried to formalize the problem of physical security, in order to extend the guarantees of provable security from algorithms and protocols to implementations. The main challenge in this case is to find relevant restrictions of the adversaries. A typical example is the one of leakage-resilient cryptography, where the assumption is that the information leakage of a single algorithm run is bounded (see, e.g. [61] for an early reference, [177] for a recent one, and many other proposals in between). Alternatively, another line of work is based on the assumption of secure pre-computations, e.g. in order to prevent "continual memory attacks", as formalized by Brakerski et al. [35] and by Dodis et al. [57]. The one-time programs introduced at Crypto 2008 are another (extreme) way to exploit such secure pre-computations [77]: they essentially correspond to a program that can be executed on a single input, whose value can be specified at run time. Nevertheless, the practical relevance of these solutions is still limited by sometimes unrealistic hardware assumptions and (mainly), by large performance overheads.

As usual in side-channel attacks, the main question regarding countermeasures against side-channel attacks is: "to what extent do they improve security

and at which cost?". For this purpose, we first propose, in Section 5.3.2, a comprehensive evaluation of the shuffling countermeasure. Next to that, we observe that both for practice-oriented and theory-oriented countermeasures, the exploitation of secure pre-computations is highly related to the problem of fast and efficient non-volatile storage. We therefore suggest, in Section 5.3, the use of a new RAM technology, i.e. the FRAM, in order to improve the performances and security of protected implementations.

5.2 SHUFFLING AGAINST SIDE-CHANNEL ATTACKS: A COMPREHENSIVE STUDY WITH CAUTIONARY NOTE

In general, shuffling can be applied to any set of independent operations. The SubBytes layer of 16 S-Boxes in the AES is a typical example. Randomizing such operations can be done in different ways. Taking the extreme cases, either the S-Boxes are executed according to a random permutation (RP) among 16! possible ones, or they are executed from a random start index (RSI) among 16 possible ones, that is then incremented. This difference is nicely illustrated with previous works on shuffled implementations of the AES. In a first paper from 2006 [87], the authors use partial shuffling and first-order masking based on S-Box pre-computation. Whereas masking is applied to the whole cipher, shuffling is only applied to the first and last rounds. Furthermore, the RSI approach is pursued, for performance reasons. In a second work from Rivain et al., higher-order masking is implemented and shuffling is mainly based on the RP approach [164]. Yet, for the MixColumns operations, only the (first-order masked) columns are shuffled, accounting for 8! possible permutations. That is, thanks to the first-order masking, they have 8 positions that can be shuffled, vs. 4 if MixColumns was not masked, and 16 for the other AES transforms. Implementation details are not given in [164], but we assume that this choice is again motivated by performance reasons. Apart from those works, shuffling was also applied to hardware implementations with 8- or 32-bit datapaths, where RSI is usually preferred as it nearly comes for free [68, 127, 136].

Following this state-of-the art, our first contribution is to improve the performances of software implementations using the RP approach (Section 5.2.1). In this respect, we start from the observation that in an unprotected block cipher implementation, one usually keeps as much data as possible in the processor registers, in order to minimize RAM access. By contrast, once random access to these registers is required (as in shuffled implementations), RAM usage is inevitable. This implies that any register access becomes a serial of load and store operations, resulting in major performance overheads. We mitigate these overheads by exploiting a different technique, which consists in manipulating the program flow. It allows us to operate on registers while at the same time randomizing the sequence of operations. In practice, we present two approaches: the first one changes the program flow "on-the-fly", while the other one re-writes the program memory prior to execution. The latter

approach can be viewed as an adaptation of the self-modifying codes used in software engineering [7], also applied to counteract side-channel attacks in [6]. Our new solutions come with contrasted performance results. Namely, the on-the-fly proposal minimizes the overall cycle count, while the program memory manipulations allow very efficient online encryption at the cost of long (possibly offline) pre-computations. For illustration, we apply these proposals (and previously published ones) to the AES Furious implementation available from [150]. Besides, we also investigate the efficient generation of (small) random permutations in low-cost microcontrollers. That is, we take a well known optimal algorithm for permutation generation and modify it slightly, in order to improve its performances. As a result, we are able to generate close-to-uniform permutations, and obtain an efficient alternative to proposals such as [45].

Next, we investigate the security of shuffling against side-channel attacks (Sections 5.2.2 to 5.2.4). Here, we start from the observation that the existing literature usually evaluates the impact of shuffling based on a so-called "integrated DPA" (aka windowing attack), introduced in [42] and applied, e.g. in [164, 185]. Intuitively, if the manipulation of a sensitive variable is spread over t time samples, its correlation with the actual leakages will be reduced by a factor $\sqrt{t}$ using such an attack, instead of t without integration. Integrating is a convenient tool for evaluation as it can be directly used to estimate the data complexity of a DPA using Mangard's formulas [120]. Yet, a possible limitation of this technique is that it treats the RSI and RP cases in the same way. Hence, a natural question is to determine whether these two approaches are indeed equivalent in general, or if advanced evaluation tools can be used to put forward additional weaknesses for RSI implementations. Our results regarding this question are summarized as follows.

First, we specialize the information theoretic and security analysis from [176] to the context of shuffled implementations. It allows us to confirm that integrated DPA is indeed suboptimal compared to a Bayesian exploitation of the leakages. While our worst-case evaluations rely on profiled attacks [41, 170], we believe they are important to moderate claims of strong security improvements provided by shuffling (e.g. the data complexity increases by a factor 360 in [19]). In particular, these results complement the previous work of Asiacrypt 2010 [182], in which such an information theoretic and security analysis was performed for masking. As a result and for the first time, we obtain lower bounds for the data complexity of standard side-channel attacks against shuffled implementations.

Second, we notice that security evaluations for masking always combine the leakage corresponding to the masked data and its masks, e.g. [131, 155]. Quite surprisingly, and to the best of our knowledge, the impact of such a scenario has not been investigated in the case of shuffling. Therefore, we include the possibility of a leakage on the permutation (or start index) manipulated when shuffling. We show that as soon as some information is leaked about them,

attacks against RSI- and RP-based implementations become significantly different, the RSI case being much easier to attack, for computational reasons.

Finally, we observe that direct leakages about the start index or permutations naturally arise in practice and can be exploited. More surprisingly, we also show the existence of "indirect leakages", coming from the different power consumption models of the hardware resources manipulating the key bytes. For example, since the 16 registers used in our shuffled Furious implementations have (slightly) different models, marginalizing the distribution of the observed leakage over the 16 AES key bytes provides information about which S-box is computed.

Summarizing, we observe that *all* previous works on shuffling reduced the size of the permutation set for some of the operations in the protected algorithm. Hence, our results bring the important cautionary note that time complexity is critical in the security evaluation of this countermeasure, as permutations with a small size can be enumerated which leads to exploitable weaknesses. In this respect, an implementation protected with RSI-based shuffling can sometimes be as weak as an unprotected one. As for the RP-based solution, we recall that it can be used as a noise amplifier for leaking devices, but never as a noise generator.

5.2.1 Efficient implementations

This section explores the software design space for shuffling the AES on an ATMEL ATmega644P microcontroller [9]. We first describe an efficient way to obtain close-to-uniform permutations in this device. Next, we show how to obtain an AES implementation for which every transform can be shuffled according to such permutations, including MixColumns (and the key scheduling algorithm if needed). Afterwards, we describe different implementations: a basic one relying on a previously proposed "double indexing" method, and two optimized ones relying on randomized execution path and program memory. We finally provide precise performance evaluations and a comparison with previous works.

Permutation generation

The first building block of a shuffled implementation is a permutation generator. From a sequence $S := \{1, \ldots, n\}$, a uniform permutation can be produced in linear time [109]. The original algorithm iterates over every element S_i (with i from 0 to $n-2$), and swaps it with a random element from the remaining tail, i.e. $\{S_i, \ldots, S_{n-1}\}$. However, sampling from $\{i, \ldots, n-1\}$ needs either a modulo operation and a random number greater than n to start from, or an approach with probabilistic run-time. We avoided this performance drawbacks by sampling from $\{0, \ldots, n-1\}$. Permuting a sequence of 16 entries following this algorithm takes 362 cycles on our device, using 8 bytes of randomness. It still allows to generate all permutations, but with a slight bias that decreases with the size of the permuted set. To estimate the impact of

this bias for different sizes of the permutation set N, we systematically sampled 10^8 permutations generated with this method, and built histograms with $N!$ bins. We then estimated the Euclidean distance between these biased histograms and a uniform distribution. In addition, we compared this situation with the one obtained with a quite minimum side-channel leakage. Namely, we assumed that the Hamming weight of the first entry of a (uniformly generated) permutation is known to be the least informative one (i.e. with half of the bits set to one). As can be observed in Table 5.1, the bias due to this small side-channel information is already significantly larger than the one due to the permutation generation algorithm. Furthermore, actual leakages in Sections 5.2.2 and 5.2.3 affect all the permutation entries, which further reduces the bias of the permutation generation algorithm compared to the one caused by physical information. Eventually, we will show in the next sections that exploiting these biases in a side-channel attack where we shuffle among 16! possible permutations is computationally hard. Therefore, we conclude that our performance optimized algorithm should not lead to a significant security reduction of the shuffling countermeasure.

Table 5.1.: Bias of the optimized permutation algorithm (OPA) vs. bias of a small SCA leakage (SSCAL).

N	3	4	5	6	7	8	9
OPA	0.04535	0.03522	0.02034	0.00993	0.00430	0.00170	0.00063
SSCAL	0.28868	0.20412	0.07454	0.03726	0.01627	0.00643	0.00234

Obtaining independent operations

Applying shuffling to an implementation requires finding sets of independent operations. In the AES case, sets of 16 independent operations naturally arise from the AddRoundKey and SubBytes transforms. By contrast, the situation is a little bit trickier for ShiftRows and MixColumns. For example, implementing ShiftRows requires one extra byte of storage in an unprotected implementation, and two in the case of RSI-based shuffling (i.e. when the permutation is "monotonous", which restricts the number of permutations to 16). But if 16 independent operations are desired, 16 bytes of temporary storage are required. As for MixColumns, four additional registers are sufficient if the state is processed column-wise, but this would then account for 4! permutations. Hence, having 16 independent operations again requires 16 bytes of temporary storage. Since our device has only 32 registers, some of which being already occupied, RAM usage becomes inevitable for shuffling. Besides, the key schedule has only four independent operations by default. This is because within one key schedule round, there are only four S-Box executions. Thus, the smallest number of indistinguishable operations is four. Yet, applications requiring on-the-fly key expansion also need an appropriate simple power analysis (SPA)

protection to prevent attacks such as [119]. In these cases, we interleaved the real key schedule with three dummy key schedules, in order to obtain 16 shuffleable operations.

Basic implementation with double indexing

Direct shuffling requires an indirect indexing of the operands. That is, a counter is used to index a permutation vector, and the result is used to index the operand vector. Thus, instead of operating on registers directly, two RAM accesses are required for each (read or write) access to operands. This naturally leads to quite large cycle counts, as in AVR devices, load and store operations take two cycles (compared to one cycle for arithmetic and logic operations). Implementing a fully shuffled AES this way results in an execution time of 30 202 cycles, excluding the key schedule. In the following we propose two different strategies in order to improve on these figures. In both cases, instructions are shuffled rather than data location, in order to allow register usage. Precisely, we are still limited by the number of available registers when performing certain transforms. But contrary to the double indexing proposal we do not always access RAM when operating on intermediate data. The first solution changes the execution path on-the-fly while the second actually rewrites the program memory (i.e. assuming that this re-writing is pre-computed, this solution can be seen as a simplified one-time program [77]).

Optimized implementation with randomized execution path

For this implementation, the assembly code of every (compound of) round transform(s) is split into 16 independent blocks of instructions. Each of the 16 blocks is augmented with a label. This allows us to identify its address in ROM. Furthermore, every transform is associated with an array of 17 16-bit words, where the first 16 words hold the addresses of the 16 blocks, and the 17th holds the address of the return instruction. The array content is initialized with the addresses of the labels at compile time. Finally, we append a flow-control macro to each of the 16 blocks. This macro performs three things: fetch an address from the array, advance the pointer to the next array entry, and jump to the fetched address.

During the execution of the cipher, we first re-order the first 16 addresses in the array, according to a previously generated permutation. Then, when we enter a transform, we set a pointer to the beginning of the array and execute the flow-control macro. This causes the execution of the first block and sets a pointer to the address of the next block. The flow-control macro is executed 16 times, until it finally looks up the address of the return instruction. In practice, we defined several sets of transforms and therefore need an address array for each of them. The first one is the compound of AddRoundKey, SubBytes, and ShiftRows. This transform reads the state from RAM and stores the result into the register file. The next one performs MixColumns and stores the result back to RAM. Afterwards, we perform one iteration of the key schedule. Similar

to ShiftRows and MixColumns, this implies additional memory requirements, because of the RotWord operation. We finally need a standalone AddRoundKey layer for the last round. As each of the address arrays need 17x2 bytes, we have an additional RAM use of 170 bytes (for technical reasons, the key schedule uses two 17x2-byte address arrays). Permuting each of these arrays takes 205 cycles, implying an overhead of 1225 cycles. Eventually, for every set of transforms, we need to load an address and jump to this address 17 times, each of which takes 6 cycles. Together with the preamble to set up the array pointer, it leads to an additional overhead of 108 cycles for each of these compounds of transforms.

Optimized implementation with randomized program memory

For this implementation, we used the self-programming capabilities of the ATmega644P microcontrollers. As the shuffling applies to independent operations, and as for each operation, the state bytes are always stored in the same registers or RAM locations, the execution order of the operations can be permuted by modifying the data corresponding to these locations in program memory. In our target controller, the program memory has to be modified one page (i.e. 256 bytes) at a time. Hence, the shuffling can be prepared in five steps. First, the page is transferred from program memory to the RAM. Afterwards, the bytes of code corresponding to state-byte locations are modified according to the permutation vector. Then, the previous version of the page is erased from program memory, and the new page is loaded into a page buffer. Finally, this page buffer is written in program memory. This process is executed before each AES execution. The main advantage of this solution is that after pre-processing of a shuffled program memory, the execution time of the AES is nearly the same as for the unprotected implementation. Minor differences come from the fact that we need to have independent operations, which implies to use RAM for the storage of some intermediate results. Its main drawback is the long pre-computation time, which accounts for approximately 18 milliseconds independently of the clock frequency. This comes from the time-consuming instructions used to erase program memory and write page buffer in memory (4.5 millisecond per page writen or erased [46]), and the low granularity of these instructions (i.e. working at the page level) in the ATMEL controllers. More flexible devices (e.g. devices with ARM architectures) would allow to improve this limitation. Note also that our target ATMEL's flash memory allows only for 10 000 re-write cycles, which could possibly lead to denial of service (DoS) attacks. If this is an issue, and depending on the actual available ROM, different areas can be used randomly and increase the number of possible encryptions by some factor. Again, alternative devices could be considered to relax this limitation. For example, the ARM LPC214x series allows already for 100 000 cycles. Note finally that, as this implementation mainly makes sense if pre-processing is allowed, it is naturally executed with a pre-computed key scheduling.

Implementation results

The performance results of our implementations are compared with previous works in Table 5.2. Namely, we use the AES Furious as reference. As for protected implementations, we considered the basic one based on double indexing and the ones of Herbst et al. and Rivain et al. However, as mentioned above, they do not allow direct comparison. Herbst et al. only protect the outer rounds (one and ten) with RSI-based shuffling, but implement masking for all the rounds and the key schedule. Rivain et al. implement higher-order masking and use a "simplified" shuffling for the MixColumns operation (they also work on a different 8051-based architecture). The implementation for which we give cycle numbers is not masked except for MixColumns. By contrast, our implementations use log-table based polynomial multiplication, and are able to shuffle all bytes during MixColumns. Not surprisingly, our implementation based on double indexing is the slowest. Its performance is comparable to the one of Rivain et al. Manipulating the program counter allows us to get a performance improvement of almost a factor five and, excluding the key scheduling, leads to encryption time only twice as slow as Rijndael Furious. As previously mentioned, the larger overheads when executing the key scheduling come from the need to execute additional dummy schedulings, in order to keep a permutation among 16! for this part of the implementation. Finally, the randomized program memory allows the fastest online encryption (i.e. excluding program re-writing).

Table 5.2.: Implementation result comparison

Implementation	Clock cycles	RAM [byte]
Furious [150]	2 739	176
Furious with KS [150]	3 546	176
Herbst et al. [87]	11 845	-
Rivain et al. [164]	29 400	-
Dbl. ind.	30 202	240
Dbl. ind. with KS	46 395	132
Rand. exec. path	6 934	394
Rand. exec. path with KS	14 834	302
Rand. prog. mem.	3299 (+≃18 msec)	480

5.2.2 Evaluation framework

We now move to the security analysis of the shuffling countermeasures and its variants. For this purpose, we rely on the evaluation framework from [176] and adapt it to capture the specificities of shuffled implementations. In order to have a fair understanding of the strengths and weaknesses of the countermeasure, we pay a particular attention to worst-case (profiled) attacks. But for

completeness, we also compare them with the integrated DPA used in previous works.

Notations

Variables are denoted with capital letters, sampled values with lowercase letters and functions with sans serif fonts. We consider the standard DPA attacks described in [122] and illustrate our notations with the case of the AES. In this context, the adversary tries to recover a 16-byte master key $\mathbf{k} = \{k_0, k_1, \ldots, k_{15}\}$, from a leakage corresponding to the first key addition and S-Box layers. In attacks against unprotected implementations, each S-Box is executed at a well defined time instant, giving rise to key leakages defined as:

$$\begin{array}{lll} L_0 & \leftsquigarrow & \mathsf{Sbox}(k_0 \oplus X_0), \\ L_2 & \leftsquigarrow & \mathsf{Sbox}(k_2 \oplus X_2) \end{array} \qquad \begin{array}{lll} L_1 & \leftsquigarrow & \mathsf{Sbox}(k_1 \oplus X_1), \\ & & \ldots \end{array}$$

That is, we have 16 leakage points (or cycles) L_c (where c is the cycle index) and 16 subkeys k_s. If we denote the part of the master key that is manipulated at time c with a variable S_c, we straightforwardly have $S_c = c$ in this unprotected case. Note that the variable nature of the leakages comes both from possible noise in the measurements and the variable (known) inputs X_i. By contrast, in the case of a shuffled implementation, the execution order of the S-Box computations is randomized according to a permutation P, leading to key leakages of the form:

$$\begin{array}{lll} L_0 & \leftsquigarrow & \mathsf{Sbox}(k_{\mathsf{P}(0)} \oplus X_{\mathsf{P}(0)}), \\ L_2 & \leftsquigarrow & \mathsf{Sbox}(k_{\mathsf{P}(2)} \oplus X_{\mathsf{P}(2)}) \end{array} \qquad \begin{array}{lll} L_1 & \leftsquigarrow & \mathsf{Sbox}(k_{\mathsf{P}(1)} \oplus X_{\mathsf{P}(1)}), \\ & & \ldots \end{array}$$

That is, we have $S_c = \mathsf{P}(c)$ with P the secret permutation that is re-generated for every new input block, e.g. with the algorithm in Section 5.2.1. In this protected case, not only leakage about the S-Box execution may be obtained, but also leakage on the permutation used in the shuffled implementation. In theory, an attack could exploit sixteen "direct" permutation leakages denoted as $L'_c \leftsquigarrow S_c$. Such notations allow us to reflect both the RSI- and RP-based shuffling methods. In the first case, we have $\mathsf{P}(c) = c + \tau \pmod{16}$, with $\tau \xleftarrow{R} [0:15]$. In the second case, P is directly picked up among the set of all 16! permutations, i.e. $\mathsf{P} \xleftarrow{R} \mathcal{P}_{16}$.

Information theoretic analysis

As a first step in our evaluation, we perform an information theoretic analysis that is aimed to capture the worst-case security of an implementation. In general, and for a fixed key byte K_s, we assume that the adversary can observe a leakage vector $\mathbf{L} = \{L_0, L_1, \ldots, L_{15}\}$. The goal of this evaluation is to obtain

an accurate estimation of the mutual information[1]:

$$\begin{aligned}\mathrm{MI}(K_s; \mathbf{L}, X) &= \mathrm{H}[K_s] - \sum_k \Pr[K_s = k] \sum_x \Pr[X = x] \\ &\cdot \int_l \Pr[\mathbf{L} = \mathbf{l} | K_s = k, X = x] \cdot \log_2 \Pr[K_s = k | \mathbf{L} = \mathbf{l}, X = x] \; dl.\end{aligned}$$

In this equation, the term $\Pr[K_s = k|\mathbf{L} = \mathbf{l}, X = x]$ is directly obtained from $\Pr[\mathbf{L} = \mathbf{l}|K_s = k, X = x]$ using Bayes' theorem. Hence, it is this last conditional leakage probability that is most critical to evaluate. For convenience, we will ignore the variable X in the rest of the paper, as it is assumed to be known for all computations. Next, we will consider two main evaluation scenarios.

1. **No permutation leakage,** i.e. the adversary gets 16 leakage cycles, and each of them could correspond to the target subkey with probability 1/16. That is:

$$\Pr[\mathbf{L} = \mathbf{l}|K_s = k] = \sum_c \frac{1}{16} \Pr[L_c = l_c|K_s = k].$$

   We will refer to this attack as case $(1.a)$. Besides, and as mentioned in introduction, a usual trick to attack shuffled implementations is to integrate over the leakage cycles. In this case, the adversary defines a variable $\overline{L} = \sum_c L_c$, and performs the attack against this variable. It boils down to consider 15 cycles out of 16 as "algorithmic noise". We will refer to this attack as case $(1.b)$.

2. **Leakage on the permutation.** In the same way as all the shares are assumed to leak in a masked implementation, it is natural to assume that the manipulation of a permutation may leak in a shuffled implementation. In practice, such leakages usually appear each time the permutation is manipulated in the microcode, e.g. when fetching the S_c'th part of the key, or when jumping to the S_c'th piece of code computing an S-Box. We now show how to perform an information theoretic evaluation in these cases. As previously, the impact of different implementations of the countermeasure affects the term $\Pr[\mathbf{L} = \mathbf{l}|K_s = k]$. For this purpose, we start with the following general formulation:

$$\Pr[\mathbf{L} = \mathbf{l}|K_s = k] = \sum_c \frac{\mathsf{f}(c, s, \mathbf{l}')}{\sum_{c'} \mathsf{f}(c', s, \mathbf{l}')} \Pr[L_c = l_c|K_s = k],$$

[1] As discussed in [160], this mutual information can only be perfectly estimated when the evaluator knows the exact leakage model of his target device. This only happens in simulated analyses (e.g. as will be performed in the next section). Whenever a practical evaluation is carried out, it is formally a "perceived information" that is evaluated, with the goal to be as close as possible to the mutual information.

with $\mathbf{l}'$ the vector of 16 leakages on the previously defined variable S_c (indicating the part of the master key used at time c). The function f essentially indicates how the knowledge available about this variable can be exploited by the adversary, as witnessed by the five examples that we now describe.

2.*a. Unprotected implementation.* In this case, we have $\mathsf{f}(c, s, \mathbf{l}') = 1$ if $c = s$ and 0 otherwise (i.e. the adversary knows exactly where each key byte is manipulated).

2.*b. Direct template attack.* In this case, we just add the permutation leakage in the conditional probabilities, yet without making any difference between the RSI and RP cases, by computing $\mathsf{f}(c, s, \mathbf{l}') = \Pr[L'_c = l'_c | S_c = s]$. Note that the case with no permutation leakage corresponds to $\mathsf{f}(c, s, \mathbf{l}') = 1/16$.

2.*c. Taking advantage of* RSI. Here, the the adversary exploits the fact that only 16 permutations are possible (out of the 16! ones), which can be enumerated. Hence, he can compute: $\mathsf{f}(c, s, \mathbf{l}') = \prod_{i=0}^{15} \Pr[L'_i = l'_i | S_i = (s - c + i) \bmod 16]$.

Contrary to the RSI case, using a RP implies that the permutation is picked up randomly among the 16! $\simeq 2^{44}$, which is significantly harder to enumerate. Hence, our following experiments will additionally consider two heuristic solutions that can be used to mitigate this issue and attack more efficiently.

2.*d. Restricted enumeration against* RP. In this case, the function f is identical to the exhaustive one, i.e. $\mathsf{f}(c, s, \mathbf{l}') = \sum_{\mathsf{p}} \prod_{i=0}^{15} \Pr[L'_i = l'_i | S_i = \mathsf{p}(i)]$, but the sum only goes over an enumerable subset of most probable p's. A *beam search* is used for this purpose [213]. This is a breadth-first search that limits the number of nodes (i.e. permutations in the sum) by pruning the least probable ones, which is done by weighting permutations p's with $\prod_{i=0}^{15} \Pr[L'_i = l'_i | S_i = \mathsf{p}(i)]$.

2.*e. Excluding heuristic.* One alternative option to simplify the enumeration is to consider that whenever $S_c = s$, we have that $c \neq c'$ implies $S_{c'} \neq S_c$. This can be reflected with: $\mathsf{f}(c, s, \mathbf{l}') = \Pr[L'_c = l'_c | S_c = s] \cdot \prod_{c' \neq c} (1 - \Pr[L'_{c'} = l'_c | S_{c'} = s])$, which, up to normalization, is equivalent to:

$$\mathsf{f}(c, s, \mathbf{l}') = \frac{\Pr[L'_c = l'_c | S_c = s]}{1 - \Pr[L'_c = l'_c | S_c = s]}.$$

Overall, an intuition on the security of different implementations is obtained by quantifying the number of possible execution orders considered by the adversary (which may be more than the actual number of permutations, if attacks do not fully exploit their structure). In the unprotected case, only one order can occur. For the direct template attack, the adversary does not

combine the different S_c informations and we implicitly have 16^{16} possible execution orders. In the RSI case, we exploit the fact that only 16 permutations are possible. The attack enumerating all possible permutations lists all 16! hypotheses. Finally, the excluding heuristic implicitly allows 16×15^{15} ones. This situation can be seen as an error correcting problem where 16 noisy values are transmitted, that can be integers from 0 to 15. The security of the countermeasure relies on a large probability of decoding error. In the RSI case, we only have 16 possible codewords, which gives us a very resilient code, lowering the probability of errors and thereby the strength of the countermeasure. For a RP, we have 16! codewords over a space of 16^{16} possible transmissions, hence increasing the probability of decoding errors.

As far as performing these attacks/evaluations in practice is concerned, case (a) is a classical template attack for which the computational complexity is usually neglected. Carrying out attacks/evaluations where L' is exploited naturally requires to build additional templates. Yet, the computational complexity of cases (b), (c) and (e) can also be neglected, as they only imply a few additional arithmetic operations. In fact, only case (d) may require intensive computations, if all permutations with non-negligible likelihood (with respect to L') are taken into account by the beam search. As will be shown in the next section, increasing the noise gradually implies that all permutation candidates have more similar likelihoods. Hence, this last attack is only applicable for low noise levels.

Security analysis

The second step of our evaluation is to perform a security analysis. It allows measuring the extent to which the different strategies listed have a strong impact on the data complexity of successful side-channel attacks. For this purpose, we apply template attacks with the key selected as:

$$\tilde{k} = \underset{k^*}{\operatorname{argmax}} \prod_{j=1}^{q} \Pr[\mathbf{L}^j = \mathbf{l}^j, \mathbf{L'}^j = \mathbf{l'}^j | K_s = k^*],$$

and we compute their success rate, in function of the data complexity q.

5.2.3 Simulated experiments

In order to gauge the impact of the proposed formulations and attacks, we first lead various experiments against simulated AES implementations. For this purpose, we re-use the notations introduced in the previous section and assume that the adversary is provided with a key leakage vector $\mathbf{L}$ with elements $L_c = \mathsf{HW}(\mathsf{Sbox}(k_{\mathsf{P}(c)} \oplus X_{\mathsf{P}(c)})) + \mathcal{N}(0, \sigma^2)$, and possibly a permutation leakage vector $\mathbf{L}'$ with elements of the form $L'_c = \mathsf{HW}(S_c) + \mathcal{N}(0, \sigma^2)$. In both cases, the second term is a Gaussian distributed random noise, with variance σ^2 that we will use as a parameter of our evaluations. Using these notations, there

Table 5.3.: Classification of the attacks

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="3">Target devices</th></tr>
<tr><th>Unp.</th><th>RSI shuf.</th><th>RP shuf.</th></tr>
<tr><td rowspan="4">adversary's means</td><td rowspan="2">L</td><td rowspan="2">UNP-TA $(2.a)$</td><td colspan="2">INT-TA $(1.b)$</td></tr>
<tr><td colspan="2">UNI-TA $(1.a)$</td></tr>
<tr><td>L,L′</td><td></td><td colspan="2">DPLEAK-TA $(2.b)$</td></tr>
<tr><td>L,L′ + comp.</td><td></td><td rowspan="2">RSIENUM-TA $(2.c)$</td><td>RESENUM-TA $(2.d)$</td></tr>
<tr><td colspan="2">EXCLUDING-TA $(2.e)$</td></tr>
</table>

are various contexts that could be investigated. As illustrated in Table 5.3, we classify them among two axes: the target device and the adversary's means.

As far as the target device is concerned, we considered the case of an unprotected implementation for reference, an RSI-based shuffled implementation and a RP-based shuffled implementation. As far as the adversary's means are concerned, we first analyzed attacks where only the key leakage vector $\mathbf{L}$ is available. Next we evaluated attacks where the permutation leakage vector $\mathbf{L}'$ is additionally provided. Finally, we quantified the efficiency gains obtained when exploiting computational power, in order to enumerate (i.e. sum over) the possible permutations. Overall, this gives rise to seven attacks:

1. Template attack against the unprotected implementation (UNP-TA), i.e. the straightforward case where S-Boxes are executed in deterministic order.

2. Template attack against integrated leakages (INT-TA), i.e. the attack against shuffled implementations previously used, e.g. in [42, 164, 185].

In these two first cases, template attacks and correlation DPA are essentially equivalent given that they exploit the same leakage model [122]. For coherence, we will keep on using template attacks everywhere. But as the experiments in Section 5.2.4 target a microcontroller with strong Hamming weight leakage dependencies, simpler (non-profiled) attacks would naturally apply as well. By contrast, the following attacks explicitly take advantage of a Bayesian description.

3. Template attack with uniform S_c (UNI-TA). In this case, the adversary follows the Bayesian strategy but does not exploit any information on the permutation (i.e. he assumes a uniform prior on the leakage cycles). Hence, the attacks still have identical efficiencies in the RSI and RP cases.

4. Template attack with direct permutation leakage (DPLEAK-TA). It corresponds to the attack $(2.b)$ described in the previous section. Here, the leakage vector $\mathbf{L}'$ is simply added in the adversary's conditional probabilities. But again, it does not distinguish between the RSI and RP cases.

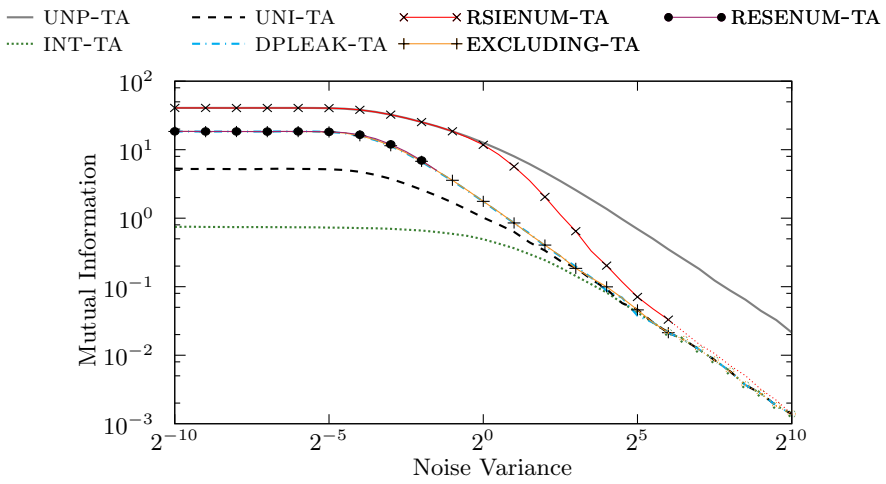

Fig. 5.1.: Mutual information versus noise variance

5. Template attack with permutation leakage enumerating the RSI permutations (RSIENUM-TA). It corresponds to the attack (2.c) in the previous section, where the adversary takes advantage of the 16 permutations that a RSI-based shuffling tolerates to combine its permutation leakages.

6. Template attacks with restricted enumeration (RESENUM-TA). It corresponds to the attack (2.d) described in the previous section. A beam search [213] is performed to enumerate the most likely permutations.

7. Template attacks with excluding heuristic (EXCLUDING-TA). It corresponds to the attack (2.e) in the previous section, where the likelihood of the permutations is weighted by simply excluding duplicates.

The result of a simulated information theoretic analysis for these different attacks is given in Figure 5.1, in function of the noise variance. Several observations can be highlighted. First, and as usual in such worst-case evaluations, the asymptotic trend only appears for large noise levels. In this respect, the main conclusion is that (unlike masking [182]), the slope of the MI curves is the same for both the unprotected and all the shuffled implementations. Intuitively, it suggests that shuffling can (at best) be used to amplify the noise existing in side-channel measurements (i.e. imply a shift of the IT curves). Besides, one can observe that for lower noise levels, significant differences arise between the different scenarios of Table 5.3. For example, it is interesting to note that even without exploiting permutation leakage, the integrated attack is less efficient than the template attack with uniform prior. It confirms that this integrated attack is suboptimal in a profiled case, and is not suited to evaluate the worst-case security of an implementation in low-noise

scenarios. Quite naturally, the distance between integrated and stronger attacks increases as permutation leakage becomes available. In this setting, the amount of information extracted is quite dependent on countermeasure implemented. If the RSI approach is chosen (and this information is exploited computationally), the implementation turns out to be as weak as an unprotected one until noise levels beyond $\sigma^2 = 2^0$. By contrast, in the RP case, the noise amplification happens earlier. In this respect, it is worth to notice the limited difference between the DPLEAK-, EXCLUDING-, and RESENUM-TAs for RP-shuffled implementations, the latter ones only bringing a small advantage. We also observe that as expected, the RESENUM-TA could only be launched until noise levels of approximately $\sigma^2 = 2^{-2}$: beyond this threshold, the large amount of permutations to enumerate with the beam search turned out to be hardly tractable. This last fact confirms the expectation in Section 5.2.1 that the small bias resulting from our efficient permutation generation algorithm should not lead to significantly improved side-channel attacks.

Note that the insecurity of RSI-based shuffling (and, to a lower extent, RP-based shuffling) for low noise levels has to be interpreted with care. What our analysis shows is *not* that the start index or permutation is trivially revealed with a template-based SPA (as the number of permutation candidates in the beam search already explodes when $\sigma^2 = 2^{-2}$). It is really the fact that the 16 leakage samples of the permutation can be exploited jointly that make these countermeasures weak. In other words, what these results show is the importance of computational power in the evaluation of shuffling: summing over 16 cases is easy, summing of 16! ones is harder, as highlighted by the different curves of the RSIENUM-TA and EXCLUDING-/RESENUM-TA information theoretic evaluations.

As a complement of information theoretic analyzes, we performed a security analysis, and computed the success rates of our different attacks, in function of the number of plaintexts measured by the adversary. This allows translating the IT curves of Figure 5.1 into data complexities. For illustration, we selected three different noise variances, corresponding to low (i.e. $\sigma^2 = 2^{-3}$), middle (i.e. $\sigma^2 = 2^0$) an large (i.e. $\sigma^2 = 2^3$) noise levels (where large refers to the fact that the IT curves are merging at this stage). The results of these simulated experiments are given in Figure 5.2 and confirm the previous observations. We again observe the weakness of the RSI-based shuffling in the low noise level case, and the lower efficiency of the integrated attack. The success rate curves also exhibit the slight advantage of the heuristic enumeration when exploiting the leakage of a RP for the smallest noise level, as well as the better behavior of the (computationally cheap) excluding heuristic when the noise increases (again, the RESENUM-TA evaluation could only be performed in the low noise case, i.e. upper figure).

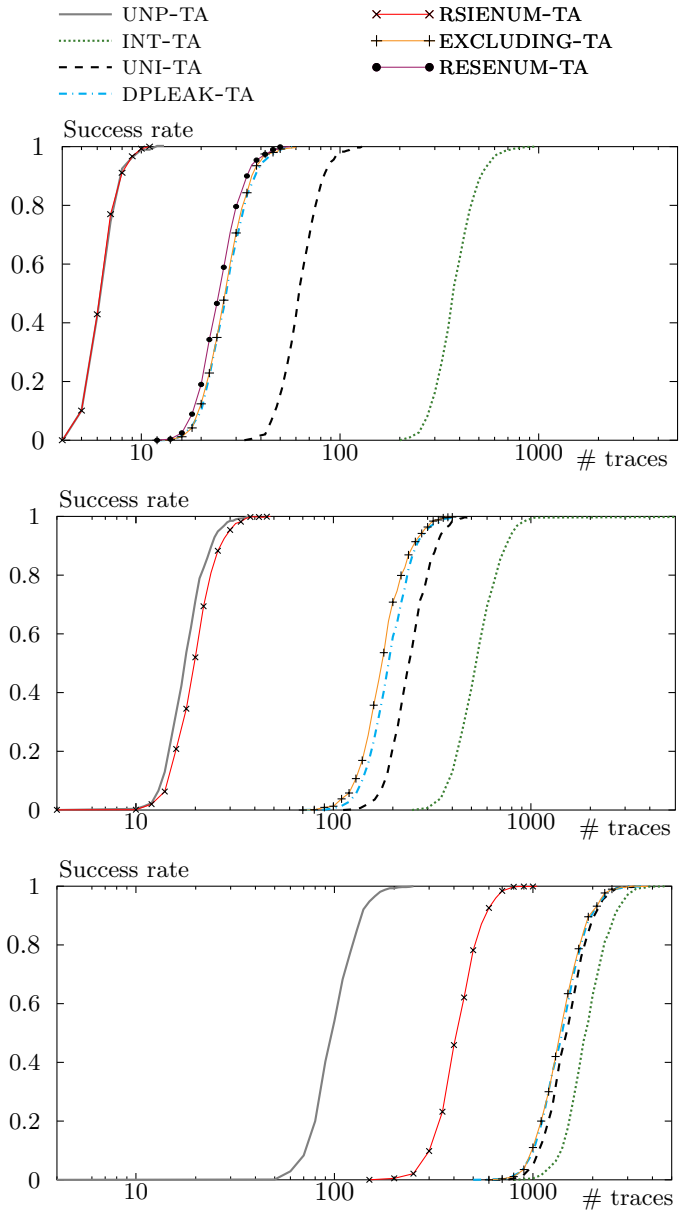

Fig. 5.2.: Success rates of simulated attacks, $\sigma^2 = 2^{-3}$ (top), 2^0 (middle), 2^{+3} (bottom).

5.2.4 Practical experiments

The previous simulated attacks naturally raise the question whether our attacks similarly apply to real world implementations. In order to validate our conclusions, we also performed these attacks against shuffled implementations of the AES, based on the randomized execution path technique of Section 5.2.1.

Our target device is an 8-bit ATMEL microcontroller, and our measurement setup was monitoring the voltage variations of this target device over a small resistor inserted in our supply circuit, with a digital oscilloscope. Based on this setup, we profiled our implementation and built the probability distributions of the vectors $\mathbf{L}$ and $\mathbf{L}'$. That is, we first estimated 16 templates corresponding to the leakages of the permutation indexes c, i.e. $\Pr[L'_c|S_c = s]$. Next, we constructed 16×16 templates for the key leakages at the output of the S-Box, i.e. $\Pr[L_c|K_s = k]$, for each value of c and s. The reason for having the 16×16 sets of key leakage templates is that these leakages behave differently when, at a given point in time, different subkeys are used. That is, we have $\Pr[L_c|K_{s_1}] \neq \Pr[L_c|K_{s_2}]$ if $s_1 \neq s_2$, and $\Pr[L_{c_1}|K_s] \neq \Pr[L_{c_2}|K_s]$ if $c_1 \neq c_2$. This fact is due to the slightly different power consumptions of different registers and memory accesses of the Furious implementation in our target device.

In order to limit the profiling efforts, our templates were kept univariate and constructed with the stochastic approach from [170], using the Hamming weight of the S-Box outputs as base vectors. Interestingly, the fact that different key bytes give rise to different templates leads to indirect leakages on the permutation. That is, we have $\Pr[L_c|K_{s_1}] \neq \Pr[L_c|K_{s_2}]$ for a fixed cycle c. By summing over the 256 key candidates, we can then obtain marginal probabilities $\Pr[L_c = l_c|K_s]$ for all key byte indexes s. This directly leads to useful information of the type:

$$\Pr[S_c = s|L_c = l_c] = \frac{\Pr[L_c = l_c|K_s]}{\sum_{s'} \Pr[L_c = l_c|K_{s'}]}.$$

Furthermore, this information is directly reflected in all the Bayesian attacks, without any modification of the descriptions in Section 5.2.2 (including UNI-TA for which direct permutation leakages are ignored). That is, just the fact that we built 16×16 templates for different s and c values allows to exploit it.

The success rates of our experimental attacks are illustrated in Figure 5.3, where the noise level corresponds to $\sigma^2 = 3.25$. We observe that in this real case study, the RSI-based shuffled implementation remains as easy to attack as an unprotected one, in our worst-case evaluation setting. Besides, we note that the indirect leakage is quite useful for the template attack with uniform prior. One important consequence of this indirect leakage is that the UNI-TA could also apply to our countermeasure with randomized program memory, even if the pre-computation was performed in a perfectly secure (i.e. leakage-free)

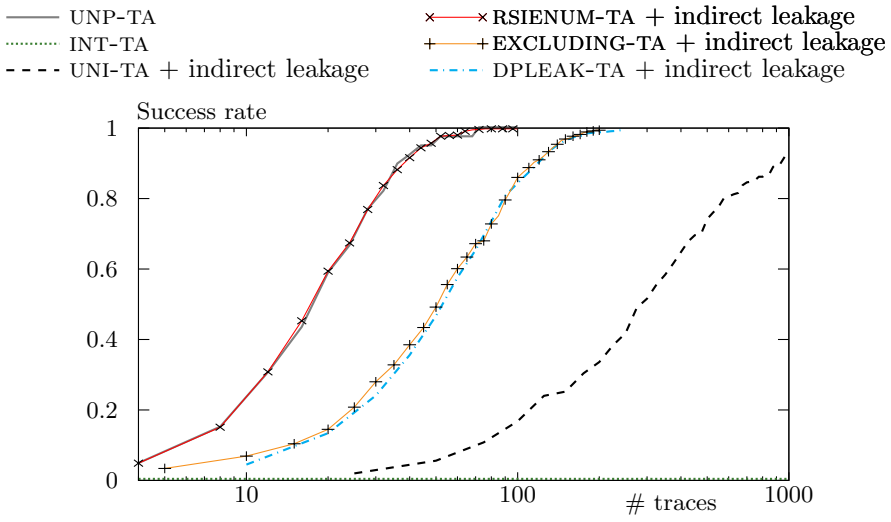

Fig. 5.3.: Success rate of actual attacks on an ATMEL AVR implementation.

environment. Interestingly, we also remark that the integrated attack is less efficient than in our simulated experiments, and is stuck to very low success rate for the data complexities we considered (yet, it eventually succeeded for larger number of measurements). This can be explained by two main reasons. First, the leakages on the permutation extracted with our templates (including the indirect ones) was larger than in our simulations, which naturally increases the gap between the integrating attack and the others. Second, the fact that different ATMEL resources leak according to different models creates an additional noise for the integrating attack, due to a modeling error (i.e. these differences are lost after integration).

5.3 EXPLOITING FRAM MEMORIES TO ENHANCE PHYSICAL SECURITY

Several existing countermeasures could benefit from secure pre-computations. Yet, the exploitation of secure pre-computations is highly related to the problem of fast and efficient non-volatile storage. In this context, a significant drawback of the mainstream flash memories is that write operations are slow and energy-consuming. Furthermore, their number of tolerated write/erase cycles is also limited (from 10k to 100k, typically), which may prevent their frequent use for cryptographic operations. Interestingly, as was presented in Chapter 2 Section 2.2.1, the recently available FRAM provides a solution to

these issues[2]. As a result, we investigate whether it can be used as a technology enabler to improve the performances and security of protected implementations. For this purpose, we first consider the shuffling countermeasure, and its instantiation for the AES algorithm based on randomized program memories proposed in Section 5.3.2. We show that FRAM allows significantly improved performances in terms of pre-computation time. Next, we discuss the application of these new memories to the *randomized look up table* (RLUT) countermeasure [179]. It can be viewed as a type of one-time program specialized to side-channel analysis, or as a generic masking scheme that provides unconditional security against side-channel attacks of all orders (i.e. independent of the statistical moment estimated by the adversary). We provide the first working implementation of this solution applied to reduced versions of the block cipher LED [84], and analyze its performances in various settings. In particular, we investigate the contexts of complete and secure pre-computations, and the tradeoffs corresponding to partial (and partially leaking) ones. We reach performances that are close to higher-order masking in the latter case [163], while complete and secure pre-computations also ensures much higher security levels at practically reachable cost. Therefore, our results suggest that FRAM is a promising solution for improving the security of low-cost tokens such as smart cards, especially when some pre-computations can be performed in a safe environment.

The rest of the section is structured as follows. Section 5.3.1 discusses our security model. Section 5.3.2 contains the implementation results of the shuffling with randomized program memory countermeasure, and their comparison with the previous work from Asiacrypt 2012. Finally, Section 5.3.3 describes the RLUT countermeasure, our proposed implementation and the various tradeoffs it provides. Note that all the developed code was made for a MSP430FR5739 microcontroller, containing 16 kilobytes of FRAM, and was tested using the MSP-EXP430FR5739 experimenter's board and Code Composer Studio 5.3.

5.3.1 Security model

The following sections mainly aim at demonstrating the efficiency of FRAM-based cryptographic computations. Yet, since we consider implementations protected against side-channel attacks, it is important to say a few words about the security model we rely on. Both for the shuffling in Section 5.3.2 and for the RLUT countermeasure in Section 5.3.3, we can consider two alternatives:

[2]Strictly speaking, FRAM is not a new technology as it was introduced as a high-security alternative to Flash memories back in the early 2000s by Fujitsu. However, FRAM-based smart cards did not make it to mass market at that time, due to excessive manufacturing costs and limited ability to reduce cell transistor size.

1. *Secure pre-computations.* That is, the permutations used in shuffling and the randomized program used in RLUT are pre-computed without leakage, prior to the execution of the cryptographic algorithm. As a result, the security of the shuffling is exactly the one analyzed at Asiacrypt 2012 [192]. And the security of the RLUT countermeasure is unconditional: even an adversary accessing the (identity) leakage of all the intermediate computations in the target implementation would not recover any information about its key.

2. *Leaking pre-computations.* That is, the permutations used in shuffling and the randomized program used in RLUT are computed online, and leaking information. In this case, the security of both countermeasures is less investigated, and essentially depends on how much information is leaked during pre-computation (in fact, the same observation holds for most countermeasures exploiting randomness, e.g. masking). Note however that the randomness used to protect these implementations is generated on-chip, and is never output. Hence, adversaries can only mount SPA attacks against it. As a result, we can informally state that our implementations will remain secure in this context, as long as one can guarantee SPA security for this part of the computation (a similar informal separation between SPA and DPA was used to argue about the security of fresh re-keying schemes, e.g. in [127]).

Note that in terms of security, the main advantage of FRAM is to make the first model more realistic. Indeed, one could (at least theoretically) imagine to implement a randomized program with SRAM memories. But in addition to performances that would most likely be poor in this case, such a solution should anyway be implemented online (hence leaking), since SRAM is volatile.

5.3.2 Improving past results: the shuffling case

Shuffling the execution order of independent operations is a possible solution to improve security of cryptographic implementations against side-channel attacks. The goal of shuffling is to distribute the intermediate cipher values over a given period of time, so that an attacker will only be able to observe a chosen intermediate value at a particular moment in time with a certain probability. A typical example of independent operations that can be shuffled is the `SubBytes` layer in the AES. Indeed, whatever the order in which each of the 16 S-Box's outputs is generated, the result of the `SubBytes` layer will not be affected.

A previous work on shuffling, proposed 3 implementations of the AES on an ATMEL ATmega644P microcontroller [192]: a basic one with double indexing, an optimized one with randomized execution path, and a variant with randomized program memory. In this work, we focus on this third proposal, for which FRAM technology provides significant improvements (the two first

ones lead to essentially similar performances, independent of the non-volatile memory used). Randomizing the program memory corresponds to rewriting the code in a randomized way before each algorithm run. In other words, a pre-computation phase modifies (inside the code) which registers and memory addresses will be used during the execution, which then remains essentially the same as an unshuffled one. While promising in principle, such an instantiation of the shuffling idea faces some limitations when implemented in Flash-based ATMEL microcontrollers. First, even when only a few bytes need to be modified, a complete memory page must be erased and rewritten, which takes a lot of time (more or less 4.5 milliseconds each time a page is written or erased). Next, the memory can only be rewritten a limited number of times (namely 10 000). Hence, FRAM-based microcontrollers are natural candidates to relax these limitations as we now detail.

Implementing the AES algorithm with randomized program memory requires three main functions. First, a permutation generator must be defined - we used exactly the same implementation as proposed in [192]. Next, it is necessary to have an AES description with well defined sets of 16 independent operations, on which the shuffling can be applied. Although such operations are easily found for `SubBytes` and `AddRoundKey`, their specification is more difficult for `ShiftRows` and `MixColumns`, for which the 16 bytes are not manipulated independently. This implies that their output cannot be stored at the same location as their input, resulting in the need of 16 additional bytes of temporary storage. Furthermore, the FRAM microcontroller we use has only 12 CPU registers, which is not enough to store a complete AES state. Therefore, each independent operation needs to access the FRAM with absolute addressing, which is more time consuming than working on registers. As for the implementations of Asiacrypt 2012, dummy key-schedule operations have also been added to the "on-the-fly" key-schedule, in order to obtain enough independent operations for this part of the implementation as well [192]. Eventually, the last function needed is the one randomizing the code before execution. This randomization was achieved by modifying the bytes of instructions referring to the cipher state's or round key's memory addresses. Interestingly, since the code and the data are both stored in the same FRAM memory, modifying some bytes of the code or some bytes of cipher state and round key takes exactly the same amount of time.

Our implementation results are available in Table 5.4 (and are given for encryption only). For reference, we first implemented an unshuffled version of the AES in the MSP430FR5739 microcontroller. Even if performance comparisons obtained with different technologies always have to be considered with care, it is worth noticing that it is slightly more time-consuming than ATMEL ones (e.g. the open source AES Furious requires 3546 cycles to execute [150]). This is mainly a consequence of the limited number of registers available in FRAM microcontrollers, leading to more frequent memory accesses. By contrast and as expected, the pre-computation time required to shuffle the code is strongly

Table 5.4.: AES program size (in bytes) and cycle counts in the MPS430FR5739.

		Code Size	Data Size	Cycle Count
Unprotected AES		1076	52	5800
Shuffled AES	Perm. Generation	194	18	2240
	Code Shuffling	418	0	2751
	AES execution	2404	146	8479
	Total	3016	164	13470

reduced, from 18 milliseconds in ATMEL devices to 0.19 milliseconds (running the chip at 16 MHz), which corresponds to a ratio of approximately 100. This is the main advantage of our implementation. Note finally the increased data size and cycle count for executing the AES in its shuffled version, which is essentially due to the previously mentioned execution of dummy key-schedules. We conclude that the overhead required to shuffle the AES algorithm based on a randomized program memory is now in line with practical applications constraints.

5.3.3 Making new results possible: the RLUT case

The previous section described how FRAM memories allow significant speedups for shuffled implementations exploiting randomized program memories. In this section, we show how similar ideas can be used to enable the efficient implementation of the RLUT countermeasure. For this purpose, we first recall the intuition behind this countermeasure, then describe its application to reduced versions of the block cipher LED, and finally discuss implementation results.

Description of the countermeasure

We will focus on the protection of a single S-Box that is the most challenging part of the countermeasure. Intuitively, it is convenient to start from the first-order Boolean masking depicted in Figure 5.4. In this scheme, a random mask m is first added to the sensitive value x which is then sent trough the combination of a bitwise key addition $\oplus$ and S-Box S. A correction function C is used (taking both $x \oplus m \oplus k$ and m as input) in order to produce the output mask q such that $\mathsf{S}(x \oplus m \oplus k) = \mathsf{S}(x \oplus k) \oplus q$. Such an implementation typically gives rise to 4 leakage points denoted as L_1, L_2, L_3 and L_4 on the figure (L_2 being the combination of two parts). It ideally guarantees that statistical moments of order 2 will have to be estimated by an adversary in order to recover secret information. The word "ideally" here refers to the fact that physical defaults such as glitches can lead to exploitable information in lower-order statistical moments [123]. For example, the leakage point L_2 on the figure corresponds to the manipulation of $x \oplus m \oplus k$ leading to $\mathsf{L}_2^a(x \oplus m \oplus k)$, and m leading to $\mathsf{L}_2^b(m)$, in parallel. It implies first-order exploitable information if

these two parts of the leakage function are not independent[3]. Boolean masking can be naturally generalized to d shares ($d = 2$ in the example of Figure 5.4), leading to an (ideal as well) data complexity increase proportional to $(\sigma_n^2)^d$, with σ_n^2 the variance of the noise in the leakage samples, as demonstrated by Chari et al. [40].

From this description, a first step towards the RLUT countermeasure is the observation that if the master key is fixed and for n-bit S-Boxes, one can replace the computation of $\mathsf{S}(x \oplus k)$ by a pre-computed table of size $2^n \times n$ (the correction function can be implemented similarly as a table of size $2^{2n} \times n$). It directly leads to the implementation of Figure 5.5, which is functionally equivalent to the previous one, but where the key addition has been "included" in a key-dependent permutations $\mathsf{P}_k(x)$. From the side-channel security point-of-view, it still corresponds to a first-order secure implementation. Next, and in order to provide unconditional security against side-channel attacks of all orders, the main idea is to replace the Boolean masking operation $x \oplus m$ by an extension to three shares denoted as $\mathsf{G}_i(x, m) = x \oplus m \oplus a_i$, where a_i is a n-bit random mask that is pre-computed in a leakage-free environment. These operations, illustrated in Figure 5.6, can also be implemented as tables of size $2^{2n} \times n$, so that the shares a_i will never be manipulated during the "online" execution of the algorithm. If the G_1 (resp. G_2) function is refreshed before each run of the protected implementation, it guarantees that no information can be extracted from the leakage points $(\mathsf{L}_1, \mathsf{L}_2)$ (resp. $(\mathsf{L}_3, \mathsf{L}_4)$). We additionally need G_1 and G_2 (i.e. their hidden a_i shares) to be independent, in order to avoid fourth-order leakages taking advantage of the correlation between the tables' inputs and outputs. Eventually, it remains to randomize the permutation $\mathsf{P}_k(x)$ (and the correction function C) in order to completely hide the key, even from identity leakage functions, as represented in Figure 5.7. This way, a non-linear S-Box can be computed securely. This leads to an implementation in which the operations G_1, G_2, R and RC are pre-computed according to Algorithm 1, and executed according to Algorithm 2. Extending this S-Box computation to a complete cipher is straightforward: we just need independent tables for all the S-Boxes. As for the linear operations, they have to be applied independently on the two shares that are explicitly manipulated by the leaking device, just as in standard Boolean masking.

Application to reduced LED

The previous section suggests that the RLUT countermeasure has high memory requirements, that strongly depend on the S-Box size used in the block cipher to protect. In particular, given a N_r-round cipher with N_s S-Boxes per round, the implementation of the RLUT countermeasure essentially requires storing:

- A *table map* that corresponds to all the a_i shares generated during pre-computation, with memory cost estimated as $(N_s \cdot N_r) \cdot 2 + N_s$ n-bit

[3] As a typical example, $\mathsf{L}_2 = \mathsf{L}_2^a + \mathsf{L}_2^b$ would correspond to an ideal implementation, while $\mathsf{L}_2 = \mathsf{L}_2^a \cdot \mathsf{L}_2^b$ would leak first-order information, as discussed in [182].

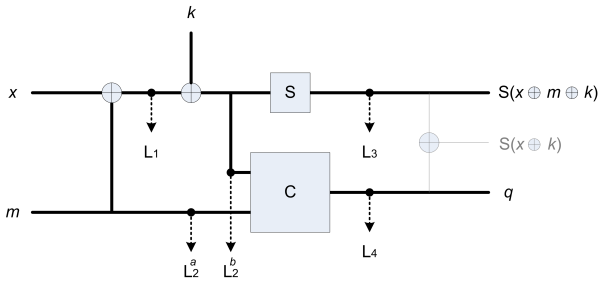

Fig. 5.4.: Boolean masking.

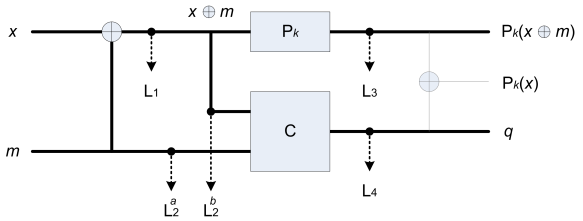

Fig. 5.5.: Boolean masking with LUTs.

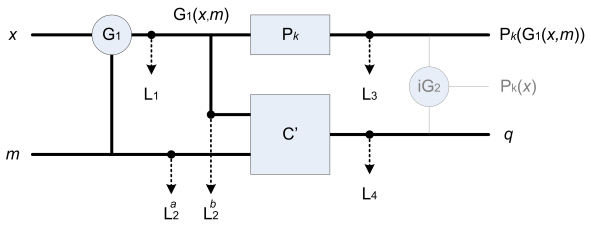

Fig. 5.6.: Randomized Boolean masking with LUTs.

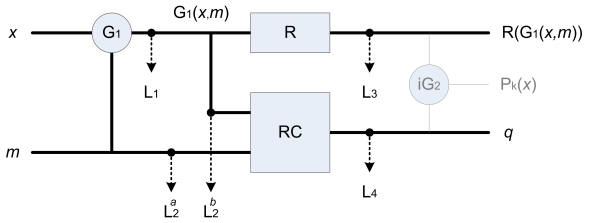

Fig. 5.7.: Randomized Boolean masking with randomized LUTs.

Algorithm 1 - Table refreshing.

- **input:** P_k.
1. Pick $a_1 \xleftarrow{R} \{0,1\}^n$;
2. Pick $a_2 \xleftarrow{R} \{0,1\}^n$
3. Pick $a_3 \xleftarrow{R} \{0,1\}^n$;
4. Pre-compute $\mathsf{G}_1(I,J) = I \oplus J \oplus a_1$;
5. Pre-compute $\mathsf{R}(I) = \mathsf{P}_k(I) \oplus a_2$;
6. Pre-compute $\mathsf{G}_2(I,J) = I \oplus J \oplus a_3$;
7. Pre-compute $\mathsf{RC}(I,J) = \mathsf{r}(I) \oplus \mathsf{p}_k(I \oplus J \oplus a_1) \oplus a_3$;
- **output:** $\mathsf{G}_1, \mathsf{R}, \mathsf{G}_2, \mathsf{RC}$.

Algorithm 2 - S-Box evaluation on input x.

- **input:** $\mathsf{G}_1, \mathsf{R}, \mathsf{RC}$.
1. Pick $m \xleftarrow{R} \{0,1\}^n$;
2. Compute $\mathsf{G}_1(x,m)$;
3. Compute $\mathsf{R}(\mathsf{G}_1(x,m))$;
4. Compute $\mathsf{RC}(\mathsf{G}_1(x,m),m)$;
- **output:** $\mathsf{R}(\mathsf{G}_1(x,m))$, $\mathsf{RC}(\mathsf{G}_1(x,m),m)$.

words (where the factor 2 corresponds to the fact that excepted for the first round, the share a_1 in Algorithm 1 is always provided by the previous round).

- A *randomized program* that corresponds to the tables R and RC, with memory cost estimated as $N_r \cdot N_s$ tables of size $2^n \times n$ and $2^{2n} \times n$, respectively.

Note that operations G_i are never explicitly used during the cipher execution, but for the first round to mask, and last round to unmask after a secure computation is completed. Following these estimations, and as discussed in [179], it is natural to consider a cipher with 4-bit S-Boxes for this purpose. In the following, we will consider reduced (16-bit) versions of the LED cipher illustrated in Figure 5.8.

While such a cipher is naturally too small for being deployed in actual applications, we use it to refine our model for RLUT performance estimates. As will be discussed in the next sections, scaling to larger number of rounds and block size (e.g. the full 64-bit LED cipher) will be possible in soon available 128- and 256-kilobyte versions of our FRAM microcontroller. In the figure, the 4-bit S-Boxes of LED are denoted as S, and its linear diffusion layer as MixColumns.

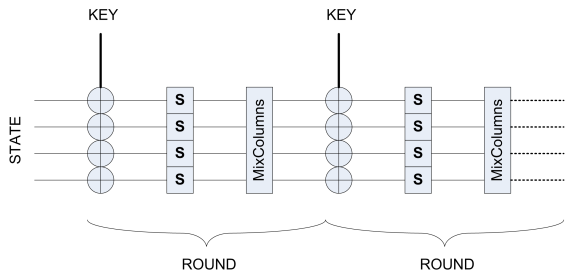

Fig. 5.8.: Reduced version of the block cipher LED.

Implementation in FRAM microcontrollers

We now describe how to implement reduced (with up to 4 rounds) LED ciphers within the 16 kilobytes of FRAM available in our MSP430FR microcontroller.

The first building block required in a RLUT-masked implementation is a randomness generator (needed to produce the a_i values of Algorithm 1). For illustration, we used a LFSR with CRC-32 polynomial for this purpose (alternative ways of generating randomness could of be considered, e.g. using a leakage-resilient PRG if leaking pre-computations are considered [61, 177]).

Next, the part computing the randomized program can be implemented quite straightforwardly, following the description in Section 5.3.3. The trickiest bit was to efficiently arrange 4-bit outputs into memory bytes, without giving any unnecessary information on the RLUT input values[4]. Using one byte to store two consecutive RLUT outputs was rejected, since accessing one or the other value in the byte would have led to different code behaviors, depending on the *least significant bit* (LSB) bit of the RLUT's input. Instead, we stored the outputs coming from two different RLUTs for the same input value in a single byte. This time, the LSB (resp. MSB) part of one byte will be accessed when an odd (resp. even) word of the state needs to be computed, giving no information on the word's value itself. Based on this strategy, the RLUTs R and RC can be generated efficiently from the cipher key, the S-Box and the table map, that are all stored in memory.

Eventually, the last piece of code concerns the execution of the block cipher itself. Again, the fact that the operations are performed on 4-bit words had to be taken into account while accessing the variables or tables stored in memory. One round of the reduced algorithm is executed by first reading the R and RC tables' outputs, corresponding to the cipher state and mask intermediate values. Then, the `MixColumn` layer is executed on each of the shares.

[4]This has no impact on the security in case of secure pre-computation, but may increase the information leakage in case of online randomization of the tables.

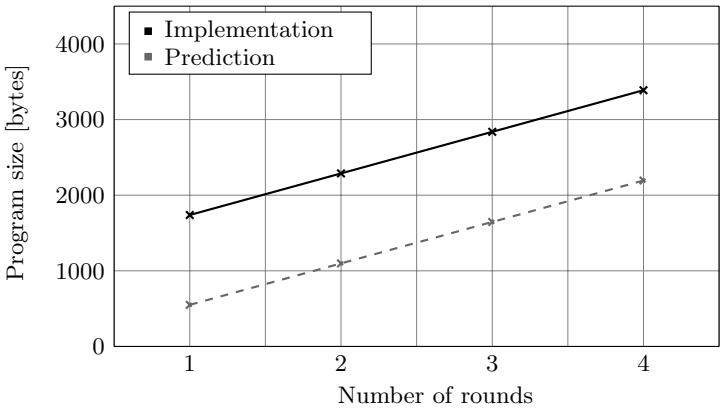

Fig. 5.9.: Program size of the LED cipher protected with RLUTs.

It is implemented using an `Xtime` table, as suggested in the specifications of LED [84].

Results and discussion

As described in Section 5.3.3, an estimation of the memory size required to store the table map and randomized program of the RLUT countermeasure can be derived from the number of rounds N_r, the number of S-Boxes per round N_s and the S-Box bit-size n. This estimation is illustrated by the dashed line of figure 5.9, in the case where $N_s = 4$, $n = 4$ and N_r varies from 1 to 4. A plain line representing the actual results we obtained for our implementation is also plotted. The two curves follow the same trend, with the offset separating them corresponding to the code size needed to implement the cipher itself (i.e. excluding the tables for which the memory requirements are growing with N_r - for detailed results refer to the full paper [106]). Interestingly, these results suggest that for any parameters N_r, N_s and n, the memory requirements needed to implement a block cipher protected with the RLUT countermeasure can be quite accurately predicted. For example, such an implementation for a full (64-bit) version of the block cipher LED (corresponding to $N_r = 32$, $N_s = 16$ and $n = 4$) would roughly require a memory size of 70 kilobytes, and could therefore be implemented in the soon available 128-kilobyte FRAM microcontrollers.

The question of accurate predictions can also be asked for pre-computation time: estimates for this metric were similarly provided in [179]. Namely, the time needed to generate the RLUTs can be approximated with $((N_s.N_r).2 + N_s) + (N_s.N_r).2^n + (N_s.N_r).2^{2n}$ "elementary operations" (where the first term corresponds to randomness generation, and the later ones correspond to the refreshing of the R and RC tables). Our actual implementation results directly

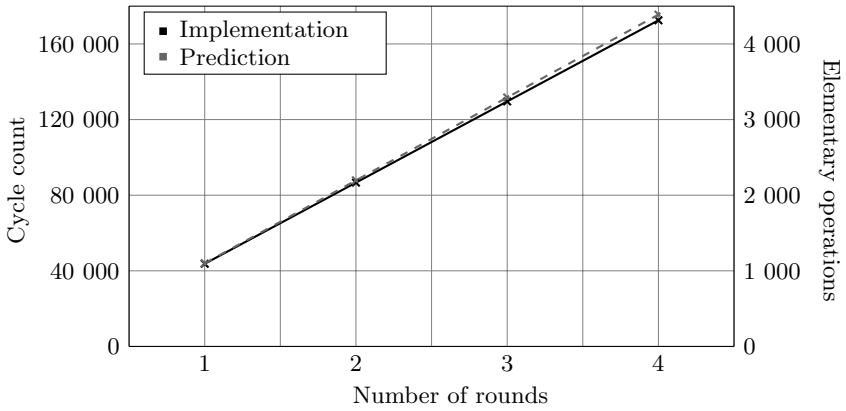

Fig. 5.10.: Pre-computation time of the LED cipher protected with RLUTs.

allow translating these "elementary operations" into a concrete number of clock cycles. The results in Figure 5.10 (the full results also reported in [106]) again confirm a nice correlation with predictions. Namely, the main difference between both curves is a factor 40, which presumably corresponds to the number of cycles needed to perform each elementary operations. Extrapolating these results to the full (64-bit) version of the LED block cipher suggests pre-computation time complexities around 140 000 elementary operations, corresponding to 5 600 000 cycles (i.e. an execution time of 35 milliseconds at 16MHz), which would be acceptable for some applications (and is likely to be improved with technology scaling).

Eventually, it is worth noticing that time and memory complexities could be reduced by exploiting some performance vs. security tradeoffs. A first solution would be an implementation for which only some rounds are masked with RLUTs, while the others use standard masking schemes. For example, one could protect only the first and last 4 rounds of (full, 64-bit) LED, i.e. 8 out of 32, resulting in an approximate reduction of the time and memory complexities down to 25% of their original values. However, this solution may be risky in front of advanced attacks such as algebraic ones, that can exploit the leakage of the middle rounds [159]. Another approach to reduce the pre-computation time consists in refreshing only a fraction of the table map (and randomized program) after each cipher execution, and to perform this refreshing randomly. Interestingly, such a tradeoff would also take advantage of the non-volatile capacities of FRAM memories, since the complete randomized program could be pre-computed offline, while only a part of it would be modified online. It is flexible since the fraction of RLUTs modified per cipher executions could be adapted to the application requirements. As an illustration, modifying 10% of the RLUTs in (full, 64-bit) LED would reduce the pre-computation time to approximately 560 000 cycles, which is getting close

to the performances of a third-order masked AES (470 000 cycles in [163]). Combining the two appraoches would of course be possible as well, e.g. by randomizing the first- and last-round tables in priority.

5.4 CONCLUSION

In this chapter, we investigated several ways of implementing side-channel countermeasures inside small (e.g. 8-bit) microcontrollers.

In a first part, we proposed two new implementations of the shuffling countermeasure. They respectively allow improved performances in terms of overall cycle count and online cycle count. From those implementations, we provided the first comprehensive evaluation of the shuffling countermeasure, including worst-case Bayesian attacks. For this purpose, we described intuitive formulas capturing the different variants of shuffling, and integrated them in a general evaluation framework from Eurocrypt 2009. These evaluation tools allowed us to show that previously used integrated attacks may not be enough to assess the security of a shuffled implementations. We put forward that simplifying the permutation generation (e.g. by using RSI rather than RP) can lead to a complete breakdown of the countermeasure if not too noisy measurements are available (which turned out to be verified in a practical case study). We also explained the computational origin of these weaknesses (i.e. their relation with the total amount of permutations that are considered in the countermeasure). We then exhibited that indirect leakages may be available in shuffled implementations, due to the different leakage models of different resources. This suggests an interesting scope of further research. Namely, since our results suggest that randomizing the order of instructions in cryptographic implementations is not always sufficient, can we design efficient ways to randomize both the execution order and the physical resources used in a cryptographic implementation?

In a second part, we put forward that FRAM is a promising technology in the context of side-channel resistant cryptographic hardware, since it enables the efficient implementation of various countermeasures taking advantage of pre-computations. The case of RLUTs is particularly relevant to illustrate this observation. Indeed, they have never been implemented so far, they will soon be applicable to complete block ciphers, and may lead to high security levels for small embedded devices, independent of hardware assumptions that may be hard to fulfill. Important scopes for further investigations include the evaluation of the security levels obtained in the context of partially leaking pre-computations. In particular, analyzing the online refreshing of partially randomized programs mentioned in Section 5.3.3 would be very useful. Besides, the design of block ciphers that are well suited to implementations with RLUTs (e.g. with light(er) non-linear layers and strong(er) linear ones) is another interesting research avenue.

PART III

INTELLECTUAL PROPERTY PROTECTION

CHAPTER 6

IP PROTECTION FOR INTEGRATED SYSTEMS USING SOFT PHYSICAL HASH FUNCTIONS

6.1 INTRODUCTION

While technology shrinking has been the enabling factor for producing increasingly powerful micro- and nano-electronic devices, it has also made the manufacturing process of these devices an increasingly difficult and expensive task. Besides, the high complexity of present hardware/software co-designs leads most system developers to assemble various pieces of hardware and software produced by different companies. This has motivated the development of new businesses, relying on the selling of *intellectual property* (IP) cores. A recent study from Semico Research estimated that the IP core market reached US$ 4 billion in 2009, i.e. a growth of 23.2% compared to the 2005 figures [161]. The advantages of re-using IP cores are enormous in terms of cost and development time reductions. But they also raise important risks regarding IP theft. Namely, the high-valued nature of hardware/software systems naturally creates a strong incentive for piracy, cloning of products and copyright infringement. In addition, the selling of IP cores implementing certain functionalities, that have to be integrated in larger developments, raise compatibility constraints that make the situation even more challenging. This is in part due to the versatility of these IPs that can be distributed under different shapes. For example, IPs can be found in high-level format (i.e. described with a high-level language) or in low-level format (i.e. as completely laid out functional blocks restricted to a given technology). As a result, various solutions have been proposed to decrease the risk of unlicensed IP usage.

Looking at low-level IPs, two main general approaches are usually suggested in the literature. A first line of research can be denoted as "permission-based". The idea is that before running, any functional system performs some test in order to make sure that it has the right permissions. The use of passwords is a typical example of such an approach. More robust ideas have also been introduced in two main directions. On the one hand, the test could check the

presence of a cryptographically-enhanced security chip (rather than a simple password), as typically suggested in [15, 117]. On the other hand, one can also take advantage of *physically unclonable functions* (PUFs) in order to make sure that the system is running on the correct (unique) piece of hardware, e.g. as suggested in [81, 174].

A second active line of research is to exploit watermarking techniques. In general, a watermark is a piece of information that is embedded in a digital signal in order to verify its authenticity or the identity of its owners. Its main evaluation criteria are *robustness* (i.e. the watermark should be detectable even if the digital signal is slightly modified by some processing, e.g. filtering), *imperceptibility* (i.e. the original signal and the masked signal should be perceptually close) and *capacity* (i.e. the embedded mark should be large). By contrast to the permission-based approach, watermarking is useful a posteriori in order to detect illegal copies at lower cost than reverse engineering. While first investigated in the context of image processing, watermarking has also been applied for software IP protection [44, 138]. More recently, it has drawn attention for hardware IP protection, e.g. [4, 97, 98].

Two types of limitations can be highlighted for these approaches. The first one mainly applies to the performance and security of permission-based protections. That is, in order to include such mechanisms in a circuit, one needs to modify its original layout. This means that some logic gates are lost (and consume power) for this purpose. Besides, as permission checks imply the addition of some circuitry in the chips, an adversary who has access to the source code (by reverse engineering or any other means) can remove this part of the designs, hence producing illegal copies without any protection. The second limitation is more general and relates to the flexibility of the solutions. Namely, for both types of IP protections, the decision to protect an implementation has to be part of the design process (and must sometimes be made early in this process). Hence, it cannot help for already deployed products.

In parallel, security enhancements that are specific to high-level IP and take advantage of software facilities have also been proposed. One typical example is source code or bitstream encryption that is used, e.g. in FPGA [58]. While this solution is useless in the case of hardwired IP, as there is no code to encrypt/decrypt, it can be a useful complement in some IP protection systems where a part of the value to preserve is software. Note that the encryption/decryption of the source code or bitstream then has to be secure against other types of physical attacks, e.g. using side-channels [135]. It is also necessary that the encryption/decryption keys are stored in a sufficiently secure memory (a problem for which the tamper resistant property of PUF could help [189]).

In this chapter, we propose a new IP protection scheme that applies both to high-level and low-level IP. It mitigates some of the limitations, and can be efficiently combined with, previously introduced solutions for this purpose. More precisely, we take advantage of two main ingredients. First, we build on the idea of soft (aka robust, aka perceptual) hash functions, that

are yet another tool used to protect the copyright of digital data (e.g. images) [71, 115, 133, 191]. Second, we exploit the physical representation of the IP, e.g. through its power consumption, as proposed with side-channel watermarks [21, 214]. In addition, we use the formalism introduced in the context of PUFs [8], in order to specify the components of our IP detection infrastructure. Combining these elements, we introduce soft physical hash functions as a powerful yet flexible way to deal with the IP of microelectronic circuits and systems. We then define the main properties of such functions, namely their *content sensitivity* and *perceptual robustness*. Next, we instantiate a first soft physical hash function with simple signal processing tools and analyze its properties based on a software experimental case study. Namely, and based on the software implementations of 10 different block ciphers that were described in Chapter 3, Section 3.2, we show experimentally that a soft physical hash function can efficiently discriminate these different implementations, while being robust to some elementary code transformations that should not modify the IP (perceptually). We finaly extend the evaluation of soft physical hash functions to the meaningful case of FPGA implementations. For this purpose, we considered the five lightweight block ciphers (namely HIGHT, ICEBERG, KATAN, NOEKEON and PRESENT) together with the standard AES, of which the ASIC designs were presented in Chapter 3, Section 3.3. This case study was mainly motivated by the goal of analyzing a challenging set of potentially similar IP cores. We additionally considered different realistic scenarios such as stand-alone designs, re-synthetized designs and designs with parasitic IPs running in parallel.

Experimental results suggest that the soft physical hash functions approach for IP detection remains a promising and flexible solution for software and hardware implementations. Depending on the cases, we successfully detected the IPs without any knowledge of their inputs, or taking advantage of the data dependencies in the power traces generated by the running algorithms, with limited number of measurements.

6.2 TYPES OF IPS AND POSSIBLE ATTACK SCENARIOS

IP cores may be provided in different forms which all have their specificities and can be found at different stages in the hardware or software design flows. In this sections, we will discuss where the IP lies in such systems. We will then describe two general attacks against IP in integrated systems.

Where does the IP lie? From the previous design flows discussed in Chapter 2, Sections 2.1.3 and 2.2.2, it appears that different types of IP are manipulated by the functional systems that are used in final applications. On the one hand, the source codes contain the high-level IP. They are easy to manipulate and provide the most readable description of algorithmic tricks used to obtain efficient implementations. In addition, they are extremely *malleable*, as it is easy to slightly modify them, or to run again the synthesis and place

and route (resp. conversion) steps, in order to produce two computing devices (resp. systems) with the same functionality and slightly different layouts (resp. machine codes). On the other hand, the layout and machine codes contain the low-level IP that is less malleable but possibly includes further optimization efforts performed during the synthesis and place and route (resp. conversion) steps. The netlist in a hardware design flow falls between these two extremes.

Attacks against the IP. Leaving technical details aside, two main types of attacks can be mounted to violate IP. First, "recovery and clone" attacks imply that the adversary just re-uses the stolen IP as such. They typically target the layouts or machine codes. Second, "recovery and modify" attacks imply that the adversary modifies the IP before re-using it in an illegal system. They typically target source codes. Clearly, the second attacks are much more powerful and, in general, harder to prevent. As mentioned in the introduction, permission-based IP protections are useless against such attacks, as the adversary can remove them in his modified design. But in cases where a watermark is somewhat "additive" (e.g. [21, 214]), the risk that an adversary identifies the watermark insertion and removes it also increases. In practice, there exist many ways in which attacks against the IP can be implemented. Reverse engineering generally comes in the first place for ASIC. A rather complete survey is given in [187]. One can also mention approaches taking advantage of side-channel analysis [55, 65, 82, 158]. But threats caused by dishonest manufacturers, producing more chips than licenced, or industrial IP theft, are other examples. There finally exist many attacks that are specific to one technology. For example, bitstream security is a serious issue for SRAM-based FPGA [204]. In order to remain generic, most of our discussions in the following section are independent of the technical solutions used to perform IP theft.

6.3 A NEW APPROACH BASED ON SOFT HASH FUNCTIONS

Looking at previous works, the protection of IP in integrated circuits and systems appears to easily take advantage of general solutions for the protection of digital data. As a result, and besides the watermarking and permission-based techniques, it is natural to investigate other tools that have been proposed to prevent the piracy of digital images. Following, we propose to exploit robust hash functions [71, 191], also known as soft hash functions [115] or perceptual hash functions [133], for this purpose. We will use the term "soft hash functions" in the rest of this work. By contrast to cryptographic hash functions, for which the output string is highly sensitive to small perturbations of the input, soft hash functions are such that similar objects should return highly correlated digests (i.e. be *perceptually robust*), while different objects should produce uncorrelated ones (i.e. be *content-sensitive*). Soft hash functions exist in keyed and un-keyed version. We will only consider the second category and leave the design of a keyed version as a scope for further research.

In practice, the soft hash functions we will consider are generally made of two main ingredients. First, a *feature vector evaluation* phase outputs an intermediate response that is expected to represent the object to characterize in the most accurate manner. In other words, this intermediate string should be very content-sensitive. Second, an *extraction phase* should apply signal processing and summarize the feature vector into a smaller output hash value that best trades content sensitivity for perceptual robustness. In addition, soft hash function infrastructures use a *detection tool* in order to compare different hashes and determine whether they correspond to the same object. The main idea that we propose here is to extract a physical feature from the objects to protect. We denote the resulting tool as a *soft physical hash function* (SPH) that will be the main component of our *IP detection infrastructure*. Interestingly, SPH can be viewed as a particular type of physical function systems, as formally defined in [8]. But whereas this previous work focuses on the robustness, unclonability and unpredictability of physical functions, we rather care about the previously mentioned perceptual robustness and content sensitivity. Relations with the formal definitions from [8] will be given in the next section. Intuitively, an IP detection infrastructure can be specified as follows:

1. *Object to protect.* This could be any type of IP, namely a source code, a netlist, a layout, or a machine code, as defined in Section 6.2.

2. *Physical feature vector evaluation.* This could be any physical emanation of the device running the IP. For example, side-channels such as the power consumption [110] or the electromagnetic radiation [74] are natural candidates.

3. *Extraction.* This could be any signal processing outputting a hash value that best represents the IP. For example, one could compress by selecting the "points of interest" in a side-channel measurement trace.

4. *Detection.* This could be any statistical tool that allows determining the level of similarity between two hash values. Again, techiques from the side-channel attack community could help, e.g. correlation [37] or template attacks [41].

In addition, one generally has to consider the context in which the IP protection mechanism will be applied. Namely, we could consider:

1. *Known or unknown inputs.* That is, can the IP claimer select the data that is manipulated by the target device during the physical feature evaluation (in which case the soft hash can characterize both operation and data dependencies)? Or is he limited to executions on random, and possibly unknown, inputs (in which case the soft hash only characterizes operation dependencies)?

2. *Known or unknown source code.* That is, does the IP claimer know the full source code he is trying to detect or does he only have access to its soft physical hash value (e.g. if the IP claimer is a third party)?

3. *Same or different technologies.* That is, are the soft hash values to be compared extracted from devices in the same technology or not?

4. *Same or different setups.* That is, are the soft hash values to be compared extracted with the same measurement apparatus or not?

Obviously, certain scenarios are more challenging than others and the perceptual robustness and content sensitivity of a SPH can significantly vary accordingly. Before entering into more details and instantiation issues, we would like to point out a few advantages of the proposed approach. In the first place, and contrary to the permission-based and watermarking approaches, using SPH does not require to modify the designs a priori. This allows improved flexibility, as one can decide even after implementation to exploit this idea for IP theft detection. It also means that implementing the protection mechanism implies no performance overhead at all. Second, SPH theoretically allow preventing some recovery and modify attacks. This is due to the fact that there is nothing to remove from a protected design, but the IP itself (i.e. the protection is fully intrinsic). Of course, the detection phase will be significantly more challenging when modifications are substantial (because content sensitivity is inherently limited once modified designs become too different), but it can at least tolerate some malleability in the objects to protect. Third, the proposed solution remains cheaper than reverse engineering, as it only requires to measure a physical feature.

6.4 DEFINITIONS

6.4.1 The IP detection infrastructure

The main goal of this section is to provide some formalism to define and study the aforementioned IP detection procedure, based on the idea of SPH. This procedure naturally fits in the generic framework for physical functions presented in [8], that is depicted in Figure 6.1. Indeed, SPH perfectly correspond to the definition of *physical function systems*. Hence, we reuse and lighten the formalism provided in [8] and apply it to our IP detection using SPH. The generic framework we propose for IP detection is illustrated in Figure 6.2, where a *soft physical hash function* is a *probabilistic procedure* $\text{SPH}(\text{IP}, \text{X})$ that returns a *hash value* h.

$$\text{SPH} : (\text{IP}, \text{X}) \mapsto h.$$

Such outputs are obtained in two steps. First, a *physical feature vector* is obtained (with the $\text{Eval}(\cdot)$ procedure), corresponding to the measurement

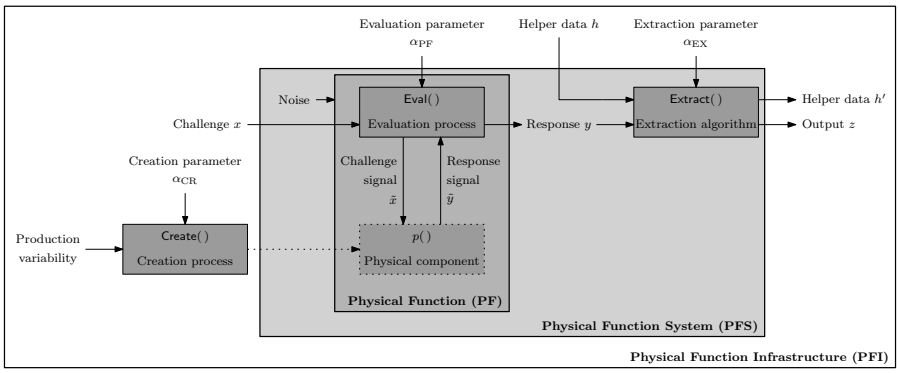

Fig. 6.1.: Generic framework for physical functions [8].

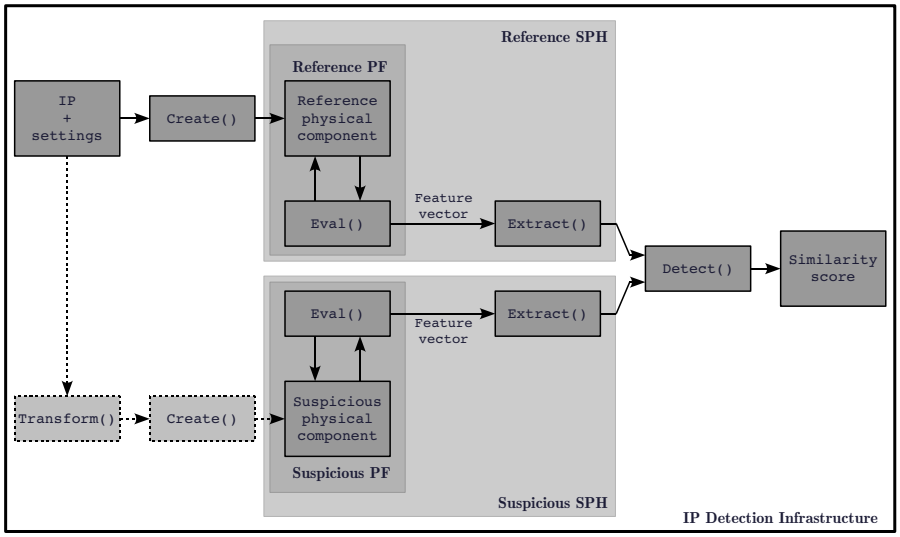

Fig. 6.2.: Generic framework for IP detection.

of some physical quantity observed during the processing of data X (that may contain more than one input) by a device running the given IP. The probabilistic nature of a SPH comes from this procedure and is due to many different reasons, such as the variability that can be observed between different devices running the same IP, or the noise in the measurements. This part of the system is the one bringing the *content sensitivity* property but, as just stated, there is a non-negligible variability that should be removed for reaching *perceptual robustness*. This is precisely the role of the Extract$(\cdot)$ procedure. During this second step, the physical feature vector is processed to extract the

robust information it contains, returning a hash value that should characterize the IP running on the device.

Next, an *IP detection infrastructure* is composed of two *soft physical hash functions* $\text{SPH}(\text{IP}_{\text{ref}}, \cdot)$ and $\text{SPH}(\text{IP}_{\text{sus}}, \cdot)$, corresponding to the reference and the suspicious devices. Each SPH outputs a hash value that should identify the IP running on the device under test. Additionally to these two SPH, a detection tool $\text{Detect}(\cdot, \cdot)$ is used to compare the hash values h_{ref} and h_{sus} returned by the SPH. This procedure outputs a *similarity score* that should be close to 1 for similar IP, and close to 0 for different ones:

$$\text{Detect} : (h_{\text{ref}}, h_{\text{sus}}) \mapsto s \in [0, 1].$$

Performance metrics are required to evaluate SPH. In the following section, we define *perceptual robustness* and *content sensitivity*, so that we will be able to quantify these properties and to determine relevant tools for extracting hash values.

6.4.2 Performance metrics

The IP detection infrastructure aims at confirming/invalidating the presence of an IP running on a suspicious device. As mentioned earlier in Section 6.2, this IP can be of different nature, such as a layout or source code for instance. The main difficulty for formally defining our problem comes from the fact that some modifications of the source code should not be considered as defining a new IP. For example, trivial modifications (e.g. changing variable names, switching registers, adding dummy operations, changing compiler options) can clearly be regarded as preserving the IP, but other modifications such as code optimizations can also be considered as improving, yet relying on, the initial IP. Defining the boundary between IP reuse and new IP definition is outside the scope of this work[1]. Here we assume the existence of such a definition and concentrate on providing a tool allowing us to decide whether a given device is based on a given IP or not, in the sense of this definition. More formally, we consider a set $\mathcal{F}_{pre}$ of *IP-preserving transformations*. The detection tool should consider as "close" an IP and its transform if the transformation function belongs to $\mathcal{F}_{pre}$, and as "distant" if the transformation belongs to $\bar{\mathcal{F}}_{pre}$.

Remark. The use of sets $\mathcal{F}_{pre}$ and $\bar{\mathcal{F}}_{pre}$ is a purely formal way of dealing with the perceptual notion of closeness involved in this problem. These sets should be complementary but obviously, in practice, we will only consider some transformations in both sets. Notice that $\bar{\mathcal{F}}_{pre}$ may contain modifications of an IP, but also transformations that simply consist in changing the IP. For instance, let IP_1 be the IP corresponding to the source code of a social network and IP_2 be the one corresponding to a plane autopilot. Then, formally, there

[1] In Section 6.6 we will run experiments on typical examples, where, on the one hand, IPs are modified and reused, and, on the other hand, different IPs are considered.

exists $\bar{F} \in \bar{\mathcal{F}}_{pre}$ such that $\text{IP}_1 = \bar{F}(\text{IP}_2)$. Nevertheless, and as will be clear in the next section, such a definition is useful to capture practical transforms that can be considered as IP-preserving, while providing the flexibility to evaluate a wide range of scenarios.

As a result, the robustness and sensitivity of a SPH are related to the set of transformations considered to be IP-preserving, to the detection tool that is used, to the threshold τ that has been fixed to make the decision, and to the data complexity N corresponding to the number of IP runs that will be measured. We denote by $\mathcal{X}_N$ the set of N-input requests that can be sent to the devices and define our evaluation metrics as follows.

Definition 1. *(τ-perceptual robustness): Let $\tau \in [0,1]$, the τ-perceptual robustness of a SPH relatively to a detection tool Det, a set of transformations $\mathcal{F}_{\text{pre}}$ and a data complexity N, is the probability that the similarity score of a reference IP and its transformation by $F \in \mathcal{F}_{\text{pre}}$ is larger than τ:*

$$\text{PeRo}_\tau^N(\text{Det}, \mathcal{F}_{\text{pre}}) \triangleq \Pr_{\text{IP}, x \in \mathcal{X}_N, F \in \mathcal{F}_{\text{pre}}}[\ \text{Det}(\text{SPH}(\text{IP},\text{X}),\ \text{SPH}(F(\text{IP}),\text{X})) \geq \tau].$$

Definition 2. *(τ-content sensitivity): Let $\tau \in [0,1]$, the τ-content sensitivity of a SPH relatively to a detection tool Det, a set of transformations $\mathcal{F}_{\text{pre}}$ and a data complexity N, is the probability that the similarity score of a reference IP and its transformation by $\bar{F} \in \bar{\mathcal{F}}_{\text{pre}}$ is smaller than τ:*

$$\text{CoSe}_\tau^N(\text{Det}, \bar{\mathcal{F}}_{\text{pre}}) \triangleq \Pr_{\text{IP}, \text{X} \in \mathcal{X}_N, \bar{F} \in \bar{\mathcal{F}}_{\text{pre}}}[\ \text{Det}(\text{SPH}(\text{IP},\text{X}),\ \text{SPH}(\bar{F}(\text{IP}),\text{X})) \leq \tau].$$

An IP detection infrastructure aims at combining an SPH to a relevant detection tool in order to ensure good performances regarding both perceptual robustness and content sensitivity. This context is the one of binary hypothesis testing where τ corresponds to a threshold, $1 - \text{PeRo}$ is non-detection error probability and $1 - \text{CoSe}$ is the false-alarm error probability.

6.5 SOFTWARE CASE STUDIES

This section demonstrates the use of SPH based on power consumption traces to perform IP detection. First, we propose a simple instance of IP detection infrastructure, that uses classical tools in signal processing and side-channel attacks. Then, we provide experimental results that confirm the content sensitivity and perceptual robustness of this instance. These results are obtained for a particular set of IP and some representative code transformations that we considered as IP-preserving.

6.5.1 An exemplary instance

We first present our instance by specifying the four elements that define an IP detection infrastructure enumerated in Section 6.3. Then, we detail the context we consider for detecting IPs. Finally, we provide information about our measurement setup.

Object to protect. As a first case study, we investigated the relevance of our IP detection based on SPH in the context of software implementations. In particular, we took advantage of the cryptographic algorithms discussed in Chapter 3 and made available as open source codes in [62]. We considered the assembly codes of these lightweight ciphers as IPs to protect, in order to analyze situations ranging from very similar algorithms to totally different ones.

Physical feature evaluation. Inspired by side-channel attacks, we characterize an IP by measuring the power consumption of a device running this IP. As the literature on power analysis attacks suggests, such a physical feature vector is expected to be highly content sensitive.

Extraction. We somehow need to compress the feature vector, removing part of the variability due to the "noise"[2], while keeping the content sensitive information. The Fourier transform is a natural candidate for such compressing and filtering purposes. The technique we propose is to apply a *fast fourier transform* (FFT) to the traces obtained, to take its modulus, and to only extract the values corresponding to frequencies below the clock frequency. Information in higher frequencies does not refer to logic gates but to the device itself. We provide below (Figure 6.4) results that emphasize the positive effect of this preprocessing.

Detection. As a detection tool, we propose to use the widespread Pearson correlation coefficient that has shown its interest in many previous works in the domain of power-based cryptanalysis [37]. Let us recall that the Pearson correlation coefficient between two vectors x and y of length n having mean values $\bar{x}$ and $\bar{y}$ is defined as:

$$\rho(x,y) \triangleq \frac{\sum_{i=1}^{n}(x_i-\bar{x})\cdot(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i-\bar{y})^2}}.$$

Context. As mentioned in Section 6.3, the IP detection infrastructure can be used in different contexts. We specify the settings of our experimental work as follows. First, inputs to the device are unknown to the detection infrastructure and a single trace is used to perform the detection. Second, the source code is also unknown to the IP detection infrastructure. Third, the same technology, i.e. an Atmel ATmega644P, has been used to perform all the measurements. Finally, the same measurement setup has been used for all these measurements.

[2] We stress that the term noise is generic. It can refer to measurement noise, but also to noise due to minor transformations of the code or the use of different inputs for the evaluation phase.

Measurement setup. We obtained power consumption traces by measuring the voltage variations around an inductance at the input of the power supply of the microcontroller. The clock frequency of the controller was fixed to 20 MHz, and the sampling frequency to 50 MHz. For each IP, a reference trace corresponding to a single encryption was stored. Next, the suspicious traces were cut or padded (by re-encrypting the ciphertext), in order to deal realistically with the (frequent) situation were IP have different cycles counts[3].

6.5.2 Testing content sensitivity

Using different ciphers

As a first step, we applied our instance of SPH on different ciphers. We chose to focus on the 10 block ciphers in Table 6.1. This first experiment aims at confirming that the proposed IP detection infrastructure indeed provides content sensitivity and perceptual robustness in a simple case, where the set of IP-preserving transformations $\mathcal{F}_{\text{pre}}$ is reduced to identity, and the set $\bar{\mathcal{F}}_{\text{pre}}$ consists in choosing another cipher in the list.

Table 6.1.: Table of ciphers used as IP.

AES	DESXL	HIGHT	KASUMI
KATAN	mCrypton	NOEKEON	PRESENT
	SEA	TEA	

For each of the 100 possible couples of IP (reference, suspicious) we measured 20 traces (corresponding to different inputs) for both IPs, and applied the IP detection tool for each of the 400 possible pairs of traces. The similarity scores obtained are plotted in Figure 6.3.

Black dots are the scores obtained when IP are compared to $F(\text{IP})$ with $F \in \mathcal{F}_{\text{pre}}$ and gray dots are the one obtained when the transformation belongs to $\bar{\mathcal{F}}_{\text{pre}}$. This picture is encouraging : both content sensitivity and perceptual robustness are reached in experiments using a single trace. Indeed, for any τ value between 0.86 and 0.27, the detection rule makes no mistake for this particular set of IP and these particular sets $\mathcal{F}_{\text{pre}}$ and $\bar{\mathcal{F}}_{\text{pre}}$.

Remark on KASUMI. The reader may have noticed that almost all reference ciphers lead to scores higher than 0.95, except for KASUMI. The explanation is quite simple. Indeed, the execution path of KASUMI depends on the data being processed. To avoid timing attacks, cryptographers usually add `nop` operations to make the implementation time-constant. While this prevents the execution time from being data-dependent, this is not the case of instructions performed: traces obtained by processing different outputs will correspond to different

[3]We first ran tests with a single execution of the IP but then it was too easy to detect IPs. Since a cipher is generally used to encrypt more than one block we chose to perform tests on looping IPs.

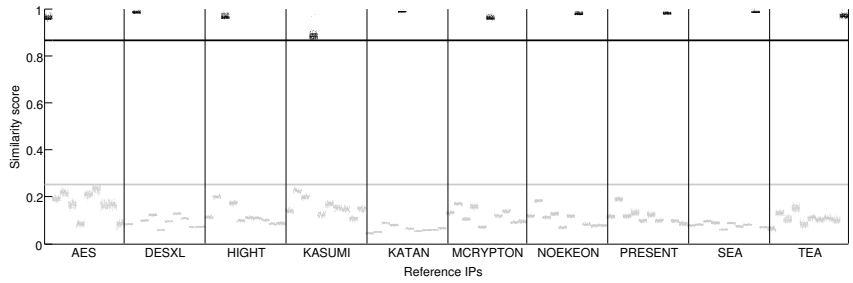

Fig. 6.3.: Similarity scores for different IP.

instruction sequences. This explains why we observe these smaller scores for KASUMI. Note also that a few scores are larger than 0.9 in Figure 6.3: they correspond to the comparison of two traces obtained with the same input.

Using different implementations of a same cipher.

As the setting of Figure 6.3 is quite favorable (since we are comparing totally different IP), we now move to a more challenging context and consider a single block cipher, namely the standard AES, and four of its implementations. Three of them are the *Fast, Furious* and *Fantastic* implementations that can be found in [150]. The fourth one is proposed in [62] and named *Francesco* in reference to the first-name of its programmer. Again, for each of the 16 possible couples of reference/suspicious IP, we performed 2500 experiments (corresponding to 50 traces for both IP). To highlight the positive effect of using an FFT, we plotted the results obtained with/without such extraction, in the right/left parts of Figure 6.4.

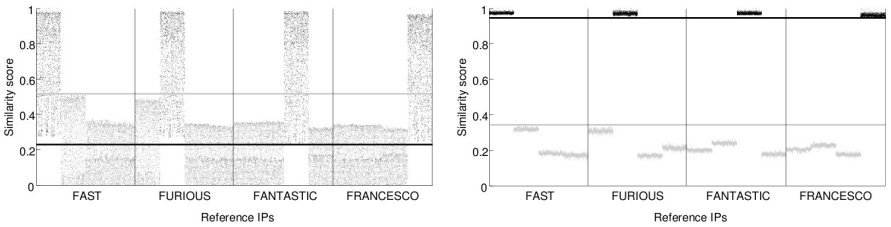

Fig. 6.4.: Similarity scores for different AES implementations using the FFT (right) or not (left).

As can easily be seen, and as expected, using the FFT in the extraction step drastically decreases the variance observed in the similarity scores in the case where traces are directly compared (left part), while keeping the content sensitivity property. Indeed, each implementation should be considered as a different IP since they are really different one from the other. This emphasizes the need of an extraction tool that compresses the feature vector to ensure

robustness: using FFT provides a wide range of thresholds (namely 0.37 to 0.95) for which both perceptual robustness and content sensitivity are equal[4] to 1 (that is, a 100% success rate) while without this tool, any value of the threshold will induce a non-zero error probability.

Yet, it remains that the content sensitivity could come from the simple fact that each implementation requires a different number of cycles to perform an encryption. In order to rule out this possibility, we additionally tuned the *Fast* and the *Furious* implementations in such a way that they have the same number of clock cycles and their only differences consist in actual round variations. For this purpose, the key expansion, first key addition, and final rounds remained the same and the inner rounds of the *Fast* implementation has been padded with `nop` operations. As can be observed in Figure 6.5, and even in this case, both implementations are detected as different IP with, again, a large range of thresholds leading to perceptual robustness and content sensitivity of 1.

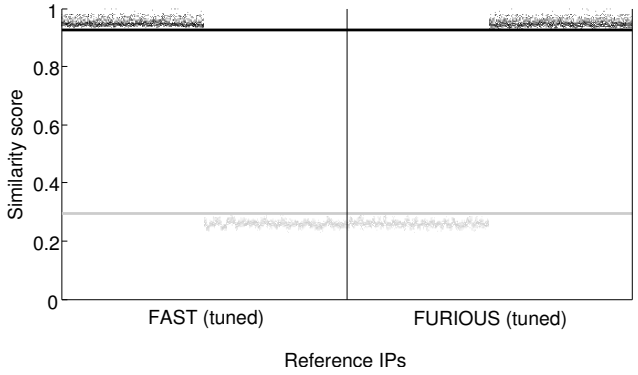

Fig. 6.5.: Similarity scores for tuned versions of *Fast* and *Furious* AES implementations.

6.5.3 Testing perceptual robustness

The previous experiments exhibit the good behavior of our SPH instance regarding content sensitivity. We now focus on its perceptual robustness. That is, can we still detect an IP when minor modifications have been applied to the original code? As previously mentioned, testing perceptual robustness requires to agree on a set of *IP-preserving* transformations. We identified two types of transformations, depending on whether they alter the performances or not:

- `addr`: i.e. changing registers or SRAM **addr**esses (non-altering);

[4]Obviously these values are only estimates since one cannot consider all possible IPs and all possible code modifications.

- `swap`: i.e. **swap**ping instructions if possible (non-altering);
- `dumo`: i.e. adding **dum**my operations **o**ut of the rounds (altering);
- `dumi`: i.e. adding **dum**my operations **i**n the rounds (altering).

We chose to apply these modifications to the *Furious* implementation of AES. Our results are given in Figure 6.6. Dots correspond to scores obtained when comparing the reference implementation (`ref`) to its modification by one or more of the listed transformations. The number of added dummy-operations is the same for both `dumo` and `dumi` (57 additional cycles). As can be seen, the proposed instance is directly robust regarding all non altering transformations. Concerning the addition of dummy operations, we can observe that increasing the encryption time does not always critically decrease the robustness (cf. `dumo`) but, that adding dummy operations inside a repetitive part of the cipher is more damaging. This observation naturally relates more to the simple nature of our extraction and detection procedures than to the very idea of SPH. Interestingly, simple tweaks allow to deal with such more challenging contexts, as illustrated in Figure 6.7.

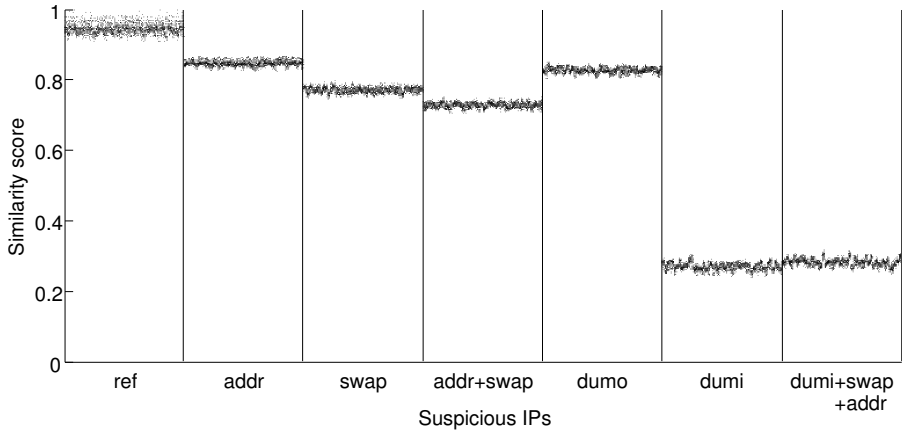

Fig. 6.6.: Similarity scores for transforms of the *Furious* implementation.

In this final figure, we slightly modified our SPH in order to first identify the block cipher rounds (using cross-correlation). Next, we applied the FFT separately on the different rounds, and used the average correlation over these rounds as detection procedure. While this approach remains clearly heuristic, the experimental results in Figure 6.7 show that it allows discriminating the addition of dummy operations within the rounds from the move towards another implementation of the AES.

To conclude, this set of experiments suggests that the concept we propose is valid and provides good results even when it is instantiated with classical tools.

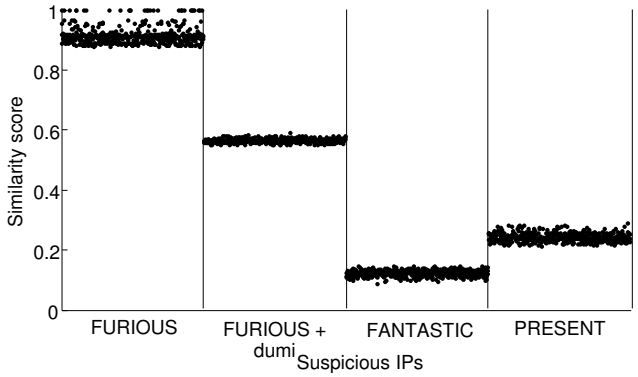

Fig. 6.7.: Improved similarity scores for the *Furious* implementation, its `dumi` transform and other IP.

Moreover, we have shown that more challenging scenarios can be handled using more advanced techniques. A deeper investigation of these scenarios is the natural next step now that this preliminary work has shown promising results.

6.6 FPGA CASE STUDIES

In this section, we analyze the application of our SPH function-based IP detection framework to FPGA designs. As previously mentioned, we considered three main scenarios for this purpose. First, we focused on the simple(st) case where the suspicious IPs are stand-alone FPGA designs corresponding to the object to protect. Second, we studied the case where the counterfeiter has re-synthetised the object to protect under a different set of constraints. Finally, we evaluated a more challenging (and realistic) context where the illegal IP is integrated in a larger system, with a parasitic design running in parallel. In the three scenarios, we first specify the components of the IP detection infrastructure we used, and then present experiments confirming its interest.

6.6.1 Stand-alone FPGA designs

Specification of the IP detection infrastructure.

We used the bitstreams of five lightweight ciphers (HIGHT, ICEBERG, KATAN, NOEKEON and PRESENT) and the AES as *objects to protect*. These bitstreams were obtained from the unrolled architectures described in Section 3.3, using the number of rounds per cycles that leads to maximum energy efficiency for each cipher, and synthetized for a Xilinx Virtex-II Pro FPGA. In this subsection, we used a default set of place-and-route options, leading to the performance results indicated in the left column of Table 6.2. We additionally used the

FPGA power consumption as *physical feature vector*. Measurement traces were obtained by measuring the voltage variations around a shunt resistor provided

Table 6.2.: Implementation results for the six considered IPs with two sets of synthesis options (default and area constrained).

	Default		Constrained	
	slices	T [ns]	slices	T [ns]
AES	1843	10.48	1687	9.28
HIGHT	573	7.76	596	8.13
ICEBERG	1123	9.04	1022	7.95
KATAN	725	7.02	650	6.87
NOEKEON	1125	5.94	1255	6.7
PRESENT	561	5.09	541	4.5

on the Sasebo-G board [162]. The device was running at a frequency of 24 MHz, and the oscilloscope sampling frequency was set at 2,5 GHz. In the following, we will characterize the SPH of each reference IP core with a feature vector made of the power consumption corresponding to one complete encryption. As for the suspicious IPs, the traces were simply obtained by measuring repeated encryptions. In this first case of stand-alone FPGA designs, the *extraction* phase directly outputs the complete traces (without performing any signal processing), and the *detection* process uses Pearson's correlation coefficient. That is, let x and y be two measurement vectors of length n with mean values $\bar{x}$ and $\bar{y}$, this coefficient is defined as:

$$\rho(x,y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}.$$

As reference traces and suspicious ones do not always have the same length, we further cropped the suspicious traces whenever longer than the reference ones[5]. The *context* in which the IP detection infrastructure is used is the following: the inputs provided to the IPs are unknown, we do not have access to the source code and the same device and measurements setup were used for all the tests.

Experimental results

Our first results are illustrated by Figure 6.8, where each column is linked to one particular reference IP compared to the six possible suspicious IPs. Each point in the figure corresponds to one experiment, i.e. one similarity score obtained from the comparison between one reference trace and one suspicious

[5]This assumes the traces to be synchronized, which can be achieved by different means, e.g. computing the correlation over a sliding window.

trace. For each suspicious-reference IP pair, 400 experiments have been performed, represented by black dots whenever considering identical IPs, and grey dots otherwise. We additionally plotted the content sensitivity threshold (i.e. the maximum score for different IPs) with a grey line, and the perceptual robustness threshold (i.e. the minimum score for similar IPs) with a black line. As can be observed, the performances in this context are already reasonably good. Yet, some non-detections and false alarms occur for "close IPs", assumably because of measurement noise in our traces. The closeness between different IPs is further emphasized in Figure 6.9, where each square in the matrix corresponds to the mean similarity score between the IPs, and the higher similarity scores are indicated with a darker color code.

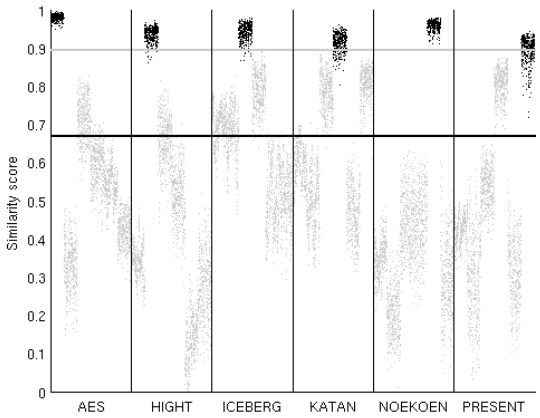

Fig. 6.8.: Similarity score scatter plot for stand-alone designs. Single reference traces / single suspicious traces / unknown inputs.

Eventually, in order to improve our performances and get rid of the noise, a natural solution is to exploit averaged traces rather than single traces, both for the reference and suspicious IPs. The result of this additional experiment (with a 10 times averaging) is given in Figure 6.10.

In this latter case, we clearly see that no false alarm nor non-detections happens anymore, and the threshold for perceptual robustness is now higher than the one for content sensitivity in all cases.

6.6.2 Re-synthetized FPGA designs

The results in the previous subsection were considering similar algorithms for which the IP cores could be detected even in a simple context (unknown inputs and source code). Quite naturally, an important question is whether such an observation is still valid in the more challenging scenario where a

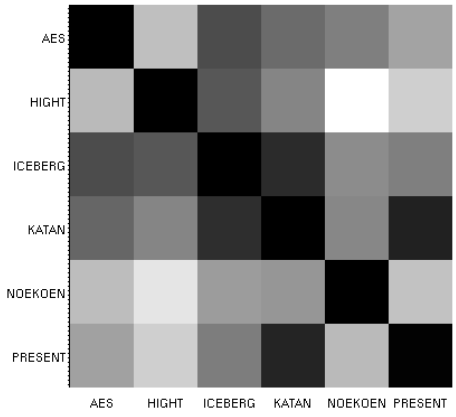

Fig. 6.9.: Similarity score correlation matrix for stand-alone designs. Single reference traces / single suspicious traces / unknown inputs.

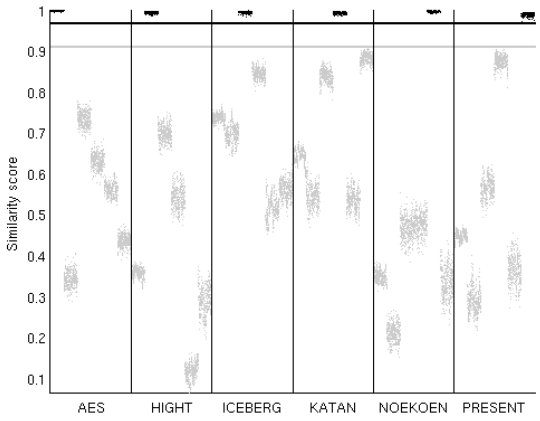

Fig. 6.10.: Similarity score scatter plot for stand-alone designs. 10 times averaged reference traces and suspicious traces / unknown inputs.

potential counterfeiter would apply some IP-preserving transformations. This is the quesion we tackle in this subsection, considering the placement and routing of our designs with a different set of constraints as an IP-preserving transformation.

Specification of the IP detection infrastructure

In this case, the object to protect is not the bitstream anymore, but rather the source code or netlist. For this purpose, we used both sets of constraints in Table 6.2. Since the previous IP detection infrastructure turned out to be insufficient in this case, we additionally considered two tweaks to improve detection. First, we moved to a known input context, allowing us to take advantage of data dependencies in the traces. Next, we used a more sophisticated extraction procedure, based on a selection of *points of interest* (POIs). Namely, we split the traces into consecutive clock cycles, using the Fast Fourier Transform to recover the rising edges of the clock signal. This was achieved by filtering the frequency spectrum around the clock frequency and its harmonics, then applying the inverse transform on the filtered signal. This preprocessing provides a sequence of peaks indicating where the rising edges of the clock signal are. Following, we were able to work on a sequence of clock cycles instead of raw side-channel traces. From this sequence, we finally extracted POIs having maximum *signal to noise ratio* (SNR) for each cycle (with the SNR defined as the variance over mean traces for different plaintexts divided by the noise variance).

Experimental results

For illustration, we first provide the similarity score correlation matrix of our re-synthetized designs obtained from the IP detection infrastructure of the previous subsection in Figure 6.11. Each line/column is now divided in two, for the two sets of synthesis options. As expected, the re-synthesis is not perfectly IP-preserving in this case. For example, the re-synthetized PRESENT design appears as a new IP. Interestingly, this issue is not really caused by differences in performances (results provided in Table 6.2 are quite similar for the two implementation contexts), but rather by the placement-and-routing illustrated in Figure 6.12.

By contrast, taking advantage of the known input context with the selection of POIs proposed in the previous paragraph directly allows getting rid of this limitation. This is illustrated in Figure 6.13 in which we only analysed the IPs giving low similarity scores in the first detection scenario. A natural explanation for the obtained results is that the operation dependencies in the power consumption, that were exploited in the previous section, are significantly affected by re-synthesis. By contrast, data dependencies that are additionally considered in the known input scenario are more robust against such transforms (since the same data has to be manipulated). In this respect, an even more challenging type of transforms would try to affect the intermediate data during the computations of an IP, without affecting its final result. We leave their investigation for further research. Note that the exploitation of data dependencies crucially depends on a good selection of POIs. For illustration we additionally provided the weaker detection results obtained with an uninformed selection in Figure 6.14.

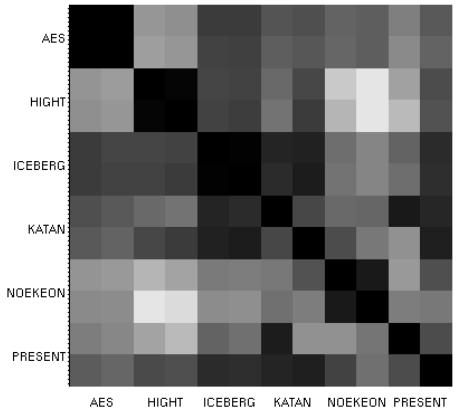

Fig. 6.11.: Similarity score correlation matrix for re-synthetized designs. Average on 10 reference traces and 10 suspicious traces / unknown inputs.

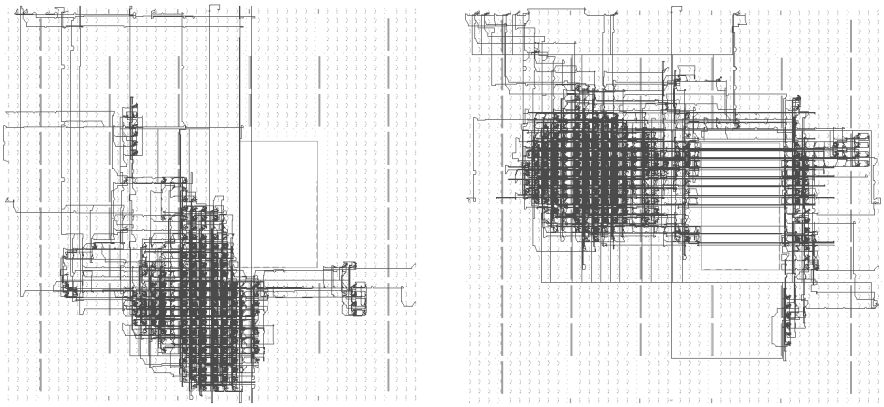

Fig. 6.12.: Floorplan of the PRESENT designs.

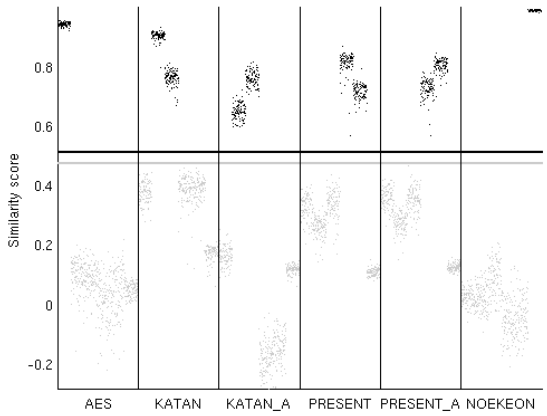

Fig. 6.13.: Similarity score scatter plot for re-synthetized designs. Single reference traces / single suspicious traces / known inputs.

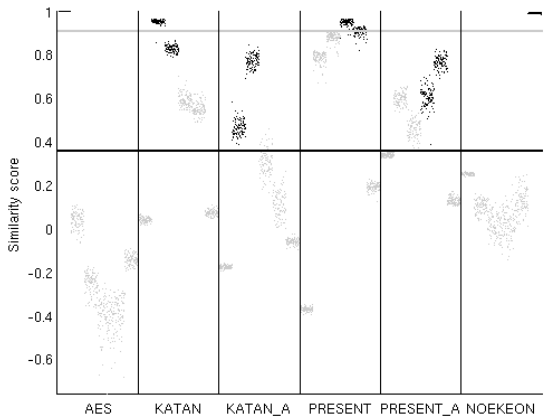

Fig. 6.14.: Same as Figure 6.13 with different POIs.

6.6.3 Parasitic IP running in parallel

Before concluding this work, we finally considered the practically important case study, where not only our suspicious IP would run on an FPGA, but also some parasitic one.

Specification of the IP detection infrastructure

We investigated the case of a Linear Feedback Shift Register (LFSR) running in parallel to PRESENT (we limited our evaluations to this IP because it was

the most challenging one in the previous subsection). The performances of the different suspicious IPs with such an additional LFSR are given in Table 6.3 for three different LFSR sizes. For the rest, we used the same IP detection infrastructure as in the previous subsection. The only difference is that we will need to work with (5 times) averaged traces in order to get rid of the "algorithmic noise" coming from the parallel execution of the LFSRs.

Table 6.3.: Implementation results for PRESENT with parallel LFSR.

	Default	
	slices	T [ns]
PRESENT + 64-bit LFSR	608	4.87
PRESENT + 512-bit LFSR	797	4.93
PRESENT + 1024-bit LFSR	1053	4.76

Experimental results

The scatter plot of this final experiment is given in Figure 6.15, where we used the same subset of reference IPs as in Figure 6.13. In all three cases and despite the presence of a parasitic LFSR, we were able to detect the suspicious PRESENT. Interestingly, we can also observe the impact of larger algorithmic noise, as the distance between the content sensitivity and perceptual robustness thresholds decreases with the LFSR size. As a counterfeited IP will be

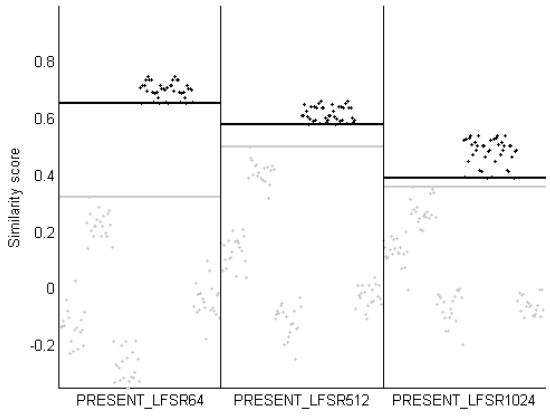

Fig. 6.15.: Similarity score scatter plot for designs with parasitic IP. 5 times averaged reference traces and suspicious traces / known inputs.

integrated in even larger designs, we can expect that the detection will still succeed given that larger amounts of traces are averaged during extraction.

6.7 CONCLUSION

This chapter introduced, defined and proposed a first instantiation of SPH infrastructure, as a flexible and efficient way to protect the IP of integrated systems. We believe that the main interest of this IP detection mechanism is its flexibility. First, it does not require to modify any piece of the original designs. Second, the decision to test a suspicious IP can be taken a posteriori (without beging considered at development time). Eventually, it is potentially useful against adversaries that can operate on high-level IP (e.g. source codes or netlists). By contrast, permission-based solutions and watermarks are hardly effective in this case.

Quite naturally, this flexibility comes at the cost of sometimes difficult detection. While preliminary experiments in this work are promising and confirm the relevance of the approach, considering more challenging scenarios (e.g. different technologies, other parasitic IPs, other measurement setups) is certainly important. It is likely that advanced extraction and detection tools will be needed in these cases. Yet, we hope that SPH functions-based IP detection can be a useful element to solve the challenge of counterfeited electronic designs, combined with other ideas with different strenghts and weaknesses.

CONCLUSIONS AND PERSPECTIVES

Implementing cryptographic algorithms on physical devices gives rise to several concerns. During this thesis, we covered three of them that can be summarized with the following questions:

1. How can we fairly compare the performances of algorithms? What is the definition of low-cost? Are lightweight cryptographic algorithms needed for embedded applications? How do they compare to standard algorithms? What knowledge do we have on the impact of certain design choices on the implementation results?

2. To what extent do countermeasures against side-channel attacks improve security and at which cost? What is the impact of new technologies on the physical security and efficiency of cryptographic implementations?

3. How to protect IPs in a way that is both flexible and secure?

In the remainder of this chapter, we will summarize the contributions related to each of these topics. We will also give some general conclusions for each of them and discuss their applicability in future works. We will conclude this chapter by a general discussion about the impact of automated implementation and evaluation tools on the confidence in the optimality of the results.

EVALUATION OF CRYPTOGRAPHIC ALGORITHMS

In Chapter 3 and 4, we evaluated the performances of two categories of cryptographic algorithms (i.e. lightweight block ciphers and hash functions) in two implementation scenarios (i.e. software and hardware implementations). In order to get fair comparisons, we first provided common implementation guidelines and methodologies for each of the four case-studies. We then implemented the cryptographic algorithms following these guidelines and proposed various performances results.

From these results we extracted some interesting conclusions. First, looking at the big picture, we realized that software implementation results are in general easier to interpret than hardware ones and similarly, that we usually have a better understanding of the impact of some design choices on the performances in the case of software than in hardware. This probably

is a consequence of the fact that in software devices, the hardware is fixed and the only possible tradeoff while implementing an algorithm concerns the implementation code size versus its cycle count. Furthermore, software implementations are easier to carry out, which allows more research in the subject. In the same line of thinking, our knowledge of hash function is still limited compared to the one we have for block ciphers. This results presumably from the research effort that has mainly been placed on the design of efficient and lightweight block ciphers over the last decade. The only exception to that trend comes from the SHA-3 competition which has allowed the development of several new hash functions. There is however still a lot to learn regarding the design of efficient hash functions.

We also tried to determine the relevance of using lightweight block ciphers with respect to the AES. Changing an already implemented algorithm may be expensive and its impact on performances may be limited even if non negligible. We therefore think that for most applications, AES is probably good enough and if area (or code size) is the limiting criteria, then the existing lightweight block ciphers are surely already efficient enough. Yet, this does not mean that we do not need new ciphers, but rather that those ciphers should bring new perspectives (e.g. ciphers that can easily be masked, ciphers with new properties such as low latency [32], ciphers with efficient and inventive key-scheduling algorithms, etc.).

Regarding the metrics, it is still difficult to define lightweight. This will highly depend on the application scenario. The results however showed that normalized metrics such as energy and throughput over area are more illustrative in the case of performance comparison than the standalone metrics. It is indeed generally possible to modify the area or code size to application needs (with an impact on other metrics), whereas in that case the relative metrics will stay more or less stable. A last conclusion we draw from the results we obtained previously is that comparative studies in a compact implementation scenario are less trivial and help revealing the complexities of the algorithms. These compact implementations should therefore always be an integral part of standardization processes.

To conclude, the comparison tools (i.e. common methodology and interfaces) provided in this work may reveal themselves extremely helpful in the future. Indeed, they allow us to easily compare any new cipher or hash function design to several other existing ones. As an example, the authors of [23] and [101] compared results obtained from the software implementation of respectively Lapin and ITUbee ciphers to some of our results. Similarly, it would be interesting to get implementation results using our framework for the two new ciphers Simon and Speck [20] which seem to provide an efficient alternative to AES and could thereby sightly modify our conclusion regarding lightweight ciphers. Next to that, our comparison tools could also easily be extended to any other kind of algorithms. For example, the presently running competition CAESAR [48], aiming at finding an authenticated cipher which is secure, applicable and robust, could greatly benefit from these guidelines

to compare the different candidates. Incidentally, we took advantage of these tools in order to rapidly get software and hardware implementation results for the proposed authenticated ciphers Scream and iScream [80]. That way, we could easily compare the performances of our tweakable block ciphers with respect to the AES. Overall, even if it is already partially proposed in competition scenario, we would suggest that the cryptographic community defines some common guidelines for fair evaluation of algorithms. These guidelines could for example include a proposed type of device on which the implementation should be carried on as well as implementation goals, common interfaces and test vectors.

IMPLEMENTATION AND SECURITY EVALUATION OF COUNTERMEASURES

In Chapter 5, we considered two different subjects. First, we investigated the security of the shuffling countermeasures. Then, we proposed the FRAM as a promising technology for countermeasures taking advantage of precomputations.

Regarding the first subject, we first proposed two new techniques for implementing the shuffling countermeasure. These techniques allow better performances than the usual ones by manipulating the program flow. More specifically, our first implementation controls the program counter in order to modify the program flow on-the-fly, whereas the second one adapts itself automatically before each execution by re-writing the program memory. We presented very good performance results for our two new approaches, with the second approach being as fast as the unprotected AES if we only consider the online encryption phase. The main shortcoming from the second approach is its need for program memory modifications, which are slow on the chosen microcontroller.

We then investigated the security against side-channel attacks of the shuffling countermeasure. In order to do so, we provided a new evaluation framework adapted to the case of shuffling. This evaluation framework allowed us to show that the previously proposed integrated attack is not the best approach for assessing the security of the countermeasure. Besides, as a natural step towards being as close as possible to the usual security evaluation used for masking (for which the leakages of all shares are exploited), we integrated the permutation leakages inside our evaluation. We also put forward that simplifying the permutation generation (e.g. by using a random start index instead of a random permutation) can lead to a complete breakdown of the countermeasure. We finally introduced the concept of indirect leakages. Indeed, different resources in a microcontroller (i.e. different registers) may have slightly different leakage models. This leads to potential stronger attacks against the countermeasure that can even be carried out against an implementation computing all the sensible information concerning the permutation

in a non leaking environment. As a conclusion, we showed by our results that shuffling should not be used as standalone countermeasure or at least not as noise generator but as a noise amplifier.

For the second subject, we first improved the results we obtained for the randomized program memory approach by the use of FRAM. This kind of RAM allows non-volatile storage as well as fast and an almost infinite re-writing capabilities. Next, we provided the first working implementation of the RLUT countermeasure applied to a reduce version of the LED block cipher. Again this implementation was mainly made possible by the use of FRAM. We then demonstrated that the memory and time required by the RLUT countermeasure can be predicted for the parameters of any ciphers using the formulas that were provided in [179]. We finally investigated the context of complete secure pre-computation as well as various possible tradeoffs corresponding to partial (and partially leaking) implementations of the countermeasure. In conclusion, we showed that if secure pre-computation (i.e. in a non-leaking environment) is possible, then the RLUT countermeasure provides unconditional security against side-channel attacks. Furthermore, the use of RLUT with partial pre-computations in a leaking environment is an interesting alternative to higher-order masking as it could presumably provide higher security margins at a similar cost.

From these contributions, we may conclude that implementing and evaluating the security of proposed countermeasure is an important topic of research. In particular, it is always interesting to evaluate the possible tradeoffs between security and efficiency and to have a better knowledge of the cost and security associated to a particular countermeasure. Some future scope of research have been proposed for the two countermeasures we have been studying. For example, for the shuffling countermeasure, it would be interesting to think of a new implementation approach preventing the indirect leakages (e.g. by shuffling the used resources additionally to the operations). From the RLUT countermeasure point of view, there is still a lot of work to be done regarding the security evaluation of the proposed tradeoffs. It would indeed be interesting to know what is the impact of partial pre-computations in a leaking environment. Finally, designing new block ciphers to which the RLUT countermeasure could easily be applied is another interesting research direction.

IP PROTECTION

In Chapter 6, we proposed a new IP protection infrastructure that applies both to high-level and low-level IPs. This infrastructure takes advantage of the side-channel leakages of the implementation combined with the concept of SPH. First, we gave some details about the existing types of IPs and the possible attack scenarios. Then, based on the definition of SPH functions, we specified our new detection infrastructure. We defined the different parts composing it as well as some performance metrics (i.e. the perceptual robustness

and content sensitivity of the scheme). This protection scheme was finally experimented on two main case studies: software and hardware IPs. For each of these case studies, we started from a simple detection scenario comprising only different unmodified IPs and increased progressively the difficulty of the detection in order to get more realistic scenarios (e.g. different synthesis of the netlist, addition of parasitic IPs, modifications on the code, etc.).

In the end, our protection infrastructure provides very promising results. Even if we sometimes needed to make use of more advanced detection tools, we were able to successfully detect all the IPs in the different detection scenarios. The main interest of this infrastructure is its flexibility since no modifications of the original design are needed. Besides, the detection mechanism can be used a posteriori and therefore does not need any considerations at the development time. Some very interesting research results could be derived from this detection infrastructure. In particular, one could consider even more challenging scenarios (e.g. different technologies, other types of parasitic IPs, different types of IP-preserving transformation, etc.). Then, evaluating the performances of other extraction and detection tools is certainly important. As an example, in [85], the use of support vector machines is investigated as a detection tool for our SPH-based infrastructure. A last possible research direction would be to amplify the IP signature with for example the addition of salware [33]. In summary, being able to determine up to which extent this protection mechanism may be applied is very important. Also, we hope that this solution (that could also be combined to other ones) will help designers in their fight against unauthorized use of IPs.

AUTOMATION VERSUS OPTIMALITY AND CONFIDENCE

To conclude this work, we would like to have a brief discussion regarding the possible approaches when it comes to efficiency or security evaluation. Usually, the evaluator wants to get the results as fast as possible. This usually is related to automation of the implementation and evaluation tools. However, how can he be confident about the optimality of the results when they come from a completely automated process?

As a first example, some high-level synthesis tools have recently been brought to light by vendors from the semiconductor industry (e.g. [207]). These tools allow a speed up of the industrial process by enabling C, C++ and SystemC specifications to be directly targeted into their FPGAs. While such tools may decrease the production time, they do not necessarily provide high confidence on the optimality of the implementation results. Some industrials may however tend to choose these automated tools as long as the possible increase in area is canceled out by the decrease in engineering costs. From a research point of view, this is however less true. Indeed, implementations results provided by the cryptographic community are often intended to be compared to each other. It is therefore important that the confidence on the

optimality of these results is high since they may later become references that will be looked at when an algorithmic choice is needed.

In the same line of thinking, regarding the implementation of countermeasure or their security evaluation, we need to a have high confidence in the optimality of the results obtained. Indeed, while having a lack of confidence on some implementation results is tolerable, missing one potential attack against a countermeasure or not taking into account some parameters while applying the countermeasure or performing a security evaluation may be devastating. Possible questions here are then: would an automated tool have given a similar security evaluation on the shuffling countermeasure than the one we proposed previously? Is it possible to automatically apply a countermeasure to any existing algorithms without any human interaction?

To these questions, we presently tend to answer no. Even if some parts of a security evaluation give good results while performed automatically (e.g. search of points of interests, dimension reduction, etc.), some other parts still need human interactions. It is indeed difficult to find tools and general evaluation scenarios that are applicable to all the countermeasures. This is also true for the application of countermesures to a particular algorithm (e.g. we saw that shuffling a code was not as straightforward as it could seem since some parts of the code needed to be modified in order to get independent operations). The search of efficient automatic evaluation and design tools (such as proposed in [18, 19, 137]) however remains a very interesting topic of research as, even if they may not necessarily give perfect confidence in the result, they could still help the evaluator in many ways.

BIBLIOGRAPHY

[1] The SHA-3 zoo. http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.

[2] *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*. IEEE Computer Society, 2010.

[3] 3rd Generation Partnership Project. Technical specification group services and system aspects, 3G security, specification of the 3GPP confidentiality and integrity algorithms, document 2: KASUMI specification (release 10), 2011.

[4] Amr T. Abdel-Hamid, Sofiène Tahar, and El Mostapha Aboulhamid. A survey on IP watermarking techniques. *Design Autom. for Emb. Sys.*, 9(3):211–227, 2004.

[5] Altera. FPGAs. Available online at http://www.altera.com/products/fpga.html.

[6] Antoine Amarilli, Sascha Müller, David Naccache, Dan Page, Pablo Rauzy, and Michael Tunstall. Can code polymorphism limit information leakage? In Claudio Agostino Ardagna and Jianying Zhou, editors, *WISTP*, volume 6633 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2011.

[7] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A model for self-modifying code. In Jan Camenisch, Christian S. Collberg, Neil F. Johnson, and Phil Sallee, editors, *Information Hiding*, volume 4437 of *Lecture Notes in Computer Science*, pages 232–248. Springer, 2006.

[8] Frederik Armknecht, Roel Maes, Ahmad-Reza Sadeghi, François-Xavier Standaert, and Christian Wachsmann. A formalization of the security features of physical functions. In *IEEE Symposium on Security and Privacy*, pages 397–412. IEEE Computer Society, 2011.

[9] Atmel. Atmel AVR 8-bit and 32-bit microcontrollers. http://www.atmel.com/products/microcontrollers/avr/.

[10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[11] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. QUARK: A lightweight hash. In Mangard and Standaert [124], pages 1–15.

[12] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. QUARK C implementation. Available online at https://www.131002.net/quark/, 2010.

[13] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE (version 1.4), 2011. http://131002.net/blake/.

[14] Jean-Philippe Aumasson, Willi Meier, and Raphael C.-W. Phan. The hash function family LAKE. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2008.

[15] Catalin Baetoniu. FPGA IFF copy protection using Dallas semiconductor/Maxim DS2432 secure EEPROMs. XAPP780, May 28, 2010.

[16] Josep Balasch, Baris Ege, Thomas Eisenbarth, Benoît Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, Thomas Pöppelmann, Francesco Regazzoni, François-Xavier Standaert, Gilles Van Assche, Ronny Van Keer, Loïc van Oldeneel tot Oldenzeel, and Ingo von Maurich. Compact implementation and performance evaluation of hash functions in ATtiny devices. In Stefan Mangard, editor, *CARDIS*, volume 7771 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2012. Open source codes available online at http://perso.uclouvain.be/fstandae/source_codes/hash_atmel/.

[17] Lejla Batina, Benedikt Gierlichs, Emmanuel Prouff, Matthieu Rivain, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. Mutual information analysis: a comprehensive study. *J. Cryptology*, 24(2):269–291, 2011.

[18] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. Sleuth: Automated verification of software power analysis countermeasures. In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES*, volume 8086 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2013.

[19] Ali Galip Bayrak, Nikola Velickovic, Paolo Ienne, and Wayne Burleson. An architecture-independent instruction shuffler to protect against side-channel attacks. *TACO*, 8(4):20, 2012.

[20] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. http://eprint.iacr.org/.

[21] Georg T. Becker, Markus Kasper, Amir Moradi, and Christof Paar. Side-channel based watermarks for integrated circuits. In Jim Plusquellic and Ken Mai, editors, *HOST*, pages 30–35. IEEE Computer Society, 2010.

[22] Daniel J. Bernstein. Chacha, a variant of salsa20. Workshop Record of SASC 2008: The State of the Art of Stream Ciphers, 2008. http://cr.yp.to/chacha.html#chacha-paper.

[23] Daniel J. Bernstein and Tanja Lange. Never trust a bunny. In Jaap-Henk Hoepman and Ingrid Verbauwhede, editors, *RFIDSec*, volume 7739 of *Lecture Notes in Computer Science*, pages 137–148. Springer, 2012.

[24] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The KECCAK reference, January 2011. http://keccak.noekeon.org/.

[25] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. KECCAK implementation overview, September 2011. http://keccak.noekeon.org/.

[26] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[27] Jean-Luc Beuchat, Eiji Okamoto, and Teppei Yamazaki. Compact implementations of BLAKE-32 and BLAKE-64 on FPGA. Cryptology ePrint Archive, Report 2010/173, 2010. http://eprint.iacr.org/.

[28] Eli Biham and Orr Dunkelman. A framework for iterative hash functions - HAIFA. Cryptology ePrint Archive, Report 2007/278, 2007. http://eprint.iacr.org/.

[29] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. SPONGENT: The design space of lightweight cryptographic hashing. *IACR Cryptology ePrint Archive*, 2011:697, 2011.

[30] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In Paillier and Verbauwhede [149], pages 450–466.

[31] David Bol. Pushing ultra-low-power digital circuits into the nanometer era. Technical report, Université catholique de Louvain, 2008. Ph.D dissertation.

[32] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - a low-latency block cipher for pervasive computing applications - extended abstract. In Wang and Sako [195], pages 208–225.

[33] Lilian Bossuet and David Hely. SALWARE: Salutary Hardware to design Trusted IC. In *Workshop on Trustworthy Manufacturing and Utilization of Secure Devices, TRUDEVICE 2013*, Avignon, France, May 2013.

[34] K.A. Bowman, X. Tang, J.C. Eble, and J.D. Menldl. Impact of extrinsic and intrinsic parameter fluctuations on CMOS circuit performance. *Solid-State Circuits, IEEE Journal of*, 35(8):1186–1193, 2000.

[35] Zvika Brakerski, Yael Tauman Kalai, Jonathan Katz, and Vinod Vaikuntanathan. Overcoming the Hole in the Bucket: Public-Key Cryptography Resilient to Continual Memory Leakage. In *FOCS* [2], pages 501–510.

[36] Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[37] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.

[38] Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - a family of small and efficient hardware-oriented block ciphers. In Clavier and Gaj [43], pages 272–288.

[39] Çetin Kaya Koç and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2000, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*. Springer, 2000.

[40] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Wiener [200], pages 398–412.

[41] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.

[42] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In Çetin Kaya Koç and Paar [39], pages 252–263.

[43] Christophe Clavier and Kris Gaj, editors. *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *LNCS*. Springer, 2009.

[44] Christian S. Collberg and Clark D. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Trans. Software Eng.*, 28(8):735–746, 2002.

[45] Jean-Sébastien Coron. A new DPA countermeasure based on permutation tables. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN*, volume 5229 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2008.

[46] Atmel Corporation. *8-bit Atmel Microcontroller with 16K/32K/64K Bytes In-System Programmable Flash - ATmega164P/V ATmega324P/V ATmega644P/V*, rev. 8011q edition, February 2013. Available online at http://www.atmel.com/Images/Atmel-8011-8-bit-AVR-Microcontroller-ATmega164P-324P-644P_datasheet.pdf.

[47] Atmel Corporation. *Atmel 8-bit AVR Microcontroller with 2/4/8K Bytes In-System Programmable Flash - ATtiny25/V ATtiny45/V ATtiny85/V*, rev. 2586q edition, August 2013. Available online at http://www.atmel.com/images/atmel-2586-avr-8-bit-microcontroller-attiny25-attiny45-attiny85_datasheet.pdf.

[48] International cryptologic research community. Competition for authenticates encryption: Security, applicability and robustness (CAESAR). http://competitions.cr.yp.to/caesar.html.

[49] P. Bulens D. Giry. Bluekrypt - cryptographic key length recommendation, October 2013. http://www.keylength.com/.

[50] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie proposal: NOEKEON, 2000. Available online at http://gro.noekeon.org/Noekeon-spec.pdf.

[51] Joan Daemen and Vincent Rijmen. The block cipher Rijndael. In Jean-Jacques Quisquater and Bruce Schneier, editors, *CARDIS*, volume 1820 of *Lecture Notes in Computer Science*, pages 277–284. Springer, 1998.

[52] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[53] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. In *Proc. first AES conference*, August 1998. Available on-line from the official AES page: http://csrc.nist.gov/encryption/aes/aes_home.htm.

[54] Ivan Damgård. A design principle for hash functions. In Brassard [36], pages 416–427.

[55] Rémy Daudigny, Hervé Ledig, Frédéric Muller, and Frédéric Valette. SCARE of the DES. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *ACNS*, volume 3531 of *Lecture Notes in Computer Science*, pages 393–406, 2005.

[56] Giacomo de Meulenaer, François Gosset, François-Xavier Standaert, and Olivier Pereira. On the energy cost of communication and cryptography in wireless sensor networks. In *WiMob*, pages 580–585. IEEE, 2008.

[57] Yevgeniy Dodis, Kristiyan Haralambiev, Adriana López-Alt, and Daniel Wichs. Cryptography against Continuous Memory Attacks. In *FOCS* [2], pages 511–520.

[58] Saar Drimer. Authentication of FPGA bitstreams: Why and how. In Pedro C. Diniz, Eduardo Marques, Koen Bertels, Marcio Merino Fernandes, and João M. P. Cardoso, editors, *ARC*, volume 4419 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2007.

[59] Saar Drimer. Security for volatile FPGAs. Technical Report UCAM-CL-TR-763, University of Cambridge, Computer Laboratory, November 2009.

[60] François Durvaux, Benoît Gérard, Stéphanie Kerckhof, François Koeune, and François-Xavier Standaert. Intellectual property protection for integrated systems using soft physical hash functions. In Dong Hoon Lee and Moti Yung, editors, *WISA*, volume 7690 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012.

[61] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-Resilient Cryptography. In *FOCS*, pages 293–302. IEEE Computer Society, 2008.

[62] Thomas Eisenbarth, Zheng Gong, Tim Güneysu, Stefan Heyse, Sebastiaan Indesteege, Stéphanie Kerckhof, François Koeune, Tomislav Nad, Thomas Plos, Francesco Regazzoni, François-Xavier Standaert, and Loïc van Oldeneel tot Oldenzeel. Compact implementation and performance evaluation of block ciphers in ATtiny devices. In Aikaterini Mitrokotsa and Serge Vaudenay, editors, *AFRICACRYPT*, volume 7374

of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2012. Open source codes available online at http://perso.uclouvain.be/fstandae/source_codes/lightweight_ciphers/.

[63] Thomas Eisenbarth, Stefan Heyse, Ingo von Maurich, Thomas Poeppelmann, Johannes Rave, Cornel Reuber, and Alexander Wild. Evaluation of SHA-3 candidates for 8-bit embedded processors. The Second SHA-3 Candidate Conference, 2010.

[64] Thomas Eisenbarth, Sandeep S. Kumar, Christof Paar, Axel Poschmann, and Leif Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 24(6):522–533, 2007.

[65] Thomas Eisenbarth, Christof Paar, and Björn Weghenkel. Building a side channel based disassembler. *Transactions on Computational Science*, 10:78–99, 2010.

[66] Adam J. Elbirt, W. Yip, B. Chetwynd, and Christof Paar. An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists. In *AES Candidate Conference*, pages 13–27, 2000.

[67] Johannes Feichtner. http://www.groestl.info/implementations.html.

[68] Martin Feldhofer and Thomas Popp. Power analysis resistant AES implementation for passive RFID tags. In Christiopher Lackner, Timm Ostermann, Michael Sams, and Ronal Spilka, editors, *Austrochip 2008*, pages 1–6, 2008.

[69] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family (version 1.3), 2010. http://www.skein-hash.info/.

[70] R. H. Fowler and L. W. Nordheim. "electron emission in intense electric fields". *Proc R. Soc. London, Ser. A*, vol. 119(781):173, 1928.

[71] Jiri Fridrich and Miroslav Goljan. Robust hash functions for digital watermarking. In *ITCC*, pages 178–183. IEEE Computer Society, 2000.

[72] Kris Gaj and Pawel Chodowiec. Comparison of the hardware performance of the AES candidates using reconfigurable hardware. In *AES Candidate Conference*, pages 40–54, 2000.

[73] Kris Gaj, Ekawat Homsirikamol, and Marcin Rogawski. Fair and comprehensive methodology for comparing hardware performance of fourteen round two SHA-3 candidates using FPGAs. In Mangard and Standaert [124], pages 264–278.

[74] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.

[75] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. SHA-3 proposal Grøstl (version 2.0.1), 2011. http://www.groestl.info/.

[76] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In Oswald and Rohatgi [145], pages 426–442.

[77] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In Wagner [194], pages 39–56.

[78] Zheng Gong, Svetla Nikova, and Yee Wei Law. KLEIN: A new family of lightweight block ciphers. In Ari Juels and Christof Paar, editors, *RFIDSec*, volume 7055 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011.

[79] Louis Goubin and Mitsuru Matsui, editors. *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *LNCS*. Springer, 2006.

[80] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, Kerem Varici, François Durvaux, Lubos Gaspar, and Stéphanie Kerckhof. SCREAM & iSCREAM: Side-channel resistant authenticated encryption with masking. submission to the CAESAR competition. First round submission available online at http://competitions.cr.yp.to/caesar-submissions.html.

[81] Jorge Guajardo, Sandeep S. Kumar, Geert Jan Schrijen, and Pim Tuyls. FPGA intrinsic PUFs and their use for IP protection. In Paillier and Verbauwhede [149], pages 63–80.

[82] Sylvain Guilley, Laurent Sauvage, Julien Micolod, Denis Réal, and Frédéric Valette. Defeating any secret cryptography with SCARE attacks. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *LATINCRYPT*, volume 6212 of *Lecture Notes in Computer Science*, pages 273–293. Springer, 2010.

[83] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2011.

[84] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED block cipher. In Preneel and Takagi [154], pages 326–341.

[85] Ludovic-Henry Gustin, François Durvaux, Stéphanie Kerckhof, François-Xavier Standaert, and Michel Verleysen. Support vector machines for improved IP detection with soft physical hash functions. In Emmanuel Prouff, editor, *COSADE*, Lecture Notes in Computer Science. Springer, 2014. To appear.

[86] Luca Henzen, Pietro Gendotti, Patrice Guillet, Enrico Pargaetzi, Martin Zoller, and Frank K. Gürkaynak. Developing a hardware evaluation method for SHA-3 candidates. In Mangard and Standaert [124], pages 248–263.

[87] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES smart card implementation resistant to power analysis attacks. In Jianying Zhou, Moti Yung, and Feng Bao, editors, *ACNS*, volume 3989 of *Lecture Notes in Computer Science*, pages 239–252, 2006.

[88] Shoichi Hirose. Some plausible constructions of double-block-length hash functions. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2006.

[89] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. Comparing hardware performance of fourteen round two SHA-3 candidates using FPGAs. Cryptology ePrint Archive, Report 2010/445, 2010. http://eprint.iacr.org/.

[90] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. Throughput vs. area trade-offs in high-speed architectures of five round 3 SHA-3 candidates implemented using Xilinx and Altera FPGAs. In Preneel and Takagi [154], pages 491–506.

[91] Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bonseok Koo, Changhoon Lee, Donghoon Chang, Jaesang Lee, Kitae Jeong, Hyun Kim, Jongsung Kim, and Seongtaek Chee. HIGHT: A new block cipher suitable for low-resource device. In Goubin and Matsui [79], pages 46–59.

[92] Texas Instruments. *MSP430FR57xx Family*, rev. 8011q edition, November rev. C. Available online at http://www.ti.com/lit/ug/slau272c/slau272c.pdf.

[93] Jeff Johnson. List and comparison of FPGA companies, 2011. Available online at http://www.fpgadeveloper.com/2011/07/list-and-comparison-of-fpga-companies.html.

[94] Bernhard Jungk and Jürgen Apfelbeck. Area-efficient FPGA implementations of the SHA-3 finalists. In Peter M. Athanas, Jürgen Becker, and René Cumplido, editors, *ReConFig*, pages 235–241. IEEE Computer Society, 2011.

[95] Bernhard Jungk and Steffen Reith. On FPGA-based implementations of Grøstl. Cryptology ePrint Archive, Report 2010/260, 2010. http://eprint.iacr.org/.

[96] Bernhard Jungk, Steffen Reith, and Jürgen Apfelbeck. On optimized FPGA implementations of the SHA-3 candidate Grøstl. Cryptology ePrint Archive, Report 2009/206, 2009. http://eprint.iacr.org/.

[97] Andrew B. Kahng, John Lach, William H. Mangione-Smith, Stefanus Mantik, Igor L. Markov, Miodrag Potkonjak, Paul Tucker, Huijuan Wang, and Gregory Wolfe. Watermarking techniques for intellectual property protection. In *DAC*, pages 776–781, 1998.

[98] Andrew B. Kahng, Stefanus Mantik, Igor L. Markov, Miodrag Potkonjak, Paul Tucker, Huijuan Wang, and Gregory Wolfe. Robust IP watermarking methodologies for physical design. In *DAC*, pages 782–787, 1998.

[99] K. Kahng and S.M. Sze. A floating gate and its application to memory devices. *Electron Devices, IEEE Transactions on*, 14(9):629–629, Sep 1967.

[100] Jens-Peter Kaps, Panasayya Yalla, Kishore Kumar Surapathi, Bilal Habib, Susheel Vadlamudi, and Smriti Gurung. Lightweight implementations of SHA-3 finalists on FPGAs. Third SHA-3 candidate conference, Mar 2012.

[101] Ferhat Karakoç, Hüseyin Demirci, and A. Emre Harmanci. Itubee: A software oriented lightweight block cipher. In Gildas Avoine and Orhun Kara, editors, *LightSec*, volume 8162 of *Lecture Notes in Computer Science*, pages 16–27. Springer, 2013.

[102] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[103] Stéphanie Kerckhof, François Durvaux, Cédric Hocquet, David Bol, and François-Xavier Standaert. Towards green cryptography: A comparison of lightweight ciphers from the energy viewpoint. In Prouff and Schaumont [156], pages 390–407.

[104] Stéphanie Kerckhof, François Durvaux, François-Xavier Standaert, and Benoît Gérard. Intellectual property protection for FPGA designs with soft physical hash functions: First experimental results. In *HOST*, pages 7–12. IEEE, 2013.

[105] Stéphanie Kerckhof, François Durvaux, Nicolas Veyrat-Charvillon, Francesco Regazzoni, Guerric Meurice de Dormale, and François-Xavier Standaert. Compact FPGA implementations of the five SHA-3 finalists. In Emmanuel Prouff, editor, *CARDIS*, volume 7079 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2011.

[106] Stéphanie Kerckhof, François-Xavier Standaert, and Eric Peeters. From new technologies to new solutions: Exploiting FRAM memories to enhance physical security. In *CARDIS*, Lecture Notes in Computer Science. Springer, 2013. To appear.

[107] Jack S. Kilby. Miniaturized electronic circuits, 06 1964.

[108] Lars R. Knudsen and Matthew Robshaw. *The Block Cipher Companion.* Information Security and Cryptography. Springer, 2011.

[109] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms.* Addison-Wesley Publishing, Boston, MA, USA, 1997.

[110] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Wiener [200], pages 388–397.

[111] P.C. Kocher. Leak-resistant cryptographic indexed key update, March 25 2003. US Patent 6,539,092.

[112] Xuejia Lai and James L. Massey. A proposal for a new block encryption standard. In *EUROCRYPT*, pages 389–404, 1990.

[113] Gregor Leander, Christof Paar, Axel Poschmann, and Kai Schramm. New lightweight DES variants. In Alex Biryukov, editor, *FSE*, volume 4593 of *LNCS*, pages 196–210. Springer, 2007.

[114] Jooyoung Lee and Je Hong Park. Preimage resistance of LPmkr with r=m-1. *Inf. Process. Lett.*, 110(14-15):602–608, 2010.

[115] Frédéric Lefèbvre, Jacek Czyz, and Benoit M. Macq. A robust soft hash algorithm for digital image signature. In *ICIP (2)*, pages 495–498, 2003.

[116] Chae Hoon Lim and Tymur Korkishko. mcrypton - a lightweight block cipher for security of low-cost RFID tags and sensors. In JooSeok Song, Taekyoung Kwon, and Moti Yung, editors, *WISA*, volume 3786 of *LNCS*, pages 243–258. Springer, 2005.

[117] Bernhard Linke. Xilinx FPGA IFF copy protection with 1-wire®SHA-1 secure memories. XAPP3826, July 21, 2006.

[118] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In *CRYPTO*, pages 31–46. Springer-Verlag, 2002.

[119] Stefan Mangard. A simple power-analysis (SPA) attack on implementations of the AES key expansion. In Pil Joong Lee and Chae Hoon Lim, editors, *ICISC*, volume 2587 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2002.

[120] Stefan Mangard. Hardware countermeasures against DPA – A statistical analysis of their effectiveness. In Tatsuaki Okamoto, editor, *CT-RSA*, volume 2964 of *Lecture Notes in Computer Science*, pages 222–235. Springer, 2004.

[121] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[122] Stefan Mangard, Elisabeth Oswald, and François-Xavier Standaert. One for all - all for one: unifying standard differential power analysis attacks. *IET Information Security*, 5(2):100–110, 2011.

[123] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In Menezes [128], pages 351–365.

[124] Stefan Mangard and François-Xavier Standaert, editors. *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *LNCS*. Springer, 2010.

[125] Mitsuru Matsui. New block encryption algorithm MISTY. In Eli Biham, editor, *FSE*, volume 1267 of *LNCS*, pages 54–68. Springer, 1997.

[126] David May, Henk L. Muller, and Nigel P. Smart. Non-deterministic processors. In Vijay Varadharajan and Yi Mu, editors, *ACISP*, volume 2119 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2001.

[127] Marcel Medwed, François-Xavier Standaert, Johann Großschädl, and Francesco Regazzoni. Fresh re-keying: Security against side-channel and fault attacks for low-cost devices. In Daniel J. Bernstein and Tanja Lange, editors, *AFRICACRYPT*, volume 6055 of *Lecture Notes in Computer Science*, pages 279–296. Springer, 2010.

[128] Alfred Menezes, editor. *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*. Springer, 2005.

[129] Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. A systematic evaluation of compact hardware implementations for the Rijndael S-Box. In Menezes [128], pages 323–333.

[130] Ralph C. Merkle. A certified digital signature. In Brassard [36], pages 218–238.

[131] Thomas S. Messerges. Using second-order power analysis to attack DPA resistant software. In Çetin Kaya Koç and Paar [39], pages 238–251.

[132] Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In Moni Naor, editor, *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2004.

[133] Vishal Monga and Brian L. Evans. Perceptual image hashing via feature points: Performance evaluation and tradeoffs. *IEEE Transactions on Image Processing*, 15(11):3452–3465, 2006.

[134] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[135] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from Xilinx Virtex-II FPGAs. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 111–124. ACM, 2011.

[136] Amir Moradi, Oliver Mischke, and Christof Paar. Practical evaluation of DPA countermeasures on reconfigurable hardware. In *HOST*, pages 154–160. IEEE Computer Society, 2011.

[137] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In Prouff and Schaumont [156], pages 58–75.

[138] Ginger Myles. Using software watermarking to discourage piracy. *ACM Crossroads*, 12(1):4, 2005.

[139] A. H. Namin and M. A. Hasan. Hardware implementation of the compression function for selected SHA-3 candidates. CACR 2009-28, 2009. http://www.vlsi.uwaterloo.ca/~ahasan/hasan_report.html.

[140] National Institute of Standards and Technology. Fips 180-3, secure hash standard. In *Federal Information Processing Standard (FIPS), Publication 180-3*, October 2008. Available online at http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf.

[141] National Institute of Standards and Technology. Data encryption standard. In *Federal Information Processing Standard (FIPS), Publication 46*, January 1977. Availble online at http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[142] National Institute of Standards and Technology. NIST special publication 800-57, recommendation for key management (revised), March 2007.

[143] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register Notices*, 72(212):62212–62220, November 2007. http://csrc.nist.gov/groups/ST/hash/index.html.

[144] Dag Arne Osvik. Fast embedded software hashing. Cryptology ePrint Archive, Report 2012/156, 2012. http://eprint.iacr.org/.

[145] Elisabeth Oswald and Pankaj Rohatgi, editors. *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154 of *Lecture Notes in Computer Science*. Springer, 2008.

[146] Daniel Otte. AVR-crypto-lib, 2009. http://www.das-labor.org/wiki/Crypto-avr-lib/en.

[147] Christof Paar and Jan Pelzl. *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2010.

[148] Bruno Paillard. An introduction to digital signal processors, 2002. Available online at http://dsp-book.narod.ru/InDSPrcs.pdf.

[149] Pascal Paillier and Ingrid Verbauwhede, editors. *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *LNCS*. Springer, 2007.

[150] B. Poettering. AVRAES: The AES block cipher on AVR controllers, 2007. Available online at http://point-at-infinity.org/avraes/.

[151] Thomas Popp and Stefan Mangard. Masked dual-rail pre-charge logic: Dpa-resistance without routing constraints. In Rao and Sunar [157], pages 172–186.

[152] Bart Preneel. MASH hash functions (modular arithmetic secure hash). In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security (2nd Ed.)*, page 761. Springer, 2011.

[153] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: A synthetic approach. In Douglas R. Stinson, editor, *CRYPTO*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1993.

[154] Bart Preneel and Tsuyoshi Takagi, editors. *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*. Springer, 2011.

[155] Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. Statistical analysis of second order differential power analysis. *IEEE Trans. Computers*, 58(6):799–811, 2009.

[156] Emmanuel Prouff and Patrick Schaumont, editors. *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*. Springer, 2012.

[157] Josyula R. Rao and Berk Sunar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *LNCS*. Springer, 2005.

[158] Denis Réal, Vivien Dubois, Anne-Marie Guilloux, Frédéric Valette, and M'hamed Drissi. SCARE of an unknown hardware Feistel implementation. In Gilles Grimaud and François-Xavier Standaert, editors, *CARDIS*, volume 5189 of *Lecture Notes in Computer Science*, pages 218–227. Springer, 2008.

[159] Mathieu Renauld and François-Xavier Standaert. Algebraic Side-Channel Attacks. In Feng Bao, Moti Yung, Dongdai Lin, and Jiwu Jing, editors, *Inscrypt*, volume 6151 of *Lecture Notes in Computer Science*, pages 393–410. Springer, 2009.

[160] Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. A formal study of power variability issues and side-channel attacks for nanoscale devices. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2011.

[161] Semico Research. Semiconductor intellectual property: The market hits its stride. http://www.design-reuse.com/news/11069/semico-research-report-semiconductor-intellectual-property-market-hits-stride.html.

[162] National Institute of Advances Industrial Science Research Center for Information Security and Technology. Side-channel attack standard evaluation board Sasebo-G - specification, October 2008. Available online at http://www.risec.aist.go.jp/project/sasebo/download/SASEBO-G_Spec_Ver1.0_English.pdf.

[163] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Mangard and Standaert [124], pages 413–427.

[164] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-order masking and shuffling for software implementations of block ciphers. In Clavier and Gaj [43], pages 171–188.

[165] Ronald L. Rivest. The md4 message digest algorithm. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1990.

[166] Phillip Rogaway and John P. Steinberger. Constructing cryptographic hash functions from fixed-key blockciphers. In Wagner [194], pages 433–450.

[167] G.A. Roland. Efficient implementation of the Grøstl-256 hash function on an ATmega163 microcontroller. Available at http://groestl.info/groestl-0-8bit.pdf, June 2009.

[168] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits. *Proceedings of the IEEE*, 91(2):305–327, 2003.

[169] Fumihiko Sano, Masanobu Koike, Shin ichi Kawamura, and Masue Shiba. Performance evaluation of AES finalists on the high-end smart card. In *AES Candidate Conference*, pages 82–93, 2000.

[170] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In Rao and Sunar [157], pages 30–46.

[171] A. D. Semenov, G. N. Gol'tsman, and R. Sobolewski. Hot-electron effect in superconductors and its applications for radiation sensors. *Superconductor Science & Technology*, 15:R1–R16, 2002.

[172] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal, Vol 28, pp. 656-715*, October 1949.

[173] Thomas Shrimpton and Martijn Stam. Building a collision-resistant compression function from non-compressing primitives. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 643–654. Springer, 2008.

[174] Eric Simpson and Patrick Schaumont. Offline hardware/software authentication for reconfigurable platforms. In Goubin and Matsui [79], pages 311–323.

[175] Michael John Sebastian Smith. *Application-specific integrated circuits*. Addison-Wesley, 1997.

[176] François-Xavier Standaert, Tal Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 443–461. Springer, 2009.

[177] François-Xavier Standaert, Olivier Pereira, and Yu Yu. Leakage-Resilient Symmetric Cryptography under Empirically Verifiable Assumptions. In Ran Canetti and Juan A. Garay, editors, *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 335–352. Springer, 2013.

[178] François-Xavier Standaert, Olivier Pereira, Yu Yu, Jean-Jacques Quisquater, Moti Yung, and Elisabeth Oswald. Leakage resilient cryptography in practice. In Ahmad-Reza Sadeghi and David Naccache, editors, *Towards Hardware-Intrinsic Security*, Information Security and Cryptography, pages 99–134. Springer, 2010.

[179] François-Xavier Standaert, Christophe Petit, and Nicolas Veyrat-Charvillon. Masking with Randomized Look Up Tables - Towards Preventing Side-Channel Attacks of All Orders. In David Naccache, editor, *Cryptography and Security*, volume 6805 of *Lecture Notes in Computer Science*, pages 283–299. Springer, 2012.

[180] François-Xavier Standaert, Gilles Piret, Neil Gershenfeld, and Jean-Jacques Quisquater. SEA: A scalable encryption algorithm for small embedded applications. In Josep Domingo-Ferrer, Joachim Posegga, and Daniel Schreckling, editors, *CARDIS*, volume 3928 of *LNCS*, pages 222–236. Springer, 2006.

[181] François-Xavier Standaert, Gilles Piret, Gaël Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. ICEBERG : An involutional cipher efficient for block encryption in reconfigurable hardware. In Bimal K. Roy and Willi Meier, editors, *FSE*, volume 3017 of *Lecture Notes in Computer Science*, pages 279–299. Springer, 2004.

[182] François-Xavier Standaert, Nicolas Veyrat-Charvillon, Elisabeth Oswald, Benedikt Gierlichs, Marcel Medwed, Markus Kasper, and Stefan Mangard. The world is not enough: Another look on second-order DPA. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 112–129. Springer, 2010.

[183] R.M. Swanson and J.D. Meindl. Ion-implanted complementary MOS transistors in low-voltage circuits. *Solid-State Circuits, IEEE Journal of*, 7(2):146–153, 1972.

[184] Stefan Tillich, Martin Feldhofer, Mario Kirschbaum, Thomas Plos, Jörn Marc-Schmidt, and Alexander Szekely. 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/.

[185] Stefan Tillich and Christoph Herbst. Attacking state-of-the-art software countermeasures-a case study for AES. In Oswald and Rohatgi [145], pages 228–243.

[186] Kris Tiri and Ingrid Verbauwhede. A logic level design methodology for a secure dpa resistant asic or fpga implementation. In *DATE*, pages 246–251. IEEE Computer Society, 2004.

[187] Randy Torrance and Dick James. The state-of-the-art in IC reverse engineering. In Clavier and Gaj [43], pages 363–381.

[188] Michael Tunstall and Olivier Benoît. Efficient use of random delays in embedded software. In Damien Sauveron, Constantinos Markantonakis, Angelos Bilas, and Jean-Jacques Quisquater, editors, *WISTP*, volume 4462 of *Lecture Notes in Computer Science*, pages 27–38. Springer, 2007.

[189] Pim Tuyls, Geert Jan Schrijen, Boris Skoric, Jan van Geloven, Nynke Verhaegh, and Rob Wolters. Read-proof hardware from protective coatings. In Goubin and Matsui [79], pages 369–383.

[190] H.J.M. Veendrick. *Nanometer CMOS ICs: From Basics to ASICs*. Springer Netherlands, 2008.

[191] Ramarathnam Venkatesan, S.-M. Koon, Mariusz H. Jakubowski, and Pierre Moulin. Robust image hashing. In *ICIP*, 2000.

[192] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In Wang and Sako [195], pages 740–757.

[193] John von Neumann. First draft of a report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4):27–75, October 1993.

[194] David Wagner, editor. *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*. Springer, 2008.

[195] Xiaoyun Wang and Kazue Sako, editors. *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*. Springer, 2012.

[196] Nicholas Weaver and John Wawrzynek. A comparison of the AES candidates amenability to FPGA implementation. In *AES Candidate Conference*, pages 28–39, 2000.

[197] Christian Wenzel-Benner, Jens Gräf, John Pham, and Jens-Peter Kaps. XBX benchmarking results january 2012. Third SHA-3 candidate conference, http://xbx.das-labor.org/trac/wiki/r2012platforms_atmega1284p_16mhz, Mar 2012.

[198] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[199] David J. Wheeler and Roger M. Needham. TEA, a tiny encryption algorithm. In Bart Preneel, editor, *FSE*, volume 1008 of *LNCS*, pages 363–366. Springer, 1994.

[200] Michael J. Wiener, editor. *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*. Springer, 1999.

[201] Wikimedia Commons. Transistor count and moore's law, 2008. Available online at http://commons.wikimedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2008.svg.

[202] Wikimedia Commons. Comparison semiconductor process nodes, 2012. Available online at http://en.wikipedia.org/wiki/File:Comparison_semiconductor_process_nodes.svg.

[203] Wikipedia. Field-programmable gate array. Available online at http://en.wikipedia.org/wiki/Field-programmable_gate_array.

[204] Thomas J. Wollinger, Jorge Guajardo, and Christof Paar. Security on FPGAs: State-of-the-art implementations and attacks. *ACM Trans. Embedded Comput. Syst.*, 3(3):534–574, 2004.

[205] Hongjun Wu. JH Documentation Website. http://www3.ntu.edu.sg/home/wuhj/research/jh/.

[206] Hongjun Wu. The hash function jh. Submission to NIST (round 3), 2011.

[207] Xilinx. Vivado ESL design. Available online at http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/.

[208] Xilinx. What is an FPGA? Available online at http://www.xilinx.com/fpga/index.htm.

[209] Xilinx. *Spartan-6 FPGA Configurable Logic Block - User Guide*, v1.1 edition, February 2010. Available online at http://www.xilinx.com/support/documentation/user_guides/ug384.pdf.

[210] Xilinx. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, v5.0 edition, June 2011. Available online at http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf.

[211] Xilinx. *Virtex-6 FPGA Configurable Logic Block - User Guide*, v1.2 edition, February 2012. Available online at http://www.xilinx.com/support/documentation/user_guides/ug364.pdf.

[212] Bob Zeidman. All about FPGAs. Available online at http://www.eetimes.com/document.asp?doc_id=1274496.

[213] Weixiong Zhang. *State-space search - algorithms, complexity, extensions, and applications.* Springer, 1999.

[214] Daniel Ziener and Jürgen Teich. Power signature watermarking of IP cores for FPGAs. *Signal Processing Systems*, 51(1):123–136, 2008.