PhD-FSTC-2017-77
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defence held on 29/11/2017 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Dumitru-Daniel DINU

EFFICIENT AND SECURE IMPLEMENTATIONS OF LIGHTWEIGHT SYMMETRIC CRYPTOGRAPHIC PRIMITIVES

Dissertation defence committee

Abstract

This thesis is devoted to efficient and secure implementations of lightweight symmetric cryptographic primitives for resource-constrained devices such as wireless sensors and actuators that are typically deployed in remote locations. In this setting, cryptographic algorithms must consume few computational resources and withstand a large variety of attacks, including side-channel attacks.

The first part of this thesis is concerned with efficient software implementations of lightweight symmetric algorithms on 8, 16, and 32-bit microcontrollers. A first contribution of this part is the development of FELICS, an open-source benchmarking framework that facilitates the extraction of comparative performance figures from implementations of lightweight ciphers. Using FELICS, we conducted a fair evaluation of the implementation properties of 19 lightweight block ciphers in the context of two different usage scenarios, which are representatives for common security services in the Internet of Things (IoT). This study gives new insights into the link between the structure of a cryptographic algorithm and the performance it can achieve on embedded microcontrollers. Then, we present the SPARX family of lightweight ciphers and describe the impact of software efficiency in the process of shaping three instances of the family. Finally, we evaluate the cost of the main building blocks of symmetric algorithms to determine which are the most efficient ones. The contributions of this part are particularly valuable for designers of lightweight ciphers, software and security engineers, as well as standardization organizations.

In the second part of this work, we focus on side-channel attacks that exploit the power consumption or the electromagnetic emanations of embedded devices executing unprotected implementations of lightweight algorithms. First, we evaluate different selection functions in the context of Correlation Power Analysis (CPA) to infer which operations are easy to attack. Second, we show that most implementations of the AES present in popular open-source cryptographic libraries are vulnerable to side-channel attacks such as CPA, even in a network protocol scenario where the attacker has limited control of the input. Moreover, we describe an optimal algorithm for recovery of the master key using CPA attacks. Third, we perform the first electromagnetic vulnerability analysis of Thread, a networking stack designed to facilitate secure communication between IoT devices.

The third part of this thesis lies in the area of side-channel countermeasures against power and electromagnetic analysis attacks. We study efficient and secure expressions that compute simple bitwise functions on Boolean shares. To this end, we describe an algorithm for efficient search of expressions that have an optimal cost in number of elementary operations. Then, we introduce optimal expressions for first-order Boolean masking of bitwise AND and OR operations. Finally, we analyze the performance of three lightweight block ciphers protected using the optimal expressions.

To my family.

Acknowledgements

This doctoral thesis would not exist without the support, help, and encouragement of many people. It is a pleasure for me to express my gratitude to all of them.

I am deeply thankful to my supervisor, Alex Biryukov. He gave me the opportunity to pursue a Ph.D. in his research group and guided me with patience through my studies. I highly appreciate the freedom and flexibility that Alex offered to me in addition to his wise advice.

I am very grateful to Johann Großschädl for his enthusiastic support, the interesting discussions we had, and the wonderful time I spent working with him on interesting research problems. Johann introduced me to the design of masking countermeasures and taught me how to write papers. I also thank him for proofreading this thesis and suggesting many corrections.

I am also grateful to Ilya Kizhvatov, who was my mentor during an exciting internship at NXP Semiconductors. He provided me an industrial view on side-channel attacks and generously shared his expertise with me. I enjoyed very much our collaboration and the interaction with the highly experienced researchers and engineers from NXP Semiconductors.

I warmly thank Yann Le Corre and André Stemper for the endless engineering discussions we had and their technical assistance. I also owe gratitude to Fabienne Schmitz, Isabelle Schroeder, and Natalie Kirf for their help with many administrative tasks.

I am lucky to have had the opportunity to write papers together with outstanding researchers: Alex Biryukov, Johann Großschädl, Ilya Kizhvatov, Dmitry Khovratovich, Yann Le Corre, Léo Perrin, Aleksei Udovenko, and Vesselin Velichkov.

I would like to thank Benedikt Gierlichs, Volker Müller, Daniel Page, and Peter Ryan for serving on my defence committee and providing valuable comments on my thesis.

Many thanks to my current and former colleagues from the University of Luxembourg for creating a good working environment: Patrick Derbez, Daniel Fehrer, Johann Großschädl, Dmitry Khovratovich, Yann Le Corre, Zhe Liu, Léo Perrin, Ivan Pustogarov, Sergei Tikhomirov, Aleksei Udovenko, Praveen Vadnala, Vesselin Velichkov, and Srinivas Venkatesh.

I am thankful to all my friends from Luxembourg for the nice time and good memories. I am grateful to my family for their unconditional support and encouragement through all these years.

The work described in this thesis would not have been possible without the funding from the Fonds National de la Recherche (FNR) through the CORE project ACRYPT (ID C12-15-4009992).

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

The Latin expression *"sapientia est potentia"*[1], which is often translated as *"knowledge is power"*, captures the great importance of knowledge for humankind. At the same time, it shows why people became interested in protecting information [2], which is one of the main ingredients of knowledge.

Cryptology (from Greek kryptós – hidden and lógos – word) is the science that studies communication and storage of data in secure and usually secret form [324]. It encompasses two highly related areas: cryptography (from the Greek gráphein – to write) and cryptanalysis (from Greek analýein – to loosen or to untie). Cryptography uses mathematical techniques to provide information security, such as confidentiality, data integrity, entity authentication, and data origin authentication. The large field of cryptography also includes algorithms and protocols designed for electronic transactions and elections or cryptocurrencies. On the other hand, cryptanalysis aims at finding weaknesses in cryptographic algorithms.

Contents

[1] The expression appeared for the first time in this form in [162].

[2] RFC 2828 [322] defines *information* as "facts and ideas, which can be represented (encoded) as various forms of data" .

1.1 From Classical to Modern Cryptography

Historically, cryptography was more of an art than science. The first forms of secret writing date to ancient Egypt about 4000 years B.C. [181, 385]. At that time, scribes were communicating by written messages using hieroglyphs. The art of secret writing using hieroglyphs was transmitted from father to son and was broken only several millenia later by Champollion [371].

The ancient Greeks used a cylinder with a strip of parchment wound around it to write secret messages. This tool (the *scytale*), invented around 500 B.C. by Spartans, represents the first *transposition cipher*. To recover the original message, one had to wrap the parchment around a rod of the same diameter and read along the axis. This cipher was used to communicate during military campaigns.

Later, the Roman emperor Julius Caesar used another technique (the *Caesar cipher*) to protect his private correspondence. It consists in replacing every letter of a message by the letter which comes three positions later in the alphabet. The initial message can be recovered by performing the inverse transformation. The Caesar cipher is regarded as the first recorded use of a *substitution cipher* [325].

The ROT13 substitution is a special case of the Caesar cipher that is typically used to hide offensive jokes or to obscure an answer to a puzzle [310]. It replaces a letter of the basic Latin alphabet, which consists of 26 letters, with the letter situated 13 positions after it in the alphabet and thus it is its own inverse.

Modern cryptography started to develop in the first half of the 20th century [183]. A famous example from that time is the *Enigma machine* invented by the German engineer Arthur Scherbius at the end of World War I and patented in 1928 [19]. Several different Enigma models were produced. Enigma I, also known as the Wehrmacht, was extensively used by German military services before and during World War II. Under the leadership of mathematician Marian Rejewski, the Poles reverse-engineered Enigma and found the first attack against it. Their efforts were continued by a British team led by mathematician Alan Turing who created the *bombe*, an electromechanical device designed to discover the daily settings of the Enigma machines used by the German troops.

In the years immediately following World War II, both cryptography and cryptanalysis continued to develop at a rapid pace, driven by several significant technical innovations such as the invention of the *von Neumann architecture* in 1945 [374] and of the *transistor* in 1947 [66]. Some other notable milestones in the history of modern cryptography are the seminal work of Claude Shannon that established the mathematical basis of information theory in 1948 [319, 320], the first public description of *public-key cryptography* by Diffie and Hellman in 1976 [101], and the publication of the Data Encryption Standard (DES) in 1977 [360], which is based on a slightly modified version of an earlier design of cryptographer Horst Feistel. As a consequence of all breakthroughs and findings in this area, a rich theory emerged that enabled a rigorous study of cryptology (cryptography and cryptanalysis) as a science.

The advent of the first *personal computer* in 1981 [257] and the rise of the *Internet* in the early 1990s set the stage for mass usage of cryptography. Nowadays, cryptography plays a crucial role in our lives since it is at the heart of computer and communication security. Without doubt, the expansion of the *Internet of Things (IoT)* envisioned for the coming years will augment even more the importance of cryptography.

While the first decades of modern cryptography were dominated by military applications, in the last decades cryptography permeated numerous domains such as commerce, banking, industry, and health care to become an essential component of many secure systems. The history of cryptology is a fascinating story in itself, but not the scope of this thesis. A good reading on the history of cryptology is the monograph of Kahn [181].

1.2 Modern Cryptography

1.2.1 Security Services

Cryptography uses a set of techniques to provide information security. A *security service* is a specific security goal that can be achieved by using cryptography. The primary objective of using cryptography is to provide the following four fundamental information security services: *confidentiality*, *data integrity*, *authentication*, and *non-repudiation*. They form a framework upon which other security services, such as *access control*, *anonymity*, or *digital signatures*, can be derived [238].

- **Confidentiality.** This security service ensures that only those authorized have access to the content of the information. Hence, it prevents an unauthorized user to access the content of the protected information. It is sometimes referred to as *secrecy*.
- **Data integrity.** Data integrity provides a mean to detect whether data has been manipulated by an unauthorized party since the last time an authorized user created, stored or transmitted it. Data manipulation refers to operations such as insertion, deletion, or substitution.
- **Authentication.** Authentication is related to identification and it is often divided into two classes: *data origin authentication* and *entity authentication*.
  - **Data origin authentication.** It gives assurance that an entity is the original source of a message. Data origin authentication implicitly provides data integrity. Sometimes, it is referred to as *message authentication*.
  - **Entity authentication.** Entity authentication assures one entity about the identity of a second entity with which it is interacting. Usually, entity authentication implies data origin authentication.
- **Non-repudiation.** Non-repudiation is a security service that prevents an entity from denying a previous action or commitment. It is very useful in situations that can lead to disputes. When a dispute arises, a trusted third party is able to provide the evidence required to settle it.

1.2.2 Kerckhoffs' Principle

The Dutch cryptographer Auguste Kerckhoffs formulated six design principles that a cipher has to satisfy [185]. The second principle is the most famous of them and states the following:

> *The security of a cryptographic system must depend only on the secrecy of the key, and not on the secrecy of the algorithm.*

There are many good arguments in favour of Kerchhoffs' principle [125, 183]. The first two arguments stem from the fact that algorithms are built in hardware or software which makes them susceptible to reverse-engineering and hard to replace. First, it is more difficult to keep the secrecy of an algorithm than to maintain the secrecy of a simple key. Second, it is more straightforward to replace an exposed key than a leaked algorithm. Third, it is much easier to share the same algorithm and use different keys to securely communicate with various entities than to use a different algorithm for each party. Finally, there are good reasons why algorithms should be published. While it is very difficult to design a good cryptographic algorithm, it is very easy to make a mistake that weakens it. Therefore, an algorithm is likely to be stronger if it has been extensively studied and no weaknesses have been found.

However, the simple fact that an algorithm is public does not imply that it is secure. It can be the case that the algorithm was not studied enough or it was already broken. The first case can be exemplified by the PC1 stream cipher published in 1997 and broken in 2012 [52] when it was already part of the DRM system of the MOBI e-book format, which was supported by the Amazon Kindle and by the free software MobiPocket. An expressive example of the latter case is the surprising use of the `ROT13` substitution, described in Section 1.1, by the eBook vendor New Paradigm Research Group to protect its documents (at least) until 2001 when this finding was presented at a hacking conference [326]. Windows XP used the same algorithm on some of its registry keys [339].

Some famous examples of proprietary algorithms that were firstly leaked or reverse-engineered and then broken are RC4 [254, 10], DST [61], KeeLoq [70, 171, 120, 4], and Megamos [372]. The last three algorithms were used for car immobilizer transponders. A detailed list of such algorithms can be found in [53].

1.2.3 Adversarial Models

In the context of encryption algorithms, the primary goal of a cryptanalyst is to break an algorithm using an *attack*. By attack we understand any technique that provides some information about the decryption key (*key recovery attack*) or the plaintext (*decryption attack*). There are many types of attacks, each with its own strengths and weaknesses. Depending on the information a cryptanalyst has in addition to the description of the cipher under analysis, we distinguish between four main categories of attacks [183, 371].

- **Ciphertext-only attack.** The only piece of information the attacker has is a set of ciphertexts produced using the same encryption key. This is the most difficult type of attack since the attacker has the least amount of information.

- **Known-plaintext attack.** In this setting, the attacker knows some plain-text-ciphertext pairs. In practice, there are many situations in which the attacker gets to know the plaintext associated to a ciphertext (e.g. after a confidential document is made public).

- **Chosen-plaintext attack.** The attacker can choose the plaintext to be encrypted and therefore she gets the corresponding ciphertext. This is a more powerful type of attack than a known-plaintext attack.

- **Chosen-ciphertext attack.** In addition to a chosen-plaintext attack, the adversary can choose arbitrary ciphertexts and she gets the corresponding plaintexts decrypted from it. Hence, this attack is more powerful than a chosen-plaintext attack.

The above-mentioned attacks belong to the class of *cryptanalytic attacks*. Another important class of attacks is represented by the so-called *generic attacks*. A generic attack is an attack that can be applied to a wide range of cryptographic algorithms without knowing the details of the attacked primitive. Examples of generic attacks are *brute-force attacks* and *dictionary attacks*. A brute-force attack consists in an exhaustive search of the correct key. The attacker encrypts a plaintext under a guessed key and checks if the resulting ciphertext matches the known ciphertext. The process is repeated until the correct key is found. Brute-force attacks are impractical given sufficiently large key sizes. A third class of attacks covers *physical attacks* that exploit implementation aspects of cryptographic algorithms. These attacks are described in detail in Section 1.5.

1.3 Cryptographic Toolkit

Cryptography can be divided into two main branches: *symmetric cryptography* and *asymmetric cryptography*. The difference between the two branches hinges on the relationship between the keys used to perform different operations. In general, symmetric cryptosystems use the same key or closely related keys for encryption and decryption. Symmetric cryptography also studies algorithms that do not require any key at all, such as hash functions. On the contrary, in asymmetric cryptosystems the encryption key is fundamentally different from the decryption key [226].

Symmetric and asymmetric algorithms have various advantages and disadvantages, but they complement each other well. Cryptographic systems commonly use algorithms from both branches to provide various security services such as those described in Section 1.2.1. A brief comparison of symmetric and asymmetric cryptography is given in Section 1.3.3.

1.3.1 Symmetric Cryptography

Symmetric cryptography is also referred to as *secret key cryptography* because the same key is used to both encrypt and decrypt the data. Secret key cryptography is the oldest form of cryptography. The algorithms studied by this branch of cryptography can be divided into five categories: *block ciphers*, *stream ciphers*, *hash functions*, *message authentication codes*, and *authenticated ciphers*.

1.3.1.1 Block Ciphers

A block cipher is a bijective function that maps a plaintext block of n bits to a ciphertext block of n bits using a key of k bits. Usually a block cipher is built by iterating a round function that depends on the secret key using a well-known structure such as a Substitution-Permutation network (SPN), a Feistel network (FN), or a Lai-Massey structure.

In practice, a block cipher is used in a *mode of operation* to encrypt plaintexts of arbitrary size. The most important modes of operation are briefly described next.

- **Cipher Block Chaining (CBC).** In this mode of operation, each plaintext block is XORed with the previous ciphertext block and then it is encrypted. Consequently, each cipher block depends on all plaintext blocks previously processed [117]. The first plaintext block is XORed with an *initialization vector (IV)*. Although CBC is sequential and requires padding of the message to a multiple of the cipher block size, it is one of the most widely used modes of operation.

- **Counter (CTR).** This mode of operation turns a block cipher into a stream cipher. A counter is encrypted by the block cipher and the result is XORed with the plaintext block to obtain the ciphertext block [102]. The counter can be any function which produces a sequence that is guaranteed not to repeat for a long time. A simple and popular method is to increment the counter value by one for each new block.

- **Counter with CBC-MAC (CCM).** Designed to provide both authentication and confidentiality, CCM supports only block ciphers with a block length of 128 bits [380, 362]. It combines the CBC-MAC (i.e. CBC mode used to generate a message authentication code) with CTR mode of encryption: CBC-MAC is first computed on the message to obtain a tag which is then encrypted together with the message using counter mode. A minor variation of the CCM, called CCM*, includes all of the features of CCM and additionally offers encryption-only and integrity-only capabilities.

- **Galois/Counter Mode (GCM).** GCM is an authenticated encryption algorithm that combines the counter mode of encryption with the Galois mode of authentication [363, 234]. It can reach high throughput thanks to its structure that allows parallel processing and efficient use of pipelining. Namely, the CTR mode and the Galois field multiplication used for authentication can be easily computed in parallel.

Block ciphers are versatile primitives and thus they can be used to build a large variety of cryptographic algorithms such as stream ciphers, hash functions, message authentication codes, authenticated ciphers, or pseudo-random number generators. The most widely used block cipher is the Advanced Encryption Standard (AES) [361], which is based on the Rijndael algorithm [98].

1.3.1.2 Stream Ciphers

Stream ciphers are inspired by the *one-time pad (OTP)* cipher, which encrypts every plaintext with a different key. OTP provides perfect secrecy if the keys used are fully random [320]. However, in practice it is difficult to share large key streams. To overcome this issue, a stream cipher generates a pseudo-random stream of bits from a short secret key. There are two types of stream ciphers: *synchronous stream ciphers* and *self-synchronizing stream ciphers*. A self-synchronizing stream cipher generates a key stream that depends on some previous ciphertext digits, while a synchronous stream cipher generates a key stream that is independent of the plaintext and ciphertext messages. Trivium is a synchronous stream cipher selected as part of the portfolio for low area hardware ciphers by the eSTREAM project and standardized by ISO/IEC [173].

1.3.1.3 Hash Functions

A hash function transforms an arbitrarily long input into a fixed length *digest*. This construction is secure if it satisfies three conditions: *collision resistance*, *preimage resistance*, and *second preimage resistance*. Collision resistance is achieved if it is computationally impossible to build two messages that hash to the same value. Preimage resistance means that it is computationally infeasible to reverse a hash function (i.e. to find a message that hashes to a given digest). Finally, second preimage resistance requires that given an input and its digest it is hard to find a different input with the same digest. Secure Hash Algorithm 2 (SHA-2) [364] and Secure Hash Algorithm 3 (SHA-3) [366] are members of the Secure Hash Algorithm family of standards, released by NIST. SHA-2 was designed by the United States National Security Agency (NSA), while SHA-3 is a subset of the broader cryptographic primitive family Keccak [46].

1.3.1.4 Message Authentication Codes

A message authentication code (MAC) or a *keyed hash function* is an algorithm that takes a key and an arbitrarily long message and produces a fixed size *tag* whose purpose is to provide message authentication. A MAC should have the *forgery resistance* property, namely, that it is computationally infeasible for an attacker to find a message and tag pair without knowing the secret key. A common mechanism for message authentication using cryptographic hash functions is HMAC [204].

1.3.1.5 Authenticated Ciphers

Authenticated ciphers are designed to simultaneously provide confidentiality, integrity, and authenticity of data. The process of using an authenticated cipher is sometimes called *authenticated encryption (AE)* or *authenticated encryption with associated data (AEAD)*. Authenticated ciphers are motivated by the fact that combining a confidentiality mode with an authentication mode in a secure way can be difficult and error prone. Examples of authenticated encryption modes based on block ciphers include CCM and GCM, which are standardized by ISO/IEC [172].

1.3.2 Asymmetric Cryptography

Asymmetric cryptography or *public-key cryptography* studies any cryptographic system that involves a pair of keys: a *public key* and a *private key*. In order to use a public-key cryptosystem one needs to generate a pair of mathematically-related keys such that it is computationally infeasible to determine the private key from the public key. The private key is kept secret, while the public key is distributed to other entities.

Public-key cryptosystems are based on mathematical problems that currently admit no efficient solution: integer factorization (RSA [297]), discrete logarithms (ElGamal [134], digital signature algorithm – DSA [365]), or discrete logarithms on elliptic curves (elliptic curve cryptography – ECC [365]). Three well-known applications of asymmetric cryptography are *public-key encryption*, *digital signatures*, and *public-key infrastructure (PKI)*.

1.3.2.1 Public-Key Encryption

Public-key encryption is realized using the public key of the recipient. Anyone can encrypt a message using this public key and the corresponding public-key encryption algorithm. Only the owner of the matching private key can decrypt the ciphertext. Public-key algorithms are typically used to encrypt small messages due to the their computational complexity. To protect a long message, one can use a public-key cryptosystem to encrypt only a secret key which is then used to encrypt the long message using a symmetric cipher.

1.3.2.2 Digital Signatures

A digital signature can be obtained by encrypting a message under the sender's private key. Typically, the message is hashed before being signed. Anyone who has the sender's public key can verify if the signature is valid or not. Besides the sender's public key, the verification function takes a signature and a message. It checks if the signature was generated from the given message using the private key of the sender. The verification function ensures that the message was not tampered with.

1.3.2.3 Public Key Infrastructure (PKI)

A public key infrastructure is a system in which the ownership of a key pair is certified by a trusted third party through a *public-key certificate*. A public-key certificate includes information about the public key, the identity of its owner, and the validity period. This information is signed by a *certification authority (CA)*, which issues, stores and revokes public-key certificates. Each participant in a PKI has to get a public-key certificate from the CA. Moreover, each participant has to know the CA's public key to be able to verify the certificates of other participants.

1.3.3 Symmetric vs. Asymmetric Cryptography

Current cryptographic systems exploit the strengths of both symmetric-key and public-key cryptography. Symmetric encryption is preferred when confidentiality is required because it is faster than public-key encryption. Moreover, symmetric algorithms usually use smaller keys than public-key algorithms. On the other hand, public-key cryptography is used to establish secure communication channels and to provide non-repudiation.

To preserve the security of a symmetric system, the key must be kept secret at both ends. Furthermore, a different key must be shared between each two entities.

Consequently, the key management might become cumbersome when communicating with a large number of entities. In contrast, in public-key cryptosystems only the private key must be secret.

Sound cryptographic practice requires that symmetric keys have to be changed more frequently than the key pairs used for public-key cryptography, which can remain unchanged for considerable periods of time. While symmetric keys can be randomly generated, keys used in a public-key system have a special structure and are usually more expensive to generate.

An overview of the main security services (see Section 1.2.1) that can be achieved using symmetric and asymmetric cryptography is given in Table 1.1. Sometimes a primitive can not provide a security service on its own, but when it is used in a mode of operation or in combination with another primitive.

Primitive	Security service			
	Confidentiality	Integrity	Authentication	Non-repudiation
Block cipher	●	◑	◑	○
Stream cipher	●	◑	◑	○
Hash function	○	◑	◑	○
MAC	○	●	◑	○
Authenticated cipher	●	●	●	○
Public-key encryption	●	◑	●	○
Digital signature	○	●	●	●

● – using only the primitive
◑ – using the primitive in a mode of operation or combined with other primitives
○ – not possible

Table 1.1: Security services provided by different symmetric and asymmetric cryptographic primitives.

1.4 Implementations

Modern cryptographic algorithms are designed to work on a binary representation of information and thus they are fairly useless on their own. They have to be implemented in actual devices that process information in the same binary representation. Usually, the implementations of cryptographic algorithms are divided into *hardware implementations* and *software implementations*. Though, a cryptographic algorithm can also be implemented using a *hardware/software codesign* which tries to exploit the synergy of hardware and software. The goal of such an implementation is to satisfy stringent design constraints such as cost or performance, while reducing the time to market [346, 306].

Any cryptographic algorithm can be implemented in software or can be built directly in hardware. The choice between hardware and software is determined by various factors such as the requirements and constraints of specific use cases, flexibility, cost, or time to market. The same criteria are considered when selecting the chip technology for hardware implementations or the programming language for software implementations.

1.4.1 Hardware Implementations

A hardware implementation describes the structure of an integrated circuit. An integrated circuit is the result of a *hardware design flow* that starts with the specification of the circuit in a *hardware description language (HDL)* such as VHDL or Verilog. A digital circuit can be implemented on a chip using a *Field Programmable Array (FPGA)* or an *Application-Specific Integrated Circuit (ASIC)*. A detailed comparison between the two technologies is given in [207].

- **Field Programmable Gate Arrays (FPGAs).** An FPGA is an integrated circuit designed to be configured using a HDL. It consists of an array of programmable logic blocks, memory elements, and routing channels. The logic blocks can be used as simple logic gates or configured to perform complex combinational functions [69].

- **Application-Specific Integrated Circuits (ASICs).** An ASIC is an integrated circuit customized for a specific use or application. Usually ASIC implementations are faster and smaller than FPGA implementations, but they are more expensive to design. However, the final cost of an ASIC decreases as more units are manufactured [329].

1.4.2 Software Implementations

A software implementation is a program that can be executed on a general-purpose processor. Nowadays, software engineers can choose the most suitable *programming language* from a rich variety of languages to create programs that implement specific algorithms. However, there are few programming languages that can be used to write code for embedded devices. Typically, the development toolchains for microcontrollers support several languages, of which the most common are *C* (high-level language) and *assembly* (low-level language).

- **C language.** C is a language that supports cross-platform programming. It is used to write various software for devices ranging from supercomputers to embedded systems. Code written in C benefits from low-level access to memory through pointers and, in general, it is efficiently mapped to machine instructions.

- **Assembly language.** An assembly (or assembler) language is strongly tied to the machine instructions of a particular architecture. Hence, an assembly language is specific to a particular architecture. Assembly code has niche uses such as for operations that can not be easily implemented or are not well optimized by a compiler.

1.4.3 Hardware vs. Software Implementations

There are some major differences between hardware and software implementations. While the development cost of software implementations is almost flat and relatively low, the development cost of hardware implementations tends to increase towards the end of the development cycle. Hence, hardware implementations, especially ASICs, are cost-effective in large volumes. Hardware has to be well implemented and debugged before being shipped to the customer. In other words, it has to be free of errors or bugs because it is difficult and expensive to change deployed hardware. In contrast, software is more flexible and usually can be updated after shipment to the customer. Unlike hardware implementations, software implementations can evolve through multiple releases and new features can be added at any time. However, hardware implementations can be faster than software implementations.

1.4.4 Implementation Efficiency

A first step towards building secure systems is to implement a set of cryptographic algorithms that provide the main security services (see Section 1.2.1) necessary to achieve the desired security properties. Typically, implementations of cryptographic algorithms need to meet some specific requirements imposed by the applications of the system. For example, an implementation for a real-time system must be able to encrypt data within a clearly defined time frame. Such application-specific requirements are precisely formulated and assessed using the appropriate metrics.

The term *efficiency* characterizes a process done well with minimum resources. Hence, we say an implementation is efficient when it does not waste the resources of the system on which it is executed. Writing an efficient implementation of a cryptographic algorithm is not straightforward. It requires a good understanding of the algorithm to be implemented as well as of the hardware or software architecture which will run the implementation. Therefore, the skills required for an efficient implementation of a cryptographic algorithm fall into the area of cryptographic engineering.

A description of meaningful metrics for efficiency of hardware implementations is given in Section 1.4.5. Similarly, Section 1.4.6 presents metrics used to assess the efficiency of software implementations.

1.4.5 Metrics for Hardware Implementations

There are various metrics that can be used to assess the efficiency of a hardware implementation. These metrics usually depend on the fabrication technology and the standard cell library. The most common ones are:

- **Area.** Area measures the size of an integrated circuit in μm^2 or *gate equivalents (GEs)*. A GE is a unit of measure that allows one to quantify the area complexity of a circuit independently of the manufacturing technology. Usually a GE is equal to the area of a two-input NAND gate.

- **Latency.** Latency (or *execution time*) gives the time required to perform an operation. It is measured in *clock cycles* or *seconds* (often ms). The amount of time is obtained by dividing the number of cycles by the operating frequency.
- **Throughput.** Throughput is the maximum rate at which an operation is performed. It is usually expressed in *bits per second (bps)*, that is, the number of output bits divided by the time required to generate those output bits.
- **Power consumption.** This metric quantifies the electric power, usually in mW, consumed by an integrated circuit to perform an operation. Typically, power consumption values can be based on estimations or simulations at the gate or transistor level provided by the hardware design tools.
- **Energy consumption.** Energy consumption of an integrated circuit, typically expressed in mJ, is equal to the product of the electric charge transferred through the circuit and the supply voltage. It can be determined experimentally by integrating the current across a shunt resistor inserted between the `Vdd` pin and the power supply. Energy per bit is obtained by dividing the energy consumption by the number of output bits.
- **Figure of merit.** There are various ways to compare the performance or efficiency of implementations using metrics such as *hardware efficiency* defined as throughput to area ratio [59, 277], *figure of merit (FOM)* defined as throughput divided by the area squared [24], or *figure of adversarial merit (FOAM)* [186] which combines the inherent security provided by cryptographic structures and components with their hardware implementation properties.

1.4.6 Metrics for Software Implementations

A metric is a way to determine the degree to which an implementation possesses some property. Hence, there is a considerable overlap between the metrics used to measure hardware and software implementations since both have some similar characteristics. The metrics used to weight software implementations depend on the target architecture or the compiler/assembler used to generate the binary code.

- **Code size.** While the size of a hardware implementation is determined by its area, the size of a software implementation is quantified by its code size. Hence, the code size measures the amount of *bytes* required to store the binary code in the non-volatile memory (e.g. flash memory, ROM) of a device.
- **Execution time.** This metric has exactly the same essence as execution time of a hardware implementation. While number of *clock cycles* is independent of the operating frequency, the actual time in *seconds* depends on the frequency of the clock signal.
- **RAM consumption.** RAM consumption gives the amount of run-time memory (in *bytes*) required for the execution of a software implementation on

a processor. RAM is a form of volatile memory that can be used to store the data of a program and its execution stack.

- **Throughput.** Throughput is a common metric for both hardware and software implementations. Typically, an algorithm can reach higher throughput rates when implemented in hardware than when it is implemented software.

- **Power consumption.** This metric gives the power consumed for an operation. Modern processors and microcontrollers have several power modes that can be used to optimize the power consumption. In contrast to some hardware design tools that can provide power consumption estimations, software toolchains usually do not have this feature.

- **Energy consumption.** Although there is a direct link between energy and power, namely power is the energy consumed per unit time, energy is a primary concern for battery operated devices since it directly influences the lifetime of the battery [114, 31].

- **Figure of merit.** The authors of [118] used a *combined metric* defined as the product of code size and execution time normalized by the block size to asses the performance of block ciphers on an 8-bit microcontroller. Two similar metrics were used to summarize the efficiency of several hash functions: the product of code size and execution time and the product of RAM consumption and execution time [26].

1.4.7 Optimization Strategies

An implementation can be optimized to make more efficient use of available resources (e.g. energy) or to achieve better results for a certain metric (e.g. throughput). Typically, an optimization requires a trade-off between different objectives of an implementation. Two common optimization techniques that can be applied to both hardware and software implementations of symmetric cryptographic algorithms are *loop unrolling* and *pipelining*.

- **Loop unrolling.** Loop unrolling is an optimization technique that can be used to reduce the execution time of an implementation at the expense of an increase in area (hardware implementation) or code size (software implementation). It replicates the loop body and adjusts the loop iteration counter accordingly.

- **Pipelining.** Pipelining consists in overlapping the execution of different operations with the aim of increasing the throughput. The circuitry is usually divided into stages and each stage performs a particular operation. Hence, pipelining takes advantage of those operations that can be executed concurrently at distinct stages of a pipeline.

There are many other optimization strategies besides the above-mentioned ones. Therefore, efficient implementations demand skilled cryptographic engineers that are able to leverage the appropriate optimization techniques necessary to meet the requirements of different use cases. Typical trade-offs for hardware implementations of symmetric cryptography are discussed in [194]. A good description of common optimization techniques for software implementations can be found in [269].

1.5 Implementation Attacks

In general, a straightforward implementation of a cryptographic algorithm is vulnerable to various types of attacks referred to as *implementation attacks* or *physical attacks*. In contrast to cryptanalytic attacks that focus on breaking a cryptographic algorithm in a *black box model*, implementation attacks work in the so-called *gray box model*. In the black box model, the attacker uses the specifications of a cryptographic algorithm (see Section 1.2.2) and some pairs of plaintexts and ciphertexts to recover the key. The gray box model assumes a more powerful attacker that has access to some information about the internal state of the algorithm in addition to the knowledge of an attacker in the black box model.

Essentially, an attack against an implementation of a cryptographic algorithm takes advantage of the physical characteristics of the device that executes the implementation. Attackers can simply observe some physical phenomena or they can manipulate some physical parameters to cause a response from the device. Physical attacks are usually much more powerful than classical cryptanalytic attacks (see Section 1.2.3). A graphical representation of the two attack models is shown in Figure 1.1.

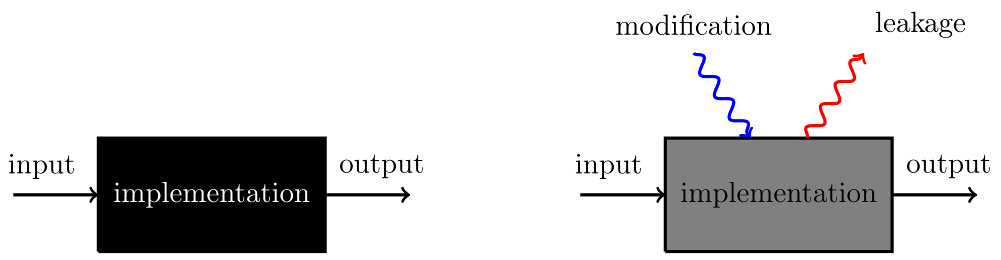


Figure 1.1: Comparison between the black box (left) and gray box (right) security models.

1.5.1 Short History

An early example of an attack that exploited the emanations of a communication equipment occurred in 1914, during World War I. At that time, the German army successfully eavesdropped on the voice communication of its enemy using the ground current of the phone lines. To reduce the weight of cable drums that the troops had to carry, the field phones were connected with a single insulated wire and the ground was used for the return circuit. This allowed the Germans to pick up the resulting voltage drop from the other side of the trenches with valve amplifiers connected to so-called *search electrodes* inserted into the ground. To prevent this attack, the combatants introduced various countermeasures such as placing the ground connections far behind the front trenches, using twisted-pair cables, reducing the line currents, and limiting the sensitivity of information communicated via field phones [34, 13, 206].

This attack does not target a cryptographic implementation. It is merely an eavesdropping attack on an insecure communication channel. Nevertheless, it is a good example of how an attacker can exploit some compromising emanations that were not that obvious in the first place. Moreover, it uses the same principle employed in power analysis attacks to measure the power consumption of a device that executes cryptographic algorithms.

According to a paper declassified by NSA in 2007 [131], attributed to Jeffrey Friedman [311], the United States of America learned about the existence of compromising emanations during World War II. The problem of compromising emanations, which was given the code name *TEMPEST*, is also mentioned in another document recently declassified by NSA [57]. It is clearly described by the following excerpt from the declassified paper:

> *Any time a machine is used to process classified information electrically, the various switches, contacts, relays, and other components in that machine may emit radio frequency or acoustic energy. These emissions, like tiny radio broadcasts, may radiate through free space for considerable distances – a half mile or more in some cases. Or they may be induced on nearby conductors like signal lines, power lines, telephone lines, or water pipes and be conducted along those paths for some distance – and here we may be talking of a mile or more.*

The problem of compromising emanations was discovered in 1943 by a researcher from Bell Labs. While he was testing the Bell-telephone mixing device 131-B2, which was used for encryption in the backbone systems of the U.S. Army and Navy, he observed a spike on an oscilloscope situated in a distant corner of the lab. After a careful examination of the spikes, he was able to recover the plaintext encrypted by the device. In a demonstration under field conditions for the skeptical military leaders who did not believe that the phenomenon can be exploited, some engineers from Bell Telephone recovered 75% of the plaintext processed by a device situated at a distance of about 25 meters.

Bell Labs was appointed to study the phenomenon in depth and propose modifications to secure the Bell-telephone mixing device 131-B2. They identified three separate phenomena and suggested three basic suppression measures: *shielding* (for radiation through space and magnetic fields), *filtering* (for conducted signals on power lines), and *masking* (for space-radiated or conducted signals). However, the application of shielding and filtering countermeasures was challenging. The modified mixer was heavy, had issues with heat dissipation and limited the access to various controls. As a result, it was never used in the field. Instead, commanders were advised to keep a control zone of about 30 meters in diameter around the their communication centers.

After World War II, the problem was forgotten and rediscovered by the CIA in 1951. At that time, a considerable effort was put into understanding the phenomena of compromising emanations and determining suppression techniques. In the next years, the progress in developing and improving new attack techniques was faster than in building countermeasures. A disturbing discovery was the threat of *acoustic* emanations since it was immediately linked to the microphones found in various strategic locations. Ordinary microphones could detect machine sounds with enough fidelity to permit exploitation.

The NSA did not declassify the entire paper [131], leaving the description of two separate, but apparently related, topics enticingly redacted. One topic is called *seismic* and the other *flooding*.

Peter Wright, a former MI5 officer, describes in his book [384] two missions that exploited compromising emanations. The first mission took place in 1956, during the Suez Crisis, when a joint operation of MI5 and GCHQ determined the initial position of several wheels of the Hagelin cipher machine installed in the Egyptian Embassy in London using a phone bug and an oscilloscope. The second espionage campaign targeted the communications of the French Embassy in London between 1960 and 1963. A broadband radio frequency tap installed on the telex cable carried a secondary faint signal that facilitated the recovery of the plaintext.

For a long time, the topic of emission security was exclusively studied by military and intelligence agencies, which treated it as a highly classified case. According to Markus Kuhn, the security risks of electromagnetic radiation were first mentioned in the open literature only in 1966 [206]. However, the broader public became aware of the problem of compromising emanations in 1985 when Wim van Eck published a paper that shows how to reconstruct the image of a cathode ray tube (CRT) display by picking up and decoding the electromagnetic interference produced by this type of equipment [369]. He made a practical demonstration of the attack using just a dipole antenna, a television receiver, and an external synchronization oscillator. Wim van Eck proposed three ways to mitigate the attack: decrease radiation level, increase noise level, and randomize the sequence in which the image is displayed on the screen. The countermeasures proposed by van Eck can be used to protect an implementation against other types of emissions as well.

The cryptographic research community started to intensively study and publish papers on various implementation attacks in the late 1990s. Firstly, Paul Kocher developed *timing attacks* in 1996 [198]. The core idea of his attacks is that one can exploit the variations in the execution time of cryptographic software to recover the secret data involved in the computations. His paper is considered to be one of the foundation bricks for research in the field of *side-channel attacks*. The following quote illustrates his view about side-channel attacks at that time:

> *In general, any channel which can carry information from a secure area to the outside should be studied as a potential risk.*

In 1997, Boneh *et al.* [60] published a theoretical model for breaking cryptographic systems by taking advantage of random hardware faults. It was followed by a related attack, called *differential fault attack*, proposed by Biham and Shamir [49]. Then, Kocher *et al.* described two *power analysis attacks*, namely *simple power analysis (SPA)* and *differential power analysis (DPA)*, in a technical report released in 1998 [199] and then in a paper published in 1999 [200].

The threat of *optical compromising emanations* was demonstrated in 2002. Kuhn was able to reconstruct the information displayed on a CRT computer monitor in a dark environment using a photomultimeter and a computer with a fast analog-to-digital converter [205]. In the same year, Loughry and Umphress showed that LED status indicators found in various electronic devices can carry a modulated optical signal that is correlated with the handled information [221].

In 2004, Asonov and Agrawal found that keyboards of computers, notebooks, telephones, and ATM pads are vulnerable to attacks that differentiate the sound emanated by different keys [20]. The attack was improved by Zhuang *et al.* in 2005 [392]. Their work was followed by an acoustic attack on matrix printers in 2010 [23] and an RSA key extraction via acoustic cryptanalysis in 2014 [138].

Murdoch showed, in 2006, that many computers reveal their CPU load via thermal leakage and that the influence of the temperature on clock skew can be remotely detected [245]. Masti *et al.* demonstrated that the processor core temperature can be used both as side-channel and covert communication channel [228].

For more details on the history of implementation attacks, we refer the reader to [206, 13, 313, 390, 389]. Numerous implementation attacks were published in the proceedings of various conferences and workshops on security or cryptography. Some good books about power analysis attacks are [223, 272, 11].

1.5.2 Classification

There are many criteria one can use to categorize implementation attacks since they use various techniques that differ in cost, time, equipment, or expertise needed. In the cryptographic engineering literature, it is common to categorize physical attacks according to two criteria [334, 223]. The first criterion classifies physical attacks into *passive* and *active* attacks depending on whether the attacker directly interacts with the target or not.

- **Passive attacks.** In a passive attack, the attacker gathers information by simply observing executions of a cryptographic algorithm on the target device which operates according to its functional specifications. Side-channel attacks fall into this category since the attacker has to observe a physical phenomenon such as electromagnetic emissions, power consumption, or acoustic emanations.

- **Active attacks.** An active attack exploits an erroneous or unexpected behavior of the target device in response to a manipulation done by the attacker to affect the execution environment or the underlying hardware. Fault attacks are active implementation attacks that induce faults into a device by means of supply voltage, external clock, temperature, light, X-rays, or ion beams [32].

A complex attack can consist of a sequence of active and passive attacks. For example, an active attack can be a preparation step for a passive attack. Examples of attacks that combine active and passive techniques are [12, 83, 299]. Compared to active attacks, passive attacks are harder to detect and do not leave any damage to the attacked device.

The second criterion distinguishes between three types of attacks based on the level of intrusion into the target device: *non-invasive*, *semi-invasive*, and *invasive* attacks. Any passive or active attack can fall into one of these three types of attacks [223].

- **Non-invasive attacks.** In a non-invasive attack, the target device is attacked only through directly accessible interfaces and thus it is not permanently altered. In other words, the attacker observes or manipulates only the environmental parameters, but not the device itself. These attacks are relatively inexpensive and thus they constitute a real threat to the security of cryptographic devices.

- **Semi-invasive attacks.** Semi-invasive attacks are characterized by a moderate level of physical intrusion. For example, the casing of a device is removed or the microchip is decapsulated to get better access to its inner components, but no direct electrical contact to the chip surface is made because the passivation layer is not damaged. All methods of decapsulation such as eroding the chip surface by mechanical or chemical means fall into this category.

- **Invasive attacks.** An invasive attack is the strongest type of attack against a device since the attacker has full control of the target device and no boundaries. The adversary can establish electrical contact with the chip and even modify the circuit. Invasive attacks are very powerful, but they require expensive equipment, which can be usually found only in specialized laboratories.

A visual representation of the classification of implementation attacks along the two dimensions, namely interaction with the target device and intrusion level, is shown in Figure 1.2. When the degree of interaction with the target device increases, the attacker is more exposed to the risk of being detected. However, this comparison has its limitations. For example, a passive attacker measuring the power consumption of a device in a certain physical location is typically more exposed than an active attacker who mounts a remote timing attack from the comfort of his home. On the other dimension, the cost of the equipment necessary to mount the attack increases proportionally with the intrusion level. The more invasive an attack is, the higher are the chances that it permanently damages a device and consequently the attack will be detected. Hence, in a broad sense, one axis quantifies the risks to which an attacker is exposed to and the other the cost of an attack.

Although, the classification of implementation attacks along the two dimensions is not perfect or complete, it serves the purpose of grouping different types of attacks by their main characteristics. Side-channel attacks are usually passive and non-invasive attacks, while fault attacks fall into the category of active attacks. Depending on the fault injection mechanism, fault attacks can be non-invasive or semi-invasive . Most of the semi-invasive and invasive attacks are active attacks (e.g. *probing attacks* [160], *reverse engineering attacks* [255], *data remanence attacks* [158, 327]). Consequently, there are few semi-invasive or invasive attacks that are passive attacks.

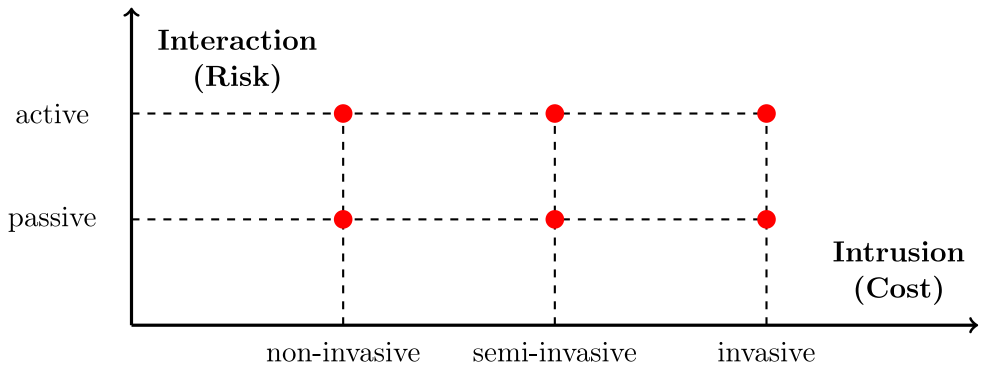


Figure 1.2: A classification of implementation attacks.

1.6 Side-Channel Attacks

Side-channel analysis attacks belong to the genre of physical attacks and exploit some auxiliary information (e.g. the power consumption of a device that executes a cryptographic algorithm) to recover the secret key [178]. Their main advantage is that they are hard to detect because usually a side-channel attacker does not interact with the target device. Hence, they fall into the class of passive and non-invasive implementation attacks. Consequently, a second advantage of side-channel attacks is that they can be mounted with relatively cheap equipment.

There are many sources of side-channel information that an attacker can exploit to break a system. The most popular ones are: timing [198], power consumption [199, 200], and electromagnetic emanations [286, 5]. These three physical effects caused by the execution of a cryptographic algorithm on a device are very popular because they are easy to observe and record. Moreover, they usually carry enough information about the secret used during the computation such that it can be recovered after a relatively low number of measurements. Less popular sources of side-channel leakage are: acoustic [138, 139], optical [128, 328], and thermal [169].

The study of the side-channel emanations from a cryptographic device with the aim of recovering the secret used during the observed computations is referred to as *side-channel analysis (SCA)*. In this work we focus only on side-channel analysis attacks that exploit power and electromagnetic emanations, which are briefly described next.

1.6.1 Power Analysis Attacks

Every electronic device needs electric power to operate. More concretely, a digital circuit needs power to transfer and process data. The power is supplied at different voltages and current intensities (currents) depending on the circuit design. At any point in time, the power consumption of a CMOS circuit consists of several components: $P_{operation}$ – an operation-dependent component, P_{data} – a data-dependent component, $P_{electronic\ noise}$ – electronic noise component, and $P_{constant}$ – a constant component.

$$P_{total} = P_{operation} + P_{data} + P_{electronic\ noise} + P_{constant} \tag{1.1}$$

The two interesting components for a side-channel attack are $P_{operation}$ and P_{data}. Power analysis attacks exploit the small variations in the power consumption of a device determined by different operations being executed and/or different data being processed.

The total power consumption of a single CMOS cell is the sum of its static and dynamic power consumption. The dynamic power consumption depends on the data being processed and usually is the dominant component of the power consumption of a cell. For example, when there is a signal transition between the input and output of a cell (i.e. $0 \mapsto 1$ or $1 \mapsto 0$), the dynamic power consumption is higher than when there is no transition (i.e. $0 \mapsto 0$ or $1 \mapsto 1$). The static power consumption is increasing in modern technologies that have very small size [223].

1.6.2 Electromagnetic Analysis Attacks

These attacks are based on the information gained from the electromagnetic radiation of the electromagnetic field generated by a device. Electromagnetic radiation propagates through space and consists of waves which are synchronized oscillations of electric and magnetic fields. In other words, the electromagnetic field is the combination of the electric field (E) and its dual, the magnetic field (H). Hence, the electromagnetic field typically carries similar information about the processed data and executed instruction as the power consumption does.

A major advantage of electromagnetic analysis attacks over power analysis attacks is that they do not require insertion of a shunt resistor in the ground or current path of the target device to perform the measurements. Moreover, an attacker can use different spots for the acquisition of the electromagnetic emanations such as chip surface [219] or decoupling capacitors [258, 29]. However, the process of identifying the best spot can be very time consuming since it depends on a combination of factors such as a careful selection of the EM probe and finding a good orientation for it. Sometimes, the electromagnetic signal has to be amplified and preprocessed, but it can give better results than power consumption measurements in the sense that the number of observations necessary to recover a secret may be lower [265].

1.6.3 Attack Toolkit

The toolkit of a side-channel attacker depends on the type of information that is exploited in an attack. In general, the attacker needs some equipment to record the side-channel information, which is usually referred to as *side-channel leakage* or simply *leakage*. She also needs some tools to process the side-channel leakage in order to break the cryptographic system.

Typically, a side-channel attack is performed in two phases: an *online phase* and an *offline phase*. In addition to these two phases, some attacks (e.g. *template attacks* [79, 290]) require a prior *profiling phase* in which the attacker uses a device similar to the one to be attacked to characterize its leakage. This phase yields the so-called *templates* which profile the power consumption or the electromagnetic emanations of the target device. In the *online* phase, the attacker measures and records the side-channel information while the target device performs cryptographic operations. For each input or output of the cryptographic algorithm, the attacker captures a set of leakage samples referred to as a *side-channel trace*. At the end of this phase, the attacker has a set of *traces* and, usually, the corresponding input or output values of the algorithm. The data collected in the *online phase* is fed into an attack algorithm in the *offline phase* to recover the key used during the observed computations. The outcome of the attack algorithm is one or several key candidates.

The time spent in the *online phase* of an attack is determined by the number of measurements recorded, which in turn depends on the measurement setup. An attacker aims to break a system using as few measurements as possible in order to reduce the risk of being detected. Typically, the *offline phase* of an attack is constrained by the technical resources of the attacker and by the number and quality (i.e. signal-to-noise ration) of the traces acquired in the previous phase.

This is a rather compact description of a side-channel attack, similar to the one in [191]. The first mention of DPA attacks [199] outlines two phases: *data collection* and *data analysis*. Other works present a more fine-grained flow for side-channel attacks. For example, the attack strategy described in [223] consists of five steps, while the one presented in [201] involves six stages.

In the following, we describe the equipment and tools required to exploit the power or electromagnetic leakage of a device. Then, we introduce some metrics which are frequently used to quantify different aspects of a side-channel attack and some tools designed for side-channel attacks.

1.6.3.1 Measurement Setup

The online phase of a side-channel attack requires a measurement setup, which typically consists of a digital sampling oscilloscope (DSO), a target device (sometimes referred to as device under test – DUT), and a computer (PC), as shown in Figure 1.3. In addition to these components, a measurement setup may include: a regulated power supply, a clock generator, an amplifier, and different types of probes.

The entire process is controlled by the PC, which sends inputs to the DUT and gets the corresponding leakage traces from the DSO. Typically, the PC communicates with the target board through a serial connection via USB. To reduce the noise generated by the communication channel, the USB connection can be replaced by an optical fiber link. The oscilloscope is connected to the computer using an Ethernet cable or an USB cable.

In a controlled environment such as a research laboratory, the attacker sets a trigger signal that is used by the oscilloscope to record the side-channel leakage. In real world settings, most of the time, there is no trigger signal available for the attacker. In such situations, she has to use other means of detecting the relevant part of the acquired emanations such as a pattern-based trigger [41]. Consequently, the traces are not aligned and thus they have to be preprocessed before they can be

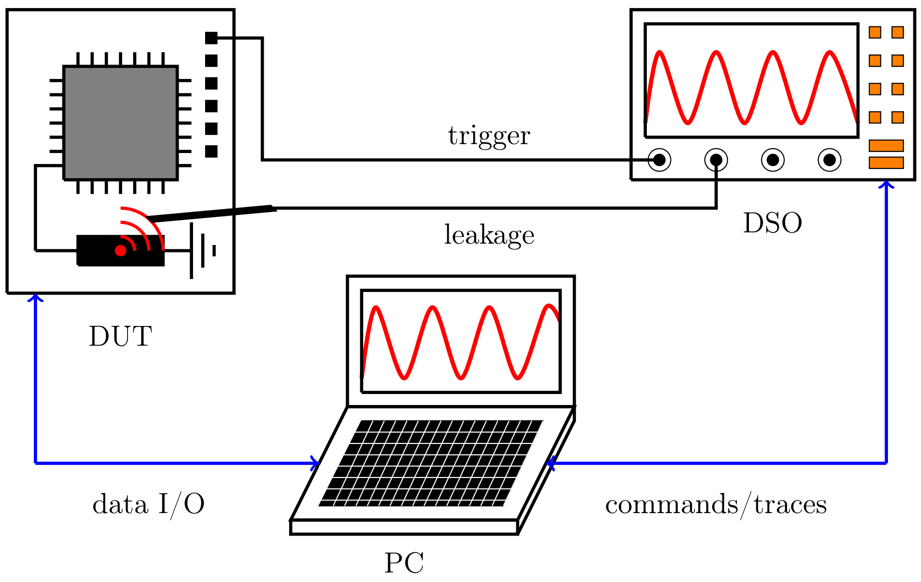


Figure 1.3: Diagram of a measurement setup used for power and EM attacks.

used in the next phase of the attack.

The attacker uses a power measurement circuit or an electromagnetic probe to measure the leakage signal [223]. Typically, the attacker inserts a resistor in the power or ground line of the target device in order to be able to measure the power consumption of the device. If the resistor is inserted in the power line, then a differential probe is used to measure the voltage drop across the resistor. When the resistor is connected in series to the device's ground line, a normal probe can be used. The electromagnetic emanations can be measured using an E-field probe, an H-field probe, or a simple coil placed in a carefully chosen spot.

A key component of the measurement setup is the digital sampling oscilloscope, which takes an analog voltage or electromagnetic signal and converts it into a digital signal. This analog-to-digital conversion is characterized by three main parameters: *input bandwidth*, *sampling rate*, and *resolution* [223]. The bandwidth of an oscilloscope gives the maximum frequency at which a signal is processed without distortion. The sampling rate is equal to the number of points of the input analog signal that are recorded in a second. The resolution of an oscilloscope is the number of possible values a sample in the digitized signal can take. Beside these three parameters, there are other important parameters such as memory size or the smallest input range that can be measured [334]. All these parameters determine the specifications of an digital sampling oscilloscope.

Typically, an attacker is interested in reducing the influence of noise on the performed measurements such that the acquired signal is very clear. There are various ways to reduce the noise level of a measurement setup that range from disabling unused features of the target device, such as a blinking LED, to using a Faraday cage to isolate the target device from the environmental noise. An important

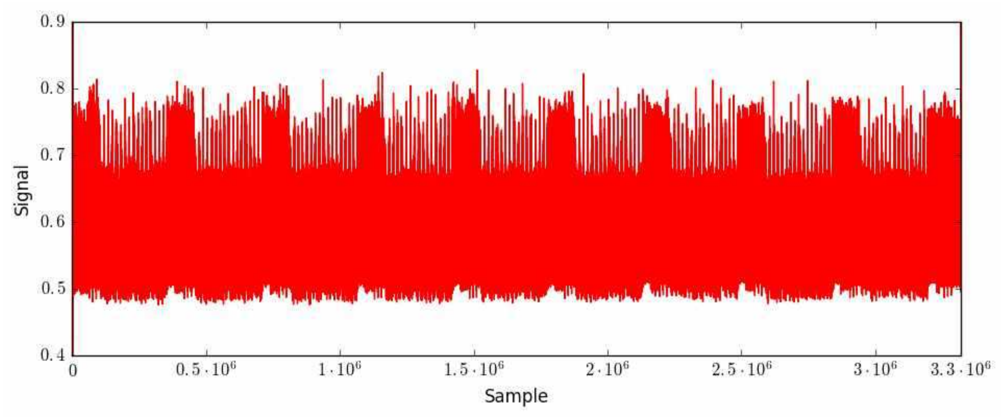


Figure 1.4: A power trace of the AES-128 sampled at 1 GS/s from an Arduino Uno.

source of noise is the power supply. Therefore, an attacker usually prefers to use a battery or a regulated power supply instead of powering a target device from an USB cable which is directly connected to a computer.

More detailed descriptions of various measurement setups can be found in [223, 334, 335]. In this work, we used different measurement setups to record the power consumption or the electromagnetic emanations of several target devices. In each chapter where we present experimental results, we briefly describe the measurement setup used.

A power consumption trace measured during the execution of the AES-128 on an Arduino Board is exemplary shown in Figure 1.4. One can easily identify the ten rounds of the AES-128 in the leakage trace acquired at a sampling rate of 1 GS/s.

1.6.3.2 Signal Processing

In practice, signal processing techniques can be applied in both phases of an attack to improve the signal-to-noise ratio of the measured side-channel information or to align the collected traces. Typically, signal processing improves the attack outcome by reducing the number of traces required to recover the secret key. The time complexity of the offline and online phases of the attack scales down accordingly. Some of the most suitable signal processing techniques for side-channel analysis attacks are: *signal filtering*, *signal alignment*, *signal compression*, and *signal averaging*.

- **Signal filtering.** The main purpose of filtering the side-channel leakage is to isolate those components of the signal that carry information about the processed data or the executed instruction. To this end, low-pass and band-pass filters are typically applied to the measured signal in order to control the frequency range and remove noise.

- **Signal alignment.** The measured leakage traces may be misaligned due to the absence of a reliable trigger signal or as the result of hiding countermeasures such as dummy operations, random delays [86], and shuffling [373]. Some of the waveform matching algorithms proposed in the literature for signal alignment are: correlation coefficient, cross-correlation, sign comparison, sum of absolute differences (SAD), least square, and interval matching [41, 223].

- **Signal compression.** Compression algorithms combine multiple samples of a single trace to reduce the size of the trace. Signal compression can help to reduce high-frequency noise and amplify signal resolution while reducing the amount of data that requires processing in subsequent steps [201]. Compression methods range from extracting the mean or maximum from a set of consecutive samples to more complex resampling techniques.

- **Signal averaging.** The main purpose of signal averaging is to reduce the noise by performing the same measurement several times. It can be done directly on the digital oscilloscope used for measurements or on a computer during or after the data collection process. Conditional averaging [218] computes in the offline phase an average trace from all traces whose associated intermediate input or output used in an attack is the same.

Signal processing techniques may combine some of the above-mentioned methods. For example, a signal compression algorithm may include a low pass filter.

1.6.3.3 Attack Algorithms

Simple Power Analysis (SPA) Attacks. Simple power analysis (SPA) [199, 200] encompasses a collection of analysis techniques that can be used to extract a secret key from one or few traces. Essentially, the attacker inspects the patterns within the power consumption traces and maps them to the structure of the executed algorithm. Typically, the attacker targets instructions that use or depend on the value of the secret key. Despite their name, SPA attacks on symmetric cryptosystems are very challenging in practice due to noise and because they require a detailed knowledge of the executed algorithm.

SPA attacks are typically applied in the context of public-key cryptography. For example, the modular exponentiation used in the RSA algorithm can reveal the private key because multiplications consume more power than squarings. Consequently, multiplications appear as higher peaks in power traces. They are performed only when the corresponding bit of the exponent is set to 1. Yet, SPA can also be used to attack insecure implementations of symmetric key algorithms. See for example the SPA attacks against insecure implementations of the key schedule of the AES [222, 370].

SPA attacks can exploit leakages that stem from both data and instructions. When the electromagnetic emanations of a target device are the source of the side-channel leakage, the corresponding attack is referred to as a *simple electromagnetic analysis (SEMA)* attack.

Differential Power Analysis (DPA) Attacks. Differential power analysis (DPA) [199, 200] uses statistical techniques to exploit the dependency between the power consumption of a device and the processed data. In contrast to SPA attacks, DPA attacks require more power traces, but they do not need a detailed knowledge of the attacked algorithm. While SPA attacks analyze the power consumption along the time axis, DPA attacks focus on the variations of the power consumption at a fixed moment of time that result from handling different data values. Usual targets for DPA attacks are symmetric-key algorithms, especially block ciphers.

Based on a *divide and conquer* paradigm, DPA attacks recover individual chunks of a secret key. Afterwards, these chunks are used to reconstruct the full secret key. The general idea is to choose an *intermediate value* (also called a *sensitive value*) of the attacked algorithm that combines a chunk of the key with a known variable input (e.g. a part of the plaintext) and to estimate its power consumption using a *power model* for all possible values of the key chunk. The resulting set of *predicted leakages* is compared with the traces collected from the target device to determine the most likely key chunk used during the observed computations. The statistical test used to infer the most likely key candidates is referred to as a *distinguisher* in the side-channel literature.

When it was initially proposed, DPA was merely a side-channel attack method that combined the information contained in several power consumption traces measured from a cryptographic device to recover the secret key used during the observed computations. Nowadays, in addition to the original definition, DPA encompasses a rich class of techniques for side-channel attacks. In general, a DPA attack is characterized by two key features: the *power model* that describes the hypothetical power consumption of a target device and the *distinguisher* used to determine the secret key.

The original DPA attack uses a *single-bit* power model and the *difference of means (DoM)* test as distinguisher. The power traces are divided into two sets depending on the value of the intermediate bit. Then, a *differential trace* is computed for each key candidate. A spike in a differential trace indicates that the corresponding key is a candidate for the correct key. The analogous attack technique that exploits the electromagnetic emanations of a device instead of its power consumption is referred to *differential electromagnetic analysis (DEMA)* attack.

Subsequent developments of DPA attacks concentrated on the two key features of DPA. Consequently, new power models were proposed to better capture the instantaneous power consumption of a target device. At the same time, the relation between the estimated and the measured power consumption was better exploited by new distinguishers.

Messerges *et al.* [240] exploited the leakage of all bits of an intermediate variable to get a better signal-to-noise ratio in the differential traces by using a *multiple-bit DPA*. The leakage traces are separated in two sets according to whether all bits in an intermediate variable are set to zero or to one. All traces that do not fit into the two sets used to compute the differential trace are discarded. Therefore, this power model requires a large number of traces. The same authors [241] exploited the relation between the power consumption of an intermediate variable and its binary representation, which was already mentioned before in several works [200, 240, 232, 87, 8]. The Hamming weight of an intermediate variable (i.e. the number of bits set to one in that variable) or the Hamming distance between two consecutive values of the same variable (i.e. the number of bits that changed between the two states of the variable) are good power models for side-channel attacks. They proposed a new technique, *generalized multiple-bit DPA*, which splits the traces used to compute the differential trace into two sets depending on whether the modeled power consumption is lower or greater than a threshold.

Various statistical tests were proposed to improve the recovery of the correct key. Coron *et al.* [87, 88] studied several leakage detection tests to check the existence of secret-correlated emanations from cryptographic devices. Mayer-Sommer evaluated the changes in power dissipation due to writing different data values into a certain memory location or register [232]. The Pearson's correlation factor was used to determine the exact instant during the execution at which the dependency between power dissipation and data is maximal. Bevan and Knudsen [47] proposed the maximum-likelihood test. Agrawal *et al.* [6] used a generalized maximum-likelihood test in the context of multi-channel attacks to identify the correct key hypothesis.

Brier *et al.* [68] combined the advantages of the previous power models and statistical techniques in a new method – *correlation power analysis (CPA)*. Their approach is based on the Hamming distance model, which encloses the particular case of Hamming weight leakages. The Pearson's correlation coefficient is used as statistical distinguisher to optimally exploit the estimated relation between the power model and actual traces [67].

Depending on the number of time samples exploited in an attack, one distinguishes between *univariate* and *multivariate* attacks. Most attacks proposed in the literature are univariate because they focus only on atomic leakages at single points in time. In contrast, multivariate attacks exploit joint statistical properties of several time samples. Unlike multivariate attacks, univariate attacks do not work on intermediate variables of a masked algorithm (see Section 1.7.1). Typically, univariate attacks require a preprocessing step which combines several atomic leakages at different points in time using a *combining function*. Multivariate attacks, such as *mutual information analysis (MIA)* [141], can be used to attack masked implementations without a preprocessing step that combines the leakeages [282]. Therefore, they are not affected by the information loss induced by the preprocessing step [78].

Mutual information analysis (MIA) uses a distinguisher based on mutual information to measure the statistical dependence between the predicted power consumption and the recorded traces. It works even if each intermediate variable leaks in a distinct manner. Hence, MIA allows the application of the most generic power model possible, namely the identity function. However, correlation-based attacks are typically more efficient than MIA in simple attack scenarios [282]. A comprehensive evaluation of MIA was conducted in [381].

Profiled Attacks. This category of attacks comprises all techniques that require a profiling phase to gain additional information about the leakage of a target device. The core idea of profiled attacks is that the outcome of a side-channel attack improves with a more precise leakage model of the target device. Indeed, profiled attacks are among the most powerful side-channel techniques.

Template attacks, introduced by Chari *et al.* [79], seek to make maximal use of a small number of traces from a target device. Although template attacks may require a large amount of initial effort to accurately model the leakage of a target, they are the strongest side-channel attack possible from an information-theoretic point of view if the noise follows a multivariate Gaussian distribution. However, experimental results show that in practice template attacks suffer from the variability caused by different devices or different acquisition campaigns [81]. Template attacks are particularly useful when few measurements can be obtained from the target device and a clone device is available for training. Template attacks have a strong advantage over non-profiled attacks when applied against masked implementations [7, 266].

Linear Regression Analysis (LRA) attacks, or *stochastic attacks*, were introduced by Schindler *et al.* [307] as an efficient alternative to template attacks in situations where the adversary already has a parameterized model for the leakage of a target device. As first observed in the original paper [307] and later confirmed by Doget *et al.* [112], LRA attacks can work without a profiling phase. Lemke and Paar [213] applied profiled LRA attacks against masked implementations, while Dabosville *et al.* [96] mounted non-profiled LRA attacks against masked implementations. An experimental comparison between template and stochastic attacks under identical conditions can be found in [142].

Other Attacks. Besides the above-mentioned attacks, there are many other attack algorithms described in the literature. For example, *collision attacks* [312] use the side-channel leakage of an implementation to detect internal collisions, which provide some information about the secret key. Nevertheless, an exhaustive study of all existing attack techniques is outside the scope of this work.

1.6.3.4 Metrics

Standaert *et al.* [336] described a framework for fair evaluation and comparison of side-channel attacks. Their framework includes two types of metrics: information-theoretic metrics and actual security metrics. The information-theoretic metrics are used to gauge the amount of information leaked by an implementation, while the actual security metrics show to what extent the leaked information can be used by an attacker. They proposed the following metrics: *conditional entropy* (information-theoretic metric), *success rate* and *guessing entropy* (actual security metrics).

Next, we briefly describe three metrics widely used in the literature for experimental evaluations of side-channel attacks, namely *success rate*, *guessing entropy*, and *number of traces*.

Success Rate. The success rate of a side-channel attack is computed as the ratio between the number of experiments in which the correct key was recovered and the total number of experiments carried out. The success rate of an order o is the probability that the key is in the first o key candidates.

Guessing Entropy. The guessing entropy gives the average number of key candidates that have to be tested to recover the correct key after a side-channel attack was performed. Hence, it measures the remaining workload of a side-channel adversary. Guessing entropy was defined by Massey [227] and then used by Köpf and Basin [203] to evaluate the effectiveness of side-channel attacks.

Number of Traces. This metric assesses the number of measurements an attacker has to perform in the online phase of an attack in order to achieve the desired outcome (e.g. a guessing entropy below a fixed threshold). Mangard *et al.* [223] provided a way to estimate the number of traces required to mount a successful DPA attack.

1.6.3.5 Tools

In this section, we present some tools designed for side-channel attacks, especially for power and electromagnetic analysis attacks. The list includes a wide range of tools from professional frameworks designed for security evaluations in accordance to various security standards such as Common Criteria [89] to software libraries written for hobbyists. The industry-grade solutions comprise everything an attacker or a security evaluator needs, from a specialized hardware measurement apparatus to highly flexible software tools. In contrast to tools designed by side-channel researchers and enthusiasts, which are typically free and open-source, the professional tools are usually very expensive.

Inspector SCA. Designed by Riscure, Inspector SCA [294] is a modular platform that combines features for acquisition, alignment, and signal processing of leakage traces. It integrates with a variety of hardware equipment and provides a set of statistical tools that can be used in side-channel attacks against major cryptographic algorithms. Customers get access to the source code of the tool and can extend its functionalities.

DPA Workstation Analysis Platform. DPA Workstation Analysis Platform [287] is designed by Rambus for security chip vendors, product companies, testing labs, and government organizations. The collected leakage traces can be examined using Simple Power and Electromagnetic Analysis (SPA/SEMA) or more powerful Differential Power and Electromagnetic Analysis (DPA/DEMA) to identify exposure of secret keys.

ChipWhisperer. ChipWhisperer [252] is a toolchain for embedded hardware security research, including side-channel and glitching attacks. It is a combination of open-source software and hardware sold at a low price. The hardware uses a synchronous capturing method, which greatly reduces the sampling rate and the data storage. This is possible because the sampling clock is synchronized to the target clock [260].

Daredevil. Daredevil [323] is a tool for (higher-order) correlation power analysis (CPA) attacks. Its distinctive feature is that it can perform CPA attacks very fast on multiple cores given a specified amount of memory. The algorithm used by the tool is selected after a careful evaluation of the computational aspects of calculating the Pearson product-moment correlation coefficient and is based on an incremental approach which extends already completed computations [64].

pysca. pysca [189] is a side-channel analysis toolbox written in Python that aims to be simple and flexible, while using a language suitable for scientific computing. It implements state-of-the-art DPA techniques and achieves good performance. Another key feature is that it facilitates visualization of various metrics for security evaluation purpose.

Jlsca. Jlsca [76] is a tool for side-channel analysis written in Julia, a dynamic programming language for numerical computing. It can be executed as a stand-alone tool or as a module inside Inspector SCA. Some of the features implemented in Jlsca are: conditional averaging, non-profiled linear regression analysis (LRA) [307, 218], incremental correlation statistics [43], and mutual information analysis (MIA) [141].

1.7 Countermeasures against Side-Channel Attacks

The discovery of side-channel attacks triggered a continuous arms race between attackers and designers of countermeasures against side-channel attacks. As a result of this race, both attack and defense techniques evolved. On the one hand, new and improved attack techniques have been proposed. On the other hand, countermeasures have been devised and improved to prevent such attacks.

The goal of side-channel countermeasures is to make the physically observable leakage of a device independent of the intermediate values of the executed cryptographic algorithm or at least to reduce the dependencies between the two.

There are various countermeasures against side-channel attacks, which can be applied at different levels: gate/circuit, architecture, system, implementation, or protocol. While some countermeasures focus on the root cause of the problem and try to prevent it, others try to increase the difficulty of attacks or to simply slow down attackers. In practice, several countermeasures are usually combined following a defense in depth approach to achieve better security against side-channel attacks [161, 295, 309].

The most widely used countermeasures against DPA attacks can be classified into two categories: *hiding* and *masking*. Hiding changes the leakage of a device with the aim of making it random or equal for all operations and data values. Although reaching perfectly random or equal leakages is an elusive goal, there are several techniques that achieve good results by introducing changes in the time dimension or the amplitude dimension of the leakage. Hiding in the time dimension can be done by randomly inserting *dummy operations* (e.g. random delays [86]) or by *shuffling* (i.e. randomly changing the sequence of operations of a cryptographic algorithm that can be performed in arbitrary order [373]). Hiding in the amplitude domain modifies the signal-to-noise ratio by either increasing the noise (e.g. performing several operations in parallel, using noise generators [202]) or by lowering the signal (e.g. filtering [317]). Masking randomizes the intermediate values that are processed by a cryptographic device. Therefore, it can be applied at the algorithmic level without changing the power consumption characteristics of the device. Firstly used as a side-channel countermeasure by Chari *et al.* [78] and Goubin and Patarin [151], masking is based on the principle of *secret sharing* introduced independently by Blakley [55] and Shamir [316].

Masking and hiding are implemented at circuit level using secure *logic styles* such as Sense Amplifier Based Logic (SABL) [354] and Wave Dynamic Differential Logic (WDDL) [355]. However, logic styles can be easily defeated when the circuit is not perfectly symmetric or the input signals of a gate do not arrive at the same time [345].

Other types of side-channel countermeasures are *protocol level* countermeasures and *leakage-resilient* cryptography. A simple approach to prevent side-channel attacks at the protocol level is to reduce the number of cryptographic operations performed using the same key [200]. For example, one can use a re-keying mechanism [236, 111] to change the secret keys frequently. The focus of *leakage-resilient* cryptography is to build primitives that can be proven to be secure against side-channel attacks under certain assumptions on the leakage model [116, 338].

1.7.1 Masking

A first step towards a masked implementation of an algorithm is to convert each sensitive intermediate value of the algorithm into a shared representation. A sensitive value x is split into n values (or *shares*) $x_1, x_2, \ldots, x_n$ such that $x = x_1 \star x_2 \star \ldots \star x_n$, with $n \geq 2$. Among these shares, $n-1$ are generated uniformly at random and the last one is computed such that the sensitive value is revealed when combining all shares. Depending on the operations used in the algorithm to be masked, the operator $\star$ can be replaced by $\oplus$ (exclusive-OR), $\boxplus$ (modular addition), or $\times$ (modular multiplication). Consequently, the masking is referred to as *Boolean masking*, *arithmetic masking*, or *multiplicative masking*. Then, each operation of the algorithm is performed on the shared representation by carefully manipulating the shares independently to ensure that no information about the sensitive value is leaked.

The number of random values in the shared representation of a sensitive value determines the order of a masking scheme. For example, a masking scheme that splits a sensitive value into two shares is referred to as *first-order masking*, or simply *masking*. In general, an n-th order masking scheme resists attacks of order n (i.e. attacks that combine the leakages of up to n points). Chari *et al.* [78] showed that the number of measurements required to attack a masked implementation increases exponentially with the number of shares in simplified, but realistic, settings.

1.7.1.1 Boolean Masking

Boolean masking is probably the most deployed side-channel countermeasure for symmetric algorithms. When protecting a symmetric algorithm that consists of a combination of linear and nonlinear operations, Boolean masking must be applied to all operations. In contrast to linear operations, which are easy to compute directly on Boolean shares, nonlinear operations are difficult to mask.

The approach to mask a nonlinear operation is specific to each operation. When the nonlinear operation is represented as a lookup table, the table can be randomized [239, 161]. In case of modular addition/subtraction, a first approach consists of three steps: convert the Boolean masks to arithmetic masks, perform the operation on arithmetic masks, and then convert the arithmetic masks to Boolean masks. This approach is costly because it requires conversions between masks [149]. A second approach is to perform the nonlinear operation directly on Boolean shares [182, 85].

Trichina [358] proposed a first-order masked AND gate, while Ishai *et al.* [174] proposed several techniques for building *private circuits* and described a way to compute a secure AND gate at any order. Then, Nikova *et al.* [253] introduced *threshold implementations* and a masked AND gate using three shares. Finally, Reparaz *et al.* [292] studied the similarities and differences between the three aforementioned schemes to propose a generalized masking scheme.

1.7.1.2 Provable Security

An important characteristic of masking schemes is that their security can be proven in certain theoretical models such as the probing model [174] or the strong non-interference (t-SNI) model [33]. These models make realistic assumptions on the leakage model of a device and the capabilities of an adversary.

The security of first-order masking schemes can be proven by showing that all the intermediate variables of an algorithm are independent of the sensitive input values. This approach can also be used for high-order masking schemes, but its complexity grows exponentially with the masking order and thus it quickly becomes too complicated. One has to show that any combination of n intermediate variables is independent from the sensitive inputs in order to prove n-th order security.

Sometimes, security proofs are very complex and hard to comprehend even for experts in the field. Hence, a minor slip in the argumentation can yield to a security flaw. Moreover, the models in which security proofs are built do not perfectly match the physical characteristics of a device nor the details of an implementation. Although provable security provides powerful tools to assess the security of masking schemes, it should be exercised with care, especially when implementing masking schemes [28]. Finally, provable security should not be the sole criterion used to determine the strength of a masking scheme. A provably secure masking scheme should not leak when correctly implemented in a device, while its implementation should be efficient and suitable for real-world applications.

1.7.2 Leakage Detection Tests

The goal of a leakage detection test is to determine whether a particular implementation is leaking or not. This is a more efficient and reliable strategy to decide if a protected implementation achieves its security objective rather than performing a battery of known key-recovery attacks against that implementation.

Even when an implementation withstands a set of known attacks, the significance of the outcome is limited since there is no guaranty that a slightly modified attack will not succeed. Certainly, it is difficult to get a good coverage of all known attacks in a battery of tests. On the other hand, leakage detection tests have the benefit of exhibiting leakages that might not yet be exploitable with the known attack techniques.

Coron *et al.* [87, 88] presented several leakage detection tests to verify the existence of secret-correlated emanations. Then, Goodwill *et al.* [148] proposed a methodology for side-channel resistance validation based on statistical hypothesis testing. The core statistical technique for their methodology is Welch's t-test [377], which is an extension of Student's t-test [343] for unequal sample sizes and unequal variances. Becker *et al.* [40] proposed an enhanced and optimized methodology which was named *Test Vector Leakage Assessment (TVLA).*

There are several types of tests that can be performed to evaluate an implementation. A *specific test* targets certain intermediate values (e.g. S-box output) of the assessed implementation. The evaluator knows the secret key and carefully chooses the inputs to activate the targeted intermediates. On the other hand, a *non-specific test* assesses the leakage of all intermediate variables of an algorithm. A *random-vs.-random test* is performed using only random inputs, while a *fixed-vs.-random test* is performed on a data set that uses fixed inputs and a data set of random inputs.

All t-test evaluations are one or two orders of magnitude faster than key-extraction attacks [40]. Moreover, t-test leakage assessments are suitable for real-time computation: the statistics can be computed as measurements are being collected. In particular, the non-specific, fixed-vs.-random t-test can identify leakages in an implementation very fast.

1.8 Internet of Things (IoT)

The *Internet of Things (IoT)* is one of the words that is on everyone's lips nowadays. The salient feature behind this buzzword is that the IoT brings Internet connectivity to a plethora of devices, also called *things*, to create new *ubiquitous* ecosystems. A *thing* can refer to a variety of devices that communicate to each other to make our lives easier. Unlike the classical Internet where all devices use the same protocol suite (i.e. TCP/IP) to exchange data, there is no standard way of communication in the IoT, the only common ground being the Internet layer (i.e. IP/IPv6) connectivity. Thus, the IoT is a highly *heterogeneous* environment with devices clustered in small networks. The communication between these networks and the Internet is currently facilitated by special-purpose hubs or gateways able to transfer the traffic between various IoT technologies and the Internet. Still in its incipient stages, the IoT is believed to fundamentally change our daily lives in the way we will interact with the surrounding environment.

1.8.1 Constraints

A distinguishing characteristic of IoT devices is that they are customized for specific applications. Typically, they are designed to operate autonomously using a limited amount of resources (e.g. energy). Therefore, these devices have to meet various constraints that are imposed by a combination of factors such as the final price of a product or the desired features.

In this context, the amount of resources that can be allocated for security services (see Section 1.2.1) is just a small fraction of the total available resources. Consequently, the cryptographic primitives used to provide the required security services must meet stringent constraints without sacrificing security. The constraints that an implementation of a cryptographic algorithm has to satisfy are usually expressed using metrics such as those defined in Section 1.4.4. Typical constraints for hardware implementations are silicon area, latency, and power consumption. Exemplary constraints for software implementations are code size, execution time, and energy consumption.

The problem of optimizing an algorithm for different criteria spans across multiple axes, with one axis for each goal that has to be achieved. While it is relatively easy to optimize along one axis, it is very hard to optimize for more than one design goal at the same time. In general, designers of cryptographic algorithms focused on optimizing along two different axes (e.g. security vs. speed, area vs. latency). For example, different trade-offs specific to hardware implementations were explored in [277, 193].

1.8.2 Lightweight Cryptography

Lightweight cryptography emerged as a new research direction that aims to address the constraints that conventional cryptography faces in the IoT context. It is widely accepted that cryptosystems play a major role in the security arena of the IoT, but they need to be designed and implemented efficiently enough so as to comply with the scarce resources of typical IoT devices. Gligor defined in [145] lightweight cryptography as *cryptographic primitives, schemes and protocols tailored to (extremely) constrained environments.*

The efficient implementation of cryptographic primitives so that they are applicable in the highly constrained regimes of various IoT devices is a challenging task since, for example, performance is conflicting with other metrics of interest such as memory footprint and code size. In addition, implementations of lightweight cryptography should withstand all known forms of attacks since lightweight cryptography is not meant to be the weakest link in the security of a system.

1.8.3 Device Types

It is important to perceive the similarities and differences between the technologies present in the IoT landscape, especially of those situated at the low-end spectrum of computational power and capabilities: *RFID (Radio Frequency IDentification)*, *NFC (Near Field Communication)*, and *contactless smart cards.* These terms are used interchangeably by many people mainly because in today's digital landscape, keeping track of the technical jargon can be overwhelming. A clear understanding of these technologies reveals to what extent devices using them can connect to the IoT and where lightweight cryptography might be useful.

1.8.3.1 Lower Bound for the Computational Power of IoT Devices

A classical *RFID tag*, which consists of an integrated circuit for storing and processing information and an antenna for receiving and transmitting signals, can communicate only with tag readers. The main characteristic of an RFID tag is its unique serial number that facilitates inventory and package tracking. Active RFID tags contain their own power source, giving them the ability to broadcast signals with a read range of up to 100 meters. Passive RFID tags are powered by the electromagnetic energy transmitted from the RFID reader and have a read range of up to 25 meters [350]. An RFID tag has minimal built-in support for security and privacy [137].

NFC technology is a newer, more finely honed version of RFID. It takes advantage of the short read range limitations (no more than a few centimeters) of its operating radio frequency (13.56 MHz). An NFC device can work in three modes: reader/writer, card emulation, and peer-to-peer. NFC tags contain data which can be read, and under some circumstances can be writable by an NFC device. The card emulation mode enables a phone to behave like a contactless card, allowing users to perform various transactions. Peer-to-peer communication is a feature that sets NFC apart from typical RFID devices [350], enabling devices to exchange information in an adhoc fashion.

A *contactless smart card* contains a small but sophisticated computer (microcontroller) that can perform certain on-card operations to provide a high level of security [137]. The multi-layer security mechanism might include tamper-resistance techniques, a dedicated cryptoprocessor, or a secure file system. Contactless smart cards have a very limited read range of up to 10 cm to prevent tracking or eavesdropping. They are powered by external devices to which they exchange data using communications technologies such as NFC.

As the cost of contactless smart cards decreases to reach soon the cost of RFID tags set at a few cents per unit, classical RFID tags may get phased out due to their drawbacks and limitations. At the same time, NFC technology progressively becomes more popular thanks to NFC-capable smartphones able to accommodate a considerable number of applications without additional costs. Currently, there are two noticeable trends for secure payments operated under financial regulations. Firstly, classical smart cards are evolving into contactless smart cards. Secondly, mobile transactions done through smartphones are growing fast.

1.8.3.2 Upper Bound for the Computational Power of IoT Devices

There are numerous IoT devices (i.e. smartphones, tablets) capable to communicate directly through the Internet. Their role in the IoT is merely to augment the user experience by facilitating interaction with different sensors and actuators. Since these are powerful devices, they do not need to satisfy any additional requirements to be able to communicate with other IoT devices.

1.8.3.3 Middle Range IoT Devices

A multitude of devices lay in between the previous two categories of IoT devices. This category includes a wide range of microcontrollers that can be used for various applications such as wireless sensors, smart homes, building management, telemedicine and healthcare. Microcontrollers are particularly interesting because they have several advantages compared to FPGAs and ASICs. First of all, they are very versatile in the sense that they can accommodate various software implementations. Moreover, the software can be updated (relatively easy if such a mechanism is in place) after the deployment the device. In the second place, the cost of writing software applications is well below the one of hardware implementations. In addition, there are more skilled software engineers than hardware engineers ready to write custom applications and thus to support the fast growth of software applications for the IoT. A third argument in favour of microcontrollers is that software applications written in high-level programming languages such as ANSI C can be ported to different microcontrollers with minor changes using the appropriate toolchain.

1.8.4 Threat Model

Designing and implementing effective security mechanisms requires a good understanding of the system to be protected. In addition to the defender's perspective, a security professional must also embrace an attacker's mindset. Often, an attacker targets those components of a system where the security is not strong enough. For example, many attack vectors in the IoT context stem from the lack of proper physical security, which exposes devices to a wide range of implementation attacks. Therefore, the attack surface of IoT systems is considerably larger than the attack surface of classical Internet-connected computers, which are typically deployed in a secure perimeter. In light of the predicted growth of the IoT to billions of connected devices in the coming years [122, 136], IoT devices must be designed to withstand a variety of attacks in order to avoid large scale security incidents.

A threat model is the result of an iterative process leading to the identification and classification of attack vectors that can be used to compromise an asset. Three of the eight classes of attack vectors described in the thread model for the IoT of Atamli and Martin [21] are essentially different implementation attacks: device tampering, signal injection, and side-channel analysis. While profiling the attackers, they identified three entities that can pose risks to the security and privacy of IoT systems: legitimate user, device maker, and malicious adversary. In an earlier work, Abraham *et al.* [2] described three classes of attackers: clever outsider, knowledgeable insider, and funded organization. For a more detailed treatment of the subject, we refer the reader to the work of Atamli and Martin [21].

1.9 Motivation

The driving force of lightweight cryptography stems mainly from its direct applications in the real world since it provides solutions to actual problems faced by designers of IoT systems. Broadly speaking, lightweight cryptographic algorithms are designed to achieve two main goals. The first goal of a cryptographic algorithm is to withstand all known cryptanalytic attacks and thus to be secure in the black box model. The second goal is to build the cryptographic primitive in such a way that its implementations satisfy a clearly specified set of constraints which depend on a case-by-case basis. The major challenge is to address both design goals at the same time since they require expertise in different domains.

Embedded IoT devices are deployed in various locations, including places with limited or no physical security. In such insecure or even hostile environments, they are an enticing target for implementation attacks. Consequently, the implementations of lightweight cryptographic algorithms that are embedded in these constrained devices have to be protected against very powerful adversaries, while retaining their efficiency.

Major standardization organizations are closely following the evolution of lightweight cryptography. The International Organization for Standards (ISO) and the International Electrotechnical Commission (IEC) have already standardized several lightweight primitives and they currently consider other algorithms for inclusion in their standards. The National Institute of Standards and Technology (NIST) has organized two workshops on lightweight cryptography [248, 249] and has recently announced the requirements for their portfolio of lightweight algorithms [247].

Enforcing security of IoT systems is very challenging task. Due to the resource constraints imposed by the use cases for which the IoT devices are built for, system architects are often left with few to no resources for securing these systems after all desired futures have been added. Solutions to many of these security problems converge to *cryptographic engineering*, a field at the intersection of cryptography, computer science, and electronic engineering. This work seeks to conciliate these contradicting requirements in order to provide secure, yet usable, embedded IoT systems.

1.10 Research Contributions

This thesis focuses on efficient and secure implementations of lightweight symmetric cryptographic algorithms for resource-constrained microcontrollers that are typically used in the IoT. It is organized in three parts that are briefly described next.

1.10.1 Part I – Efficient Implementations

The four chapters of this part are centered around FELICS (***F****air* ***E****valuation of* ***L****ightweight* ***C****ryptographic* ***S****ystems*), an open-source benchmarking framework for software implementations of lightweight cryptographic primitives on embedded devices.

Chapter 2. We introduce FELICS, a free and open-source benchmarking framework designed for fair and consistent evaluation of software implementations of lightweight cryptographic primitives for embedded devices. The framework is very flexible thanks to its modular structure, which allows for an easy integration of new metrics, target devices and evaluation scenarios. It consists of four modules that can currently assess the performance of lightweight block ciphers, stream ciphers, authenticated ciphers, and hash functions on three widely used microcontrollers: 8-bit AVR, 16-bit MSP and 32-bit ARM. The extracted metrics are execution time, RAM consumption and binary code size. FELICS has a simple user interface and is intended to be used by cipher designers to compare new primitives with the state of the art. The extracted metrics are very detailed and assist embedded software engineers in selecting the best cipher to match the requirements of a particular application. The tool aims to increase the transparency and trust in benchmarking results of lightweight primitives and facilitates a fair comparison between different primitives using the same evaluation conditions.

Chapter 3. We use FELICS to benchmark various implementations of 19 lightweight block ciphers, namely AES, Chaskey, Fantomas, HIGHT, LBlock, LEA, LED, Piccolo, PRESENT, PRIDE, PRINCE, RC5, RECTANGLE, RoadRunneR, Robin, SIMON, SPARX, SPECK, and TWINE. Then, we propose a *figure of merit* according to which all evaluated candidates can be ranked. Our results give new insights to the question of how well these lightweight block ciphers are suited to secure the IoT. We also draw conclusions about which design strategies are the most promising ones for the IoT.

Chapter 4. We introduce the SPARX family of lightweight block ciphers. SPARX is the first ARX design that has provable security arguments and competitive performance on resource-constrained devices. In this chapter, we elaborate on the implementation-related characteristics of SPARX and how software efficiency influenced the final design. Then, we provide implementation details and results for two instances of SPARX that use a 128-bit key and two different block sizes, namely SPARX-64/128 and SPARX-128/128.

Chapter 5. We evaluate the cost of the main building blocks of a symmetric cryptographic algorithm to determine the most efficient ones. This chapter provides a detailed insight into the efficiency of software implementations of lightweight symmetric cryptography. The contribution of this chapter is particularly valuable for designers of new lightweight ciphers because they can make design decisions based on both security and efficiency using our results.

Impact. FELICS already has impact in the research community. Many people contributed optimized implementations and several designers of new algorithms used FELICS for the evaluation of their software implementations, while the evaluation results are becoming a common reference in the literature. Moreover, NIST is interested in using FELICS for a fair comparison of candidates for their recommended portfolio of lightweight algorithms for the IoT.

1.10.2 Part II – Side-Channel Attacks

Chapter 6 An important criterion to assess the suitability of a lightweight cipher with respect to SCA is the amount of leakage available to an adversary. In this chapter, we analyze the efficiency of different selection functions that are commonly used in Correlation Power Analysis (CPA) attacks on symmetric primitives. To this end, we attacked implementations of the lightweight block ciphers AES, Fantomas, LBlock, Piccolo, PRINCE, RC5, Simon, and Speck on an 8-bit AVR processor. By exploring the relation between the nonlinearity of the studied selection functions and the measured leakages, we discovered some imperfections when using nonlinearity to quantify the resilience against CPA. Then, we applied these findings in an evaluation of the "intrinsic" CPA-resistance of unprotected implementations of the eight mentioned ciphers. We show that certain implementation aspects can influence the leakage level and try to explain why. Our results shed new light on the resilience of basic operations executed by these ciphers against CPA and help to bridge the gap between theory and practice.

Chapter 7 We show that most implementations of the AES present in popular open-source cryptographic libraries are vulnerable to side-channel attacks, even in a network protocol scenario when the attacker has limited control of the input. We present an algorithm for symbolic processing of the AES state for any input configuration where several input bytes are variable and known, while the rest are fixed and unknown as is the case in most secure network protocols. Then, we describe an optimal algorithm that can be used to recover the master key using Correlation Power Analysis (CPA) attacks. Our experimental results raise awareness of the insecurity of unprotected implementations of the AES used in network protocol stacks.

Chapter 8 We perform the first side-channel vulnerability analysis of the Thread networking stack. We leverage various network mechanisms to trigger manipulations of the security material (i.e. cryptographic keys) or to get access to the network credentials. Then, we choose the most feasible attack vector to build a complete attack that combines network specific mechanisms and Differential Electromagnetic Analysis. When successfully applied on a Thread network, the attack gives full network access to the adversary. We evaluate the feasibility of our attack in a TI

CC2538 setup running OpenThread, a certified open-source implementation of the stack. The full attack does not succeed due to a fortunate side-effect that is not related to security. Finally, we summarize the problems we found in the protocol with respect to side-channel analysis, and suggest a range of countermeasures to prevent our attack and the other attack vectors we identified during the vulnerability analysis. This chapter provides a useful lesson to designers of IoT protocols and devices.

1.10.3 Part III – Side-Channel Countermeasures

Chapter 9 The best known expressions for Boolean masking of bitwise operations are relatively compact, but even a small improvement of these expressions can significantly reduce the performance penalty of more complex masked operations such as modular addition on Boolean shares. Consequently, protected implementations of ciphers that use better expressions get more efficient. We present and evaluate new secure expressions for performing bitwise operations on Boolean shares. To this end, we describe an algorithm for efficient search of expressions that have an optimal cost in number of elementary operations. We show that bitwise AND on Boolean shares can be performed using less instructions than the best known expressions, while the best known expression for bitwise OR is optimal. More importantly, our expressions do no require fresh random values.

Part I

Efficient Implementations

Chapter 2

FELICS – Fair Evaluation of Lightweight Cryptographic Systems

Contents

2.1 Introduction

The imminent expansion of the Internet of Things is creating a new world of smart devices in which security implications are very important. If we consider that brain stimulator circuits and heart pacemakers may be directly connected to a network to provide physicians with useful information in establishing and adjusting the therapy without physical examination of the patient, security plays a crucial role since unauthorized access to these critical devices can be life-threatening. The health sector is just one of the domains where the number of IoT devices is expected to grow significantly. Other IoT applications include supply chain management, smart homes, green cities and many more.

Besides the security aspects, the IoT introduces new challenges in terms of energy and power consumption. Thus the lightweight cryptographic primitives designed for IoT-enabled devices must consume few resources, while providing the claimed level of security. In the recent past, the research community's interest for lightweight cryptography increased and as a result many lightweight algorithms were designed and analyzed from the security perspective. The implementation effort focused on selecting the best design constructions in order to reduce the resource consumption, evaluating the performance figures achieved by hardware and software implementations on different platforms, and analyzing and improving the protection against side-channel attacks.

Looking back at NIST contests for the selection of new cryptographic standards [250, 246], we can see that weak designs from a security perspective were disqualified after the first evaluation phase. In the following stages, the remaining algorithms had similar security margins and thus new evaluation criteria were necessary. This is the moment where hardware and software evaluation of the candidates plays a very important role. As is pointed in [132], the final ranking of candidates is closely related to the hardware and software performance figures. Since benchmarking frameworks allow for consistent evaluation, they are important not only in the selection process of new cryptographic standards, but also for a fair comparison of ciphers' performance in given usage scenarios.

NIST organized two workshops [248, 249] on lightweight cryptography to discuss the security and resource requirements of applications in constrained environments and potential future standardization of lightweight primitives. Considering the increasing market of IoT devices and the industry's need for a standard to secure IoT applications, tools designed to extract the performance figures of lightweight primitives on different platforms under the same conditions are required. These tools help cryptographers to evaluate proposed designs with respect to previous ones and can be used to break the tie between the candidates in the subsequent phases of the selection process. Based on the feedback following the two workshops, NIST decided to create a portfolio of lightweight ciphers that fit into clearly defined use cases specific to the IoT. Moreover, they already announced a call for cryptographic primitives suitable for two different profiles [247].

2.1.1 Research Contribution

Firstly, we analyze previous benchmarking frameworks to identify the strengths and weaknesses of each one. We formulate a set of design goals that are required for a fair evaluation of lightweight primitives on different platforms under the same conditions. Then, we describe the structure of our benchmarking framework, extracted metrics and target devices. For each of the extracted metrics and for each supported device we describe the methodology and tools used.

FELICS (***F**air **E**valuation of **L**ightweight **C**ryptographic **S**ystems*) [93] is a free, open-source and flexible framework that assesses the performance of C and assembly software implementations of lightweight primitives on embedded devices. Thanks to the modular design, the framework can easily accommodate new metrics, usage scenarios, or target devices. It is the core of an effort to increase transparency in lightweight algorithms' performance and aims to facilitate fair comparison of the assessed algorithms. In the past three years we maintained a web page [93] where the tool can be downloaded and up-to-date results of the assessed primitives can be found. Soon after its initial publication, the framework has become a valuable resource and reference point for comparing the efficiency of lightweight ciphers.

To the best of our knowledge, this is the only free and open-source benchmarking framework designed for fair and consistent evaluation of software implementations of lightweight primitives on various IoT embedded devices in the same usage scenarios. As the IoT field is expected to have a major growth in following years, FELICS will help to provide the research community and industry with fair and detailed performance figures of lightweight primitives.

2.2 Related Work

Over time, several benchmarking frameworks have been designed to ease the evaluation of cryptographic primitives on different hardware or software platforms. In addition to these benchmarking frameworks, survey and benchmarking papers [209, 121, 194, 184, 231] were published. In this section we describe the previous work that helped us in designing the proposed framework. For each project analyzed we present the design requirements and constraints, the extracted metrics and the methodology used to ensure a fair and consistent evaluation.

2.2.1 BLOC Project

The BLOC project [74] aims to study the design of block ciphers dedicated to constrained environments. During the project, a paper [75] about the performance evaluation of lightweight block ciphers for wireless sensor nodes was published. The C implementations of the studied ciphers along with the source code used to extract the analyzed metrics are available for free. The target device is the 16-bit MSP430F1611 [348] microcontroller, commonly used in sensor nodes.

The three metrics considered (execution time, RAM requirement and code size) are extracted for a set of 17 ciphers. The cycle count is measured using the cycle accurate simulator `MSPDebug`. The RAM requirement is given by the stack usage for running the encryption key schedule, encryption and decryption operations. The stack consumption is computed by debugging the program execution on the `MSPDebug` simulator using `msp430-gdb`. Breakpoints are inserted at the beginning and at the end of the program execution and afterward the number of modified words in memory is computed. The data required to store the cipher state, the master key and round keys are not included in the RAM requirement. The code size is given by the `text` section of the binary file and is extracted using the `msp430-size` tool. The metric extraction is done automatically through Bash scripts and the results are exported into LaTeX tables similar to those used in the paper [75].

Analyzing the project source code, we inferred that the framework has some major drawbacks. Firstly, the RAM requirement given in the paper and on the project website is wrongly computed because the framework implementers assume that the `unsigned int` data type requires one byte instead of two on a 16-bit MSP430F1611 [348] microcontroller. Thus the RAM requirement provided in the paper is half of the actual value. Secondly, the library is not flexible at all and it does not allow easy addition of new devices or metrics. The provided library does not have a set of requirements that each implementation should follow and there is no common interface for assessing the performance of the implemented ciphers. Without a clear evaluation methodology, reference implementations that process one block at a time are compared with bit-sliced implementations that process several blocks in parallel. Thirdly, some implementations of the studied ciphers do not verify the test vectors (e.g. LBlock). We wrote a patch that fixes the identified issues and sent it to the authors of the project. The patch was applied to the public repository [192].

The project has the merit of being one of the first attempts to evaluate a set of lightweight block ciphers on an embedded device. It also contains a large collection of implementations of lightweight ciphers available for free.

2.2.2 eBACS Project

The eBACS (ECRYPT Benchmarking of Cryptographic Systems) [45] project aims at measuring the speed of a wide variety of cryptographic primitives on personal computers and servers. The developed framework, SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives), integrates features for measuring the execution time of hash functions (eBASH), stream ciphers (eSTREAM), authenticated ciphers (eBAEAD), and public-key systems (eBATS). It provides a large collection of cryptographic implementations. The open and free source code of the framework is written in C with inline assembly, Bash and Python.

The project web page provides information on how to submit new implementations as well as how to collect data for existing implementations using the framework. The requirements that the implementation of a cryptographic primitive has to fulfill in order to be evaluated using the framework are very well described and ensure a consistent evaluation of all implementations across all considered target platforms.

The framework allows benchmarking of C, C++, and assembly implementations.

It automatically compiles the source code using different compilers and compiler options. The cycle count metric is computed using inline assembly instructions for each of the supported platforms. Because execution time is the only metric extracted, the submitted implementations are optimized only for speed. The results are extracted for different input data lengths across all compilers and compiler options and saved in a database, which is merely a text file that contains a line for each implementation evaluated. A line consists of a set of entries, separated by spaces, that allows to identify the measurement conditions and the corresponding results.

The framework represents a first step towards consistent evaluation of cryptographic primitives. Thanks to the fair and clear evaluation methodology, it has been used as source of inspiration for other similar projects. One of the framework's strengths is given by the large number of computers with different architectures and characteristics used for the result collection process, while the main shortcoming is that it is able to extract only the execution time. The framework is used to measure the performance of authenticated encryption schemes in the context of the CAESAR competition [71].

2.2.3 XBX Project

The XBX (eXternal Benchmarking eXtension) project [378] is an extension of SUPERCOP that allows benchmarking of hash functions on different microncontrollers. The XBX framework is written in C, Perl and Bash and uses the same interfaces for the implemented algorithms and generated results as SUPERCOP. The hardware layer consists of XBD (eXternal Benchmarking Device) and XBH (eXternal Benchmarking Harness) that communicate with each other using either I^2C or UART and digital I/O lines. The XBH is connected to the PC running the XBS (eXternal Benchmarking Software) using the Ethernet port. Where more compilers are available, XBX retains the SUPERCOP capability to benchmark the same implementation using different compilers and compiler options.

Besides extending the eBASH capabilities to microcontrollers, XBX extracts two more metrics for the analyzed hash implementations: binary code size and RAM consumption. The code size is obtained through static analysis of the generated binary file. The RAM requirement is the sum of stack consumption and static RAM requirement obtained from the application binary. The cycle count values are subject to measurement errors because they are not extracted directly from the target devices, but from the XBH [379]. Most of the benchmarked algorithms are taken from SUPERCOP.

The XBX is the first project to uniformly measure the performance of software implementations of cryptographic primitives built for different embedded devices using the same evaluation methodology. The results given in the report [379] are gathered for eight different devices with 8-bit, 16-bit and 32-bit CPUs from all major vendors and they were used in the second round of the SHA-3 competition [246]. The project is not active anymore, but its source code is public [208].

2.2.4 ATHENa Project

The ATHENa (Automated Tool for Hardware EvaluatioN) project [359] aims at fair, comprehensive and automated evaluation of cryptographic cores developed using hardware description languages such as VHDL or Verilog. The goal of the framework is to spread knowledge and awareness about good performance evaluation practices in order to make the comparison of competing algorithms fairer and more comprehensive.

The open-source benchmarking environment is described in [133]. It is inspired from the eBACS project [45] and consists of a set of scripts written in Perl and Bash aimed at automated generation of optimized results for multiple hardware platforms. The metrics considered are area, throughput, and execution time, while the primary optimization target is throughput to area ratio.

The framework can be used under Windows or Linux operating systems and supports different target FPGA families from Xilinx, Altera, and Actel. It allows running all steps of synthesis, implementation, and timing analysis in batch mode and performs automated optimization of results aimed at one of the three optimization criteria: speed, area, and throughput to area ratio. The generated results can be exported in CSV, Excel and LaTeX formats.

During the SHA-3 contest [246] the tool played an important role due to the comprehensive results generated and published [165]. Besides having been used during the SHA-3 competition, the framework is ready for the evaluation of authenticated encryption candidates from the CAESAR competition [71] and preliminary results are available on the project website [359]. We note that although it provides comprehensive performance figures, it does not require revealing the source code. While this decision is meant to protect intellectual property, it narrows the transparency of the results.

2.2.5 ECRYPT II Project – Performance Evaluation on ATtiny45

During the ECRYPT II project, two papers [118, 26] presenting the performance evaluation of block ciphers and hash functions with applications in ubiquitous computing on an Atmel AVR ATtiny45 8-bit microcontroller were published. The implementations written in assembly language are available for free [119, 27]. Although the authors of the two papers formulate a list of common constraints to be able to compare the performance, some of the guidelines were not always followed.

The papers consider the following metrics: code size, RAM usage and execution time. A combined metric is computed as the product of code size and execution time divided by the block size for block ciphers. For hash functions the combined metrics are given by the product of code size and execution time and the product of RAM usage and execution time. For block ciphers the average energy consumption is computed by integrating the measured current consumption. The energy consumption for all studied ciphers is strongly correlated with the cycle count values.

The tools and methodology used to extract the main metrics are not described. Although the common interfaces used for the evaluation of the implementations are provided, no scripts to help with the metric collection process are provided. The use

	eBACS	ECRYPT II	BLOC	XBX	FELICS
Code size	✗	✓	✓	✓	✓
RAM	✗	✓	✓	✓	✓
Execution time	✓	✓	✓	✓	✓
AVR	✗	✓	✗	✓	✓
MSP	✗	✗	✓	✓	✓
ARM	✗	✗	✗	✓	✓
PC	✓	✗	✗	✗	✓
Evaluation scenarios	✗	✗	✗	✗	✓
Actively maintained	✓	✗	✗	✗	✓

Table 2.1: Comparison of software benchmarking frameworks.

of assembly language for the implementations of algorithms has the advantage of illustrating the lightweight aspects of the studied ciphers better than C implementations, but at the same time it limits the code portability.

2.3 Motivation

The lightweight designs published in the literature give different performance figures on different platforms and different evaluation conditions. The exact methodology used to extract the figures is often unclear. Considering that the performance figures are usually reported for different devices and that the measured operations and measurement conditions differ from paper to paper, it is very hard to use the given values to compare different designs.

The lack of comparative performance figures creates the need for a fair and consistent way of extracting performance figures for lightweight ciphers. The results obtained using the same assessment methodology can be used to compare different algorithms. Using the performance values, cipher designers can infer which design constructions are better on different architectures. At the same time, the results can help engineers to select the best cipher for a given use case.

While the first proposed lightweight ciphers were mainly geared for hardware efficiency, in the last years, we notice that the focus is moving to lightweight ciphers designed for efficiency in software (see Section 3.3). This new design direction for lightweight ciphers reinforces the need of reliable and accurate performance figures.

Table 2.1 summarizes the main characteristics of the software benchmarking frameworks described in Section 2.2 and FELICS. While many frameworks extract the three metrics considered (code size, RAM, and execution time), few frameworks support IoT devices such as the 8-bit AVR, 16-bit MSP, or the 32-bit ARM. Except for FELICS, none of those frameworks is actively maintained. A distinguishing feature of FELICS is that it uses the concept of *evaluation scenario* to benchmark an implementation in various use cases inspired from the real world.

2.4 Goals

FELICS was created to allow the comparison of software implementations of lightweight ciphers on different embedded devices commonly used in the IoT. Its key characteristics are:

Fair Evaluation. To ensure a fair evaluation a clear assessment methodology was formulated. The methodology indicates the requirements that each implementation should follow and how each metric is extracted for each supported device. Although sometimes the methodology can be considered restrictive, it has been formulated to ensure a fair evaluation for every implementation.

Consistent Evaluation. The same methodology is used to assess the performance of all implementations of a primitive type. Thus the evaluation is consistent across all the target embedded devices for all studied usage scenarios. The consistent evaluation allows easy comparison of the performance figures between similar implementations of ciphers. It also facilitates the correct ranking of the ciphers' implementations using different criteria.

Accurate Measurements. The framework provides accurate measurements. To achieve this goal, the tools used to extract the metrics and the measurement conditions are precise. The simulators are cycle accurate and the tools used for measurements on development boards are carefully calibrated.

Open Source. To increase the trust in the measurements, the framework source code is open. Anyone can analyze the source code, can detect and fix coding bugs, or even contribute to the tool development with new modules and features.

Comprehensive Results. The extracted metrics are very detailed and aim to help understanding how different parts of an algorithm's implementation are affecting the performance. Embedded software engineers can use the comprehensive results to select the best trade-offs for a specific use case.

Flexible. The framework uses a modular architecture that facilitates further development. FELICS is designed to allow future development of new modules for assessing other types of cryptographic primitives. It also allows integration of new target devices and metrics. The process of integrating a new cipher implementation is very easy and can be done following the methodology and requirements of the framework.

Automatic Evaluation. The framework is able to verify if an implementation follows the formulated requirements. It can automatically check if an implementation verifies the test vectors provided by an implementer for all target devices. The process of collecting the performance figures is suitable for batch processing. The user can extract the results for a given list of ciphers and for a given list of architectures.

AVR	MSP	ARM
avr-gcc 4.8.2	msp430-gcc 4.6.3	arm-none-eabi-gcc 4.8.2

Table 2.2: Compilers used to build the implementations for each target device.

2.5 Benchmarking Framework

2.5.1 Structure

FELICS is written in C with inline assembly, Bash and Python and is designed to work on Linux operating systems. It allows benchmarking of C and assembly implementations that follow a given set of requirements. The C programming language was selected because of its widespread adoption and portability. If we consider that usually the reference implementations of ciphers are provided in C language, then it is the natural choice to reach a wide group of users. Moreover, the framework can target multiple embedded devices used in the IoT context with a single implementation, which limits the development effort. FELICS also facilitates the benchmarking of highly optimized assembly implementations, which are platform dependent.

The usage scenarios are written in C, while the cipher implementations can be written in C or assembly. Each module has a makefile that can build an implementation for a given architecture and scenario using a given compiler optimization level. The framework contains a collection of Bash scripts that allow to fully automate the metric extraction process. Python scripts were used to perform operations that were too complicated or impossible to be done in Bash. FELICS is able to automatically generate the binary code, to check the implementation's correctness using the provided test vectors, and to extract the implementation metrics for the supported devices.

The current version of the framework includes a core module and four specialized modules for evaluating lightweight block ciphers, stream ciphers, authenticated encryption algorithms, and hash functions. Thanks to the modular structure depicted in Figure 2.1, FELICS can be easily extended with new modules capable to measure other primitives. Each module uses the services of the core module and provides it with scripts for batch processing.

2.5.1.1 Core Module

The core module is the heart of the framework and provides the tools necessary to collect the metrics for each of the supported devices. Each implementation is built automatically using different compiler optimization levels (`-O3`, `-O2`, `-O1` and `-Os`) and the metrics for each compiler optimization level are reported. The compilers used for each target device and the compiler versions are given in Table 2.2.

The complete list of used tools and tool versions organized by extracted metric for each supported device is given in Table 2.3. Since the framework is subject to

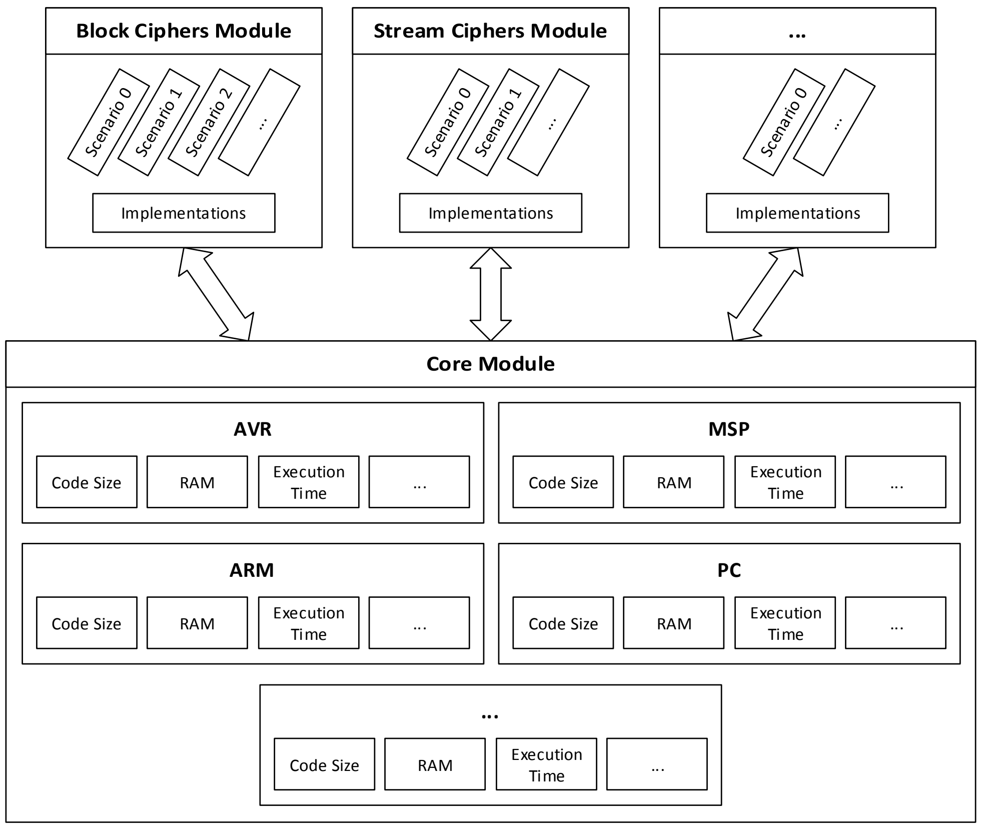


Figure 2.1: Modular structure of FELICS.

changes and improvements, we refer the reader to the FELICS web page [93] for updated information on compilers and tools.

The role of the core module is to facilitate a smooth integration of new target devices and extracted metrics. It allows the user to collect the results for one or more of the modules that are integrated in the framework.

In addition to the supported embedded devices, the module gives the possibility to debug and evaluate cipher implementations built for personal computers. This feature is mainly added to reduce the complexity of the implementation and integration process and to ease the task of users.

In order to achieve the described design goals, each module formulates a specific set of requirements that every implementation should follow. Even though the requirements create additional constraints and limit the possibility to benchmark highly optimized implementations (e.g. bit-sliced implementations), they ensure a fair and consistent evaluation across all implementations.

The core module has a configuration file, `conf.sh`, that provides the other modules with information about the tools used to extract the analyzed metrics for each target device. At the same time, each module implements the `get_results.sh`

AVR	MSP	ARM
	Code size	
avr-size 2.23.1	msp430-size 2.21.1	arm-none-eabi-size 2.24
	RAM	
simavr 1.5	MSPDebug 0.25	J-Link GDB Server V5.00l
avr-gdb 7.6.50	msp430-gdb 7.2	arm-none-eabi-gdb 7.6.50
	Execution time	
Avrora 1.7.117	MSPDebug 0.23	Arduino Due board

Table 2.3: Tools used to extract the metrics for each target device.

script that can be called from the core module to extract the performance figures in batch mode.

2.5.1.2 Block Ciphers Module

The module allows the evaluation of software implementations of lightweight block ciphers. Each implementation of a block cipher has to use the function prototypes from Listing 2.1 for the basic operations. In order to enable the framework to extract the metrics for each of the four operations, each operation has to be implemented in a separate C file. If the decryption key schedule is the same as the encryption key schedule, the decryption key schedule function has to be defined as an empty function. In the case the cipher does not have a key schedule, the encryption key schedule must be defined as a function that copies the master key into the round keys. The encryption and decryption operations are done in place to reduce the RAM consumption and the key should not be modified after running the key schedule.

```c
void RunEncryptionKeySchedule(uint8_t *key, uint8_t *roundKeys);
void Encrypt(uint8_t *block, uint8_t *roundKeys);
void RunDecryptionKeySchedule(uint8_t *key, uint8_t *roundKeys);
void Decrypt(uint8_t *block, uint8_t *roundKeys);
```

Listing 2.1: Required function signatures for block ciphers.

The block size, key size, round keys size and the number of rounds of the cipher have to be defined in the `constants.h` file. Other constants used by the implementation should be declared in the same header file, while the definitions can be added to constants.c or any other *.c file, except for the predefined C files. The constants can be stored in flash or RAM memory of the device and should be read with the macros provided by the framework. FELICS automatically checks if each implementation verifies the test vectors given in the test_vectors.c file.

The implementation information file, implementation.info, provides implementation details to the framework such that the common code and data to be considered just once when extracting the metrics. One can split an implementation into as many files as wanted if each implementation file is correctly listed in the implementation.info file. The implementation information file indicates if a key schedule is used for encryption and decryption.

A template cipher implementation and a file describing the module requirements is provided with the module to help users to integrate new implementations. The process of integrating an existing C or assembly implementation is thus very easy and consists in filling the functions of the template cipher with the implementation of an actual cipher as described in the README file.

The module contains three evaluation scenarios, but can be easily extended with new evaluation scenarios. The cipher operation (Scenario 0) evaluates the basic operations performed by a block cipher. In this scenario a block of data is encrypted and then decrypted using the provided test vectors. The communication protocol (Scenario 1) assumes the encryption and decryption of 128 bytes of data using the CBC mode of operation. This scenario is suitable for secure communication in the IoT context and considers the limitations of IEEE 802.15.4 [170] and ZigBee [393] protocols used in sensor networks. The challenge-response authentication protocol (Scenario 2) is created to evaluate the cost of authentication in the IoT context by using a block cipher in CTR mode to encrypt 128 bits of data. No key schedule is required because the cipher round keys are precomputed and stored in the device's flash memory.

Because the communication protocol and challenge-response scenarios assume the encryption of 128 bytes and 128 bits of unpadded data, respectively, the block size of the cipher in bits has to be equal to or a submultiple of 128.

2.5.1.3 Stream Ciphers Module

The performance figures of stream ciphers can be extracted for each stream cipher implementation that defines the functions described in Listing 2.2. The definition of each function has to be placed in a separate C file. The implementation of the encryption function must be able to process at least one byte. The encryption process is done in place to reduce RAM consumption. The cipher master key should not be modified after running the setup.

```
void Setup(uint8_t *state, uint8_t *key, uint8_t *iv);
void Encrypt(uint8_t *state, uint8_t *stream, uint16_t length);
```

Listing 2.2: Required function signatures for stream ciphers.

The cipher state size, key size and initialization vector size have to be defined in the `constants.h` file. The constants used by the stream cipher must be declared in `constants.h` file and defined in the `constants.c` or any other `*.c` file, except for the predefined `*.c` files. The implementer can choose to store the constants in flash or RAM and has to use the corresponding macro to read the constants. The test vectors used by FELICS to check the correctness of the implementation should be defined in the `test_vectors.c` file.

Integrating a new stream cipher implementation is very easy because the user is provided with a template implementation of a stream cipher and a file with implementation instructions. The implementer has just to fill the functions from the template cipher with the source code of the cipher according to the requirements described in the `README` file.

FELICS parses the `implementation.info` file to ensure the common source code and constants are counted only once in the extracted metrics. The implementation of each of the required functions can be split into several files provided that the implementation information is correctly given in the `implementation.info` file.

Two evaluation scenarios are implemented for this module, but new scenarios can be added at any time with minimal effort. The cipher operation (Scenario 0) is evaluated using the provided test vectors. The communication protocol (Scenario 1) is designed to secure the communication between wireless sensor nodes and consists in encryption of 128 bytes of data. Because the evaluation conditions are similar to the one used for block ciphers, these scenarios can also be used to compare the performance figures of block and stream ciphers.

2.5.1.4 Authenticated Ciphers Module

This module assesses the performance of authenticated ciphers that implement the functions specified in Listing 2.3. Each of the six functions must be implemented in a separate C file. This application programming interface (API) is designed to facilitate detailed measurements of the main structural components of an authenticated cipher. It was inspired by the lightweight submissions to the CAESAR competition [71]. Since the call for submissions did not impose strict requirements on the structure of authenticated ciphers, designers used various structures. Consequently, it was challenging to design an API that can easily accommodate implementations of algorithms that do not have exactly the same structure.

```
void Initialize(uint8_t *state, uint8_t *key, uint8_t *nonce);
void ProcessAssociatedData(uint8_t *state, uint8_t *associatedData, uint8_t lenght);
void ProcessPlaintext(uint8_t *state, uint8_t *message, uint8_t length);
void ProcessCiphertext(uint8_t *state, uint8_t *message, uint8_t length);
void Finalize(uint8_t *state, uint8_t *key);
void TagGeneration(uint8_t *state, uint8_t *tag);
```

Listing 2.3: Required function signatures for authenticated ciphers.

An implementer has to specify the structural properties of the algorithm in the `implementation.info` file such that the module can address the differences in the structure of various authenticated ciphers. One must indicate the common code and data in the implementation file to allow the module to add them a single time to the computed metrics.

The encryption and decryption are done in place to reduce the memory requirement. Most of the algorithms proposed in the CAESAR competition [71] can not process the input message in chunks of different sizes. Therefore, they impose that the same message lengths are used for both encryption and decryption. One can implement an authenticated cipher or port an existing implementation by following the steps described in the `README` file.

This module supports five usage scenarios. Scenario 0 is used to automatically test the correctness of an implementation using the provided test vectors. Different use cases specific to IoT communication protocols such as ZigBee [393] are evaluated in three scenarios. Scenario 1 describes a use case where a cipher encrypts and authenticates 128 bytes of data. In addition to the message, the cipher authenticates 128 bits of associated data (e.g. an IPv6 address). Scenario 2 assumes only encryption and authentication 128 bytes of data. In Scenario 3, the cipher is used only to encrypt 128 bytes of data. Finally, Scenario 4 describes a challenge-response protocol where 128 bits of associated data are authenticated.

2.5.1.5 Hash Functions Module

This module benchmarks implementations of hash functions that define the three functions described in Listing 2.4. Each of the three functions must be implemented in its own C file. The constants and code shared between these functions must be listed in the `implementation.info` file. The `README` file provides details on how to implement a hash function for this module starting from a template implementation. The update function must be implemented such that the resulting hash is the same regardless of how the input message is split into chunks.

```
void Initialize(uint8_t *state);
void Update(uint8_t *state, uint8_t *message, uint16_t length);
void Finalize(uint8_t *state, uint8_t *digest);
```

Listing 2.4: Required function signatures for hash functions.

The hash function module supports three evaluation scenarios and new scenarios can be easily added. The correctness of an implementation is verified in Scenario 0 using the given test vectors. Scenario 1 assesses the performance of an implementation that hashes messages of 16, 128, and 1024 bytes. The short message (i.e. 16 bytes) corresponds to a challenge-response protocol, while the long message (i.e. 1024 bytes) represents the maximum block size of a firmware update [63]. The common amount of data exchanged in an IoT communication protocol influenced the choice of the medium-sized message (i.e. 128 bytes). In Scenario 2, a hash function is used to

Characteristic	AVR	MSP	ARM
Model	ATmega128	MSP430F1611	Cortex-M3
CPU	8-bit RISC	16-bit RISC	32-bit RISC
Frequency (MHz)	16	8	84
Registers	32	16	21
Architecture	Harvard	von Neumann	Harvard
Flash (KB)	128	48	512
SRAM (KB)	4	10	96
EEPROM (KB)	4	–	–
Supply voltage (V)	4.5 – 5.5	1.8 – 3.6	1.6 – 3.6

Table 2.4: Key characteristics of the three microcontrollers used by FELICS.

generate a 16-byte message authentication code (MAC) using a 16-byte key.

2.5.2 Export Formats

The framework can export the extracted results for each scenario and target architecture in several formats in order to allow the user to analyze and post process the results. The supported formats are: raw data table, CSV file, XML file compatible with Microsoft Office Excel and LibreOffice Calc, MediaWiki table and LaTeX table. New formats can be easily added should the need arise. An archive with latest results in all mentioned formats is available for download on the FELICS web page [93]. On the same web page, a Python script for processing the CSV results can also be found. It allows the ranking of existing ciphers' implementations using the Figure of Merit (FOM) defined in Section 3.4, but it can be easily modified to compute other values of interest.

2.6 Target Devices

The IoT is populated by billions of devices that are equipped with a highly diverse and largely incompatible range of hardware platforms. In fact, the microcontroller population of the IoT is much more heterogeneous than the processor population of commodity computers, where the Intel architecture enjoys a market share of over 90%. Since there is no single dominating platform in the IoT, it is essential that a lightweight block cipher achieves consistently good performance on a variety of 8, 16, and 32-bit microcontrollers. It is also essential that a benchmarking framework is capable to collect implementation results from a wide range of platforms. Our framework supports the AVR ATmega128 [22] as example of an 8-bit architecture, the TI MSP430F1611 [348] as representative of a 16-bit platform, as well as the ARM Cortex-M3 [17] as example of a 32-bit RISC machine. However, the benchmarking framework can be easily extended to support further platforms. Table 2.4 gives the main characteristics of each target device.

2.6.1 8-bit AVR ATmega128 Microcontroller

The AVR ATmega128 [22] microcontroller manufactured by Atmel uses a CPU with a RISC architecture and an on-chip two-cycle multiplier. Most of the 133 instructions require a single cycle to execute. The rich instruction set is combined with the 32 8-bit general purpose registers (`R0 - R31`) with single clock access time. Six of the 32 8-bit registers can be used as three 16-bit indirect address register pointers (`X`, `Y` and `Z`) for addressing the data space. The instructions are executed within a two-stage, single-issue pipeline: while one instruction is executed, the next instruction is pre-fetched from the program memory. Therefore, one instruction is executed every clock cycle. The Arithmetic Logic Unit (ALU) operations are divided into three main categories: arithmetic, logic and bit manipulation functions. All 8-bit registers are directly connected to the ALU, allowing two independent registers to be accessed in one instruction executed in one clock cycle.

It has a modified Harvard architecture where program and data are stored in separate physical memory regions located at different physical addresses. The separate memories and buses for program and data maximize the performance and parallelism. The memory includes 128 KB of flash, 4 KB of SRAM and 4 KB of EEPROM. The data memory can be addressed using five different modes: direct, indirect, indirect with displacement, indirect with pre-decrement and indirect with post-decrement. An access to SRAM is performed in 2 CPU cycles.

Being among the best 8-bit microcontrollers in terms of power consumption when it entered the market, the Atmel ATmega128 provides a highly flexible and cost effective solution to many embedded control applications from building and home automation to medical and healthcare systems. It is working at supply voltages between 4.5 V and 5.5 V and has six different software-selectable power modes of operation.

2.6.2 16-bit MSP430F1611 Microcontroller

The MSP430F1611 [348] microcontroler produced by Texas Instruments has a CPU with RISC architecture and 16 16-bit registers. Four of the registers are dedicated to program counter, stack pointer, status register and constant generator, while the remaining 12 registers (`R4 - R15`) are general-purpose registers. The 52 instructions with three formats (dual operand, single operand, jump) and seven addressing modes (register, indexed, symbolic, absolute, indirect, indirect auto-increment, immediate) can operate on byte and word data. The register to register operations take one clock cycle. The number of clock cycles required to perform an instruction depends on the instruction format and addressing mode used.

The von Neumann memory of MSP430 has one shared address space for special function registers, peripherals, RAM and flash memory. It includes 48 KB of flash and 10 KB of SRAM. The flash memory is bit, byte and word addressable and programmable.

Designed for low-cost and low-power embedded applications, it requires a supply voltage between 1.8 V and 3.6 V and can reach a frequency of 8 MHz. It has one active mode and five software selectable low-power modes of operation. Typical applications include industrial control, sensor systems, and hand-held meters.

2.6.3 32-bit ARM Cortex-M3 Microcontroller

The Arduino Due board [15] uses the 32-bit Atmel SAM3X8 Cortex-M3 [129] RISC CPU that executes Thumb-2 instructions. The instruction set allows high code density and reduced program memory requirements. The processor has a three-level pipeline (instruction fetch, instruction decode and instruction execute) and 13 general purpose registers (`R0` - `R12`).

The Harvard memory architecture includes 512 KB of flash organized in two blocks of 256 KB and 96 KB of SRAM divided into two banks of 64 KB and 32 KB. The processor enables direct access to single bits of data in simple systems by implementing a technique called bit-banding. It supports two operating modes (thread and handler) and two levels of access to the code (privileged and unprivileged) enabling the implementation of complex systems without sacrificing security.

Specifically designed to achieve high system performance in power sensitive embedded applications, such as automotive systems, industrial control systems and wireless networking, the processor operates at a maximum frequency of 84 MHz. The recommended supply voltage ranges between 1.6 V and 3.6 V.

2.7 Metrics

The tree metrics considered can be extracted in batch mode for a list of implementations, usage scenarios and target devices using the `collect_cipher_metrics.sh` script. We added support for these metrics because they outline the lightweight characteristics of the evaluated implementations. Derived or secondary metrics such as power and energy consumption were not included in the initial release, mainly because they are closely related to the basic metrics.

Detailed and accurate results are generated for each operation required by the corresponding module separately and for the all operations together. The comprehensive results can be used by embedded software engineers to decide what cipher operations should be implemented for a particular device and application. Where cycle accurate and free software simulators of the target embedded devices exist, they are preferred to development boards because of usability reasons. While a software simulator can be downloaded and installed easily, a development board involves an acquisition and configuration cost. Next we describe how each metric is extracted for the considered target devices.

2.7.1 Code Size

The code size is measured in bytes and quantifies the amount of storage an operation occupies in the non-volatile memory (e.g. flash memory) of the target device. To extract the code size for each target device, the frameworks uses the GNU `size` tool, which lists the section sizes and the total size in bytes for a given binary file. The binary code size is given by the sum of the `text` and `data` sections. The `text` section of the binary file contains the code, while the `data` section stores global initialized variables, which are loaded from flash into RAM at run time. The `bss` section of the binary file is not considered since the framework forbids the utilization of global uninitialized variables. The code size of the `main` function, where all operations are put together, is not considered because it is the same for all studied ciphers.

The framework is able to determine the common parts using the implementation information file and considers them just once in the extracted code size value. Hence, FELICS encourages code reuse and the computed program footprint is accurate.

FELICS uses `avr-size`, `msp430-size` and `arm-none-eabi-size` to extract the code size for AVR, MSP and ARM, respectively. The exact version of each tool is given in Table 2.3. The code size extraction process is completely automated and can be done using the `cipher_code_size.sh` script for a given cipher implementation and a given evaluation scenario.

2.7.2 RAM

The RAM consumption is split into stack requirement and data requirement. The stack consumption gives the maximum value of the RAM used to store local variables and return address after interrupts and subroutine calls. The data requirement represents the static RAM and is given by the size of the constants stored in the target device's RAM. It includes the data specific to each scenario such as data to encrypt, master key, round keys or initialization vectors. The heap is not used at all because the framework does not permit any dynamically-allocated variables.

The static RAM consumption is computed from the `data` section of the binary file using GNU `size`. As in the case of code size, using the implementation information file, FELICS considers the global initialized variables just once when they are used in several operations. The stack consumption is measured using the appropriate `gdb` client and the target device simulator or development board. Before the function call for the measured operation, the stack is filled with a memory pattern. Then, at the end of the function's execution, the values in the stack area are compared with the memory pattern and the number of modified bytes gives the stack consumption. Hence, the measurement method takes into account the function arguments that are passed on the stack. The measured operation's return address is not considered since it is insignificant and the same for all ciphers on a given target device. The client and server tools used for computing the stack requirement are given in Table 2.3. The `cipher_ram.sh` script is able to extract the RAM requirement for a given cipher in a given usage scenario.

Another way to compute the stack requirement is to statically analyze the assembly instructions generated by the compiler and build the call graph for the measured function. For each entry in the call graph the maximum stack consumption is computed and stored. The stack usage of the measured function is given by the call path with the maximum stack requirements. This method is not able to solve recursive function calls and calls to functions from the standard C library. On the other hand, using the `gdb` client with a well tested simulator is less error prone than a tool developed from scratch.

2.7.3 Execution Time

The execution time measures the number of CPU clock cycles spent on executing a given operation. The metric is extracted using either cycle accurate software simulators of the target microcontrollers or development boards.

The execution time is computed as the absolute difference between the system timer's number of cycles at the end of the measured operation and at the beginning of the measured operation. To extract the number of cycles spent to execute the measured operations, FELICS simulates the cipher operation using the cycle accurate simulator `Avrora` [357, 356] for AVR and the cycle accurate simulator `MSPDebug` [108] for MSP. For ARM, the framework inserts additional C and assembly code to read the system timer's number of ticks at the beginning and at the end of each measured operation and then executes the program on an Arduino Due [15] board. The measurement process on the ARM board was carefully adjusted to obtain accurate and precise results. We draw attention to the fact that extracted values for ARM may vary depending how the C code is translated into assembly instructions and how data is aligned in memory for different usage scenarios. Information about the used simulators is provided in Table 2.3. The `cipher_execution_time.sh` script extracts the execution time for a given cipher implementation and scenario.

2.8 Summary

In this chapter, we introduced FELICS, a free and open-source benchmarking framework for fair and consistent evaluation of cryptographic primitives. It is primarily motivated by the lack of comparative performance figures for lightweight cryptographic algorithms measured from different embedded devices. Our aim is to increase the trust and transparency of results obtained by different algorithms and to ensure an independent environment for assessing the performance of new designs. FELICS facilitates the comparison of performance figures between different ciphers due to the consistent evaluation methodology.

Currently the framework is able to benchmark lightweight block ciphers, stream ciphers, authenticated encryption functions, and hash functions on three different embedded devices. The extracted metrics for each device and evaluation scenario are: binary code size, RAM consumption, and execution time. Thanks to its modular design, FELICS is very flexible and can be easily extended to benchmark new lightweight primitives, to extract new metrics, to collect the performance figures for other target devices, or to evaluate the implemented algorithms in new usage scenarios. The framework source code together with the source code of the implemented ciphers and the corresponding performance figures are available on a website [93].

FELICS borrows and improves concepts from previous frameworks and, at the same time, adds new ideas and features. The result is a better framework for fair and consistent evaluation of cryptographic primitives. To the best of our knowledge, FELICS is the first benchmarking framework to evaluate lightweight primitives for the IoT context in different usage scenarios. It also provides full transparency in the performance figures of assessed implementations by publishing the results and the corresponding source code on the project website [93].

Possible additional features include: addition of new modules to allow benchmarking of other cryptographic primitives (e.g. public-key algorithms), extraction of new metrics (e.g. energy consumption), or support for other embedded devices. Another direction for further development is to add support for development boards where software simulators are currently used. Since the framework is free and its source code is available, anyone interested can contribute to the tool development by implementing new features, reporting issues, and fixing bugs.

Since its initial release in March 2015, the framework has been constantly updated and improved. The feedback and comments received from the users of FELICS helped us to improve the framework as well as its online documentation. FELICS attracted implementers from around the world, which contributed more than 60 implementations of lightweight ciphers.

Chapter 3

Fair Evaluation of Lightweight Block Ciphers

Contents

3.1 Introduction

In this chapter we present a survey of lightweight block ciphers along with software benchmarking results obtained on embedded 8, 16, and 32-bit microcontrollers. We consider three metrics of interest: execution time, memory (i.e. RAM) requirements, and binary code size.

To ensure a fair and consistent evaluation, we used the FELICS framework introduced in Chapter 2. Following the spirit of the well-known and widely-used eBACS system [45], we made FELICS available to the cryptographic research community. Our benchmarking tool is "open" in various aspects. First, it is possible to upload implementations of new ciphers as well as new (i.e. improved) implementations of ciphers that are already included. Second, the tool was developed from the ground up with the goal of supporting a wide range of embedded platforms through both cycle-accurate instruction set simulation and actual measurements on development boards. Currently, our tool includes cycle-accurate instruction set simulators for

AVR ATmega and TI MSP430, as well as an ARM development board equipped with a Cortex-M3 processor. We use GCC for all these platforms, but other compilers could be supported as well. Third, our tool is also open with respect to the evaluation metrics. Currently, it can evaluate three basic metrics, namely execution time, RAM footprint, and binary code size. Other metrics can be derived thereof or are, at least, closely related. For example, the energy consumption of a block cipher executed on an embedded processor operating in a certain power mode can be estimated by the product of execution time, supply voltage, and average power dissipation. However, since our framework supports development boards, it could be extended to acquire more accurate energy figures by simply measuring the processor's power dissipation while it executes a cryptographic algorithm.

Our benchmarking toolsuite accepts source codes written either in "pure" ANSI C or in C with inlined assembly sections for the three processor architectures mentioned above. In this way, the toolsuite supports various trade-offs between performance and portability. At one end of the spectrum are highly-optimized implementations for which the complete encryption/decryption function consists of hand-crafted assembly code. Assembly programming allows one to fully exploit the architectural features of a processor and, in this way, reach peak performance. The speed-up due to the integration of hand-crafted assembly code is especially pronounced if a cipher performs a large number of operations that are significantly less efficient in C than in assembly language (e.g. multi-word arithmetic, certain bit manipulations). Benchmarking results obtained from carefully-optimized assembly implementations played an important role in the evaluation of candidates for cryptographic standards like the AES [250] and SHA-3 [246], and this will also be the case for future standardization activities in the area of lightweight cryptography for the IoT [247]. However, an implementation of a cipher written in assembly language is architecture-dependent and, consequently, not portable. At the opposite end of the performance-portability spectrum are "pure" C implementations, which are highly portable but, in general, less efficient than their hand-crafted assembly counterparts.

While the importance of benchmarking hand-optimized assembly implementations is out of dispute, we argue that it makes also sense to benchmark portable C implementations of lightweight ciphers. Our argument is twofold and based on the specific properties and constraints of the IoT. First, it has to be noticed that there is no single dominating hardware platform in the IoT, in contrast to the "conventional" Internet of commodity computers, where the Intel architecture has a market share of over 90%. In fact, the IoT is populated by billions of heterogenous devices with largely incompatible processors and different operating systems. Supporting a large number of platforms with optimized assembly code is tedious and error-prone since, for each processor architecture, a separate code base needs to be written, tested, debugged, and then maintained. In the light of ever-increasing time-to-market pressure, cryptographic engineers may value the portability of C code more than the performance of assembly code. Our second argument is related to the steadily increasing research interest in lightweight ciphers with new designs being published (almost) every month. Implementations written in C often serve as proof-of-concept in the design phase of a new primitive to explore e.g. different candidates for a round function. Benchmarks generated from C implementations allow cipher designers to quickly evaluate the impact of various design options (e.g. round function, number of rounds) on execution time, RAM footprint and code size. In this way, designers can already assess in an early phase of the design cycle how a new primitive may compare with the state of the art.

We report detailed benchmarking results for a total of 19 lightweight block ciphers, namely the AES [250], Chaskey [244], Fantomas [153], HIGHT [167], LBlock [386], LEA [166], LED [157], Piccolo [321], PRESENT [58], PRIDE [9], PRINCE [62], RC5 [296], RECTANGLE [391], RoadRunneR [35], Robin [153], Simon [36], Sparx [106], Speck [36], and TWINE [344]. Our rationale for selecting exactly the mentioned 19 ciphers is twofold; first, each of these candidates has some special property or feature that makes it interesting for applications in the IoT. Second, they cover a wide range of different design strategies and approaches. Our evaluation considers two application scenarios or use cases; the first relates to the encryption of messages transmitted in a Wireless Sensor Network (WSN) and the second is a simple challenge-response authentication protocol with applications in e.g. object identification or access control. To accommodate the different requirements of these application scenarios, we evaluated at least two versions of most of the 19 ciphers, including a low-memory variant and a speed-optimized variant. The former can be seen as a "minimalist" implementation that favors low memory footprint and small code size over performance. On the other hand, the second implementation includes certain optimizations that increase code size and/or memory footprint (e.g. partial loop unrolling, use of small lookup tables) with the goal of improving performance. Roughly half of the implementations were written from scratch by us, whereby we put a comparable effort into optimizing each cipher to ensure a consistent and fair evaluation. The other half was either taken from other open-source projects or contributed by the designers of the algorithms or by volunteers; in all these cases we carefully reviewed the source codes and further optimized them whenever possible. In this way, we tried to minimize the impact of varying programming skills and experience. Most of our implementations are faster or on par with the best execution times reported in the literature on the three platforms we consider. Therefore, the implementations form a solid code base for the benchmarking of lightweight block ciphers.

3.1.1 Our Contributions

We survey a total of 19 lightweight block ciphers and study, in particular, their suitability for software implementation on resource-restricted devices. This set of ciphers covers a wide range of different design principles and includes a number of recent proposals with interesting properties, e.g. Simon/Speck [36], Robin/Fantomas [153] and Sparx [106]. We collected between two and up to 24 implementations of each cipher to account for different trade-offs between execution time, RAM footprint, and code size. For nine out of the 19 ciphers we have not only C implementations, but also optimized assembly code for the three platforms we consider. In total, our repository includes over 250 implementations, of which we developed roughly half from scratch. The source code of all our implementations is available under GPL and can be downloaded from the CryptoLUX wiki using the given link[1].

Third, we report detailed performance, RAM footprint, and code size figures of the 19 ciphers, which we generated with the help of our benchmarking toolsuite. In addition, we define two typical usage scenarios that aim to resemble security-related operations commonly performed by real-world IoT devices. The results we obtained shed a new light on the relative efficiency of lightweight block ciphers because:

1. some of our implementations are significantly faster or smaller than that of other survey and benchmarking efforts, and

2. we include a few designs that have been published only very recently.

Since lightweight cryptography is a rapidly progressing area of research, we also maintain a web page [93] with the most recent results, which gets automatically updated when users provide new implementations. Our framework allows the user to define a custom *Figure of Merit (FOM)* according to which an overall ranking of ciphers can be assembled. The FOM metric can assign different weights to execution time, RAM footprint, and code size, and may even consider (cryptanalytic) security aspects.

Our results allow one to infer some interesting relations between cipher design principles and performance figures, and, in this way, contribute to a better understanding of how to design and implement lightweight block ciphers.

3.2 Benchmarking Framework

Most papers introducing a new block cipher report some kind of results of some kind of performance evaluation on some kind of platform using some kind of implementation. These results are then used by the authors to claim that the proposed cipher has some kind of "advantage" over existing ciphers or compares "favorably" with the state of the art. However, such comparisons are little meaningful in the real world since it is not easily possible to take differences in the characteristics of the target platforms or differences in the simulation/measurement conditions into account. Consequently, it is difficult to assess the relative efficiency of the numerous proposals for lightweight ciphers in a fair and consistent fashion. This motivated us to develop FELICS (see Chapter 2), which allows for a unified evaluation of a large number of candidates by collecting accurate and comprehensive results for execution time, RAM footprint, and code size. The toolsuite is currently able to extract these metrics from implementations for 8-bit AVR, 16-bit MSP430, and 32-bit ARM Cortex-M processors, but other platforms could be supported as well. We make the full source code of the benchmarking framework available under GPL to facilitate its acceptance in the cryptographic research community and to maximize transparency in the evaluation of lightweight block ciphers.

[1]All results reported in this chapter are based on version 1.1.20 of the FELICS framework, which can be downloaded from `https://www.cryptolux.org/index.php/File:FELICS.zip`

As stated in the previous section, we consider benchmarking results obtained with C implementations to be useful for cipher designers and for cryptographic engineers who prefer portable C code over platform-optimized assembly code. Since cipher designers tend to write reference implementations in ANSI C, the effort of evaluating a new cipher boils down to adapting the C source code to meet the requirements of the framework. However, benchmarks generated with C implementations do often not reflect the full potential of a lightweight cipher because ANSI C can not efficiently express multi-word arithmetic operations and certain bit manipulations. In addition, the quality of the C compiler (i.e. its ability to apply sophisticated optimizations) may impact the relative performance of lightweight ciphers. To mitigate these issues, and to serve cryptographic engineers who are primarily interested in high speed rather than high portability, the toolsuite supports the benchmarking of hand-optimized assembly implementations for the three considered platforms. We had both C and assembly implementations available for nine of the 19 lightweight ciphers we benchmarked; the remaining 10 ciphers were evaluated using C source codes only. In total, we analyzed more than 250 different C and assembly implementations of 19 lightweight block ciphers. We make the full source code of all implementations available under GPL to ensure the reproducibility of our results and, in this way, increase the transparency and trustability of our evaluation process.

3.2.1 Usage Scenarios

Besides the evaluation of the four basic operations of a block cipher (i.e. encryption, decryption, encryption key schedule, and decryption key schedule), the benchmarking framework also supports more advanced forms of assessment based on usage scenarios. A usage scenario should implement some common security service with practical relevance for the IoT and utilize the basic cipher operations. In this way, it is possible to obtain realistic benchmarking results that are meaningful in the real world. The results reported in Section 3.4 are based on two simple usage scenarios, which we describe below. Further usage scenarios can be easily added thanks to the modular design of the benchmarking framework.

3.2.1.1 Scenario 1: Communication Protocol

This scenario covers the need for secure communication between two IoT devices such as two sensor nodes in a WSN. It is assumeed that the sensitive data is encrypted and decrypted using a lightweight block cipher in CBC mode of operation. Since standard communication protocols for the IoT, such as IEEE 802.15.4 [170] and ZigBee [393], are characterized by low transmission rates and small packet sizes, we assume the plaintext to have a length of 128 bytes (i.e. 1024 bits) in this scenario. There is no need for a padding scheme because the length of the plaintext is a multiple of both 64 and 128 bits, which are the two block sizes we consider in this chapter. Furthermore, we assume that the master key resides in RAM and that the round keys (obtained through the operation for key schedule) are also kept in RAM for later use by the encryption or decryption operation. The plaintext and initialization vector for CBC mode shall also be in the device's RAM at the beginning of the scenario. In order to reduce the RAM footprint, the encryption is performed in place, which means the plaintext gets overwritten by the ciphertext (and vice versa for decryption). However, the key schedule does not modify the master key.

3.2.1.2 Scenario 2: Challenge-Response Authentication

This scenario is inspired by a simple authentication protocol where an IoT device proves that it is in possession of a secret key by encrypting a challenge using a block cipher. In real-world settings, the IoT device can, for example, be an RFID tag (see e.g. [123]) or a smart card. In this scenario we assume that a lighteight block cipher is used in CTR mode to encrypt 128 bits of data. The device has the full round key stored in flash memory, which means there is no need to store the master key and also no key schedule operation has to be performed. Both the 128-bit plaintext to be encrypted and the counter value are held in RAM at the beginning of the execution. In order to reduce the RAM footprint, the encryption is done in place, i.e. the plaintext gets overwritten by the ciphertext.

3.3 Analyzed Ciphers

Since our aim is to contribute to a better understanding of the relation between basic design methodologies for lightweight ciphers and the resulting software performance on resource-limited IoT devices, we selected 19 ciphers that represent a wide variety of design approaches based on Substitution-Permutation Networks (SPNs) and Feistel Networks (FNs). A classical example of an SPN is the AES [250, 98], but other designs for the S-box and the linear layer are possible, as demonstrated by PRESENT [58], Robin, and Fantomas [153]. The overall structure of an SPN-based cipher can also vary while still maintaining a round function consisting of an S-box layer and a linear layer: LED [157] adds key material every four rounds only, while PRINCE [62] implements a property called α-reflection, which minimizes the overhead for decryption on top of encryption. Furthermore, it is also possible to build an SPN using only modular Addition, Rotation, and XOR (ARX), as was done by the designers of SPARX [106]. An FN, on the other hand, can be designed by utilizing a small SPN as the Feistel function, as in LBlock [386] and Piccolo [321], or with simple arithmetic and logical operations, as in SIMON [36] and ARX designs like HIGHT [167], RC5 [296], and SPECK [36]. These operations may be data-dependent like in RC5. A variant of the FN is the Generalized FN, which uses more than two branches. The way the branches are mixed at the end of each round can consist of a simple rotation (HIGHT) or a dedicated permutation optimizing diffusion (TWINE [344], Piccolo). A high number of branches allows the use of very simple Feistel functions like in TWINE and HIGHT.

Besides representing a wide variety of different design approaches, most of the 19 lightweight ciphers we selected for our evaluation have a certain property or feature that makes them particularly interesting for use in the IoT. We intentionally did not restrict our selection to software-oriented ciphers and included some designs that were developed for efficiency in hardware, e.g. Piccolo, PRESENT, and PRINCE.

Cipher	Year	Block size	Key size	Round key size	Rounds	Security level	Type	Target
AES	1998	128	128	1408	10	0.70	SPN	SW, HW
Chaskey	2014	128	128	0	8/16	0.87/0.43	Feistel	SW
Fantomas	2014	128	128	0	12	NA	SPN	SW
HIGHT	2006	64	128	1088	32	0.81	Feistel	HW
LBlock	2011	64	80	1024	32	0.72	Feistel	HW, SW
LEA	2013	128	128	3072	24	0.63	Feistel	SW, HW
LED	2011	64	80	0	48	NA	SPN	HW, SW
Piccolo	2011	64	80	864	25	0.56	Feistel	HW
PRESENT	2007	64	80	2048	31	0.84	SPN	HW
PRIDE	2014	64	128	0	20	NA	SPN	SW
PRINCE	2012	64	128	192	12	0.83	SPN	HW
RC5*	1994	64	128	1344	20	0.80	Feistel	SW
RECTANGLE	2015	64	80/128	1664	25	0.72	SPN	HW, SW
RoadRunneR	2015	64	80/128	0	10/12	0.5/0.58	Feistel	SW
Robin/Robin⋆	2014	128	128	0	16	1/NA	SPN	SW
SIMON	2013	64	96/128	1344/1408	42/44	0.71/0.70	Feistel	HW, SW
SPARX	2016	64/128	128	1600/4224	24/32	0.62/0.68	Feistel	SW
SPECK	2013	64	96/128	832/864	26/27	0.73/0.74	Feistel	SW, HW
TWINE	2011	64	80	1152	36	0.64	Feistel	HW, SW

* We use RC5 with increased number of rounds, RC5-20.

Table 3.1: Overview of the 19 lightweight block ciphers considered in this evaluation. Block, key and round key sizes are expressed in bits. Security level is the ratio of the number of rounds broken in a single key setting to the total number of rounds.

The device population of the IoT is very heterogenous and shows extreme differences in terms of computational capabilities and resources. Some devices are so constrained that cryptographic operations can only be implemented in hardware (e.g. RFID tags), while other devices are powerful enough to run cryptographic software at acceptable speed. Since all these devices should be able to interact and communicate securely with each other, they have to use one and the same cipher. In order to be suitable for the IoT, a lightweight block cipher needs to be efficient in both hardware and software. Thus, it makes sense to evaluate the software performance of hardware-oriented ciphers and vice versa. In the following, we give an overview of the 19 lightweight ciphers we selected for benchmarking and describe how they can be implemented in software. The main characteristics of the candidates are summarized in Table 3.1.

AES. The AES is standardized by NIST and the by far most-widely used block cipher today. It has an SPN structure with an internal state of 128 bits represented in the form of a (4×4)-byte matrix. The `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey` functions operate on the cipher's state [250, 98]. To date, the best single-key cryptanalysis of AES-128 is a meet-in-the-middle attack on seven rounds out of ten [100]. Size-optimized implementations of the AES put the S-box and the round constants in lookup tables since they occupy just slightly more than 256 bytes. The source code of our size-optimized implementation mostly follows the cipher pseudocode on all three considered architectures. Since T-tables are very large (4 KB for either encryption or decryption), we did not include such implementations.

Chaskey. The Chaskey cipher is based on the π permutation of the Chaskey MAC algorithm [244] that is currently considered for standardization by ISO/IEC. Said π permutation is a generalized FN and uses ARX operations on 32-bit words. The cipher has an Even-Mansour structure, which means there is no key schedule but the master key is simply XORed to the internal state before and after π is applied. Chaskey-LTS (Long Term Security) has twice as many rounds as Chaskey and is recommended as a fallback in the case of cryptanalytic breakthroughs. Currently, the best attack against Chaskey is a differential-linear attack on seven out of eight rounds [214]. We benchmarked the C implementation provided by the designers, which is straightforward thanks to the simple structure of the cipher. In addition, we developed implementations in assembly language from scratch. The execution times of both can be improved by unrolling several rounds at the cost of larger code size.

Fantomas. Fantomas is a 128-bit cipher belonging to the family of LS-designs [153]. Its linear layer consists in the parallel application of so-called “L-boxes,” while the S-box is designed to simplify the implementation of masking, a countermeasure against Differential Power Analysis (DPA). There is no key-schedule; the master key is simply added in every round. At the time of writing this chapter, there was to our knowledge no attack against Fantomas. A software implementation of Fantomas usually combines lookup-table based L-boxes with bit-sliced S-boxes, which are computed using a Feistel structure. Storing the four 512 B L-boxes in RAM instead of flash improves the execution time by a quarter on AVR and ARM. Our implementations are based on the C source code provided by the designers.

HIGHT. The lightweight cipher HIGHT is a generalized FN with an ARX structure. More precisely, the Feistel functions perform only logical XOR and bitwise rotations. The output of the Feistel functions is combined with the other branches using either XOR or addition modulo 2^8 [167]. An impossible differential attack breaks 26 out of 32 rounds of HIGHT [268]. All implementations we benchmarked follow closely the specification from [159], which modifies the design of the original paper [167]. The 128 7-bit δ constants are either computed when the key-schedule function is called or precomputed and stored in flash or RAM. An entirely unrolled version with inlined auxiliary round functions $\mathtt{F}_0$ and $\mathtt{F}_1$ requires only half of the cycles of the reference implementation. When implemented in assembly language, the execution time decreases by 50% on MSP and by 10% on AVR and ARM, respectively.

LBlock. LBlock is an FN with 32 rounds. The Feistel function consists of a logical XOR with the round subkey, a substitution layer of eight different S-boxes, and a permutation of eight nibbles. Furthermore, the content of one of the branches is rotated by eight bits in each round. The chosen design trade-offs between security and performance led not only to hardware efficiency but also software efficiency [386]. To date, the best cryptanalytic result is obtained through an impossible differential attack against 23 out of 32 rounds [65]. The benchmarked LBlock implementations follow the specification from [386]. Optimization strategies include performing operations on 8, 16 or 32 bits when possible, storing the S-boxes in flash or RAM, and unrolling the loops. The best execution time on ARM is achieved by the fully-unrolled implementation using 32-bit operations, with the S-boxes stored in RAM.

LEA. The block cipher LEA uses a generalized FN with four 32-bit branches [166]. Designed for high-speed software encryption on 32-bit platforms, the cipher can be efficiently implemented in hardware as well. The designers mention a boomerang attack against 15 rounds, which is, to our knowledge, the best cryptanalytic result to date. The benchmarked assembly implementations are based on three different optimization strategies: fast execution time, small code size, and a trade-off between speed and size. These optimizations are facilitated by LEA's simple structure requiring only 32-bit operations.

LED. The AES-based cipher LED is aimed at very compact hardware implementation while maintaining reasonable performance in software. It represents the state by a (4×4)-nibble matrix and uses similar round transformations as the AES, except that they are nibble-oriented. A distinguishing characteristic of LED is the absence of a key schedule; the round keys are simply replaced by a part of the master key [157]. To the best of our knowledge, there are no attacks on LED-80. However, there is a differential attack that covers 16/32 rounds of LED-64 and 24/48 rounds of LED-128 [237]. The structural attack breaking 32/48 rounds of LED-128 proposed in [107] is unlikely to be adaptable to LED-80. Our LED implementation combines the `SubCells`, `ShiftRows`, and `MixColumnsSerial` operations into a table lookup to reduce execution time.

Piccolo. Piccolo has a generalized FN structure with four 16-bit branches. To improve diffusion, Piccolo uses a byte permutation between rounds. Piccolo's Feistel function consists of two S-box layers separated by a diffusion matrix [321]. The currently best attack against Piccolo-80 is a meet-in-the-middle attack on 14 rounds, which was presented by the designers. Our Piccolo implementation follows closely the description provided in [321]. The arithmetic in $GF(2^4)$ uses only XORs and two small lookup tables for multiplication by two and three. Both the S-box and the key schedule constants are stored in lookup tables. No specific loop unrolling is applied.

PRESENT. PRESENT has an SPN structure and comes with a bit-oriented permutation layer. The nonlinear layer is based on a single 4-bit S-box that was designed for efficiency in hardware [58]. A truncated differential attack against 26 out of 31 rounds of PRESENT is described in [56]. Since the S-box is quite small, a lookup table is used in all our implementations. However, its combination with a bit permutation over a 64-bit word is difficult to optimize without introducing extremely large lookup tables (up to 1 MB for decryption). The size-optimized implementation resembles the cipher's pseudocode and was taken from [74]. In general, the bit-oriented design of PRESENT makes C implementations very slow unless one can afford huge lookup tables. Our assembly implementations take advantage of bit-manipulation instructions that the target devices support. On AVR, the assembly implementation is around 12 times faster than the C counterpart, while the MSP assembly version is even 19 times faster than the C code.

PRIDE. The block cipher PRIDE is an SPN with a strong linear layer and a bit-sliced S-box, which are optimized for 8-bit microcontrollers [9]. It uses the so-called FX construction with the same key for pre- and post-whitening and a different key as basis for the round keys. A differential attack on 19 out of 20 rounds is described in [387]. The designers contributed a C implementation using only 8-bit operations. PRIDE's simple key schedule can be performed on the fly to reduce the RAM requirements at the cost of execution time. The S-box requires only bitwise operations, and also the linear layer consisting of four transformations (one for every 16 bits of the state) can be implemented efficiently in software.

PRINCE. Similar to PRIDE, PRINCE is an FX construction, whereby the first two subkeys are used as whitening keys, while the third subkey is the 64-bit key for a 12-round SPN called PRINCE$_{core}$. PRINCE introduced the α-reflection property: encryption with a given key corresponds to decryption with a related key [62]. To date, the best cryptanalytic result is a multiple differential attack on ten out of the twelve rounds [72]. We implemented PRINCE as described in the original paper [62, 72]. The optimization strategies we considered include the use of 8, 16, 32, and 64-bit operations where possible and different amounts of loop unrolling. We obtained the best performance with fully unrolled implementations based on 8-bit operations for AVR and 16-bit operations for MSP. On ARM, the best execution times were achieved using a partially unrolled version with 32-bit operations.

RC5. RC5 is an FN that uses data-dependent rotations [296]. Though RC5 was designed before lightweight ciphers became popular, it is obviously suitable for resource-constrained devices, which is confirmed by its widespread use in sensor networks [273]. The block and key sizes, as well as the number of rounds, can be chosen freely. We use RC5-32/20/16, i.e. a version of RC5 that operates on two 32-bit words with a total of 20 rounds (40 half-rounds) and a 16-byte key. The number of rounds was chosen so as to have a security margin of 0.80. RC5-32/12/16 can be attacked using differential cryptanalysis as demonstrated in [51]. This attack can be extrapolated to 18 rounds, but would require almost the full codebook (2^{64} ciphertexts). RC5 was implemented by slightly adapting the reference code provided in [296]. Because of its elegant and simple design, there are not many possibilities for optimization. To explore different trade-offs, we fully unrolled the cipher's operations and precomputed the encryption-key-schedule array `S` to store it in flash or RAM.

RECTANGLE. The block cipher RECTANGLE is an SPN that can be efficiently implemented in both hardware and software thanks to its bit-sliced structure [391]. The nonlinear layer applies a 4-bit S-box to each column of the state, which is represented as a (4×16)-bit matrix, while the linear layer rotates each row by a different amount. A differential attack that covers 18 out of 25 rounds is described by its designers. RECTANGLE was implemented in C and assembly by its designers using different optimization strategies. The bit-sliced S-box is relatively fast in software because it uses only bitwise operations. On the other hand, the simple linear layer consists of three rotations of 16-bit words by 1, 12, and 13 bits, which can be efficiently implemented on 8, 16, and 32-bit architectures.

RoadRunneR. RoadRunnerR has an FN structure that can be efficiently implemented on 8-bit microcontrollers in a bit-sliced fashion [35]. The Feistel function is an SPN, which consists of four 4-bit S-box layers, three linear layers, and three key additions. There exists a high-probability truncated trail covering five rounds of RoadRunneR, which can be exploited to attack a 7-round variant of RoadRunneR-128 [388]. RoadRunneR can be easily implemented thanks to its simple structure designed for 8-bit microcontrollers. The Feistel function consists of a bit-sliced S-box and a linear layer; they use only bitwise operations and rotations of 8-bit values by 1 bit and are, thus, very efficient. The round keys can be computed on the fly to reduce the RAM requirement.

Robin. Robin is a 128-bit block cipher similar to Fantomas, but its "L-boxes" are involutions. The lookup table-based diffusion layers and the structure of the S-boxes makes this family of ciphers good candidates for Boolean masking in bit-sliced software implementations [153]. There exists a set of weak keys of density 2^{-32} for this cipher, which, if used, leads to an attack on the full primitive [210]. In response to the so-called invariant subspace attack [210], the designers of Robin proposed Robin* [176], in which the 8-bit round constant is replaced by a 128-bit round constant. Robin was implemented in different ways that are based on the C code provided by its designers. The two L-boxes are stored in flash or RAM, while the S-box layer is computed at each round using the Feistel structure. Robin* requires more memory and is also slower than the original Robin due to expensive derivation of the 128-bit round constants.

SIMON. SIMON uses an FN structure with a simple round function performing bitwise XOR, bitwise AND and circular left shifts. It is optimized for high performance in hardware implementations, but achieves excellent results in software as well [36]. Differential attacks on 30 out of 42 rounds of SIMON-64/96 and on 31 out of 44 rounds of SIMON-64/128 are presented in [80]. Optimized implementations of SIMON written in assembly (for AVR and MSP) and C (for ARM) were provided by its designers. The very simple structure of SIMON enables various trade-offs between code size and execution time by combining a different number of rounds in one loop iteration.

SPARX. The block cipher SPARX is an SPN designed on basis of the recently introduced Long Trail Strategy (LTS), which allows the use of a large and relatively weak S-Box rather than a small and strong one. The ARX-based S-box consists of one unkeyed SPECK-32 round, while the linear layer is inspired from that of NOEKEON [97]. Its authors described an integral attack based on Todo's division property covering 15 out of 24 rounds of SPARX-64/128 and 22 out of 32 rounds of SPARX-128/128 [106]. SPARX can be implemented using various optimization strategies thanks to its simple and flexible structure. We explored different trade-offs between execution time and code size by rolling/unrolling the rounds of a step function and performing one or two step functions at once. For a detailed description of SPARX and its implementations, we refer the reader to Chapter 4.

SPECK. SPECK is designed to achieve excellent results in hardware and in software, especially when executed on resource-constrained microcontrollers. It uses a Feistel structure in which both branches are modified at each round using bitwise XOR, modular addition, and circular shifts in both directions [36]. The best cryptanalytic results against SPECK-64/96 and SPECK-64/128 are differential attacks targeting 19 and 20 rounds out of 26 and 27, respectively [332]. SPECK has a very simple round function that is extremely fast and takes just a few bytes of code. The optimized implementations of SPECK were written in assembly (AVR and MSP) and C (ARM) and provided by the designers. Depending on the optimization goal, one or several round functions can be unrolled to improve the execution time at the cost of a minor increase in code size.

TWINE. TWINE is a generalized FN with 16 branches. The Feistel function simply consists of a key addition and the application of a 4-bit S-box. The linear layer is a nibble permutation with much higher diffusion than a nibble rotation as used for example in HIGHT. The cipher's design aims at small footprint in hardware implementations and small ROM/RAM consumption in software [344]. The best attack on TWINE-80 is a multi-dimensional zero-correlation linear attack on 23 out of 35 rounds [376]. TWINE is a very simple cipher so that the speed-optimized implementation is only marginally larger than the size-optimized one. It uses 4-bit branches which, in the authors' implementation [344], reside in separate bytes (so that the entire state is twice as large). We wrote a size-optimized implementation from scratch. Both implementations are small enough to run on all platforms.

3.4 Results

In this section, we firstly describe our evaluation methodology, including the Figure of Merit (FOM) we developed to rank the candidates, and then we present and discuss the benchmarking results of 19 ciphers in the two scenarios described in Section 3.2.1. Block sizes of 64 bits were used when available, otherwise 128 bits were used. We only evaluated cipher versions with a key length of at least 80 bits, which we consider the minimum security level acceptable for IoT applications.

3.4.1 Methodology

At the time of writing this chapter, our repository contained between two and 35 implementations for each cipher, and more than 250 in total. We benchmarked all of them on each of the three devices in each scenario. It is possible to order the implementations according to their execution time, RAM footprint, or code size in any particular scenario on any device and we maintain a separate interactive web page [93] where all these ordering options can be chosen. We have aggregated the data by the following principles, which seem to be the most interesting ones:

- In Scenario 1, we implemented the full encryption and decryption including key schedule. Then, for each implementation i and device d, we calculate the performance parameter $p_{i,d}$. The value $p_{i,d}$ aggregates the three metrics $M = \{$ execution time, RAM consumption, code size $\}$ as follows:

$$p_{i,d} = \sum_{m \in M} w_m \frac{v_{i,d,m}}{\min_i(v_{i,d,m})}, \tag{3.1}$$

  where $v_{i,d,m}$ is the value of metric m for implementation i on device d; w_m is the *relative weight* of metric m and $\min_i(v_{i,d,m})$ represents the minimum value of the metric m from all considered implementations of all considered ciphers on the same device d. For each cipher and each device we set $w_m = 1$ (the framework also allows one to choose other weights for the metrics; for example the results in Table 3.3 are computed using a higher weight for execution time than for RAM footprint and code size) and select the implementation with the smallest $p_{i,d}$. Finally, for each cipher and the selected set of implementations i_1, i_2, i_3 (one for each device) we calculate the Figure of Merit (FOM) value as the average performance value over the three devices.

$$\text{FOM}(i_1, i_2, i_3) = \frac{p_{i_1,AVR} + p_{i_2,MSP} + p_{i_3,ARM}}{3} \tag{3.2}$$

  Then, we sort the ciphers according to their FOM value (Table 3.2).

- In Scenario 2, we also select for each cipher and device the best implementation. First, we select the most balanced implementation using Equation (3.1) and $w_m = 1$ (Table 3.4). In Table 3.5 we calculate $p_{i,d}$ a bit differently:

$$p_{i,d} = \sum_{m \in \{\text{code, RAM}\}} w_m \frac{v_{i,d,m}}{\max_i(v_{i,d,m})}, \tag{3.3}$$

  where $\max_i(v_{i,d,m})$ is the maximum value of flash memory (for the code size metric) or RAM (for the RAM metric) available on device d (see Section 2.6).

Thus, we essentially measure the fraction of the available memory occupied by the implementation. Finally, in Table 3.5, the best implementation of a cipher is the one with the smallest RAM footprint and code size, respectively.

Cipher			AVR			MSP			ARM			
	Block [b]	Key [b]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	FOM
Encryption + Decryption (including key schedule)												
Chaskey	128	128	*1328*	*229*	***20622***	*900*	*222*	***16674***	**438**	**236**	**9851**	4.0
Chaskey-LTS	128	128	*1328*	*229*	*33102*	*904*	*222*	*25394*	**438**	**236**	12859	4.6
Speck	64	96	*966*	*294*	*39875*	***556***	*288*	*31360*	492	308	15427	5.1
Speck	64	128	***874***	*302*	*44895*	*572*	*296*	*32333*	444	308	16505	5.2
Simon	64	96	*1084*	*363*	*63649*	*738*	*360*	*47767*	600	376	23056	7.0
Simon	64	128	*1122*	*375*	*66613*	*760*	*372*	*49829*	560	392	23930	7.2
RECTANGLE	64	80	*1152*	*352*	*66722*	*812*	*398*	*44551*	*664*	*426*	*35286*	8.0
RECTANGLE	64	128	*1118*	*353*	*64813*	*826*	*404*	*44885*	*660*	*432*	*36121*	8.0
LEA	128	128	*1684*	*631*	*61020*	*1154*	*630*	*46374*	*524*	*664*	*17417*	8.3
Sparx	64	128	*1198*	*392*	*65539*	*966*	*392*	*36766*	*1200*	*424*	*40887*	8.8
Sparx	128	128	*1736*	*753*	*83663*	*1118*	*760*	*53936*	*1122*	*788*	*67581*	13.2
HIGHT	64	128	*1414*	*333*	*94557*	*1238*	*328*	*120716*	*1444*	*380*	*90385*	14.8
AES	128	128	*3010*	*408*	*58246*	*2684*	*408*	*86506*	*3050*	*452*	*73868*	15.8
Fantomas	128	128	3520	227	141838	2918	222	85911	2916	268	94921	17.8
Robin	128	128	2474	229	184622	3170	238	76588	3668	304	91909	18.7
Robin*	128	128	5076	271	157205	3312	238	88804	3860	304	103973	20.7
RC5-20	64	128	3706	368	252368	1240	378	386026	624	376	36473	20.8
PRIDE	64	128	1402	369	146742	2566	**212**	242784	2240	452	130017	22.8
RoadRunneR	64	80	2504	330	144071	3088	338	235317	2788	418	119537	23.3
RoadRunneR	64	128	2316	**209**	125635	3218	218	222032	2504	448	140664	23.4
LBlock	64	80	2954	494	183324	1632	324	263778	2204	574	140647	25.2
PRESENT	64	80	*2160*	*448*	*245232*	*1818*	*448*	*202050*	*2116*	*470*	*274463*	32.8
PRINCE	64	128	2412	367	288119	2028	236	386781	1700	448	233941	34.9
Piccolo	64	80	1992	314	407269	1354	310	324221	1596	406	294478	38.4
TWINE	64	80	4236	646	297265	3796	564	387562	2456	474	255450	40.0
LED	64	80	5156	574	2221555	7004	252	2065695	3696	654	594453	138.6

Table 3.2: Results for Scenario 1. Encrypt and decrypt 128 bytes of data using CBC mode. Results of assembly implementations are in italics. For each cipher, an optimal implementation on each architecture is selected. The Figure of Merit (FOM) takes into account the three metrics (Code, RAM, and Time) on all platforms (AVR, MSP, and ARM). The smaller the FOM, the better the implementations of the cipher.

Defining a fair Figure of Merit that considers various trade-offs is a challenging task. The *Figure of Adversarial Merit (FOAM)* introduced in [186] combines inherent security provided by cryptographic structures and components with their implementation properties allowing the comparison of security-time-area trade-offs

Cipher			AVR			MSP			ARM			
	Block	Key	Code	RAM	Time	Code	RAM	Time	Code	RAM	Time	FOM
	[b]	[b]	[B]	[B]	[cyc.]	[B]	[B]	[cyc.]	[B]	[B]	[cyc.]	
Encryption + Decryption (including key schedule)												
Chaskey	128	128	*1328*	*229*	***20622***	*900*	***222***	***16674***	472	240	**9313**	5.4
Chaskey-LTS	128	128	*1328*	*229*	*33102*	*904*	***222***	*25394*	*576*	***228***	*11076*	6.5
Speck	64	96	***966***	*294*	*39875*	*664*	*290*	*29611*	492	308	15427	7.5
Speck	64	128	*1112*	*302*	*41103*	***592***	*298*	*31832*	**444**	308	16505	7.8
Simon	64	96	*1084*	*363*	*63649*	*758*	*362*	*47266*	600	376	23056	10.7
Simon	64	128	*1122*	*375*	*66613*	*780*	*374*	*49328*	560	392	23930	11.0
LEA	128	128	*1684*	*631*	*61020*	*1154*	*630*	*46374*	696	644	16192	11.5
RECTANGLE	64	80	*1152*	*352*	*66722*	*832*	*400*	*44050*	*664*	*426*	*35286*	12.4
RECTANGLE	64	128	*1118*	*353*	*64813*	*846*	*406*	*44384*	*660*	*432*	*36121*	12.5
Sparx	64	128	*1426*	*392*	*61955*	*986*	*394*	*36265*	*1200*	*424*	*40887*	13.4
Sparx	128	128	*1736*	*753*	*83663*	*1710*	*758*	*46640*	*2290*	*784*	*53109*	19.6
AES	128	128	*3010*	*408*	*58246*	*2684*	*408*	*86506*	*3080*	*452*	*73579*	23.6
HIGHT	64	128	*1414*	*333*	*94557*	*1258*	*330*	*120215*	*1444*	*380*	*90385*	25.1
Fantomas	128	128	5892	267	111677	4164	234	56788	4604	308	70142	26.3
Robin	128	128	4944	271	146149	3170	238	76588	3572	1312	74665	28.5
Robin⋆	128	128	5076	271	157205	3312	238	88804	3724	1316	85247	31.1
RC5-20	64	128	3706	368	252368	1240	378	386026	624	376	36473	37.0
PRIDE	64	128	3384	373	111155	2918	380	226135	2240	452	130017	38.8
RoadRunneR	64	80	2504	330	144071	3088	338	235317	2788	418	119537	39.2
RoadRunneR	64	128	2316	**209**	125635	2952	362	218909	2504	448	140664	39.8
LBlock	64	80	2954	494	183324	1632	324	263778	2204	574	140647	43.7
PRESENT	64	80	*2160*	*448*	*245232*	*1838*	*450*	*201549*	*2528*	*502*	*270464*	59.3
PRINCE	64	128	5358	374	243396	4174	240	357423	4372	504	201136	62.3
TWINE	64	80	4236	646	297265	3796	564	387562	2456	474	255450	70.8
Piccolo	64	80	1992	314	407269	1354	310	324221	1596	406	294478	71.9
LED	64	80	5156	574	2221555	7004	252	2065695	3696	654	594453	264.8

Table 3.3: Results for Scenario 1 (encryption of 128 bytes of data using CBC mode) when using different weights w_m for the three metrics in Equation (3.1) to compute the performance parameter p_i, d. Namely, the code size and the RAM size have the weights $w_{code} = w_{RAM} = 1$, while the cycle count has the weight $w_{cycle} = 2$. Results of assembly implementations are in italics.

of hardware implementations. Although the FOAM is only suitable for hardware implementations, a similar metric could be defined for software by replacing area by RAM consumption and/or code size.

3.4.2 Discussion of Results

In Scenario 1 ("bulk encryption"), the top-3 ciphers based on the FOM score are Chaskey, Speck, and Simon; the FOM score of these ciphers is less than half of the FOM score of the AES. Recall that the FOM score takes into account all three

Cipher			AVR			MSP			ARM			
	Block [b]	Key [b]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	FOM
Balanced (globally efficient)												
Chaskey	128	128	*624*	*80*	***1465***	*388*	*70*	***1153***	***216***	*76*	***524***	4.4
Chaskey-LTS	128	128	*624*	*80*	*2265*	*390*	*70*	*1690*	***216***	*76*	*648*	5.0
Speck	64	96	*506*	*53*	*2647*	***328***	***48***	*1959*	256	**56**	1003	5.1
Speck	64	128	***452***	*53*	*2917*	*332*	***48***	*2013*	276	60	972	5.2
Simon	64	96	*600*	*57*	*4269*	*460*	*56*	*2905*	416	64	1335	7.0
Simon	64	128	*608*	*57*	*4445*	*468*	*56*	*3015*	388	64	1453	7.2
LEA	128	128	*906*	*80*	*4023*	*722*	*78*	*2814*	*520*	*112*	*1171*	8.0
RECTANGLE	64	128	*602*	*56*	*4381*	*480*	*54*	*2651*	*444*	*76*	*2365*	8.5
RECTANGLE	64	80	*606*	*56*	*4433*	*480*	*54*	*2651*	*444*	*76*	*2365*	8.5
Sparx	64	128	*662*	***51***	*4397*	*580*	*52*	*2261*	*654*	*72*	*2338*	8.7
Sparx	128	128	*1184*	*74*	*5478*	*1036*	*72*	*3057*	*1468*	*104*	*2935*	13.0
RC5-20	64	128	1068	63	8812	532	60	15925	372	64	1919	14.8
AES	128	128	*1246*	*81*	*3408*	*1170*	*80*	*4497*	*1348*	*124*	*4044*	14.9
HIGHT	64	128	*636*	*56*	*6231*	*636*	*52*	*7117*	*670*	*100*	*5532*	15.9
Fantomas	128	128	2496	108	5919	1920	78	3602	2184	184	4550	19.6
Robin	128	128	2530	108	7813	1942	80	4913	2188	184	6250	23.0
Robin⋆	128	128	2580	106	8052	1980	80	5262	2272	196	6417	23.7
RoadRunneR	64	80	1420	61	7329	1536	76	13034	1900	172	7234	25.5
PRIDE	64	128	2064	91	5727	1842	68	13108	1592	148	7446	25.6
RoadRunneR	64	128	1184	59	6289	1724	74	13266	1436	164	8573	26.3
LBlock	64	80	1440	64	11183	804	58	16101	1220	284	9015	28.7
PRESENT	64	80	*1294*	*56*	*16849*	*1072*	*58*	*12347*	*1222*	*80*	*17105*	38.6
PRINCE	64	128	1362	72	20060	1576	76	24246	1384	280	15165	44.0
Piccolo	64	80	1114	72	25820	784	70	20081	688	112	17965	44.2
TWINE	64	80	1528	64	21701	1922	136	23662	1180	156	15673	44.6
LED	64	80	2548	267	135061	4422	104	121850	2172	352	35891	149.2

Table 3.4: Results for Scenario 2. Encrypt 128 bits of data using CTR mode. Results of assembly implementations are in italics. For each cipher, an optimal implementation on each architecture is selected. The Figure of Merit (FOM) takes into account the three metrics (Code, RAM, and Time) on all platforms (AVR, MSP, and ARM). The smaller the FOM, the better the implementations of the cipher.

metrics (i.e. execution time, RAM footprint, and code size) and does so across three platforms (AVR, MSP, and ARM). Of course, when looking at performance, RAM footprint, or code size individually, or when looking at AVR, MSP, or ARM individually, the specific ranking can differ significantly from the overall ranking based on the FOM score. Furthermore, it has to be taken into account that several (up to 35) different implementations exist for one and the same cipher. Since these implementations are based on different optimization strategies, they can (and usually do) perform differently on the three platforms. It may also happen that one and the same cipher is slower on 16-bit MSP than on 8-bit AVR (e.g. HIGHT, AES, RC5), which is not a mistake but simply the result of considering RAM equally important as execution time. On each platform, we collected our benchmarking results using the implementation that achieved the best (i.e. smallest) FOM score.

When having a closer look at the results on AVR, it turns out that the top-ranked algorithms are very similar in terms of RAM footprint, which means the overall rank is primarily determined by execution time and code size. Speck has roughly twice the execution time of Chaskey, while Simon carries a performance penalty by a factor of approximately three. A somewhat surprising result is that the AES beats Simon on AVR, but its high performance comes at the expense of relatively large code size. Also LEA and Sparx are slightly faster than Simon when comparing the versions with 64-bit blocks and 128-bit keys. All other ciphers have an execution time that is more than three times worse than that of Chaskey. The situation is somewhat similar on MSP in the sense that Chaskey is the fastest cipher, followed by Speck. Simon is again on the sixth position, outperformed by RECTANGLE, LEA and Sparx with 64-bit blocks. However, the MSP results also show a disadvantage of Chaskey, namely its relatively large code size, which is roughly twice the size of Speck. On the other hand, in terms of RAM footprint, PRIDE, RoadRunneR, and Fantomas perform very well on the MSP430 platform. Finally, on ARM, the winners in the performance competition are Chaskey, Speck, and LEA. In addition, these three ciphers also have the top positions in terms of code size, which is mainly due to their extremely simple round function operating on 32-bit words. All other algorithms are both slower and larger than LEA.

The overall ranking in Scenario 2 ("challenge-response authentication"), shown in Table 3.4, is similar to that of Scenario 1. The three top spots are held by the same ciphers in the same order, i.e. Chaskey is the best overall performer and Speck the runner-up. Simon secured the third place, even though on all three platforms some other ciphers show better execution times. However, Simon profits from its relatively small code size and low RAM footprint. Positions 4 to 6 are held by LEA, RECTANGLE and Sparx with FOM scores that are between 1.82 and 1.98 times worse than Chaskey's FOM score. All other ciphers have a FOM score that is more than three times higher than that of Chaskey. Table 3.5 summarizes the results of the implementations with minimal RAM footprint and code size for each of the 19 ciphers. Speck turns out to be the most lightweight candidate and, therefore, the best choice for applications where size is the primary constraint. On all three platforms, Speck has a code size of below 500 bytes and a RAM footprint of at most 60 bytes. On the other hand, as shown in Table 3.5, when RAM footprint and code size are of primary concern and execution time does not matter much, then Speck is clearly the best choice. Also Simon is size-wise consistently good on all three platforms.

Caveats. The results of any "survey-and-benchmark" work in lightweight cryptography, including ours, always reflect the state of research at a certain time, namely the time when it was written. However, the efficient implementation of (lightweight) ciphers is an active area of research that is likely to provide new approaches for speeding up one or more of the 19 candidates considered in this chapter. The AES

Cipher			AVR			MSP			ARM		
	Block [b]	Key [b]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]
Small code size & RAM											
AES	128	128	*1246*	*81*	*3408*	*1170*	*80*	*4497*	952	140	38191
Chaskey	128	128	*624*	*80*	***1465***	*388*	*70*	***1153***	**180**	76	**785**
Chaskey-LTS	128	128	*624*	*80*	*2265*	*390*	*70*	*1690*	**180**	76	961
Fantomas	128	128	1712	76	9689	1412	74	5506	1384	100	8335
HIGHT	64	128	*636*	*56*	*6231*	*636*	*52*	*7117*	528	88	14244
LBlock	64	80	864	55	17933	804	58	16101	586	84	13818
LEA	128	128	*906*	*80*	*4023*	*722*	*78*	*2814*	528	88	1714
LED	64	80	1162	95	284063	950	102	170135	758	120	114723
Piccolo	64	80	1042	74	32603	784	70	20081	688	112	17965
PRESENT	64	80	*1294*	*56*	*16849*	802	62	513599	582	80	209946
PRIDE	64	128	834	62	14234	944	68	22551	656	96	16310
PRINCE	64	128	1384	70	20812	1518	70	27311	1158	136	22826
RC5-20	64	128	742	65	22635	524	54	23318	372	64	1919
RECTANGLE	64	128	*648*	*54*	*4665*	*480*	*54*	*2651*	464	68	3004
RECTANGLE	64	80	*648*	*54*	*4665*	*480*	*54*	*2651*	464	68	3004
RoadRunneR	64	80	1420	61	7329	628	88	67497	540	140	21475
RoadRunneR	64	128	1112	58	7023	620	82	25577	598	80	16334
Robin	128	128	1710	78	12513	1406	72	7051	1400	112	10070
Robin*	128	128	1754	80	14285	1452	76	8634	1432	112	11679
SIMON	64	96	*534*	*57*	*4521*	*416*	*56*	*3199*	324	**56**	2587
SIMON	64	128	*542*	*57*	*4709*	*424*	*56*	*3323*	340	60	2308
SPARX	64	128	*662*	***51***	*4397*	*496*	*54*	*2623*	*482*	*76*	*3434*
SPARX	128	128	*1212*	*73*	*5602*	*904*	*80*	*3273*	*932*	*108*	*4085*
SPECK	64	96	***448***	*53*	*2829*	***328***	***48***	*1959*	256	**56**	1159
SPECK	64	128	*452*	*53*	*2917*	*332*	***48***	*2013*	264	**56**	1029
TWINE	64	80	788	56	42434	850	56	45273	530	72	29986

Table 3.5: Results for Scenario 2. Encrypt 128 bits of data using CTR mode. Results of assembly implementations are in italics. For each cipher, an optimal implementation on each architecture is selected.

serves as a good example on how progress in software optimization techniques can yield significantly more efficient implementations. Similar progress could also make one or more of our lightweight ciphers much faster than anticipated today. This is the very reason why we maintain a web page [93] where the reader can find up-to-date benchmarking results and cipher rankings. Furthermore, our results reflect, to a certain degree, also the programming skills of the implementers and how much effort they put into optimization. We invite the cryptographic research community to send us improved implementations of the 19 ciphers covered in this chapter. In addition, we also welcome implementations of new ciphers.

3.4.3 Comparison with other Benchmarking Results

Many of the ciphers we study in this paper have already been evaluated on AVR, MSP, or ARM processors before, either separately or within some other benchmarking project. It is not easily possible to compare performance figures across various frameworks and implementations because the evaluation methodology is usually different and also the optimization efforts typically vary. The importance of a consistent evaluation framework and methodology becomes quickly evident when taking the AES counter-mode implementation for Cortex-M3 processors in [314, Section 3] as example. This implementation uses the T-table approach in combination with a careful optimization of the memory accesses and achieves, according to [314], an average execution time of 659.4 clock cycles for a single-block encryption with a 128-bit key. However, this cycle count was only reached by configuring the Cortex-M3 processor to have a reduced number of wait states for memory accesses, which favors implementations using T-tables, but limits the maximum frequency the processor can be clocked with. On the other hand, our benchmarking framework operates the Cortex-M3 with the full wait states (so that it can be clocked with its maximum frequency) and reports an execution time of 1641 clock cycles for this T-table implementation. In addition, it must be taken into account that using T-tables entails a large memory footprint, which worsens the FOM score. This also explains why an implementation using only Sbox look-ups can reach a better FOM score than the T-table approach, despite the fact that T-tables have the potential to reduce the execution time by a factor of more than two.

The most notable differences between our benchmarks and previous implementation results obtained on AVR/MSP/ARM are the following. The BLOC project's [74] MSP implementations of LBlock, Piccolo, and Twine are slightly worse than ours, whereas the implementations of AES, HIGHT, and PRESENT are much slower. On the other hand, the AVR assembly implementations of PRESENT and AES from the ECRYPT project [119, 121] are slightly slower than our assembly implementations, while our implementation of HIGHT is twice as fast as the assembly implementation from [121] and ten times faster than the assembly implementation from [119].

3.5 Summary

In this chapter, we presented a survey and benchmark of 19 lightweight block ciphers based on two usage scenarios that are common for secure communication in the IoT. In particular, we studied their implementation aspects on representative 8, 16, and 32-bit platforms.

The metrics (binary code size, RAM footprint and execution time) are extracted using the FELICS benchmarking framework introduced in Chapter 2. For full transparency, the source code of the framework, together with the implementations of the evaluated ciphers, are available under an open-source license. We strongly encourage the community to use and contribute to our framework, since it allows easy integration and evaluation of new C and assembly implementations. We are committed to maintaining a web page [93] that summarizes the latest results obtained by each submitted implementation.

Based on the benchmarking results, we inferred some interesting information regarding the link between the design decisions and performance figures. In particular, our results show that state-of-the-art designs based on simple operations (addition/AND, rotation, and XOR) like Chaskey, Speck and Simon are not only very fast, but also extremely small in terms of both code size and RAM requirements. Furthermore, they perform consistently well on all three platforms, which makes them excellent candidates for a lightweight cipher to secure the IoT.

Designers of new ciphers should focus on simple round functions that use as few operations as possible and reach a good security level after several iterations. The most efficient operations to be employed are the bitwise logical operations and modular addition. The cost of rotations depends on the target architecture and the rotation amount. One should use rotations by some carefully chosen values (e.g. 7, 8, 9, 15, or 16 for a 32-bit word) to reduce the execution time and code size on architectures that support only rotations by one bit at a time. The above-mentioned operations do not require any memory access, provided that the cipher's state can be kept into the internal registers. Finally, lookup tables of any size should be avoided as they increase the code size and/or RAM footprint at the cost of a memory load. These requirements lead to the following three categories of designs: ARX – Add-Rotate-XOR (e.g. Chaskey, Speck, LEA, Sparx), AndRX – AND-Rotate-XOR (e.g. Simon), and bit-sliced (e.g. RECTANGLE).

Further work may include the addition of new ciphers, integration of countermeasures against physical attacks, extending the framework's capabilities to benchmark other lightweight symmetric primitives (stream ciphers, hash functions, authenticated encryption algorithms) and the support of additional processors.

Chapter 4

On the Efficiency of the SPARX Family of Lightweight Block Ciphers

Contents

4.1 Introduction

SPARX is a family of lightweight block ciphers designed to be secure and yet efficient on a variety of resource-constrained devices. As the name suggests, SPARX is a Substitution-Permutation (SP) network and uses only three operations: addition, rotations, and XOR (ARX). Thanks to the *Long Trail Strategy (LTS)* [106], SPARX is the first ARX cipher that was designed to have provable bounds against differential and linear cryptanalysis. The family has three instances depending on the block and key sizes.

While the primary design goal of SPARX was security, the efficiency of its software implementations on 8-, 16-, and 32-bit microcontrollers played an important role in shaping this family of lightweight block ciphers. Hence, in this chapter we elaborate on the implementation-related characteristics of SPARX and how software efficiency influenced the final design. Then, we provide implementation details and results for two instances of SPARX, namely SPARX-64/128 and SPARX-128/128.

4.2 Short Description

We use SPARX-n/k to refer to the instance of SPARX that operates on a block of n bits and has a key of k bits. The high-level structure of SPARX depicted in Figure 4.1 works on 32-bit words. Hence, the cipher's state is represented on $w = n/32$ words, while the key consists of $v = k/32$ words. The encryption function is obtained by iterating n_s times a *step*, which consists of a substitution layer and a linear mixing layer. The substitution layer applies r_a *rounds* of a so-called ARX-box (preceded by a key addition) to each word of the state, while the linear layer mixes the cipher's state. The linear layer of the last step is followed by the addition of a post-whitening key to the cipher's state. The key schedule, shown in Figure 4.2, uses the function K_v to generate v round keys in each iteration. The main parameters of the three instances of SPARX are summarized in Table 4.1.

	SPARX-64/128	SPARX-128/128	SPARX-128/256
# State words w	2	4	4
# Key words v	4	4	8
# Steps n_s	8	8	10
# Rounds r_a	3	4	4
# Round keys words	50	132	164

Table 4.1: SPARX parameters.

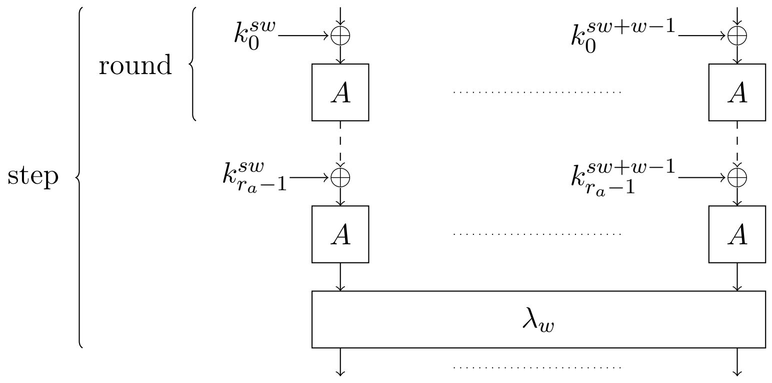


Figure 4.1: The structure of SPARX encryption.

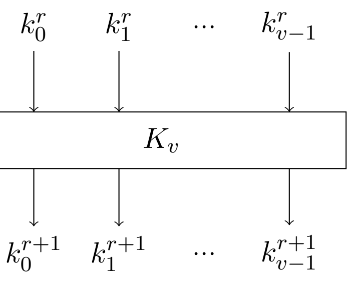
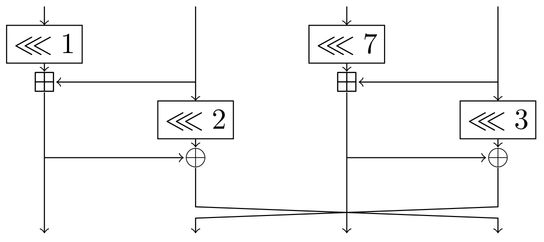


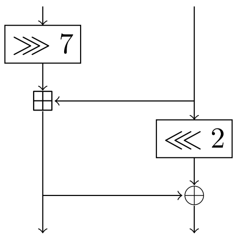
Figure 4.2: The structure of SPARX key schedule.

4.3 Choosing the ARX-box A

Once the high-level structure (see Figure 4.1) of the cipher was set, we had to choose an ARX-box that provides good cryptographic properties (i.e. minimizes the differential and linear probabilities), while facilitating efficient implementations. We considered two possible structures, namely MARX-2 and SPECKEY, which were introduced by Biryukov *et al.* [274]. MARX-2 is based on the MIX function of Skein [124], while SPECKEY is a variant of SPECK-32. The two structures are shown in Figure 4.3.



(a) MARX-2.



(b) SPECKEY.

Figure 4.3: The candidate 32-bit ARX-boxes, MARX-2 and SPECKEY. The branch size is 8 bits for MARX-2 and 16 bits for SPECKEY.

Although both structures process a 32-bit word, their branches have different sizes. MARX-2 has four 8-bit branches, while SPECKEY uses two 16-bit branches. Another major difference between the two stems from the number of elementary operations used. MARX-2 needs two additions modulo 2^8, two bitwise XORs, and four rotations (by 1, 2, 3, and 7 bits to the left). Moreover, MARX-2 also performs a branch swap. SPECKEY requires only one addition modulo 2^{16}, one bitwise XOR, and two rotations (by 2 bits to the left and by 7 bits to the right). The execution time in number of clock cycles of both MARX-2 and SPARX is given in Table 4.2. These values correspond to implementations where each branch is stored in its own register. Consequently, the implementations of these two structures do not require additional registers. For more details on the cost of each elementary operation, we refer the reader to Chapter 5.

The above-mentioned implementation properties of MARX-2 and SPECKEY

	AVR	MSP	ARM
MARX-2	20	25	17
SPARX	16	9	7

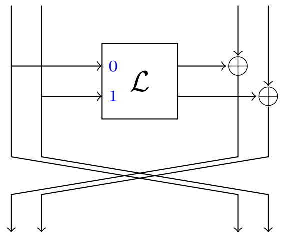
Table 4.2: Comparison between the execution time (in cycles) of MARX-2 and SPECKEY.

indicate that SPECKEY yields much better implementations than MARX-2. Therefore, we selected SPECKEY for the ARX-box of SPARX. An even better choice for the overall implementation efficiency would have been a primitive that operates on 32-bit branches because this word size yields better execution times on all platforms (see Section 5.3.2). However, when we designed the cipher, there was no such construction with provable differential and linear bounds for enough rounds.

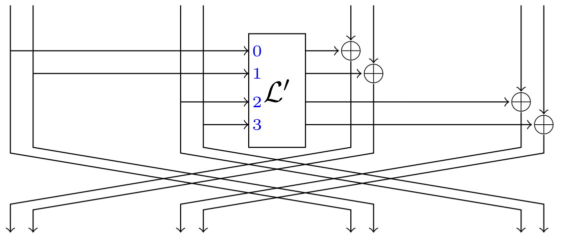
In order to reduce the pressure on registers, we decided to execute all the ARX-boxes of a branch before processing another branch of the same step. This approach reduces the number of stack operations required to switch between branches when the state does not fully fit into the available registers of a microcontroller. On the other hand, it does not negatively affect the performance of an implementation when there are enough registers to store the entire state of the cipher.

4.4 Choosing the Linear Layer λ_w

The linear layer was selected such that it minimizes the probability of differential and linear trails as well as the number of steps of the integral characteristic found with the division property. The Feistel functions $\mathcal{L}$ and $\mathcal{L}'$ are used by the linear layers λ_2 and λ_4, respectively. They provide diffusion and yield efficient implementations on 8-, 16-, and 32-bit architectures. Both $\mathcal{L}$ and $\mathcal{L}'$ rely on a Lai-Massey structure. The function $\mathcal{L}'$ is a generalization of $\mathcal{L}$, which is borrowed from NOEKEON [97]. The linear layer λ_2 used by SPARX-64/128 is shown in Figure 4.4a, while the linear layer λ_4 used by SPARX-128/128 and SPARX-128/256 is shown in Figure 4.4b. The two Feistel functions $\mathcal{L}$ and $\mathcal{L}'$ are shown in Figure 4.5.



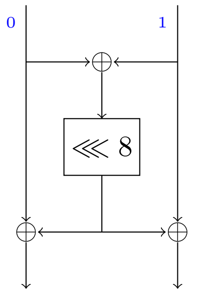
(a) λ_2.



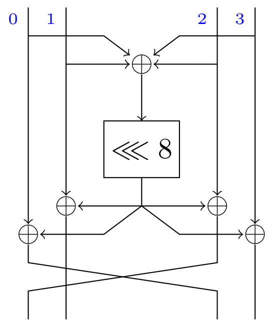
(b) λ_4.

Figure 4.4: The linear layers of SPARX-64/128 and SPARX-128/128.

The $\mathcal{L}$ transformation maps a 32-bit value x to $x \oplus (x \lll 8) \oplus (x \ggg 8)$. Its



(a) $\mathcal{L}$.



(b) $\mathcal{L}'$.

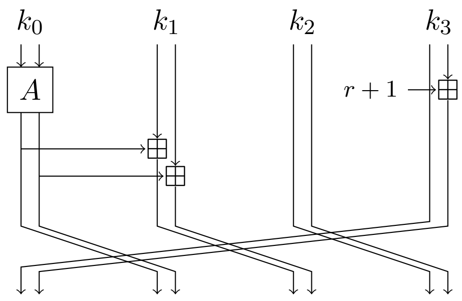
Figure 4.5: The Feistel functions used by the linear layers of SPARX.

alternative representation shown in Figure 4.5a works on two 16-bit branches a and b, which are transformed to $a \oplus \big((a \oplus b) \lll 8\big)$ and $b \oplus \big((a \oplus b) \lll 8\big)$, respectively.

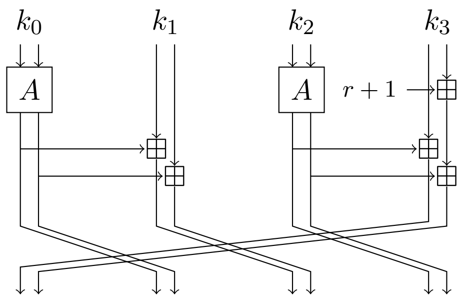
Similarly to $\mathcal{L}$, the $\mathcal{L}'$ function can be defined in two ways. The representation shown in Figure 4.5b transforms the four 16-bit branches a, b, c, d to $c \oplus t$, $b \oplus t$, $a \oplus t$, and $d \oplus t$, where $t = (a \oplus b \oplus c \oplus d) \lll 8$. In order to define the representation of $\mathcal{L}'$ that works on 32-bit values, let $\mathcal{L}'_\star$ be the function that skips the final branch swap of $\mathcal{L}'$. If $x \mid\mid y$ is the concatenation of two 32-bit words and $t = \big((x \oplus y) \ggg 8\big) \oplus \big((x \oplus y) \lll 8\big)$, then $\mathcal{L}'_\star(x \mid\mid y) = x \oplus t \mid\mid y \oplus t$. Furthermore, $\mathcal{L}'_\star$ can be written using $\mathcal{L}$ as follows: $\mathcal{L}'_\star(x \mid\mid y) = y \oplus \mathcal{L}(x \oplus y) \mid\mid x \oplus \mathcal{L}(x \oplus y)$.

4.5 Key Schedule

The key schedule was designed to quickly mix the bits of the master key into the round keys. It reuses the ARX-box A to reduce the code size of software implementations. The additions modulo 2^{16} provide diffusion, while the addition of the round number prevents slide attacks [54]. The expensive operations of the key schedule K_v are the branch swaps. The key schedule of SPARX-64/128 is shown in Figure 4.6a, while the key schedule of SPARX-128/128 is given in Figure 4.6b.



(a) Key schedule of SPARX-64/128.



(b) Key schedule of SPARX-128/128.

Figure 4.6: The key schedules of SPARX-64/128 and SPARX-128/128.

It is important to note that one iteration of the key schedules of SPARX-64/128 and SPARX-128/128 generates four key words of 32 bits each. On the other hand, one step of SPARX-64/128 and SPARX-128/128 requires 6 and 16 round key words, respectively. The final key addition uses 2 round key words for SPARX-64/128 and 4 round key words for SPARX-128/128. Hence, the number of key words generated by the key schedule of SPARX-64/128 is not synchronized with the number of key words required in the step function. Consequently, its rolled implementations consume more code size (to compute the last two round keys outside the main loop) or waste some clock cycles (to execute one full iteration and discard two key words).

4.6 Implementation

Next we describe how SPARX can be efficiently implemented on three resource-constrained microcontrollers widely used in the IoT, namely the 8-bit Atmel ATmega128, the 16-bit TI MSP430, and the 32-bit ARM Cortex-M3. These devices are described in Section 2.6. We support the described optimization strategies with performance figures extracted from assembly implementations of SPARX-64/128 and SPARX-128/128 using the FELICS open-source benchmarking framework (see Chapter 2). We refer the reader to Chapter 3 for comparative performance figures extracted from software implementations of 19 lightweight block ciphers.

4.6.1 Main Components

In order to efficiently implement SPARX on a resource-constrained embedded processor, it is important to have a good understanding of its instruction set architecture (ISA). The number of general-purpose registers determines whether the entire cipher's state fits into registers or if a part of it has to be held in RAM. Memory operations are generally slower than register operations, consume more energy and increase the vulnerability of an implementation to side-channel attacks as shown in Chapter 6. Thus, the number of memory operations should be reduced as much as possible. Ideally the state should only be read from memory at the beginning of the cryptographic operation and written back at the end. The three targets we implemented SPARX for have 32 8-bit, 12 16-bit, and 13 32-bit general-purpose registers, which result in a total capacity of 256 bits, 192 bits, and 416 bits for AVR, MSP, and ARM, respectively.

The SPARX family's simple structure consists only of three components: the ARX-box A and its inverse A^{-1}, the linear layer λ_2 or λ_4 (depending on the version), and the key addition. The key addition (bitwise XOR) does not require additional registers and its execution time is proportional to the ratio between the operand width and the target device's register width. The execution time in cycles and the number of registers required to perform A, A^{-1}, λ_2, and λ_4 on each target device are given in Table 4.3.

The costly operation in terms of both execution time and number of required registers is the linear layer. The critical point is reached for the 128-bit linear layer λ_4 on MSP, which requires 13 registers. Since this requirement is above the number

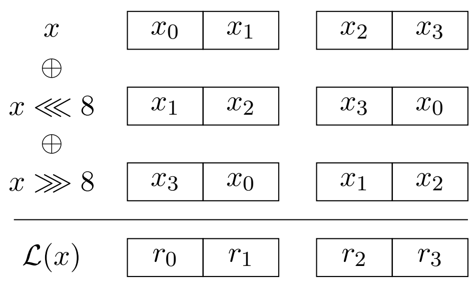
Component	AVR		MSP		ARM	
	Cycles	Registers	Cycles	Registers	Cycles	Registers
A	16	$4+1$	9	2	11	$1+3$
A^{-1}	19	4	9	2	12	$1+3$
λ_2 – 1-step	24	$8+1$	11	$4+3$	5	$2+1$
λ_2 – 2-steps	12	8	7	$4+1$	3	2
λ_4 – 1-step	48	$16+2$	36	$8+1$	16	$4+5$
λ_4 – 2-steps	24	$16+2$	13	$8+1$	12	$4+4$

Table 4.3: Performance characteristics of the main components of SPARX.

of available registers, a part of the state has to be saved on the stack. Consequently, the execution time increases by 5 cycles for each `push` – `pop` instruction pair.

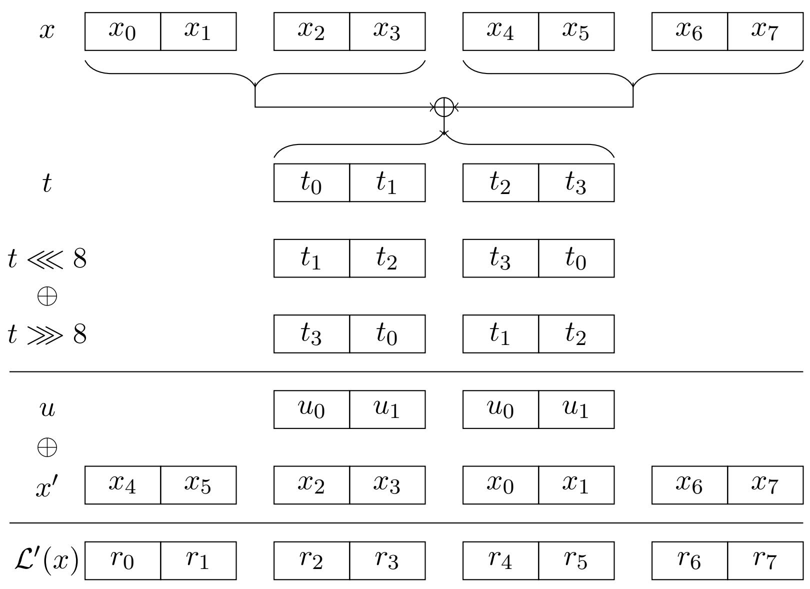
A 2-step implementation uses a simplified linear layer without the most resource-demanding part – the branch swaps. It processes the result of the left branch after the first step as the right branch of the second step and similarly the result of the right branch after the first step as the left branch of the second step. This technique reduces the number of required registers and improves the execution time at the cost of an increase in code size. The performance gain is a factor of 2 on AVR, 2.7 on MSP, and 1.3 on ARM.

The linear transformations $\mathcal{L}$ and $\mathcal{L}'$ exhibit interesting implementation properties. For each platform there is a different optimal way to perform them. The optimal way to implement the linear layers on MSP is using the representations from Figure 4.5a and Figure 4.5b. On ARM the optimal implementation performs the rotations directly on 32-bit values. The function $\mathcal{L}$ can be executed on AVR using 12 XOR instructions and no additional registers. On the other hand, the optimal implementation of $\mathcal{L}'$ on AVR requires 2 additional registers and takes 24 cycles. The steps required to efficiently compute $\mathcal{L}$ and $\mathcal{L}'$ on an 8-bit AVR microcontroller are shown in Figure 4.7



$$r_0 = x_0 \oplus x_1 \oplus x_3 \qquad r_1 = x_0 \oplus x_1 \oplus x_2$$
$$r_2 = x_1 \oplus x_2 \oplus x_3 \qquad r_3 = x_0 \oplus x_2 \oplus x_3$$

Figure 4.7: Computation of $\mathcal{L}$ on 8-bit registers.



$$u_0 = x_1 \oplus x_3 \oplus x_5 \oplus x_7 \qquad u_1 = x_0 \oplus x_2 \oplus x_4 \oplus x_6$$
$$r_0 = u_0 \oplus x_4 \qquad r_1 = u_1 \oplus x_5 \qquad r_2 = u_0 \oplus x_2 \qquad r_3 = u_1 \oplus x_3$$
$$r_4 = u_0 \oplus x_0 \qquad r_5 = u_1 \oplus x_1 \qquad r_6 = u_0 \oplus x_6 \qquad r_3 = u_1 \oplus x_7$$

Figure 4.8: Computation of $\mathcal{L}'$ on 8-bit registers.

and Figure 4.8, respectively. Each intermediate variable is an 8-bit value.

The linear layer performed after the last step of SPARX can be dropped without affecting the security of the cipher, but it turns out that it results in poorer overall performance. The only case where this strategy helps is when top execution time is the main and only concern of an implementation. Thus we preferred to keep the symmetry of the step function and the overall balanced performance figures.

4.6.2 Flexibility

The salient implementation-related feature of the SPARX family of ciphers is given by the simple and flexible structure of the step function depicted in Figure 4.1, which can be implemented using different optimization strategies. Depending on specific constraints, such as code size, speed, or energy requirements to name a few, the rounds inside the step function can be rolled or unrolled; one or two step functions can be computed at once. The main possible trade-offs between the execution time and code size are explored in Table 4.4.

Except for the 1-step implementation of SPARX-128/128 on MSP, which needs RAM memory to save the cipher's state, all other RAM requirements are determined only by the process of saving the register context on the stack at the beginning of the measured function. Thus, the RAM consumption of a pure assembly implementation would be zero, except for the 1-step rolled and unrolled implementations of SPARX-128/128 on MSP.

Implementation		AVR			MSP			ARM		
	Block [b]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]
1-step rolled	64	1789	**248**	2	1088	**166**	14	1370	**176**	28
1-step unrolled	64	1641	424	**1**	907	250	12	1100	348	**24**
2-steps rolled	64	1677	356	2	1034	232	10	1331	304	28
2-steps unrolled	64	**1529**	712	**1**	**853**	404	**8**	**932**	644	**24**
1-step rolled	128	4553	**504**	11	2809	**300**	26	3463	**348**	44
1-step unrolled	128	4165	1052	**10**	2353	584	24	2784	884	40
2-steps rolled	128	4345	720	11	2593	432	18	3399	620	40
2-steps unrolled	128	**3957**	1820	**10**	**2157**	1004	**16**	**2377**	1692	**36**

Table 4.4: Different trade-offs between the execution time and code size for encryption of a block using SPARX-64/128 and SPARX-128/128. Minimal values are given in bold.

Due to the 16-bit nature of the cipher, performing A and A^{-1} on a 32-bit platform requires a little bit more execution time and more auxiliary registers than performing the same operations on a 16-bit platform. The process of packing and unpacking a state register to extract and store back the two 16-bit branches of A or A^{-1} adds a performance penalty. This cost is amplified by the fact that the flexible second operand of an instruction can not be used with a constant to extract the least or most significant 16 bits of a 32-bit register. Thus, an additional masking register is required. One can use the `movt` instruction to perform the masking of the 16-bit values without an additional register, but the implementations of A and A^{-1} will require more instructions and hence the execution time and code size will increase.

The simple key schedules of SPARX-64/128 and SPARX-128/128 can be implemented in different ways. The most efficient implementation turns out to be the one using the 1-iteration rolled strategy. Another interesting approach is the 4-iterations unrolled strategy, which has the benefit that the final permutation is achieved for free by changing the order in which the registers are mapped to the generated key words. This strategy increases the code size by up to a factor of 4, while the execution time is on average 25% better.

Although we do not provide performance figures for SPARX-128/256, we emphasize that the only differences with respect to implementation aspects between SPARX-128/256 and SPARX-128/128 are the key schedules and the different number of steps.

4.6.3 Evaluation

We evaluate the performance of our implementations of SPARX using FELICS in the two usage scenarios described in Section 3.2.1. The key performance figures are summarized in Table 4.5. The balanced results are achieved using the 1-step implementations of SPARX-64/128 and SPARX-128/128.

	AVR			MSP			ARM			
Block [b]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	Time [cyc.]	Code [B]	RAM [B]	FOM
Scenario 1 – Encryption of 128 bytes of data using CBC mode										
64	30256	358	10	16113	338	22	19131	456	56	8.8
128	37984	614	19	24056	404	36	30466	428	68	13.2
Scenario 2 – Encryption of 128 bits of data using CTR mode										
64	4397	662	51	2261	580	52	2338	654	72	8.7
128	5478	1184	74	3057	1036	72	2935	1468	104	13.0

Table 4.5: The performance figures of the balanced (globally efficient) implementations of SPARX-64/128 and SPARX-128/128 according to the Figure of Merit (FOM) defined in FELICS.

4.6.4 Comparison

We compare the performance of SPARX with the results available on the Triathlon Competition website [92] at the time of writing. Up-to-date results are available at [92]. As can be seen in Table 4.6 the two instances of SPARX perform very well across all platforms and rank very high in the FOM-based ranking. The superior competitors are the NSA designs SIMON and SPECK, Chaskey, RECTANGLE and LEA, but, apart from RECTANGLE, none of them have been designed with provable security in mind.

Besides the overall good performance figures in the two usage scenarios, the

Rank	Cipher	Block size	Key size	Scenario 1 FOM
1	Chaskey-LTS	128	128	4.6
2	SPECK	64	128	5.2
3	SIMON	64	128	7.2
4	RECTANGLE	64	128	8.0
5	LEA	128	128	8.3
6	**SPARX**	**64**	**128**	**8.8**
7	**SPARX**	**128**	**128**	**13.2**
8	HIGHT	64	128	14.8
9	AES	128	128	15.8
10	Fantomas	128	128	17.8

Table 4.6: Top 10 best implementations in Scenario 1 (encryption key schedule, encryption and decryption of 128 bytes of data using CBC mode) ranked by the Figure of Merit (FOM) defined in FELICS. The smaller the FOM, the better the implementation.

following results are worth mentioning:

- the execution time of Sparx-64/128 on MSP is in the top 3 of the fastest ciphers in both scenarios thanks to its 16-bit oriented operations;
- the code size of the 1-step rolled implementations of Sparx-64/128 and Sparx-128/128 on MSP is in the top 5 in both scenarios as well as in the small code size and RAM table for Scenario 2;
- the 1-step rolled implementation of Sparx-64/128 breaks the previous minimum RAM consumption record on AVR in Scenario 2;
- the execution time of the 2-steps implementation of Sparx-64/128 in Scenario 2 is in the top 3 on MSP, in the top 5 on AVR, and in the top 7 on ARM; it also breaks the previous minimum RAM consumption records on AVR and MSP.

4.7 Test Vectors

Test vectors are shown as 16-bit words in hexadecimal notation in Table 4.7. Reference C implementations of the three instances of Sparx are available on GitHub [95]. All optimized implementations described in this chapter are included in FELICS [93]. More resources about the Sparx family of lightweight block ciphers can be found on the primitive's web page [94].

Sparx-64/128				
key	`0011 2233`	`4455 6677`	`8899 aabb`	`ccdd eeff`
plaintext	`0123 4567`	`89ab cdef`		
ciphertext	`2bbe f152`	`01f5 5f98`		
Sparx-128/128				
key	`0011 2233`	`4455 6677`	`8899 aabb`	`ccdd eeff`
plaintext	`0123 4567`	`89ab cdef`	`fedc ba98`	`7654 3210`
ciphertext	`1cee 7540`	`7dbf 23d8`	`e0ee 1597`	`f428 52d8`
Sparx-128/256				
key	`0011 2233`	`4455 6677`	`8899 aabb`	`ccdd eeff`
	`ffee ddcc`	`bbaa 9988`	`7766 5544`	`3322 1100`
plaintext	`0123 4567`	`89ab cdef`	`fedc ba98`	`7654 3210`
ciphertext	`3328 e637`	`14c7 6ce6`	`32d1 5a54`	`e4b0 c820`

Table 4.7: Sparx test vectors.

4.8 Other Implementations

Damian Gryski [99] implemented SPARX in Go, while Frank Denis [130] and Quininer Kel [285] implemented SPARX in Rust.

4.9 Summary

Given its simple and flexible structure as well as its very good overall ranking in the Triathlon Competition of lightweight block ciphers [92], the SPARX family is suitable for applications on a wide range of resource-constrained devices. The absence of look-up tables reduces the memory requirements and provides some intrinsic resistance against power analysis attacks as described in Chapter 6.

SPARX was designed to achieve two goals, namely security against known cryptanalytic attacks and efficiency on resource-constrained microcontrollers. In this chapter, we showed that software efficiency played an important role in the design phase of SPARX and influenced many design choices. However, as in many lightweight designs, the main limiting factor was the theoretical framework used to prove the cipher's security. In other words, we could have designed much more efficient ciphers, but we could not prove their security against the main cryptanalytic attacks.

Chapter 5

Efficient Lightweight Symmetric Cryptography for Embedded IoT Systems

Contents

5.1 Introduction

In a broad sense, the IoT connects devices with various computational capabilities ranging from small RFID tags to very powerful smartphones and tablets. A more precise scope of the IoT covers only devices with very low computational capabilities and middle range IoT devices, but not the very powerful ones. We focus on devices that support software implementations (i.e. microcontrollers) for the advantages that these implementations have over hardware implementations in a dynamic environment such as the IoT. In this chapter, we consider the three embedded devices used by FELICS. A description of these devices is provided in Section 2.6. Table 5.1 summarizes the number of general-purpose registers and the total capacity of these registers for each of the three devices.

MCU	Register size (bits)	GPRs	Number of GPRs	Capacity (bits)
AVR	8	`R0` – `R31`	32	256
MSP	16	`R4` – `R15`	12	192
ARM	32	`R0` – `R12`	13	416

Table 5.1: General-purpose registers (GPRs) of each target device.

One of the main problems raised by the emergence of the IoT is the need for secure and efficient cryptographic algorithms that meet the security requirements and the design constraints of IoT systems. On the one hand, there is the need for basic security functions such as confidentiality, integrity, and availability in these information ecosystems. On the other hand, the use cases for which most of the IoT devices are built impose numerous constraints ranging from the physical dimensions of a device and its battery lifetime to acceptable query response time or throughput. The large body of research on lightweight cryptography aims to conciliate these two conflicting requirements.

5.1.1 Our Contribution

In this chapter, we study how to design lightweight symmetric algorithms that lead to efficient software implementations on embedded IoT devices. We provide a detailed analysis of the costs associated with the main building blocks used to construct lightweight symmetric primitives. Moreover, we give the optimal instruction sequences in terms of both execution time and number of registers required for all operations frequently used in lightweight symmetric cryptography, especially for rotations which are not optimized by C compilers. The optimal implementation of rotations demand a tremendous effort and a very good understanding of the instruction set architecture of each target device.

To the best of our knowledge, this is the first work to provide such a detailed insight into the efficiency of software implementations of lightweight symmetric cryptography. The contribution of this chapter is particularly valuable for designers of new lightweight ciphers because they can make design decisions based on both security and efficiency using our results.

5.2 Efficient Implementations

The efficiency of lightweight cryptographic primitives is determined by the properties of the underlying building blocks. The basic building blocks for lightweight symmetric algorithms are the individual operations that are combined to achieve the desired cryptographic properties. We classify the possible operations in the following four categories: bitwise operations, modular arithmetic operations, rotations, and table-based lookups. The operand size varies between 4 bits for small S-boxes to 32 bits for modular addition. Frequently used operand sizes are 8, 16, and 32 bits.

The evaluation of lightweight block ciphers [105] conducted using the FELICS framework [104] has shown that, in general, assembly implementations give better results than pure C implementations, especially on 8-bit AVR and 16-bit MSP. Although more demanding and less portable than C implementations, assembly implementations allow the implementer to fully control the allocation of registers and give her access to all instructions supported by the target microprocessor. On the contrary, the *GNU Compiler Collection (GCC)* does not always manage efficiently the available registers; thus stack memory is used to compensate for the inefficient allocation of registers. Moreover, the GCC is not able to generate efficient assembly code for certain bit manipulations such as rotations.

For each of the basic operations considered next, we provide the execution time in number of CPU cycles and, when relevant, the number of additional registers required to perform the given operation for different operand sizes. When a particular operation can be implemented in different ways to leverage various trade-offs between the execution time and the number of registers necessary to perform the given operation, we give the same priority to the execution time and the number of used registers. Moreover, when such trade-offs are possible we list the both options (i.e. the high speed implementation that requires more registers and the implementation that uses less registers, but more execution time) such that one can choose a trade-off. The figures correspond to assembly implementations optimized for the above-mentioned criteria. Indeed, the execution time for each operation is optimal in the sense that there is no faster way of performing that operation. Similarly, the number of registers required to perform each operation is minimal, meaning that there is no way to perform that operation using less registers.

5.2.1 Bitwise Operations

Bitwise operations such as NOT ($\neg$), AND ($\wedge$), OR ($\vee$), and exclusive-OR ($\oplus$) can be performed very fast on any microcontroller. When the operand size is less than or equal to the target device's register size, the execution of a bitwise operation takes a single clock cycle. When the operand size is greater than the register size, the execution time of a bitwise operation is equal to the ratio between the operand size and register size because the operand has to be processed in chunks less than or equal to the register size. None of these operations requires additional registers. Table 5.2 summarizes the execution time of the considered bitwise operations for different operand sizes.

Operand size	8 bits				16 bits				32 bits			
Operation	$\neg$	$\wedge$	$\vee$	$\oplus$	$\neg$	$\wedge$	$\vee$	$\oplus$	$\neg$	$\wedge$	$\vee$	$\oplus$
AVR	1	1	1	1	2	2	2	2	4	4	4	4
MSP	1	1	1	1	1	1	1	1	2	2	2	2
ARM	1	1	1	1	1	1	1	1	1	1	1	1

Table 5.2: Execution time of bitwise operations.

To perform a bitwise NOT, AND, OR, and XOR using the 8-bit registers of the AVR microcontroller, one needs the following instructions: `com` (one's complement), `and`, `or`, and `eor`, respectively.

On an MSP microcontroller, the instructions `inv`, `and`, `bis` (bit set or OR), and `xor` perform the same operations (NOT, AND, OR, and XOR) on 16-bit registers. In addition to these instructions, the MSP microcontroller supports the `bic` (bit clear or AND NOT) instruction that takes a single cycle and two bytes of code, thus saving one clock cycle and two bytes of code compared to chaining of `inv` and `and` instructions.

The instructions `mvn`, `and`, `orr`, `eor` are used on the 32-bit registers of ARM to perform NOT, AND, OR, and XOR, respectively. The ARM microcontroller provides two additional bitwise instructions: `bic` (bit clear or AND NOT) and `orn` (OR NOT). Each of these instructions combines two basic bitwise operations into a single instruction to save one clock cycle and four bytes of code.

5.2.2 Modular Arithmetic Operations

The most frequent modular arithmetic operations used by lightweight ciphers are addition and subtraction. For an operand size of n bits, the modular addition and subtraction are performed modulo 2^n. Common choices for n are 8, 16, or 32. The execution time and number of additional registers required to perform modular additions and subtractions for different values of the operand size are given in Table 5.3.

Addition and subtraction of two n-bit values using n-bit registers are straightforward on the three microprocessors. Any of these two operations can be performed using a single instruction (`add`/`sub`) with no need for additional registers.

When the operand size n is a multiple of the register size m, i.e. $n = k \cdot m$ with $k > 1$, then the whole operation is performed in k steps. In the first step, the least significant bits of the two operands are added/subtracted without considering the carry bit. In the subsequent steps, the same operation is repeated, but the carry bit from the previous step is considered (i.e. `adc`/`sbc` – AVR, ARM; `addc`/`subc` – MSP). These steps use k instructions and consequently the execution time is k cycles. The code size is $2 \cdot k$, $2 \cdot k$, and $4 \cdot k$ bytes for AVR, MSP, and ARM, respectively.

Whenever the operand size n is smaller than the register size m, there is at least

MCU	Operand size	8 bits		16 bits		32 bits	
	Operation	$\boxplus$	$\boxminus$	$\boxplus$	$\boxminus$	$\boxplus$	$\boxminus$
AVR	Cycles	1	1	2	2	4	4
	Registers	0	0	0	0	0	0
MSP	Cycles	3	3	1	1	2	2
	Registers	0	0	0	0	0	0
ARM	Cycles	2	2	2	2	1	1
	Registers	0	0	0	0	0	0

Table 5.3: Execution time and number of additional registers required to perform modular arithmetic operations.

one additional instruction (`and`) necessary to ensure that the final result is n bits long. Hence, an 8-bit addition/subtraction on the MSP microcontroller takes two extra cycles and four bytes of code. The same operation takes one extra cycle and four bytes of code on the ARM microcontroller.

The result of an 16-bit addition/subtraction of on ARM can not be masked using `and` instruction with an immediate value. Though, the `movt` instruction can be used to write a 16-bit immediate value (i.e. 0) to the top halfword of a register, without affecting the bottom halfword.

When a 32-bit register of an ARM microcontroller stores a single 8-bit value, then the total cost of an 8-bit addition/subtraction is 2 cycles. One can perform two 8-bit additions/subtractions at the same time on ARM provided that the values are properly arranged in registers. The additional cost of masking the result is three cycles, an additional register, and twelve bytes of code because the mask has to be loaded (`ldr`) into a register (two cycles and eight bytes of code) since it can not be used as an immediate value. If the mask can be kept in the auxiliary register, then the additional cost of subsequent operations is one clock cycle and four bytes of code.

5.2.3 Rotations

It is crucial for the performance of a lightweight primitive to choose the best rotation amounts possible [106, 39] because the cost of rotations greatly varies as a function of the operand size, rotation amount, and target microcontroller. This simple principle is hard to apply in practice in the absence of a detailed analysis of the cost associated with different rotations. Depending on the operand size, we distinguish between rotations of 8-bit values (Table 5.5), rotations of 16-bit values (Table 5.6), and rotations of 32-bit values (Table 5.7). For each possible rotation amount, we give the execution time and the number of additional registers required to perform the given operation.

We follow two optimization goals of equal priority. The first optimization goal is the execution time and consequently some rotations can also be performed using more

Operand size	MCU	Operation	Amount	Metric	Trade-offs		
16-bit	AVR	$\ggg$	3	Cycles	10	12	
				Registers	1	0	
		$\ggg$	5	Cycles	11	12	
				Registers	1	0	
32-bit	AVR	$\lll$	5	Cycles	21	23	25
				Registers	3	1	0
		$\ggg$	4	Cycles	20	24	
				Registers	2	0	
32-bit	MSP	$\lll$	11	Cycles	15	18	
				Registers	1	0	
		$\ggg$	6	Cycles	13	18	
				Registers	1	0	
		$\ggg$	11	Cycles	16	18	
				Registers	1	0	

Table 5.4: Trade-offs between the execution time and number of additional registers required to perform various rotations.

clock cycles but less additional registers. Hence, the execution time is optimal in the sense that it is not possible to perform the same rotations using less clock cycles. The second optimization goal is to use the minimal number of registers possible for each rotation. Therefore, the execution time increases, but the pressure on the registers decreases. These two extreme cases show the trade-offs one can make between execution time and register usage. In general, each additional register required for an operation can be traded for a pair of stack operations (see Section 5.2.5), which increases the execution time. Consequently, for brevity, we list only those trade-offs that give a gain according to this principle in Table 5.4 and we show them in italics in the other tables (Table 5.6 and Table 5.7). However, one can easily explore all possible trade-offs by simply using the values of the basic rotations displayed in bold since all other rotations can be performed using a combination of basic rotations.

For an operand of n bits, we consider left ($\lll$) and right ($\ggg$) rotations by an amount up to $\lceil n/2 \rceil$ bits. To be able to efficiently implement any rotation on the three target devices, it is important to know the supported instructions that can be used to perform rotations for each device. Finding the optimal values for the execution time and number of registers required is an iterative process that requires a tremendous effort. Thus, besides providing the cost of rotations, we also provide the optimal implementations for all basic rotations of 8-, 16-, and 32-bit values in Appendix A, Appendix B, and Appendix C, respectively.

5.2.3.1 8-bit Operand on AVR

The basic rotation amounts for rotation of an 8-bit value on AVR are 1 and 4. Since the microcontroller does not have special instructions to rotate the content of a register by one or more bits, rotations by one bit to the left and to the right take two and three cycles respectively.

The rotation of an 8-bit register by 1 bit to the left can be done using a logical shift to the left by one bit (`lsl`) followed by an addition with carry of zero (`__zero_reg__` or `R1`).

The rotation of an 8-bit register by 1 bit to the right can be performed using three instructions as follows: store the least significant bit of the register to the `T` flag of the status register (`bst`), rotate through carry the content of the register to the right by one bit (`ror`), and load the bit `T` from the status register into the most significant bit of the register (`bld`).

The rotation of an 8-bit value by four bits to the left or to the right takes only one clock cycle on AVR thanks to the `swap` instruction that exchanges the two nibbles of a byte.

5.2.3.2 16-bit Operand on AVR

The are three basic rotation amounts for rotation of a 16-bit value on AVR: 1, 4 and 8. On the AVR microcontroller, a 16-bit value is stored in two 8-bit registers.

The rotation of a 16-bit operand by 8 bits to the left or to the right can be done in 3 clock cycles using three XORs (`eor`) and no additional register.

The rotation of a 16-bit value by 1 bit to the left uses the same principle employed to rotate an 8-bit value by 1 bit to the left; additionally, it requires a rotation by one bit to the left through carry (`rol`). Similarly, the rotation of a 16-bit value by 1 bit to the right makes use of the instructions to rotate an 8-bit value by 1 bit to the right; additionally, it requires a rotation by one bit to the right through carry (`ror`).

The optimal rotations by four bits to the left and to the right can be done in three steps using an auxiliary register. Firstly, the content of the two registers is swapped. Then, the values of the two registers are XORed in the auxiliary register. Depending on the rotation direction, the low or high nibble of the auxiliary register is enabled. Finally, the auxiliary register is XORed to the two registers holding the value to be rotated.

5.2.3.3 32-bit Operand on AVR

The basic rotations for a 32-bit value are rotations by 1, 5, 8, and 16 to the left and rotations by 1, 4, 8, and 16 to the right. A 32-bit value is stored in four registers on AVR.

The rotation of a 32-bit value by 1 bit uses the same technique as the rotation of a 16-bit value by 1 bit, but requires two more rotations by one bit through carry.

The rotations by 4 bits can be done by extracting each nibble of the initial 32-bit value and inserting them on the correct positions in the four registers. These

MCU	Operation	Metric	Amount			
			1	2	3	4
AVR	$\lll$	Cycles	**2**	4	4	**1**
		Registers	**0**	0	0	**0**
	$\ggg$	Cycles	**3**	5	3	**1**
		Registers	**0**	0	0	**0**
MSP	$\lll$	Cycles	**2**	4	6	8
		Registers	**0**	0	0	0
	$\ggg$	Cycles	**2**	4	6	8
		Registers	**0**	0	0	0
ARM	$\lll$	Cycles	**2**	**2**	**2**	**2**
		Registers	**0**	**0**	**0**	**0**
	$\ggg$	Cycles	**2**	**2**	**2**	**2**
		Registers	**0**	**0**	**0**	**0**

Table 5.5: Execution time and number of additional registers required to perform 8-bit rotations.

operations take 20 cycles and require two additional registers to perform a 4-bit rotation.

The rotation of a 32-bit value by 5 bits to the left can done in 21 clock cycles using the hardware multiplier of the AVR microcontroller. The whole operation requires three additional registers and consumes one byte of RAM, since the `__zero_reg__` (`R1`) has to be saved on the stack.

The rotations by 8 bits can be done efficiently by moving the content of each register to its neighbour. These operations can be done using five `mov` instructions and an auxiliary register.

The rotations by 16 bits can be done by simply swapping the content of the four registers using three `movw` instructions and two additional registers. The `movw` instruction copies the content of two registers into another two register in a single clock cycle.

5.2.3.4 8-bit Operand on MSP

The basic rotations for an 8-bit value on MSP are rotations by 1 bit to the left and to the right. Each of these rotations can be done in two steps. The left rotation by one bit consists of an arithmetic rotation to the left by one bit (`rla.b`) followed by the addition of the carry bit to the previous result (`adc.b`). The right rotation by one bit sets the carry bit to the least significant bit of the initial value (`bit`) and then rotates the byte stored in register right through carry (`rrc.b`).

MCU	Operation	Metric	Amount							
			1	2	3	4	5	6	7	8
AVR	$\lll$	Cycles	**3**	6	9	**7**	10	11	7	**3**
		Registers	**0**	0	0	**1**	1	0	0	**0**
	$\ggg$	Cycles	**4**	8	*10*	**7**	*11*	9	6	**3**
		Registers	**0**	0	*1*	**1**	*1*	0	0	**0**
MSP	$\lll$	Cycles	**2**	4	6	8	7	5	3	**1**
		Registers	**0**	0	0	0	0	0	0	**0**
	$\ggg$	Cycles	**2**	4	6	8	7	5	3	**1**
		Registers	**0**	0	0	0	0	0	0	**0**
ARM	$\lll$	Cycles	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**1**
		Registers	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	$\ggg$	Cycles	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**1**
		Registers	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Table 5.6: Execution time and number of additional registers required to perform 16-bit rotations.

5.2.3.5 16-bit Operand on MSP

There are two basic rotations amounts for rotations of a 16-bit value on MSP, namely 1 and 8. Each 16-bit value fully occupies a register.

The rotations of a 16-bit value by 1 bit use the same two steps as rotations of an 8-bit value by one bit, but, for rotations of 16-bit values, the instructions are applied to the whole register and not only to its low byte as in the case of 8-bit values.

The rotation of an 16-bit value by 8 bits can be done in a single clock cycle by swapping the two bytes of a register (`swpb`).

5.2.3.6 32-bit Operand on MSP

The basic rotations amounts for performing rotations of a 32-bit value on MSP are 1, 8, and 16. A 32-bit value is stored in two registers.

The rotations of a 32-bit value by 1 bit use the same techniques presented for rotation of 16-bit values on the same microprocessor, but require one additional instruction per operation: rotate left through carry (`rlc`) for the rotation to the left by one bit and rotate right through carry (`rrc`) for the rotation to the right by one bit.

To rotate a 32-bit value by 8 bits, one extracts the four bytes of the two input registers and places them in the correct position of the output registers. These sequence of operations requires instructions to swap the bytes of a register (`swpb`), to move the content of a register into another register, and to XOR the content of two

MCU	Op.	Metric	Amount															
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
AVR	⋘	Cycles	**5**	10	15	20	***21***	17	11	**5**	10	15	20	25	21	15	9	**3**
		Registers	**0**	0	0	0	***3***	1	1	**1**	1	1	1	1	2	2	2	**2**
	⋙	Cycles	**6**	12	18	***20***	20	15	10	**5**	11	17	23	23	18	13	8	**3**
		Registers	**0**	0	0	***2***	1	1	1	**1**	1	1	1	2	2	2	2	**2**
MSP	⋘	Cycles	**3**	6	9	12	15	12	9	**6**	9	12	*15*	15	12	9	6	**3**
		Registers	**0**	0	0	0	0	1	1	**1**	1	1	*1*	0	0	0	0	**0**
	⋙	Cycles	**3**	6	9	12	15	*13*	10	**7**	10	13	*16*	15	12	9	6	**3**
		Registers	**0**	0	0	0	0	*1*	1	**1**	1	1	*1*	0	0	0	0	**0**
ARM	⋘	Cycles	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
		Registers	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	⋙	Cycles	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
		Registers	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Table 5.7: Execution time and number of additional registers required to perform 32-bit rotations.

registers. An additional register is necessary to properly manipulate the four bytes.

The rotation of a 32-bit value by 16 bits comprises three XORs and does not require additional registers.

5.2.3.7 ARM

Surprisingly, the GCC compiler does not generate optimal code for rotation of an n-bit value, with $n < 32$, by k bits to the right. Namely this operation is done in two cycles using an additional register, but it can be done without the additional register. The assembly code generated by the compiler performs a logical shift right by k bits into the additional register and updates the conditional flags (`lsrs`). Then, it does a bitwise OR (`orr`) between the additional register and the original register shifted to the left (`lsl`) by $n - k$ bits. The rotations to the left are performed using the same sequence of instructions by simply swapping the amounts of the two shifts.

The optimal sequence of instructions required to rotate a n-bit value, with $n < 32$ by k bits to the right uses the `bfi` instruction to replace k bits of the data register starting at the low-bit position n, with k bits starting from bit 0. Then, it performs a rotation to the right by k bits on the same register. The rotations to the left can be performed using the same sequence of instructions by simply changing the amounts of the two instructions from k to $n - k$.

The rotation of a 16-bit value by 8 bits to the left or to the right can be done in a single clock cycle without any additional register using the `rev16` instruction that reverses the byte order in each halfword (16 bits) of the register.

MCU	Chunk	4-bit lookup		8-bit lookup	
		Cycles	Registers	Cycles	Registers
AVR	0	6	2	5	2
	1	7	2	–	–
MSP	0	4	0	4	0
	1	11	0	5	0
	2	5	0	–	–
	3	12	0	–	–
ARM	0 – 3	4	2	4	2
	4 – 7	4	2	–	–

Table 5.8: Execution time and number of additional registers for table lookup operations. All bits of a register are used to store data. Data chunks are counted in increasing order from least significant bit to most significant bit.

The ARM microcontroller has a barrel shifter able to perform rotations of 32-bit values to the right by any amount within a single clock cycle. This can be accomplished using the `ror` instruction. Moreover, a shift or rotation followed by a bitwise operation or a modular addition/subtraction can be done in a single cycle thanks to the flexible second operand being passed through the barrel shifter.

5.2.4 Table-Based Lookups

Most frequently, lookup tables are used for the nonlinear layer of a cipher. Sometimes, lookup tables are also used for the linear layer. A lookup table T returns an n-bit value $T(x)$ for the m-bit input value x. Typically, the values of m and n are equal to four or eight, but they can also be different as in case of Robin, Fantomas, and Mysterion [153, 176]. The lookup table can be stored in flash memory or in RAM. If the code for generation of the lookup table is smaller than the table itself, then the lookup table can be computed during the initialization of the device and stored in RAM to save flash memory. Such a trade-off is possible, for example, for the S-box of SKINNY [42].

Table lookups are usually slower than most of the operations performed directly on registers because they require at least one memory access. The execution time and the number of additional registers for different table lookups are given in Table 5.8.

For the AVR microcontroller, the execution time of a load operation depends on the memory type: a load from flash (`lpm`) takes 3 clock cycles, while a load from RAM (`ld`) takes 2 clock cycles. The results reported consider only loads from flash.

5.2.4.1 8-bit Table on AVR

The naive way of doing a table lookup takes 7 cycles and 10 bytes of code. If the table is aligned in memory (on a 256 bytes boundary), then a lookup can be performed in 4 cycles (+1 cycle for loading the table's memory address) and takes 4 bytes of code (+2 bytes for loading the table's memory address).

5.2.4.2 4-bit Table on AVR

Two 4-bit table lookups are slower than a single 8-bit table lookup on AVR because processing the two nibbles of a byte requires additional instructions to extract each nibble from a register and to store them back. The overhead of a straightforward implementation using two 4-bit table lookups is 13 clock cycles and 22 bytes of code compared to a straightforward implementation using a single 8-bit table lookup. The high nibble requires one additional instruction (1 clock cycle and 2 bytes of code) compared to the low nibble. A possible trade-off between memory and execution time is to increase the table size from 2^4 bytes to 2^8 bytes to be able to perform two 4-bit lookups at a time using a single 8-bit lookup.

5.2.4.3 8-bit Table on MSP

When a single 8-bit value is kept in a register, the execution time of an 8-bit table lookup is 3 clock cycles and its code size is 4 bytes. If each register stores two bytes of data, the overhead (associated with extracting the two bytes of a 16-bit register and inserting them back into the register) is 8 clock cycles and 16 bytes of code compared to processing two registers that hold a single 8-bit value each. On an MSP microcontroller it is not possible to perform two 8-bit lookups at the same time, because the size of the lookup table (2^{16} bytes) exceeds the available memory (both flash and RAM).

5.2.4.4 4-bit Table on MSP

To perform 4-bit table lookups when a 16-bit register contains four data values on MSP, one has first to extract the correct nibble of a byte, perform the lookup, and then store the nibble back. While the lookup alone takes only 3 cycles, the process of extracting the nibble and storing it back adds an overhead of between 1 and 9 cycles depending on the nibble position in the register. To speed up the lookups, the initial lookup table of 2^4 bytes can be replaced with a lookup table of 2^8 bytes to facilitate two 4-bit lookups at the same time.

5.2.4.5 8-bit Table on ARM

Assuming that a 32-bit register is used to store four data bytes, a straightforward implementation of an 8-bit lookup on ARM requires 5 cycles and 4 additional registers. When several loads are done one after the other, the instructions can pipeline their address and data phases. Consequently, the execution time is decreased to $2 + 3 \cdot N$ cycles, where N is the number of consecutive load instructions. If a 32-bit register

Instruction	AVR	MSP	ARM
`push`	2	3	$1 + N$
`pop`	2	2	$1 + N$

Table 5.9: Execution time of stack operations. N is the number of registers in the register list to be loaded or stored.

stores a single 8-bit value, then the execution time of an 8-bit table lookup is 2 cycles when instructions are not pipelined and $1 + N$ cycles when N instructions are pipelined.

5.2.4.6 4-bit Table on ARM

The technique described above can be employed on ARM to perform 4-bit table lookups. The size of the lookup table can be increased to 2^8 or 2^{16} bytes to perform two or four lookups at the same time.

5.2.5 Stack Operations

The stack is the program memory (RAM) used to keep track of the address to which each active subroutine should return control when it finishes executing. In addition to this, the stack is used to pass parameters to a subroutine or for local data storage.

When the available registers of a microcontroller are not enough to keep all data, then stack operations are necessary to store the content of registers on the stack and load them back. The `push` instruction stores the content of a register on the stack. Similarly, the `pop` instruction loads a register from the stack.

The execution time of the stack operations is provided in Table 5.9. Whenever possible, stack operations should be avoided because they are slow and increase the memory consumption. On an ARM microcontroller the `push` and `pop` instructions can receive a list of registers. In this case, thanks to the pipeline, the execution time is $1 + N$ cycles, where N is the number of registers in the register list to be loaded or stored. The code size of a stack instruction is 2 bytes on any of the three platforms.

5.3 Discussion

5.3.1 Choosing the Best Operations

The reader can easily see that the most efficient operations are bitwise logical operations, followed by modular arithmetic operations. In general, rotations are the worst operations, especially rotations of large operands by values at equal distance from the neighbouring basic rotation amounts. At the same time, different rotation amounts offer different trade-offs between security (differential and linear probabilities) and efficiency. Hence, it is important to carefully chose the best rotation amounts as described in [38, 106].

Although the execution time for 8-bit table lookup operations is not very high, the size of the table adds a significant penalty on the code size or RAM consumption. On the other hand, the 4-bit lookup tables have a lower footprint, but a higher execution time and thus they are not very efficient.

5.3.2 Choosing the Best Word Size

An important design decision with a profound impact on the performance of a lightweight symmetric algorithm is the so-called *word size*. In this context, the *word size* gives the operand size and should not be confused with the word size of a processor, which is typically equal to the number of bits that can be stored in a register.

The most efficient implementations on a particular microcontroller are reached when the *word size* is equal to the register size of that microcontroller. The worst results are obtained when the register size is greater than the operand size. Consequently, when the *word size* is a multiple of the register size, the efficiency of the implementation is in between the the best and worst possible figures.

It is desirable that an implementation of a lightweight symmetric algorithm is efficient on a wide range of microcontrollers characterized by different register sizes. To achieve this goal, a designer should chose the cipher's *word size* equal to the largest register size since this has no influence on the platforms with the largest register size and adds only minor penalties for devices with smaller register sizes.

5.3.3 Substitution Layer

Typically, the substitution layer is implemented using table lookups. Another option is to use the algebraic normal form (ANF) of an S-box, which requires only bitwise operations. The popularity of the second option significantly increased in the last years because, compared to lookup tables, implementations based on ANF do not require memory (flash or RAM) to store the mappings between inputs and outputs. Moreover, they achieve similar execution times, but may require more registers. This has led to bit-sliced implementations of S-boxes in ciphers such as PRIDE [9], RECTANGLE [391], or RoadRunneR [35].

5.3.4 Linear Layer

Sometimes, the linear layer is the most expensive transformation used by a cipher because it requires additional registers to permute the bits of the cipher's state. Thus, special attention must be payed to the selection of the linear layer, in particular for ciphers designed to be implemented on the 16-bit MSP microcontroller, which has only 12 general-purpose registers.

5.3.5 Cipher's State

A critical property for software efficiency of a lightweight cipher is its state size, namely whether the state fits into the available registers or not. An efficient implementation should keep the full state of a cipher in registers and, in addition to this, it should have enough registers to handle round keys and other operations that may require temporary registers. In this way, the overhead generated by register spills can be avoided. This goal is difficult to achieve on MSP, which has a register capacity of only 192 bits as shown in Table 5.1.

5.3.6 Structure

Block ciphers represent a major fraction of the lightweight symmetric primitives designed so far. They can be built using different structures, the most widely used being: Substitution-Permutation network (SPN), Feistel network (FN), and Lai-Massey. When analyzing these three structures, one can see that Feistel networks are software friendly. Typically, each operation affects only one of the two branches of a Feistel network and thus the number of temporary registers is minimal compared to a Lai-Massey structure, which updates both branches at the same time, or an SPN structure, which uses heavy linear layers. In addition, Feistel ciphers usually have simpler round functions and consequently need more rounds than SPN ciphers. Nevertheless, there are exceptions from these observations. For example, bit-sliced designs favour efficient software implementations regardless of their structure (RECTANGLE [391] is an SPN, RoadRunneR [35] is a FN).

Usually, ARX designs (which use only modular addition/subtraction, rotations, and bitwise XOR) have very efficient software implementations (e.g. Chaskey [244]). A variation of this design strategy that replaces modular addition/subtraction with bitwise AND leads to very efficient software implementations as well (e.g. SIMON [36]).

5.4 Summary

In this chapter, we have conducted a comprehensive analysis of the implementation cost associated with basic building blocks of a lightweight symmetric algorithm. At the same time, we provided optimal cost implementations for rotations of various operand sizes (8, 16, and 32 bits) on three microcontrollers (8-bit AVR, 16-bit MSP, and 32-bit ARM) widely used for IoT applications.

This chapter provides detailed insights into efficiency and security of software implementations of lightweight symmetric cryptography. The comprehensive analysis can aid the development of better lightweight symmetric cryptographic algorithms for software implementations.

Part II

Side-Channel Attacks

Chapter 6

Resilience to Correlation Power Analysis Attacks

Contents

6.1 Introduction

For a long time, it was widely believed in the cryptographic community that side-channel attacks are primarily an implementation problem rather than a design problem, i.e. there is little that can be done from a designer's perspective to eliminate or reduce the leakage of sensitive information. However, some recent research results have started to challenge this view. So does the work described in this chapter. More concretely, it shows that the operations of a symmetric algorithm, which are selected in the design phase, influence differently its resistance to side-channel attacks.

Previous research at the intersection between lightweight cryptography and SCA focused (almost) exclusively on the AES, i.e. there exist only few papers that deal with attacks or countermeasures for other ciphers. In particular, the study of the SCA-resistance of software implementations of lightweight ciphers did not keep pace with the high number of new proposals. In [30], the resilience of the AES and three lightweight block ciphers that share some characteristics (namely KLEIN, LED, and PRESENT) is investigated against profiled single-trace attacks. Unprotected hardware implementations of Simon and LED were analyzed with respect to DPA in [318]. An evaluation of both an unmasked and a masked implementation of Simon for FPGAs was reported in [48]. In [315], the vulnerability of PRINCE and RECTANGLE against DPA is studied. A second line of research focused on the design of new ligthweight primitives that can be efficiently protected against DPA via masking; representative examples include PICARO [276], Zorro [140], and the LS-designs Robin and Fantomas [153].

The above-mentioned studies on DPA attacks against (lightweight) ciphers other than the AES were mainly "isolated" efforts in the sense that they were carried out on different execution platforms with different measurement setups and different analysis frameworks. A comparative (and consistent) study of the DPA-vulnerability of lightweight block ciphers based on power traces acquired from the same target device is, to our knowledge, still missing. However, such a study would allow one to answer the question of whether different ciphers are equally difficult to attack or not (and if not, why not). Furthermore, we could not find a detailed analysis of the power leakage of basic operations (e.g. arithmetic and logical computations, table lookups) executed in the round function of common lightweight ciphers. Thus, in this chapter, we first try to answer the following questions:

- How do the theoretical metrics used to assess leakage relate to real-world attack results?
- Which operation leaks more?

Then, we apply the answers of these questions to illustrate how eight lightweight ciphers (namely AES, Fantomas, LBlock [386], Piccolo [321], PRINCE [62], RC5 [296], as well as Simon and Speck [37]) behave with respect to CPA. These eight ciphers were selected from the portfolio of lightweight symmetric algorithms evaluated in [105] using the FELICS framework [93]. The two main selection criteria were high performance and to have a variety of different design strategies.

In this chapter we focus on CPA attacks against unprotected implementations. We say that an implementation leaks more than another implementation when it is easier to attack the first implementation using CPA than the second one, i.e. the correct key can be recovered with less effort.

All results and findings we describe in this chapter are based on CPA attacks performed with power consumption traces that were captured on an evaluation board equipped with an 8-bit AVR microcontroller. Our choice for this specific platform is motivated by the widespread use of the 8-bit AVR architecture in resource-limited environments and its particular relevance in the context of the IoT (e.g. wireless sensor nodes). A better understanding of the actual leakage of different operations on 8-bit AVR microcontrollers could influence the design of new lightweight ciphers for the

IoT and the implementation of more effective and less costly SCA countermeasures. For example, it is a known fact that the AES leaks significantly due to its highly nonlinear S-box [73], but modern lightweight ciphers generally use smaller S-boxes with lower nonlinearity compared to the AES, and thus one might expect that they leak less. However, an actual confirmation of this assumption with measured traces is still lacking.

We remark that the evaluation of candidates for the NIST SHA-3 standard considered besides security and performance on various hardware and software platforms also SCA resistance as a selection criterion (see e.g. [44, 395] for some concrete results). Currently, a number of standardization bodies, including NIST, are either considering or have already started the process to standardize lightweight symmetric primitives for the IoT. In this context, it makes sense to compare different aspects of potential candidates, including the SCA resistance of (unprotected) software implementations, before deploying them on millions or even billions of devices. This chapter contributes to a better understanding of how to design lightweight block ciphers that have a better *intrinsic* resistance against side-channel attacks.

6.1.1 Research Contributions

Firstly, we quantify the leakage generated by the execution of different instructions on an AVR processor, aiming to identify the instructions that leak most. Then, we compare the power consumption leakage of basic operations widely used by lightweight ciphers. For each operation, we analyze the relation between our experimental results, the nonlinearity of the operation, and the size (in bits) of the attacked intermediate value.

Secondly, we provide a fair comparison of the resilience of eight lightweight block ciphers against CPA attacks. Knowing which instructions and operations leak more, and knowing all implementation details of the eight ciphers helps to identify the weakest point of each cipher, which can be attacked with maximal efficiency. Our experimental results show that, in some cases, the actual leakage is lower than expected due to certain implementation-related aspects.

The practical approach we follow has the benefit that it gives more realistic results compared with simulated power traces, where the noise is modeled in a deterministic way, which favors the attacker. Thus, our work sheds new light on the resilience of different operations against CPA attacks, and we illustrate this for a set of eight lightweight block ciphers. To the best of our knowledge, there has been no similar effort published in the literature.

6.2 Preliminaries

We use the following operators for the corresponding (bitwise) logical operations: "$\wedge$" for AND, "$\vee$" for OR, "$\oplus$" for XOR. The operators "$\boxplus$" and "$\boxminus$" denote a modular addition and a modular subtraction, respectively. The two functions $\mathsf{MSB}(x)$ and $\mathsf{LSB}(x)$ are used to extract the most and the least significant byte from a stream of bits x, respectively. We represent the S-box layer of a block cipher α by S_α, which may involve the application of one or more S-boxes in parallel, depending on the input size and the specifications of the cipher. The symbol $L^{-1}_{i,Fantomas}$ stands for the result of the inverse linear layer of Fantomas computed with L-box i, where $i \in \{0, 1\}$. Finally, $\mathsf{HW}(x)$ denotes the Hamming weight of x, whereas $\mathsf{HD}(x, y) = \mathsf{HW}(x \oplus y)$ is the Hamming distance between x and y.

Definition 6.2.1 (Iterated Block Cipher)**.** *An iterated block cipher, sometimes called a product cipher, is a block cipher obtained by iterating r times a round function $R : \{0, 1\}^n \to \{0, 1\}^n$, each time with its own key $K_i \in \mathcal{K}$, where $\mathcal{K}$ is called round key space. The cipher block size is n bits, the number of rounds is equal to r, $X^{(0)}$ is the plaintext, and $X^{(r)}$ is the ciphertext. It works as follows:*

$$X^{(i)} = R_{K_i}(X^{(i-1)}) \text{ for } 1 \le i \le r$$

Definition 6.2.2 (Selection Function)**.** *In the context of side-channel attacks, a selection function gives the intermediate result, also referred to as sensitive value $\phi_k = \varphi(x, k)$, which is used by the attacker to recover the secret key. It depends on a known part x of the input $X^{(i-1)}$ of the round function R_{K_i} and on an unknown part k of the round key K_i.*

The attacker computes the intermediate values ϕ_k for a fixed (either known or chosen) input x and for all possible subkeys k. The bit-size $|k|$ of the subkey k determines the memory complexity m of the side-channel attack. Then, she uses the sensitive values $\phi_1, \phi_2, \ldots, \phi_{2^{|k|}}$ and the side-channel leakage to guess the subkey k^* used during the actual computations on the target device. The higher the number of inputs x for which the attacker manages to measure the leakage, the higher the chances to recover the subkey k^*. Usually, the selection functions are chosen to be easy to compute, typically at the first round of the encryption or decryption operation.

Definition 6.2.3 (Correlation Power Analysis (CPA))**.** *Given a set of power traces and the corresponding sets of intermediate values $\phi_1, \phi_2, ...\phi_{2^{|k|}}$, Correlation Power Analysis (CPA) aims at recovering the secret subkey k^* using a correlation factor between the measured power samples and the power model of the computed sensitive values.*

The concept of CPA was studied as an improvement of DPA and formalized in [68]. A power model is used to describe the hypothetical power consumption of the target device as a function of the intermediate value ϕ_k considering the device's power consumption characteristics. The Hamming weight (HW) model is more common for software implementations, whereas the Hamming distance (HD) model is generally used for hardware devices.

6.2.1 Theoretical Metrics for the SCA Resistance of S-Boxes

In the definitions introduced in this subsection, we denote by "+" the addition of integers in $\mathbb{Z}$ and by "$\oplus$" the addition mod 2. We will also use "+" for the addition of two vectors in $\mathbb{F}_2^n$ since there is no ambiguity. For a pair of vectors $a = (a_1, a_2, \ldots, a_m)$ and $b = (b_1, b_2, \ldots, b_m)$ from $\mathbb{F}_2^m$, the scalar product $a \cdot b$ is defined as $a \cdot b = \oplus_{i=1}^m a_i \cdot b_i$.

One way to achieve nonlinearity in symmetric cryptographic primitives is to use S-boxes. Formally, an S-box is an (n, m) function $F : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$ that maps n input bits to m output bits. If $m = 1$, then F is nothing else than a Boolean function. For any given (n, m) function F, we denote by $(F_1, F_2, \ldots, F_m)$ the coordinate functions of F, such that $F(x) = (F_1(x), F_2(x), \ldots, F_m(x))$, where $F_i : \mathbb{F}_2^n \mapsto \mathbb{F}_2$ for $1 \leq i \leq m$. The *derivative* of F with respect to a vector a in $\mathbb{F}_2^n$ is the function $D_aF : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$ such that $D_aF(x) = F(x) + F(x + a)$. The *Walsh transform* of F is the function $W_F(u, v) = \sum_{x \in \mathbb{F}_2^n} (-1)^{v \cdot F(x) + u \cdot x}$, while the *cross-correlation transform* of Boolean functions F_i and F_j with respect to a vector $a \in \mathbb{F}_2^n$ is defined as $C_{F_i,F_j}(a) = \sum_{x \in \mathbb{F}_2^n} (-1)^{F_i(x)+F_j(x+a)}$.

Definition 6.2.4 (Nonlinearity)**.** *The nonlinearity of an (n, m) function F is defined as:*

$$\mathsf{NL}(F) = 2^{n-1} - \frac{1}{2} \max_{\substack{u \in \mathbb{F}_2^n \\ v \in F_2^{m*}}} |W_F(u, v)| \tag{6.1}$$

Nonlinearity characterizes the resistance of F against linear cryptanalysis [230]. The higher the nonlinearity of a function, the more resistant the function is to linear cryptanalysis. It is widely accepted that the higher the nonlinearity of a function F, the more information it leaks through side channels.

Definition 6.2.5 (Transparency Order)**.** *The Transparency Order of an (n, m) function F, where n and m are two positive integers, is:*

$$\mathsf{TO}(F) = \max_{\beta \in \mathbb{F}_2^m} \left(\left| m - 2\mathsf{HW}(\beta) \right| - \frac{1}{2^{2n} - 2^n} \sum_{a \in \mathbb{F}_2^{n*}} \left| \sum_{\substack{v \in \mathbb{F}_2^m \\ \mathsf{H}(v)=1}} (-1)^{v \cdot \beta} W_{D_aF}(0, v) \right| \right)$$

The *Transparency Order* was introduced in [279] to "quantify" the resistance of an S-box against DPA attacks using the Hamming weight power model. In general, the smaller the transparency order of F, the higher is its resistance to DPA attacks. $\mathsf{TO}(F)$ satisfies the following relation: $0 \leq \mathsf{TO}(F) \leq m$.

Definition 6.2.6 (Improved Transparency Order)**.** *The Improved Transparency Order of a balanced (n, m) function F is defined as:*

$$\mathsf{ITO}(F) = \max_{\beta \in \mathbb{F}_2^m} \left(m - \frac{1}{2^{2n} - 2^n} \sum_{a \in \mathbb{F}_2^{n*}} \sum_{j=1}^{m} \left| \sum_{i=1}^{m} (-1)^{\beta_i + \beta_j} C_{F_i,F_j}(a) \right| \right)$$

The *Improved Transparency Order* addresses the limitations identified in the initial definition of TO [77].

Definition 6.2.7 (DPA Signal-to-Noise Ratio)**.** *The DPA Signal-to-Noise Ratio of function F is defined as:*

$$\mathsf{SNR}(F) = m2^{2n} \left(\sum_{a \in \mathbb{F}_2^n} \left(\sum_{i=0}^{m-1} \left(\sum_{x \in \mathbb{F}_2^n} (-1)^{F_i(x) + x \cdot a} \right) \right)^4 \right)^{-\frac{1}{2}}$$

The *DPA Signal-to-Noise Ratio* was proposed in [154] as a way to model the information leakage of CMOS circuits using the tools of traditional cryptanalysis. The SNR increases when the resistance of an S-box to linear and differential cryptanalysis increases. A novel definition of the SNR based on the maximum likelihood estimator was introduced in [155].

For an extensive comparison of the metrics proposed to assess the intrinsic resistance to side-channel analysis attacks of a given S-box at the design stage, we refer the reader to [341].

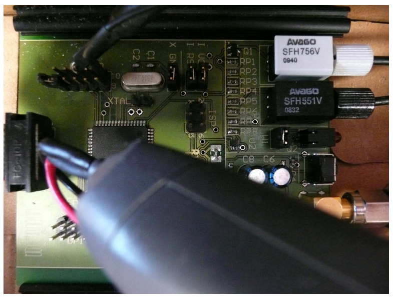
6.3 Evaluation Framework

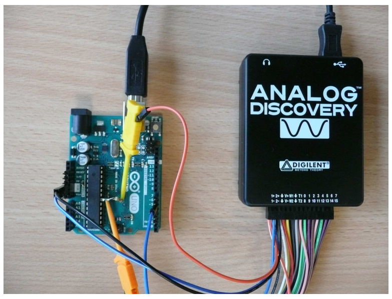
6.3.1 Measurement Setup

All experiments reported in this chapter were performed using two different measurement setups. The main difference between the two setups stems from the budget spent on equipment. The first setup costs more than \$5,000 and is referred to as a high-cost setup, while the second setup is worth less than \$300 and thus it is deemed to be a low-cost setup.

The first setup consists of on an evaluation board equipped with an 8-bit ATmega2561 processor clocked at 16 MHz as shown in Figure 6.1a. A regulated power supply provides the 5 V supply voltage required for the operation of the board. The evaluation board and the computer used to control the measurements are connected through optical fiber. We placed the board in a Faraday cage to reduce the environmental noise. The measurements of the power traces were performed with a LeCroy waveRunner 104MXi digital sampling oscilloscope using a differential probe.

The second setup uses an Arduino Uno board based on the ATmega328P microcontroller and an Analog Discovey oscilloscope from Digilent as shown in Figure 6.1b. Unlike the first setup, the second setup does not use noise reduction techniques.


(a) First setup.



(b) Second setup.

Figure 6.1: Measurement setups.

We mounted the CPA attacks against the ANSI C implementations of the selected ciphers available in the FELICS framework [93]. The only modification of the original C source codes we made was the insertion of a trigger signal to indicate the beginning and the end of the side-channel relevant portion of the power traces. To have a common ground for comparison, we assumed that the attacker needs to recover the 32 bits of the round key $K_1 = \text{0x01234567}$ for all eight block ciphers. Note that, in all of our experiments, we acquired the same number of traces, namely q for the encryption of q known plaintexts.

6.3.2 Metrics

To ensure a fair and uniform side-channel evaluation of the selected ciphers, we used the evaluation methodology for key-recovery attacks proposed in [336]. In that paper, two different types of evaluation metrics are defined: an information-theoretic metric quantifying the amount of information that leaks from a given implementation, and an actual security metric, which quantifies how well the leaked information can be used by the attacker.

Since we conducted a practical evaluation based on leakages acquired from a target board using the described setup instead of attacks based on simulated power traces, the actual security metrics (i.e. success rate and guessing entropy) are sound for our study. We do not use the information-theoretic metric from [336] (i.e. conditional entropy) because it involves profiling the target device in order to approximate the probability distribution of the leakage, which reduces the applicability of the attack to a certain class of devices. Moreover, both the template creation and the approximation of the probability distribution for all leakage samples are computationally intensive.

We recall that side-channel attacks are generally performed using a divide-and-conquer approach. The adversary attacks a subkey class κ with $|\kappa| \ll |\mathcal{K}|$ using the selection function $\varphi(x, k)$ and q measurements. As result she gets a guess vector $g = [g_1, g_2, \ldots, g_{2^{|k|}}]$ for the subkey k with the possible candidates sorted in descending order, the most-likely subkey candidate being g_1, and the least-likely subkey candidate being $g_{2^{|k|}}$. The following two metrics quantify the amount of effort required to recover the correct subkey k^* from the guess vector. Consequently, they serve as an indicator of how efficient an attack is in the case of q measurement queries.

Definition 6.3.1 (Success Rate). *The success rate of order o, $o \leq 2^{|k|}$, of a side-channel key recovery attack is defined as:*

$$\mathsf{SR}_o(k^*, g) = \begin{cases} 1, & \textit{if } k^* \in [g_1, g_2, \ldots, g_o] \\ 0, & \textit{otherwise} \end{cases}$$

Definition 6.3.2 (Guessing Entropy). *The guessing entropy of a side-channel key recovery attack is:*

$$\mathsf{GE}(k^*, g) = \log_2 i, \textit{ such that } k^* = g_i \textit{ for } g_i \in [g_1, g_2, \ldots, g_{2^{|k|}}]$$

Given an implementation $\mathcal{C}$ to be evaluated using N experiments with the maximum number of measurement queries q, the memory complexity m, and the time complexity t, Algorithm 1 shows in detail how the mean success rate of order

o, i.e. $\overline{\mathsf{SR}^i_o}$, and the mean guessing entropy, i.e. $\overline{\mathsf{GE}^i}$, can be computed for i power consumption traces. The results are accompanied by the respective standard errors $\mathsf{SE}_{\overline{\mathsf{SR}^i_o}}$ and $\mathsf{SE}_{\overline{\mathsf{GE}^i}}$. Unless otherwise specified, the results in this chapter are based on $N = 100$ experiments, each with $q = 2000$ queries. Both the time complexity t and memory complexity m were determined by guesses of at most 8-bit subkeys of the round key K_1, where k^* is the actual key used by the implementation $\mathcal{C}$.

Algorithm 1 CPA evaluation algorithm

```
Input: $\mathcal{C}$, $k^*$, $q$, $m$, $t$, $N$
Output: $\overline{\mathsf{SR}^i_o}$, $\overline{\mathsf{GE}^i}$, $\mathsf{SE}_{\overline{\mathsf{SR}^i_o}}$, $\mathsf{SE}_{\overline{\mathsf{GE}^i}}$
 1: for $j$ in $[1, N]$ do
 2:     $\mathsf{AcquirePowerTraces}(\mathcal{C}, k^*, q)$
 3:     for $i$ in $[5, q]$ do
 4:         $g = \mathsf{CPA}(\mathcal{C}, i, m, t)$
 5:         compute and store $\mathsf{SR}^{j,i}_o(k^*, g), \mathsf{GE}^{j,i}(k^*, g)$
 6:     end for
 7: end for
 8: for $i$ in $[5, q]$ do
 9:     compute $\overline{\mathsf{SR}^i_o} = \frac{1}{N}\sum_{j=1}^{N} \mathsf{SR}^{j,i}_o(k^*, g), \overline{\mathsf{GE}^i} = \frac{1}{N}\sum_{j=1}^{N} \mathsf{GE}^{j,i}(k^*, g)$
10:     compute $\mathsf{SE}_{\overline{\mathsf{SR}^i_o}}, \mathsf{SE}_{\overline{\mathsf{GE}^i}}$
11: end for
```

6.4 Quantifying the Leakage

Using the first measurement environment described before, we quantify the leakage of different instructions to find out which instruction gives the "best" target in the power traces when performing a CPA attack. For this purpose, we define the *correlation coefficient difference* $\delta = c_{k^*} - c_{k^\diamond}$ as the difference between the correlation coefficient of the correct key k^*, i.e. c_{k^*}, and the correlation coefficient of the most likely key guess $k^\diamond$, i.e. $c_{k^\diamond}$, with $k^\diamond \neq k^*$.

In this work, we use the correlation coefficient difference to quantify the leakage of different selection functions in the context of CPA attacks. We selected this metric because it is simple and describes well the result of a CPA attack. Moreover, it can be applied to both measured and simulated traces. For a better understanding of how the metric works, we give a graphical representation of the correlation coefficient difference spectrum in Figure 6.2. Our metric works regardless the sign of the correlation coefficient of the correct key k^*, i.e. c_{k^*}, and the correlation coefficient of the most likely key guess $k^\diamond$, i.e. $c_{k^\diamond}$. However, the correlation coefficient difference should be used with care in other contexts such as when comparing distinguishers [293].

The *mean correlation coefficient difference* $\bar{\delta}$ is the arithmetic mean of all values of the correlation coefficient difference δ such that each value of δ corresponds to a different correct key (k^*) and all possible values of $\mathsf{HW}(k^*)$ are considered once.

For the measurements we used a simple assembly code fragment that contains

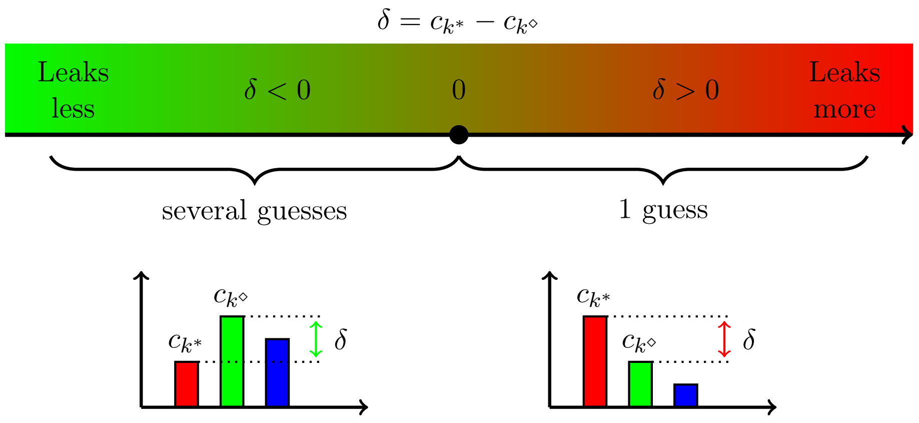


Figure 6.2: Correlation coefficient difference spectrum.

the targeted assembly instruction guarded by several `nop` instructions to reduce the noise from other operations such as the communication between the board and the computer or the peaks of the trigger signal. The measurements were done with values of the correct key k^* such that $\mathsf{HW}(k^*)$ runs through all possible values once. For a fixed value of the input plaintext x and key k^*, we averaged eight power measurements of the analyzed instruction to get a single power trace. The plaintext took all possible values from 0x00 up to 0xFF; thus the number of traces q is 256. We performed $N = 10$ experiments for each value of k^*.

6.4.1 Understanding the Device's Leakage

Understanding the device's leakage requires to understand how different assembly instructions executed by the processor can impact the power consumption of the device. For this purpose, we evaluated two instructions that operate on registers (namely `and` and `add`) as well as three instructions that require access to memory (namely `lpm`, `ld`, and `st`). The `and` instruction performs a bitwise AND of two 8-bit words, while the `add` instruction executes a modular addition of two 8-bit words. Loading an 8-bit word from the flash memory of the device into a register can be achieved through the `lpm` instruction, whereas loading an 8-bit quantity from RAM into a register requires a `ld` instruction. Finally, the `st` instruction writes the content of an 8-bit register to memory. We used the AES S-box with the index value given by the plaintext XORed with the key to perform the memory accesses.

Our results given in Table 6.1 show that the memory-access instructions leak a lot more information about the secret key than the register instructions. The writing of a register to memory leaks most, followed by the loading of a word from memory. At the other end of the spectrum is the `and` instruction, which is leaking approximately 20 times less than the `add` instruction (see Table 6.1 and Figure 6.3). We also observed that increasing the number of power traces does not significantly change the values of δ.

Instr.	Correct key									$\bar{\delta}$	$\mathsf{SE}_{\bar{\delta}}$
	`0x00`	`0x01`	`0x03`	`0x07`	`0x0F`	`0x1F`	`0x3F`	`0x7F`	`0xFF`		
`and`	-0.798	-0.643	-0.577	-0.518	-0.465	-0.392	-0.329	-0.178	-0.016	-0.435	0.183
`add`	0.190	-0.218	-0.160	-0.079	-0.053	0.001	0.049	0.041	0.001	-0.025	0.093
`lpm`	0.376	0.312	0.271	0.219	0.174	0.169	0.164	0.156	0.143	0.220	0.062
`ld`	0.244	0.200	0.178	0.225	0.215	0.226	0.215	0.195	0.222	0.213	0.015
`st`	0.596	0.581	0.578	0.577	0.566	0.594	0.603	0.585	0.592	0.586	0.008

Table 6.1: Correlation coefficient difference $\delta = c_{k^*} - c_{k^\diamond}$ between the correlation of the correct key (i.e. c_{k^*}) and the correlation of the most likely key (i.e. $c_{k^\diamond}$) where $k^\diamond \neq k^*$ for different Hamming weights of the correct key k^* ($\bar{\delta}$ and $\mathsf{SE}_{\bar{\delta}}$ are the mean and the standard error for a 95% confidence interval, respectively).

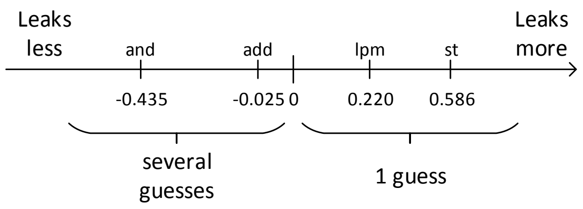


Figure 6.3: Correlation coefficient difference spectrum for four assembly instructions.

Although these experiments may remind the reader about template attacks (where the attacker creates in the profiling phase leakage templates for various instructions), we stress that we did not perform actual template attacks, but we used a technique inspired by classical template attacks to quantify the leakage of different assembly instructions. Our results indicate that an attacker should target the store of a sensitive value to increase the success rate of the attack.

6.4.2 Comparison of Different Selection Functions

We now extend the previous experiments to different selection functions, whereby we target the writing of the selection function's result to memory using the `st` instruction, which, as we saw, has the highest leakage. Table 6.2 summarizes the nonlinearity NL and the mean correlation coefficient difference $\bar{\delta}$ for a total of 16 different selection functions, which are divided into four groups. Detailed values for different correct keys can be found in Table 6.3.

The first group of selection functions comprises the three logical operations AND, OR, and XOR, which all have a negative value for the mean correlation coefficient difference $\bar{\delta}$. This means that using one of these logical operations as a selection function for a CPA attack is not a very good option. As our results show, only the AND and OR, but not XOR, are sometimes able to recover the correct key k^*, whereby AND is slightly more efficient than OR.

One can notice the contrast between the huge nonlinearity of the AND and OR selection functions on the one side, and all other selection functions listed in Table 6.2

on the other side. It is also interesting to note that these high values of nonlinearity are accompanied by (relatively) poor values for the correlation coefficient difference. In the case of the bitwise logical operations, it seems the high nonlinearity values do not provide the useful leakage one normally would expect. This contrasts with the conventional wisdom saying that the higher the nonlinearity of a selection function, the more information it leaks in SCA.

Selection function	n	m	NL	$\bar{\delta}$	$\mathsf{SE}_{\bar{\delta}}$
$\varphi_1(x,k) = x \wedge k$	16	8	16384	-0.005	0.074
$\varphi_2(x,k) = x \vee k$	16	8	16384	-0.018	0.060
$\varphi_3(x,k) = x \oplus k$	16	8	0	-0.153	0.168
$\varphi_4(x,k) = x \boxplus k$	16	8	0	0.127	0.011
$\varphi_5(x,k,c) = x \boxplus k \boxplus c$	17	8	0	0.121	0.010
$\varphi_6(x \oplus k) = S_{AES}(x \oplus k)$	8	8	112	0.586	0.008
$\varphi_7(x \oplus k) = S_{LBlock}(x \oplus k)$	4	4	4	0.342	0.008
$\varphi_8(x \oplus k) = S_{LBlock}(x \oplus k)$	8	8	64	0.235	0.006
$\varphi_9(x \oplus k) = S_{Piccolo}(x \oplus k)$	4	4	4	0.339	0.019
$\varphi_{10}(x \oplus k) = S_{Piccolo}(x \oplus k)$	8	8	64	0.259	0.006
$\varphi_{11}(x \oplus k) = S_{PRINCE}(x \oplus k)$	4	4	4	0.269	0.010
$\varphi_{12}(x \oplus k) = S_{PRINCE}(x \oplus k)$	8	8	64	0.138	0.004
$\varphi_{13}(x \oplus k) = \mathsf{LSB}(L^{-1}_{1,Fantomas}(x \oplus k))$	8	8	0	0.087	0.015
$\varphi_{14}(x \oplus k) = \mathsf{MSB}(L^{-1}_{1,Fantomas}(x \oplus k))$	8	8	0	0.041	0.014
$\varphi_{15}(x \oplus k) = \mathsf{LSB}(L^{-1}_{2,Fantomas}(x \oplus k))$	8	8	0	0.136	0.007
$\varphi_{16}(x \oplus k) = \mathsf{MSB}(L^{-1}_{2,Fantomas}(x \oplus k))$	8	8	0	0.083	0.018

Table 6.2: Leakages of different selection functions (n and m are the input and output size of the selection function in bits, NL is the nonlinearity of the selection function, $\bar{\delta}$ is the mean correlation coefficient difference, and $\mathsf{SE}_{\bar{\delta}}$ is the standard error for a 95% confidence interval).

The modular addition is similar to the XOR operation; the main difference is the carry propagation in the case of modular addition. Although the nonlinearity of the two modular addition selection functions in Table 6.2 is zero, there are components of these functions that reach high nonlinearity because of the carry propagation. For clarity, it should be mentioned that all the components of the XOR selection function have a nonlinearity equal to zero, and that the nonlinearity of an (n,m) function is determined by the component having the lowest nonlinearity. By nonlinearity of a component of an (n,m) function F, we mean the nonlinearity of F computed for a fixed vector $v \in \mathbb{F}_2^{m*}$ as shown in Equation (6.1); see Table 6.4 for details. This exhibits another imperfection of the nonlinearity metric when used to compare various selection functions regarding side-channel leakage. We note that considering the carry bit c from a previous operation when using selection function φ_5 (`adc` instruction) does not improve the correlation coefficient difference compared with φ_4

Selection	Correct key								
function	`0x00`	`0x01`	`0x03`	`0x07`	`0x0F`	`0x1F`	`0x3F`	`0x7F`	`0xFF`
φ_1	-0.225	0.098	0.086	0.057	-0.031	-0.052	-0.001	0.011	0.007
φ_2	0.006	-0.005	-0.002	-0.073	-0.002	0.026	0.015	0.072	-0.202
φ_3	-0.145	-0.160	-0.173	-0.190	-0.167	-0.152	-0.142	-0.125	-0.124
φ_4	0.129	0.134	0.134	0.127	0.150	0.125	0.117	0.096	0.131
φ_5	0.121	0.120	0.147	0.125	0.113	0.109	0.111	0.141	0.110
φ_6	0.597	0.582	0.578	0.577	0.566	0.595	0.603	0.586	0.593
φ_7	0.341	0.343	0.338	0.354	0.337	–	–	–	–
φ_8	0.234	0.223	0.228	0.249	0.230	0.245	0.244	0.233	0.234
φ_9	0.319	0.331	0.361	0.350	0.338	–	–	–	–
φ_{10}	0.252	0.245	0.264	0.256	0.263	0.268	0.264	0.255	0.268
φ_{11}	0.265	0.257	0.273	0.273	0.278	–	–	–	–
φ_{12}	0.139	0.135	0.146	0.143	0.136	0.142	0.129	0.145	0.131
φ_{13}	0.094	0.089	0.079	0.061	0.061	0.080	0.105	0.099	0.120
φ_{14}	0.036	0.027	0.026	0.028	0.018	0.047	0.060	0.062	0.069
φ_{15}	0.144	0.121	0.137	0.127	0.129	0.145	0.134	0.151	0.143
φ_{16}	0.078	0.073	0.072	0.037	0.074	0.093	0.120	0.100	0.100

Table 6.3: Detailed leakages for different selection functions φ_i as defined in Table 6.2.

(`add` instruction). The modular addition selection function successfully recovered the secret key in all our test cases and should thus be preferred over logical operations.

A further group of selection functions is composed of the substitution layers of different lightweight block ciphers. These selection functions clearly leak the most with respect to CPA. In fact, the selection function using the S-box of the AES has the highest leakage among all studied selection functions. For ciphers using 4-bit S-boxes, we considered two different selection functions: one with an 8-bit input and one with a 4-bit input. The 8-bit selection functions based on the substitution layer of LBlock, Piccolo and PRINCE leak two times less than the selection function using the AES S-box. Surprisingly, although our target device has an 8-bit architecture, the 4-bit selection functions φ_7, φ_9, φ_{11} leak more than the 8-bit selection functions of the same substitution layers.

The selection functions based on the L-boxes of Fantomas are analyzed in a fourth group since they are linear operations, which are generally expected to leak less than nonlinear operations. To our surprise, this group (which consists of the last four selection functions listed in Table 6.2) leaks more than the logical operations and is on a similar level with the modular addition. Thus, they can be considered as selection functions when performing CPA attacks.

We remark that in [212], the basic algebraic group operations XOR, addition modulo 2^n, and modular multiplication are studied in the context of multi-bit CPA attacks using simulated power traces. Then, selection functions based on the addition

NL	Number	Proportion (%)
0	1	0.39
16384	26	10.20
24576	100	39.22
28672	112	43.92
30720	16	6.27

(a) $\varphi_4 : \mathbb{F}_2^{16} \mapsto \mathbb{F}_2^8$, $\varphi_4(x,k) = x \boxplus k$

NL	Number	Proportion (%)
0	1	0.39
32768	26	10.20
49152	100	39.22
57344	112	43.92
61440	16	6.27

(b) $\varphi_5 : \mathbb{F}_2^{17} \mapsto \mathbb{F}_2^8$, $\varphi_5(x,k,c) = x \boxplus k \boxplus c$

Table 6.4: Nonlinearity (NL) of the components of the modular addition (selection functions φ_4 and φ_5 from Table 6.2). By nonlinearity of a component of an (n, m) function F, we mean the nonlinearity of F computed for a fixed vector $v \in \mathbb{F}_2^{m*}$ as in Equation (6.1). "Number" denotes how many components have the given nonlinearity NL, "Proportion (%)" is the proportion of the given nonlinearity NL with respect to the nonlinearity of all components of F.

modulo 2^{16} and multiplication modulo $2^{16} + 1$ are applied to an implementation of IDEA running on an 8-bit AVR processor. In the case of the modular addition, the characteristics of the correlation coefficients for practical attacks do not correspond to the simulated ones due to signal superposing.

Through these experiments, we revealed some interesting aspects about the leakage of the studied selection functions with respect to CPA. In contradiction to intuitions based on nonlinearity, we made the following observations:

- The bitwise logical AND and OR operations leak much less than expected and do not always reveal the secret key.
- For block ciphers that use 4-bit S-boxes, a 4-bit selection function is more efficient than an 8-bit selection function.
- The linear lookup tables (i.e. L-boxes) used by Fantomas leak more than expected and can be considered as selection functions for CPA attacks.

The lessons we learned from these experiments helped us a lot to select the appropriate leakage functions to attack the eight lightweight block ciphers we briefly describe in the following section.

6.5 Analyzed Ciphers

We chose the eight lightweight ciphers included in our evaluation according to the following criteria. Firstly, we selected the ciphers from those that achieved good software performance in the Triathlon competition [105]. Besides selecting the ciphers for our CPA study from the ones evaluated in [105], we also used the provided C source codes. This approach has the advantage that all ciphers are implemented according to a common set of guidelines and by the same team of developers, and therefore all implementations had undergone a similar level of optimization. Secondly, we chose our ciphers from the two major structural classes, namely Feistel Networks (FN) and Substitution-Permutation Networks (SPN) with the goal of having many different design approaches with unique features or properties. For example, PRINCE introduced the α-reflection property, which means that a message encrypted under a certain key can only be decrypted with a related key. RC5 introduced data-dependent rotations, while Fantomas is the first instance of the so-called LS-designs.

The main characteristics of the studied ciphers are given in Table 6.5. A short description of each cipher can be found in Section 3.3. Half of the eight ciphers use substitution boxes; Table 6.6 summarizes the most important properties of each S-box. In the following we describe for each cipher the selection function we used to attack it.

Cipher	Block size (bits)	Key size (bits)	Rounds	Structure	Target platform	Attacked operation
AES	128	128	10	SPN	SW, HW	S-box lookup
Fantomas	128	128	12	SPN	SW	L-box lookup
LBlock	64	80	32	Feistel	HW, SW	S-box lookup
Piccolo	64	80	25	Feistel	HW	S-box lookup
PRINCE	64	128	12	SPN	HW	S-box lookup
RC5	64	128	20	Feistel	SW	modular addition
SIMON	64	96	42	Feistel	HW, SW	bitwise AND
SPECK	64	96	26	Feistel	SW, HW	modular subtraction

Table 6.5: Main characteristics of the analyzed lightweight ciphers.

AES. The 8-bit selection function we used in our experiments targets the result of the S-box lookup in the first round of encryption.

Fantomas. Because there are four possible 8-bit inputs for the same MSB or LSB of the output of the 16-bit L-boxes used for the linear layer, we had to attack both the MSB and LSB to recover the key. The selection function targets the inverse linear layer at the first round of decryption.

LBlock. The 4-bit selection function is given by the result of the substitution layer at the first round of encryption.

Piccolo. The 4-bit selection function targets the result of the first substitution layer of the first round function of encryption.

PRINCE. The 4-bit selection function we used targets the substitution layer applied to the initial state XORed with the whitening key k_0 and round key k_1 at the first round of $PRINCE_{core}$. Thus, the attacker recovers the key $k^* = k_0 \oplus k_1$.

Cipher	S-box	NL	TO	ITO	SNR
AES	S	112	7.860	6.916	9.600
LBlock	s_0	4	3.667	2.567	2.946
	s_1	4	3.667	2.567	2.807
	s_2	4	3.667	2.567	2.807
	s_3	4	3.667	2.567	2.946
	s_4	4	3.667	2.567	2.946
	s_5	4	3.667	2.567	2.807
	s_6	4	3.667	2.567	2.946
	s_7	4	3.667	2.567	2.946
Piccolo	S	4	3.667	2.567	3.108
PRINCE	S	4	3.400	2.333	2.129

Table 6.6: Properties of the S-boxes of four analyzed ciphers. The values of TO, ITO, and SNR have a similar behavior as the value of NL for different S-boxes, but they have a different granularity. Thus, the study of NL with respect to CPA holds also for TO, ITO, and SNR, which are variations of NL.

RC5. The selection function for RC5 targets the modular addition of the round key before the first encryption round. To avoid correlations with the reading the round key from memory instead of modular additions, we wrote the selection function in assembly language to measure just the leakage generated by the targeted operation.

Simon. To increase leakage, we attacked the composition of the XOR and AND operations at the end of the first round of decryption because at that time the intermediate value is written to memory.

Speck. The used selection function gives the result of the modular subtraction of the two Feistel branches in the first decryption round. The attacker can take advantage of the memory-write operation of the result of the selection function rotated by 8 bits to the left.

6.6 Experimental Results

We distinguish between two main classes of lightweight ciphers with respect to their implementations' resistance against CPA. The first class contains ciphers that are implemented using lookup tables, while the second class comprises designs that involve only three operations (addition/AND, rotation, and XOR), which generally leak less than table lookups.

First Class. The first class can be further divided into three different categories of ciphers. The *first category* contains the AES, whose 8-bit S-box leaks much more than any other considered selection function. Our attacks required only 59 power traces to recover the four key-bytes with 100% success rate. The *second category* consists of the three lightweight ciphers LBlock, Piccolo, and PRINCE, each using one or more 4-bit S-boxes for the substitution layer. All members of this category leak enough information to make the recovery of the key with a small number of traces possible. On average, a little bit more than 100 traces were enough to get the subkeys of these ciphers with 100% success rate. However, two subkeys of LBlock and two subkeys of Piccolo required a lot more traces since the sensitive results of the selection functions are not written to memory after the targeted operation and hence the attacker correlates the reading of the S-box content (i.e. `ld` instruction) instead of the writing of the S-box output (i.e. `st` instruction). The *third category* is represented by ciphers that use linear lookup tables, e.g. Fantomas. Our attack against the implementation of Fantomas is a multi-target attack [229] because a normal attack failed to recover two bits of each attacked subkey. The multi-target attack enabled us to reveal the four key-bytes using 165 traces with 100% success rate.

Second Class. The second class covers RC5, SIMON, and SPECK, for which we were not able to recover the full secret key due to reduced leakage. If we consider, for example, the attacks to obtain the fourth key byte $k^* = 0\text{x}67$ using $q = 2000$ traces, our experiments for RC5 and SIMON gave a mean guessing entropy $\overline{GE}$ of 1.58 and 3.05, respectively. However, in the case of SPECK, we managed to reveal k^* using 1345 traces with 100% success rate.

The assembly code generated from the C implementations of these ciphers executes four consecutive `st` instructions, which entails signal superposing. We tried to "cancel" this effect by reducing the frequency of the processor, but we had no success. Although the insertion of `nop` instructions between the stores improved the results, we decided to not use these modified implementations in our experiments because they give the attacker an unreasonable advantage and affect therefore the fair comparison with the ciphers from the first class.

Given the small size of the state of these designs and the rather simple operations they carry out, we investigated the possibility of keeping the whole state in registers during the entire encryption process. The 64-bit block version of both SIMON and SPECK can be implemented in assembly without having to execute a single `st` instruction between the start and the end of the encryption operation. This approach significantly reduces the amount of leakage available to the attacker. But this leakage reduction optimization can not be applied to 128-bit block implementations of RC5, SIMON, and SPECK due to the restricted register space available on an 8-bit microcontroller. For RC5, we also tried the butterfly attack proposed in [394] on the modular addition, but the results were worse than when using the classical CPA attack.

We performed the described attacks also with a "low-cost" setup consisting of an Arduino Uno board and an Analog Discovery oscilloscope with a built-in differential probe. The Arduino board gets its supply voltage through an USB connection, which is also used for the communication with the computer that controls the trace acquisition process. We did not employ any noise reduction techniques. The experiments with the low-cost setup produced similar results for the ciphers in the first class, except for Fantomas, but required more traces due to the increased noise levels. For example, the AES key could be recovered with 80% success rate using 36 power traces with the first setup, but 58 traces were necessary with the second (i.e. low-cost) setup. Similarly, to retrieve the PRINCE key with the same success rate, the first setup needed 65 traces, while the second setup required 85 traces. For the ciphers from the second class, the low-cost setup yielded much worse results. When using 5000 traces, the mean guessing entropy for the attack against RC5 increased from 3.68 (low noise) to 22.29 (high noise). Similarly, for Simon we got $\overline{GE} = 9.97$ in the noise-reduced setting and $\overline{GE} = 16.44$ with the cheap equipment.

All our experiments were conducted on unprotected implementations of the ciphers. However, many security-critical applications require countermeasures against SCA attacks, e.g. masking. In this context, it is known that linear and Boolean operations, such as those performed by Fantomas, RC5, Simon, and Speck, can be masked with relatively low overheads in terms of execution time and code size. On the other hand, masking a nonlinear S-box like that of AES generally entails a significant performance and code-size penalty. Somewhere in the middle between these two extremes are LBlock, Piccolo, and PRINCE.

6.7 Summary

Following a practical approach, we investigated the leakage of various selection functions widely used in existing lightweight ciphers for an 8-bit processor. We analyzed how these results relate to the intuition about side-channel leakages based on the nonlinearity of the selection function. Thereby, we identified three imperfections of leakage evaluation based on nonlinearity, namely for AND and OR bitwise operations, for 4-bit S-boxes, and for linear lookup tables.

Using the knowledge gained from the evaluation of selection functions, we attacked unprotected software implementations of eight well-known lightweight ciphers, namely AES, Fantomas, LBlock, Piccolo, PRINCE, RC5, Simon, and Speck. We grouped the results of our experiments into two classes according to the observed resistance against CPA attacks. The unprotected implementation of AES was broken with the smallest number of power traces, followed by the implementations of lightweight ciphers using 4-bit S-boxes, and thereafter the implementation of Fantomas, whose L-boxes required slightly more traces than the 4-bit S-boxes. On the other hand, the implementations of RC5, Simon, and Speck leaked less as we could not recover the full key for any of them. We also demonstrated that different implementation options can increase the resilience of lightweight block ciphers against power analysis attacks.

The software implementations of the three designs that do not use lookup tables (i.e. RC5, Simon and Speck) are characterized by a certain level of "intrinsic" resilience against CPA. They can also be efficiently masked with relatively small impact on execution time and code size. These features make constructions based solely on addition/AND, rotation, and XOR excellent candidates for the implementation of lightweight block ciphers for the IoT.

Chapter 7

Correlation Power Analysis Attacks on Communication Protocols

Contents

7.1 Introduction

Side-channel attacks use observations made during the execution of an implementation of a cryptographic algorithm to recover secret information. From the multitude of side-channel attacks, Correlation Power Analysis (CPA) [68] stands out as a very efficient and reliable technique. Its success is augmented by the minimally invasive methods employed for the acquisition of the side-channel information. Some of the most frequently used sources of side-channel leakage are the power consumption or the electromagnetic (EM) emissions of a device under attack.

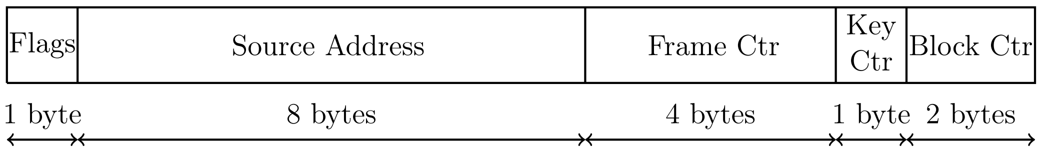


Figure 7.1: The first input block for the AES-CTR and AES-CCM modes used in IEEE 802.15.4 [170].

Nowadays, AES [250] is the most popular symmetric cryptographic algorithm in use. It is widely deployed to secure data in transit or at rest. Various network protocols rely on the AES in different modes of operation to provide security services such as confidentiality and authenticity. The usage spectrum of the AES stretches from powerful servers and personal computers to resource-constrained devices such as wireless sensor nodes. While the security of the algorithm and its implementations have been placed under scrutiny since it was standardized by NIST, with a few notable exceptions, most of the previous work focused on the AES itself and less on the usage of the AES in complex systems.

By far, most of the experimental results reported in the side-channel literature are for implementations of the AES. They usually assume the attacker has full control of the AES input. This is not the case in a real world communication protocol, when often a major part of the input is fixed and only few bytes are variable. Moreover, sometimes the attacker cannot control these variable bytes and she has to passively observe executions of the targeted algorithm without being able to trigger encryptions of her own free will. With the notable exceptions of [175, 259], the security of communication scenarios based on the AES against side-channel attacks has not been thoroughly analyzed so far. Thus, in this chapter we analyze how much control of the AES input does an attacker need to recover the secret key of the cipher by performing a side-channel attack against a communication protocol.

Numerous standards for communication in the Internet of Things (IoT) such as IEEE 802.15.4 [170] and LoRaWAN [220] use the AES to encrypt and authenticate the Medium Access Control (MAC) layer frames. The 802.15.4 standard uses a variant of the AES-CCM [380, 115], while LoRaWAN uses AES-CMAC [331]. The same CCM mode is used with the AES to encrypt the IPsec Encapsulating Security Payload (ESP) [168]. According to [305] the security architecture of IEEE 802.15.4 relies on four categories of security suites: none, AES-CTR, AES-CBC-MAC, and AES-CCM. A typical input for the AES-CTR and AES-CCM modes used in the IEEE 802.15.4 protocol is shown in Figure 7.1. In this particular example, an attacker can manipulate up to 12 bytes of the input (`Source Address` and `Frame Counter`), while the other input bytes (`Flags`, `Key Counter` and `Block Counter`) are fixed.

The attack on IEEE 802.15.4 wireless sensor nodes described in [259] assumes the control of only four input bytes (`Frame Counter`), while the remaining input bytes are constant. Thus the following question arises:

> *How many input bytes should an attacker change in the injected messages in order to fully recover the master key without triggering any network protection mechanism?*

While numerous network protocols use the AES to secure the communication between end nodes, major cryptographic libraries such as OpenSSL [261] and ARM mbed TLS [18] do not have a side-channel protected implementation of the AES for devices that do not support the AES-NI [163] instruction set as is the case with most IoT devices. Therefore, an elaborate analysis of the security of the unprotected implementations of the AES used in communication protocols is necessary. Only such a careful analysis can assess the impact of side-channel attacks on the security of real world systems using unprotected implementations of the AES.

In this chapter, we chose to focus on CPA attacks thanks to their efficiency and reliability. We opted for a non-invasive measurement setup and hence we selected the EM emissions of the target processor as source of side-channel leakage. The target is an ARM Cortex-M3 processor mounted on a STM32 Nucleo [340] board from STMicroelectronics. These processors are widely used for low-power applications and meet the requirements for use in the IoT.

The IoT will be a security nightmare if the whole information ecosystem is not designed with security in mind. While many communication protocols for the IoT are in formative stages, the threat model of the IoT is less understood despite it is widely accepted that its attack surface is large. Although we focus on a particular side-channel attack (i.e. power/EM), other side-channel attacks such as timing, fault, cache or data remanence attacks might pose a similar or even a higher threat for the security of the IoT ecosystem. Attacks that do not exploit side-channel information, such as those used to compromise Internet-connected computers, should not be neglected since they have certain advantages over side-channel attacks. Thus, our work adds another piece to the security puzzle of the IoT by showing the need for side-channel countermeasures to prevent a somehow overlooked threat.

7.1.1 Research Contributions

This chapter presents a thorough analysis of the scenarios in which an attacker can mount a DPA attack against software implementations of the AES used to secure various communication protocols. Firstly, we present an algorithm for symbolic processing of a given input state of the AES. The algorithm outputs the number of rounds and the bytes that must be attacked to recover the secret key. Then, using this algorithm we perform a classification of all possible inputs depending on the number of rounds that must be attacked in order to recover the master key. The result is a set of 25 independent evaluation cases. Secondly, we describe an optimal algorithm that uses the above-mentioned symbolic representation to recover the master key of the AES using CPA attacks. The algorithm explores all possible combinations of input key bytes and discards the invalid key candidates, thus yielding only the correct master key if enough power traces with a good signal-to-noise ratio are provided. Afterwards, we evaluate the results of the attack algorithm in each of the 25 evaluation cases identified in the classification step using traces from an ARM Cortex-M3 processor.

Our results show that popular implementations of the AES found in well-known and widely used cryptographic libraries can be broken using CPA attacks. The only requirement is that a part of the AES input is known and variable, while the rest is constant, which is a common scenario in communication protocols. Knowledge of the AES implementation strategy improves the attack results, but it is not crucial.

7.2 Preliminaries

7.2.1 Description of the AES

We give a brief description of the AES [250] to recall relevant aspects of the algorithm and to introduce the notation used in this chapter. For more details on the AES algorithm, we refer the reader to the official specifications.

The AES is a version of the Rijndael cipher [98] with 128-bit blocks and three different key lengths: 128, 192, and 256 bits. The round function is applied to the 4×4 byte state matrix 10, 12, or 14 times depending on the key length. It comprises four transformations: `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`. The final round function does not include the `MixColumns` transformation.

Let $s_{i,j}$ be the state byte located at row i and column j $(0 \leq i, j \leq 3)$, k_l the corresponding round key byte $(l = 16 \cdot r + i + 4 \cdot j)$ and r the round number. After application of the `AddRoundKey` transformation, each byte of the state becomes $s'_{i,j} = s_{i,j} \oplus k_l$, where the "$\oplus$" symbol denotes bitwise exclusive or of two 8-bit values. The nonlinear `SubBytes` operation transforms each byte of the state using an 8-bit S-box S as follows: $s'_{i,j} = S[s_{i,j}]$. The `ShiftRows` transformation performs a rotation of row i by i bytes to the left. In the `MixColumns` transformation, a polynomial multiplication over $GF(2^8)$ is applied to each column of the state matrix. The symbol "$\bullet$" is used for multiplication of two polynomials in $GF(2^8)$, while $\{01\}$, $\{02\}$, and $\{03\}$ are 8-bit vectors representing elements from $GF(2^8)$.

The key schedule expands the master key into the 16-byte round keys. The round constant array `Rcon` contains the powers of $\{02\}$ in $GF(2^8)$ as described in the specifications. The structure of the AES encryption is given in Algorithm 2. Algorithm 3 describes the AES key schedule at the byte level when using a 16-byte master key.

7.2.2 Attacking Temporary Key Bytes

To attack the AES in counter mode, Jaffe introduced a technique that propagates a DPA attack to later rounds. It can be used when just few bytes of the AES input are known and variable, while the others are fixed (constant) and unknown [175]. Next we briefly describe how the unknown fixed bytes can be incorporated into a

Algorithm 2 AES encryption

```
Input: $state$, $round\_keys$
 1: AddRoundKey($state$, $round\_keys[0]$)
 2: for $r = 1$ to $R - 1$ do                    ▷ $R$ is the total number of rounds
 3:     SubBytes($state$)
 4:     ShiftRows($state$)
 5:     MixColumns($state$)
 6:     AddRoundKey($state$, $round\_keys[r]$)
 7: end for
 8: SubBytes($state$)
 9: ShiftRows($state$)
10: AddRoundKey($state$, $round\_keys[R]$)
11: return $state$
```

Algorithm 3 AES key schedule for a 16-byte master key

```
Input: $key$
 1: $rk[0] = key$
 2: for $i = 1$ to $R$ do                         ▷ $R$ is the total number of rounds
 3:     $rk[i][0] = rk[i-1][0] \oplus$ SubBytes($rk[i-1][13]$) $\oplus$ Rcon$[i-1]$
 4:     $rk[i][1] = rk[i-1][1] \oplus$ SubBytes($rk[i-1][14]$)
 5:     $rk[i][2] = rk[i-1][2] \oplus$ SubBytes($rk[i-1][15]$)
 6:     $rk[i][3] = rk[i-1][3] \oplus$ SubBytes($rk[i-1][12]$)
 7:     for $j = 4$ to $15$ do
 8:         $rk[i][j] = rk[i-1][j] \oplus rk[i][j-4]$
 9:     end for
10: end for
11: return $rk$
```

round key byte to recover a temporary key byte. Then, using these temporary key bytes the attack can be carried to later rounds until enough round key bytes are recovered to reverse the key schedule.

Using a CPA attack an adversary can recover only those key bytes that are XORed with variable and known state bytes in the `AddRoundKey` transformation. The gist of Jaffe's technique is that an attacker can still recover a temporary key byte when an input byte of the `AddRoundKey` transformation is the result of the `MixColumns` transformation applied to at least one known and variable input byte while the other input bytes are unknown and constant.

To better illustrate how this technique works, let us consider the first state byte $s'_{0,0}$ after performing the first round function:

$$s'_{0,0} = (\{02\} \bullet s_{0,0}) \oplus (\{03\} \bullet s_{1,1}) \oplus (\{01\} \bullet s_{2,2}) \oplus (\{01\} \bullet s_{3,3}) \oplus k_{16}$$

Suppose now that the input bytes $s_{0,0}$ and $s_{1,1}$ are known and variable (key bytes k_0 and k_5 were successfully recovered using a side-channel attack on the `SubBytes` transformation of the first round), while the other input bytes ($s_{2,2}$ and $s_{3,3}$) are unknown, but fixed. Thus $s'_{0,0}$ can be written as $(\{02\} \bullet s_{0,0}) \oplus (\{03\} \bullet s_{1,1}) \oplus k'_{16}$, where the constant part is included in the temporary key k'_{16} that will be recovered by attacking the `SubBytes` transformation of the second round; $k'_{16} = (\{01\} \bullet s_{2,2}) \oplus (\{01\} \bullet s_{3,3}) \oplus k_{16}$. The temporary key k'_{16} enables the computation of four state bytes in the following round. In this way, the attack is carried to the next rounds until all state bytes are known; consequently, the real key bytes can be recovered.

The technique works similarly when three input bytes are known and variable. Though, when only one input byte is known and variable, the attacker will recover the same two equally likely key candidates for two bytes of the same column of the cipher state. For example, when only $s_{3,3}$ is known and variable while the other input bytes are unknown and fixed, then $s'_{0,0} = (\{01\} \bullet s_{3,3}) \oplus k'_{16}$ and $s'_{1,0} = (\{01\} \bullet s_{3,3}) \oplus k'_{17}$. Thus attacking either of the two, an attacker will get two equally likely key bytes (k'_{16} and k'_{17}). If the state bytes are not processed in order by the `SubBytes` transformation, the attacker will not know which key byte corresponds to $s'_{0,0}$ and which key byte corresponds to $s'_{1,0}$.

7.2.3 Software Implementations of the AES

There are various ways to implement the AES in software depending on the execution time, code size and RAM consumption requirements. Other factors that influence the implementation strategy are the cipher mode of operation and the number of plaintext blocks to be encrypted. Schwabe and Stoffelen [314] identified four different strategies to implement the AES in software: traditional, T-tables, vector permute, and bit slicing. In this chapter, we consider the following two implementation approaches for the AES that are relevant for a secure communication protocol:

- The *straightforward implementation* (**S-box** strategy) performs the four round transformations as described above. The substitution layer is implemented using a 256-byte lookup table based on S-box S. This implementation approach is suitable for 8-bit architectures.

- The *table based implementation* (**T-table** strategy) uses four lookup tables (T_0, T_1, T_2, and T_3) of 1024 bytes each to perform the `SubBytes`, `ShiftRows`, and `MixColumns` operations at the cost of 16 table lookups, 16 masks and 16 XORs per round, except for the final round. A low memory alternative uses just one T-table, but performs 12 additional rotations per round. This strategy was initially described by the designers of Rijndael [98]. It leads to very fast implementations on 32-bit platforms.

We did not analyze bit-sliced or vector permute implementations because such implementations are uncommon in cryptographic libraries due to the following limitations. The bit-sliced implementations process at least two blocks in parallel and thus they can be applied only to non-feedback modes of operation. The vector permute implementations require support of vector permute instructions, but most of the resource-constrained microcontrollers for the IoT do not support such instructions.

An analysis of the existing AES implementations used by different open-source cryptographic libraries is given in Table 7.1. The default implementations of the

Library	Language	Version	Release	AES-NI	T-table
Botan [288]	C++	2.1.0	Apr 2017	✓	✓
cryptlib [90]	C	3.4.3	Feb 2017	✓	✓
Crypto++ [91]	C++	5.6.5	Oct 2016	✓	✓
Libgcrypt [146]	C	1.7.6	Jan 2017	✓	✓
libtomcrypt [143]	C	1.17	Apr 2017	✗	✓
libsodium [215]	C	1.0.12	Mar 2017	✓	✗
mbed TLS [18]	C	2.4.2	Mar 2017	✓	✓
Nettle [251]	C	3.3	Oct 2016	✓	✓
OpenSSL [261]	C	1.1.0e	Feb 2017	✓	✓
wolfCrypt [383]	C	3.10.2	Feb 2017	✓	✓

Table 7.1: A summary of the existing AES implementations used by open-source cryptographic libraries written in C/C++. All the T-table implementations are vulnerable to the attack described in this chapter.

AES for platforms that do not support the AES-NI [163] instructions in popular cryptographic libraries such as OpenSSL [261, 262] or mbed TLS [18, 144] use the T-table approach. Except for libsodium [215], all other cryptographic libraries analyzed have an implementation of the AES based on the T-table strategy. Moreover, these implementations are not protected against side-channel attacks such as DPA or cache attacks. It is well known that unprotected implementations of cryptographic algorithms are an easy target for DPA attacks. Recently, researchers from Rambus Cryptography Research Division have shown that even an unprotected software implementation based on AES-NI instructions can be attacked with DPA [302]. The T-table implementations of the AES are vulnerable to various cache attacks as shown in [264, 216]. Although the unprotected T-table implementations are vulnerable to side-channel attacks, nine out of the ten libraries considered in Table 7.1 have such an implementation of the AES.

7.2.4 Measurement Setup

For all experimental results reported in this chapter we used a STM32 Nucleo [340] board from STMicroelectronics. It has a 32-bit ARM Cortex-M3 processor clocked at 8 MHz, 512 KB of flash, 80 KB of RAM and 16 KB of EEPROM. The measurement of the electromagnetic emissions was performed from a spot above the chip using a Langer RF-K 7-4 H-field probe as shown in Figure 7.2. The target board executed software implementations of the AES. The signal was amplified by 30dB and then sampled at 500 MS/s using a Teledine LeCroy WaveRunner 8254M-MS oscilloscope. We did not use any noise reduction technique. The board was powered through an USB cable, which was also used to control the device under test (DUT).

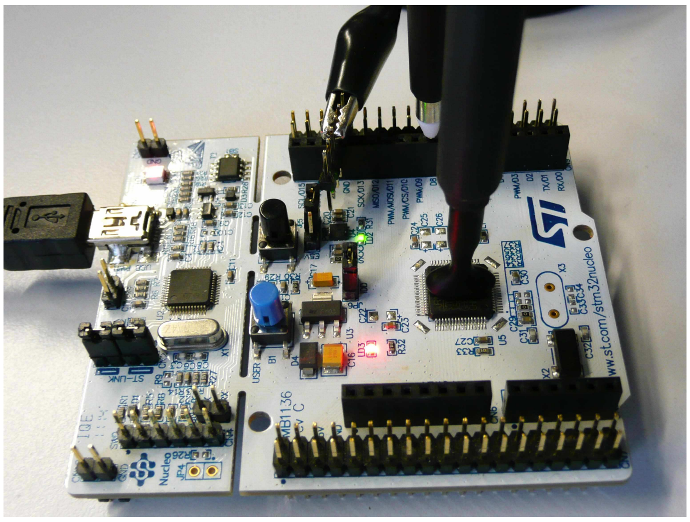


Figure 7.2: The device under test (DUT).

7.3 Quantifying the Leakage

We introduced the correlation coefficient difference metric in Section 6.4 to analyze the leakage of different selection functions in the context of CPA. The correlation coefficient difference δ gives the difference between the correlation coefficient of the correct key and the correlation coefficient of the most likely key guess, where the most likely key is different from the correct key.

We use the correlation coefficient difference to quantify the leakages of two selection functions: φ_1 based on the AES S-box and φ_2 based on the AES T-table. The two selection functions are defined below:

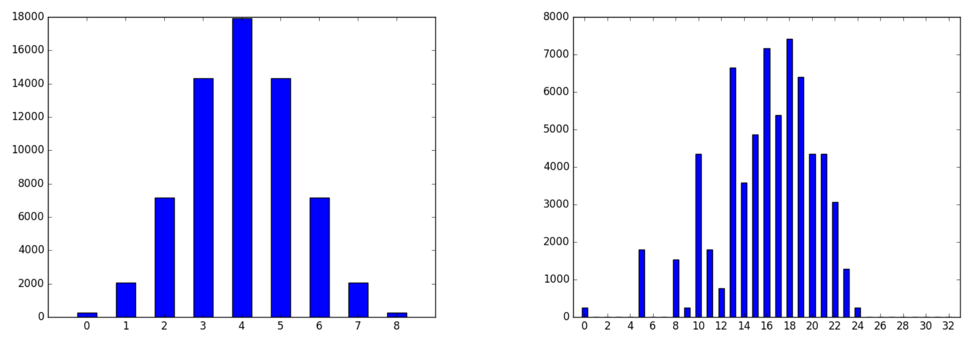
$$\varphi_1 : \mathbb{F}_2^8 \mapsto \mathbb{F}_2^8, \quad \varphi_1(x \oplus k) = S(x \oplus k)$$

$$\varphi_2 : \mathbb{F}_2^8 \mapsto \mathbb{F}_2^{32}, \quad \varphi_2(x \oplus k) = T(x \oplus k)$$

	Correct key									$\bar{\delta}$	$\mathsf{SE}_{\bar{\delta}}$
	0x00	0x01	0x03	0x07	0x0F	0x1F	0x3F	0x7F	0xFF		
φ_1	0.146	0.126	0.108	0.156	0.126	0.960	0.153	0.140	0.084	0.126	0.020
φ_2	0.104	0.072	0.143	0.074	0.070	0.126	0.078	0.044	0.028	0.082	0.028

Table 7.2: Correlation coefficient difference δ between the correlation of the correct key and the correlation of the most likely key [50], for different Hamming weights of the correct key; $\bar{\delta}$ and $\mathsf{SE}_{\bar{\delta}}$ are the mean and the standard error for a 95% confidence interval, respectively. The leakages are acquired from an ARM Cortex-M3 processor.

When using simulated leakages, the values of the correlation coefficient difference are 0.813 and 0.7 for φ_1 and φ_2, respectively. These values are the same regardless



(a) S-box. (b) T-table.

Figure 7.3: Distribution of the Hamming weight of the output of the AES (a) S-box and (b) T-table for all possible input combinations.

of the correct key used. In the simulated environment, the leakages of the two selection functions are very high and the difference between them is about 14% of the first one. On the other hand, the mean correlation coefficient difference $\bar{\delta}$ for different values of the correct key using leakages acquired from an ARM Cortex-M3 processor is given in Table 7.2. The measurements were performed at a sampling rate of 500 MS/s using assembly implementations of the analyzed selection functions. Increasing the sampling rate to 1 GS/s does not significantly improve the results. The mean correlation coefficient difference $\bar{\delta}$ is positive for both selection functions, which means they leak enough information about the secret key such that an attacker can recover the key byte using only one key guess. In practice, the selection function based on the AES S-box leaks about 50% more than the selection function based on the AES T-table. This can be explained by analyzing the distribution of the Hamming weight of the two selection functions for all possible input combinations (See Figure 7.3).

The reader can easily observe in Figure 7.3a that the distribution of values in the case of the AES S-box follows a binomial distribution. On the other hand, the distribution of values in the case of the AES T-table shown Figure 7.3b does not resemble a binomial distribution. Moreover, there are 14 out of 32 possible output values that never occur (i.e. 1, 2, 3, 4, 6, 7, 25, 26, 27, 28, 29, 30, 31, and 32). Each Hamming weight value can be seen as a predicted power consumption that is used in a CPA attack. Therefore, the 8-bit output of an S-box can generate any of the 9 possible power levels, while the 32-bit output of a T-table generates only 54.54% of the 33 possible power levels. Consequently, it is easier for a CPA attacker to determine which 8-bit input corresponds to the 8-bit output of an S-box than to the 32-bit output of a T-table. For this reason, the leakage of φ_1 is greater than the leakage of φ_2 as quantified using the correlation coefficient difference. This means that a CPA attack against an implementation based on the T-table strategy requires more effort (i.e. power traces) compared to a CPA attack against an implementation based on the S-box strategy.

7.4 Generating the Evaluation Cases

In this section we describe the algorithm for symbolic processing of a given initial state to determine the number of rounds required to recover the master key of the AES. We used this algorithm to explore all possible attack cases and to choose the relevant evaluation cases for our scenario. The algorithm relies on the following symbolic representation of a byte located at row i and column j of the AES state at the start of round r:

$$s_{i,j}^r = \begin{cases} 0, & \text{the corresponding key byte can not be recovered} \\ 1, & \text{the corresponding key byte can be recovered} \\ -n, & n \text{ temporary key bytes can be recovered} \end{cases}$$

Thus, the byte $s_{i,j}^r$ is variable if its symbolic representation is different from 0 and fixed (constant) when its symbolic representation is 0. Due to the `MixColumns` transformation, each column of the state at round $r+1$ can be expressed as a function of four bytes of the state at round r. At the start of round $r+1$ each byte of the state is updated using the following rules:

- if the number of variable input bytes is 0, then the symbolic representation of the output byte is set to 0;
- if the number of variable input bytes is 1, then the symbolic representation of the output byte is updated as follows:
  - if the variable input byte is multiplied by $\{01\}$ in the `MixColumns` transformation, then the symbolic representation of the output byte is set to -2^{p+1}, where p is the number of independent input pairs. A new pair is added to the output byte;
  - else, the symbolic representation of the output byte is set to -2^p;
- if the number of variable input bytes is 2 or 3, then the symbolic representation of the output byte is set to -1;
- if the number of variable input bytes is 4, then the symbolic representation of the output byte is set to 1.

Besides updating the symbolic representation of the state, the algorithm keeps a list of key pairs for each byte of the state and carries this list into the next round. The algorithm stops when the symbolic representation of all bytes in a round is 1. It outputs the symbolic representation of the state and the associated key pairs. These can be used to compute the number of rounds required to recover the master key and the number of possible master keys. The pseudocode for the algorithm is given in Algorithm 4. The algorithm returns the processed state and the associated set of pairs. Using this output, an attacker knows what key bytes have to be attacked in each round of the AES, the number of rounds to be attacked, and the maximum number of possible master keys.

Algorithm 4 Symbolic processing of an initial state

```
Input: $state$                    ▷ Initial state: 0 – fixed byte, 1 – variable byte
 1: function ProceessColumn($r, pairs, i_0, i_1, i_2, i_3, o_0, o_1, o_2, o_3$)
 2:     Compute the number of variable inputs for $i_0, i_1, i_2, i_3$: $var\_in$
 3:     Update $pairs$
 4:     if $var\_in == 0$ then                          ▷ No key bytes recovered
 5:         $state[r][o_0] = state[r][o_1] = state[r][o_2] = state[r][o_3] = 0$
 6:     else if $var\_in == 1$ then          ▷ 4 temporary key bytes recovered; new pair
 7:         Compute the number of independent pairs: $p$
 8:         $pairs = pairs \cup \{new\_pair\}$
 9:         $state[r][o_0] = state[r][o_1] = state[r][o_2] = state[r][o_3] = -2^p$
10:         if $state[r][o_i] == state[r][o_j] == (\{01\} \bullet state[r-1][i_t]) \oplus k'$ then
11:             $state[r][o_i] = state[r][o_j] = -2^{p+1}$
12:         end if
13:     else if $var\_in \in \{2, 3\}$ then               ▷ 4 temporary key bytes recovered
14:         $state[r][o_0] = state[r][o_1] = state[r][o_2] = state[r][o_3] = -1$
15:     else if $var\_in == 4$ then                      ▷ All 4 key bytes recovered
16:         $state[r][o_0] = state[r][o_1] = state[r][o_2] = state[r][o_3] = 1$
17:     end if
18: end function
19: function ProcessRound($r, pairs$)
20:     ProcessColumn($r, pairs, 0, 5, 10, 15, 0, 1, 2, 3$)
21:     ProcessColumn($r, pairs, 4, 9, 14, 3, 4, 5, 6, 7$)
22:     ProcessColumn($r, pairs, 8, 13, 2, 7, 8, 9, 10, 11$)
23:     ProcessColumn($r, pairs, 12, 1, 6, 11, 12, 13, 14, 15$)
24: end function
25: function RoundKeyRecovered($r$)
26:     for $i = 0$ to 15 do
27:         if $0 \geq state[r][i]$ then return False
28:         end if
29:     end for
30:     return True
31: end function
32: function Process($state$)
33:     $pairs = \emptyset$
34:     for $r = 1$ to $R$ do                    ▷ $R$ is the total number of rounds
35:         if RoundKeyRecovered($r - 1$) then return $state, pairs$
36:         end if
37:         ProcessRound($r, pairs$)
38:     end for
39: end function
40: return Process($state$)
```

	Round 1				Round 2				Round 3				Round 4			
State	1	0	0	0	-1	0	0	0	-1	-2	-4	-2	1	1	1	1
	0	0	0	0	-2	0	0	0	-2	-2	-2	-2	1	1	1	1
	0	0	0	0	-2	0	0	0	-2	-1	-2	-4	1	1	1	1
	0	0	0	0	-1	0	0	0	-1	-1	-4	-4	1	1	1	1
Pairs	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	S_3	S_4	S_1	S_6	S_6	S_7	S_7
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	S_1	$\emptyset$	$\emptyset$	$\emptyset$	S_2	S_3	S_1	S_1	S_6	S_6	S_7	S_7
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	S_1	$\emptyset$	$\emptyset$	$\emptyset$	S_2	$\emptyset$	S_1	S_5	S_6	S_6	S_7	S_7
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	S_4	S_5	S_6	S_6	S_7	S_7

$$S_1 = \{1\}; \quad S_2 = \{2\}; \quad S_3 = \{3\}; \quad S_4 = S_1 \cup \{4\} = \{1,4\};$$
$$S_5 = S_1 \cup \{5\} = \{1,5\}; \quad S_6 = S_3 \cup S_5 = \{1,3,5\}; \quad S_7 = S_2 \cup S_4 = \{1,2,4\}$$

Figure 7.4: Symbolic processing of an initial state.

Figure 7.4 gives a graphical representation of how the algorithm works when only the first byte of the initial state is variable and known, while the other bytes are fixed and unknown. By attacking the result of the `SubByte` transformation applied to the first byte of the state in the first round, the key byte k_0 is recovered. This recovered key byte allows a carry of the attack to the second round where four key bytes $(k'_{16}, k'_{17}, k'_{18}, k'_{19})$ can be recovered by attacking the result of the `SubBytes` transformation. Because the attacker cannot distinguish between k'_{17} and k'_{18}, a new pair $S_1 = \{1\}$ is added to the corresponding state bytes. Then, the attacker targets the third round, where she can recover temporary key bytes for all state bytes. The pair S_1 from previous round affects all bytes of the third and fourth column of the state and thus the corresponding pairs are updated accordingly. In addition, new pairs are added when the attacker can not distinguish between key candidates as shown in Figure 7.4. In the fourth round, the attacker is able to recover all round key bytes. Then, having all the round key bytes of the fourth round, she can reverse the AES key schedule to get the master key.

The attacker has to build 2^p possible round keys, where p is the number of independent pairs associated with the state bytes of the last attacked round. For the example in Figure 7.4, the number of possible keys is 2^5 because $card(S) = card(S_6 \cup S_7) = card(\{1,2,3,4,5\}) = 5$. Thus, in addition to the number of rounds to attack, the algorithm for symbolic processing of an initial state gives the number of possible master keys to be recovered by an attacker. Though, the attacker does not have to check all 2^p candidates to see which one is the correct one since she can discard the wrong candidates based on the difference between the correlation coefficients of the first two key candidates as we will show in Section 7.5.

Bytes	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
min(Rnds)	4	4	4	3	3	3	3	3	3	3	3	3	3	3	3	1
Prop. (%)	100	100	100	14.1	35.2	55.9	72.7	84.7	92.3	96.7	98.9	99.8	100	100	100	100
max(Rnds)	4	4	4	4	4	4	4	4	4	4	4	4	3	3	3	1
Prop. (%)	100	100	100	85.9	64.8	44.1	27.3	15.3	7.7	3.3	1.1	0.2	100	100	100	100
Trade-off	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗

Table 7.3: Possible attack outcomes for different number of bytes ("Bytes") controlled by attacker. "Rnds" is the number of rounds that have to be attacked in order to recover the master key. "Prop. (%)" is the proportion of a given evaluation case with respect to all possible input configurations for a fixed number of bytes controlled by attacker.

Using the algorithm for symbolic processing of an initial state we evaluated all possible input combinations. More precisely, we considered all configurations of the initial state when the attacker controls i bytes of the input for $i \in [1, 16]$. When the attacker controls i bytes, there are $\binom{16}{i}$ possible input configurations. This results in $2^{16} - 1$ possible configurations of the initial state in total. Then, we divided these inputs into equivalence classes (evaluation cases) depending on the number of rounds that must be attacked in order to recover the master key. The results are summarized in Table 7.3. When the attacker controls between four and eleven bytes of the input, a trade-off between the input configuration and the number of rounds to be attacked is possible. When this is the case, the proportion of possible input configurations shows which evaluation case is more likely to appear if the initial state is chosen at random. Thus, when the attacker controls only four or five bytes of the input, it is crucial to carefully choose an input configuration from the limited set of possible input configurations that minimize the number of rounds to be attacked.

We give an example of a possible initial state for each of the 25 distinct evaluation cases identified after processing all possible input combinations in Table 7.4. Any possible input configuration for the AES encryption falls into one of these evaluation cases depending on the number of bytes controlled by attacker and the number of rounds that must be attacked in order to recover the master key.

7.5 The Attack

The attack we present in this section uses the symbolic representation of the AES state (described in Section 7.4) in conjunction with CPA attacks to recover individual bytes of the AES round keys. After executing Algorithm 5, the attacker has all round key bytes of round R. Thus, she is able to recover the master key of the cipher by reversing the key schedule.

The algorithm follows the symbolic representation of the state to infer which key bytes must be attacked and how many key candidates it should yield for each

Algorithm 5 The attack algorithm

```
Input: $state$    ▷ Initial state: 0 – fixed byte, 1 – variable byte
Input: $\lambda = (plaintexts, traces)$    ▷ Recorded in the acquisition phase
 1: $state, pairs$ = PROCESS($state$)    ▷ Symbolic processing (Algorithm 4)
 2: $known\_pairs = \emptyset, \quad mapped\_pairs = \emptyset$
 3: $keys[2^p] = \emptyset, \quad valid\_keys[2^p] = True$    ▷ $p$ is the number of independent pairs
 4: for $r = 1$ to $R$ do    ▷ $R$ is the number of rounds to be attacked
 5:     for $i = 0$ to 15 do
 6:         if $state[r][i] \neq 0$ then
 7:             if $pairs[r][i] == \emptyset$ then    ▷ No pair
 8:                 $keys[0, \cdots, 2^p - 1][r][i] = \mathsf{CPA}(\lambda, keys[0], r, i)$
 9:             else if $pairs[r][i] \subseteq known\_pairs$ then    ▷ Known pair(s)
10:                 if $i \notin mapped\_pairs[pairs[r][i]]$ then
11:                     $mask = 0, \quad temp\_keys = \emptyset, \quad \alpha_{max} = -1$
12:                     for $pair \in pairs[r][i]$ do
13:                         $mask = mask \vee 2^{pair-1}$
14:                     end for
15:                     for $j \in [0, 2^p - 1]$ do
16:                         if $valid\_keys[j]$ and $temp\_keys[j \wedge mask] == \emptyset$ then
17:                             $temp\_keys[j \wedge mask], \alpha = \mathsf{CPA}(\lambda, keys[j], r, i)$
18:                             if $\alpha > \alpha_{max}$ then
19:                                 $\alpha_{max} = \alpha$
20:                             end if
21:                         end if
22:                         $valid\_keys[j][r][i] = temp\_keys[j \wedge mask]$
23:                     end for
24:                     for $j \in [0, 2^p - 1]$ do
25:                         if $abs(state[r][i]) == 1$ and $\alpha + \beta < \alpha_{max}$ then
26:                             $valid\_keys[j] = False$
27:                         end if
28:                     end for
29:                 end if
30:             else    ▷ New pair
31:                 $mask = 2^{pairs[r][i] - new\_pair}, \quad k_1 = k_2 = \emptyset$
32:                 for $j \in [0, 2^p - 1]$ do
33:                     if $k_1[j \wedge mask] == \emptyset$ then
34:                         $k_1[j \wedge mask], k_2[j \wedge mask] = \mathsf{CPA}(\lambda, keys[j], r, i)$
35:                     end if
36:                     if $j \wedge 2^{new\_pair-1}$ then
37:                         $keys[j][r][i] = k_1[j \wedge mask], \quad keys[j][r][i'] = k_2[j \wedge mask]$
38:                     else
39:                         $keys[j][r][i'] = k_2[j \wedge mask], \quad keys[j][r][i'] = k_1[j \wedge mask]$
40:                     end if
41:                 end for
42:                 $known\_pairs = known\_pairs \cup new\_pair$
43:                 Add $(i, i')$ to $mapped\_pairs[new\_pair]$
44:             end if
45:         end if
46:     end for
47: end for
48: return $keys[i]$, where $valid\_keys[i] == True$ for $i \in [0, 2^p - 1]$
```

Case	Bytes	Rounds	Possible initial state
0	1	4	[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1	2	4	[1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
2	3	4	[1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
3	4	3	[1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
4	4	4	[1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
5	5	3	[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
6	5	4	[1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
7	6	3	[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
8	6	4	[1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
9	7	3	[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10	7	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
11	8	3	[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
12	8	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0]
13	9	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
14	9	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0]
15	10	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]
16	10	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0]
17	11	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
18	11	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0]
19	12	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]
20	12	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1]
21	13	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0]
22	14	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0]
23	15	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
24	16	1	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Table 7.4: All evaluation cases with an example of a possible initial state for each evaluation case. "Bytes" gives the number of bytes controlled by attacker; "Rounds" gives the number of rounds that have to be attacked to recover the master key.

attacked key byte. By tracking the pairs associated with the recovered key bytes, the algorithm is able to discard all impossible round keys, thus saving computational resources. Indeed, the algorithm uses an optimal number of CPA attacks to recover the master key.

Initially, the set of known pairs is empty and all possible keys are considered valid. The algorithm keeps track of 2^p possible keys, where p is the total number of independent pairs in the symbolic representation of the state at round R.

The main loop of the algorithm runs through all rounds that must be attacked. At each round, the key bytes corresponding to variable state bytes are attacked to recover one or more temporary key bytes or a round key byte. Depending on the pairs associated with the byte to be attacked, there are three possible cases:

- **No pair.** If the symbolic representation does not have a pair associated with the byte of the state to be used for the attack, then the algorithm will recover a single key byte, which is distributed to all possible keys.
- **New pair.** If one of the pairs associated with the byte under attack is not present in the set of known pairs, then the algorithm will recover 2^u possible values for the corresponding key byte, where u is the number of known independent pairs associated with the byte under attack. The number of known pairs determines the number of CPA attacks to be performed. Using a mask based on the existing pairs and a mask for the new pair, the algorithm correctly distributes the recovered key byte values to all possible keys. The new pair is added to the set of known pairs and the two indexes of the state affected by the recovered temporary keys are mapped to this new pair. This mapping prevents the computation of the same temporary keys twice.
- **Known pairs(s).** In the case where the t independent pairs associated with the key byte to be attacked are known but not mapped to the current state byte, the algorithm performs 2^t CPA attacks. Then, it distributes the attack results (the recovered key and the difference between the correlation coefficients of the first two most likely key candidates α) to the corresponding bytes of all possible keys. Afterwards, the possible keys for which the value of α is less than the maximum observed value α_{max} minus a threshold β are marked as invalid. In this way, only the combination of keys yielding the highest correlation peak is selected. At this moment, the input pairs are solved in the sense that the algorithm can uniquely assign each of the two temporary keys of a pair to the corresponding state bytes. As a consequence, the algorithm will not further process the possible keys marked as invalid. Thus, this optimization improves the algorithm efficiency by reducing the number of performed CPA attacks.

Finally, the algorithm returns all possible keys, which are marked as valid. If the threshold β tends to zero, the algorithm will return only one possible key. When the quality of the side-channel acquisition is good (i.e. high signal-to-noise ratio) and there are enough power traces, the algorithm yields the correct key.

7.5.1 Optimality

We prove that our algorithm uses the minimum number of CPA attacks possible to recover the master key and thus is optimal. Hence, the lower bounds provided in Table 7.5 are optimal.

Theorem 7.5.1. *Algorithm 5 performs an optimal number of CPA attacks to recover the 16-byte master key of the AES.*

Proof. The only way an attacker can recover the 16-byte master key of the AES is to recover all key bytes of a round r and then to reverse the key schedule. Since the function that derives the round keys of round i from the round keys of round $i-1$ is bijective, knowledge of all round key bytes of a round r leads to the knowledge of the master key.

Let us assume that Algorithm 5 uses n individual CPA attacks for a given initial state and it is not optimal. Thus, there exists at least one algorithm that is able to recover the master key using only m CPA attacks, with $m < n$. We show next that such an algorithm does not exist. If there exists an algorithm that uses less CPA attacks than Algorithm 5, then this algorithm attacks at least one key byte less. But if it does so, then the attack can not be carried to later rounds any more because the state byte corresponding to the unrecovered key yields unknown and variable state bytes after the `MixColumns` transformation. These bytes can not be recovered using a CPA attack and thus the attack fails. As a consequence, there is no algorithm that uses less CPA attacks than Algorithm 5. □

7.5.2 Choosing the Best Attack Strategy

For up to seven bytes controlled by the attacker, our attack algorithm (Algorithm 5) is more efficient than the classic attack algorithm where all possible key bytes are attacked to recover the master key. The improvement varies between 15% and 68% of the number of CPA attacks required by the classic attack. When an attacker has control of more than seven input bytes, our algorithm performs the same number of CPA attacks as the classic attack. At the same time, our algorithm gives a unique master key, provided that there are enough traces with a high signal-to-noise ratio available. This is not the case for a classic attack unless an additional mechanism to discard invalid keys, as the one in Algorithm 5, is employed.

	Bytes	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
min(Rnds)	Classic attack	150	104	188	80	66	52	46	40	41	42	43	44	45	46	47	16
	Algorithm 5	48	42	48	38	38	38	39	40	41	42	43	44	45	46	47	16
	Improvement	102	62	140	42	28	14	7	0	0	0	0	0	0	0	0	0
max(Rnds)	Classic attack	150	104	188	110	72	50	51	52	53	54	55	56	45	46	47	16
	Algorithm 5	48	42	48	48	49	50	51	52	53	54	55	56	45	46	47	16
	Improvement	102	62	140	62	23	0	0	0	0	0	0	0	0	0	0	0

Table 7.5: The number of individual CPA attacks required to recover the master key for different number of bytes (Bytes) controlled by attacker; “min(Rnds)”/“max(Rnds)” and “Bytes” precisely identify the evaluation case. “Classic attack” does not use the optimizations introduced in “Algorithm 5” to discard the invalid keys. “Improvement” gives the number of CPA attacks saved by an attacker using Algorithm 5 over an attacker using “Classic attack”.

An attacker willing to reduce the duration of the offline phase of the attack (without increasing the number of rounds that must be attacked) can use the results in Table 7.5 in corroboration with the data in Table 7.3 to adjust the attack accordingly. More precisely, if an attacker is able to control up to n bytes of the AES input, she can choose to control m ($m \leq n$) bytes of the input because m variable bytes minimize the number of CPA attacks required to recover the master key. This decision has to be made before performing the side-channel acquisition since it influences the chosen inputs. Another argument in favor of using less variable input bytes is that the attack is much more difficult to detect if the injected packets have fewer variable bytes and mimic the appearance of a normal network traffic. For example, when $n = 12$, an attacker can choose $m = 4, 5,$ or 6 to reduce the complexity of the offline attack from 44 to 38 individual CPA attacks, while still attacking just three rounds. The result is an overall improvement of the attack efficiency by 14% over the classic attack.

An even better decision can be made with the help of experimental results for different configurations of the input from a similar target to the one to be attacked in addition to the results presented so far. For this reason, in the next section we determine experimentally the number of traces required to recover the master key for each evaluation case using EM leakages from an ARM Cortex-M3 processor.

7.6 Results

7.6.1 Electromagnetic Leakage

For the experimental evaluation, we considered two unprotected implementations of the AES written in ANSI C. The first implementation uses table lookups for the S-box, while the second one uses the T-table strategy. For each of the 25 evaluation cases we measured up to 2000 EM traces. The acquisition took about 90 minutes for an evaluation case. The samples were split into files corresponding to the AES round number. Then, we mounted the attack presented in Algorithm 5 using an increasing number of traces in the interval $[100, 2000]$ with a step of 100 traces until the guessing entropy converged to zero.

For each implementation we considered two selection functions based on the AES S-box and T-table, respectively. The minimum number of traces for which the guessing entropy becomes zero and remains stable is pictorially shown in Figure 7.5 for each evaluation case. All attacks recovered the full 16-byte master key with less than 1600 EM traces. In general, the master key was recovered with fewer traces when the selection function perfectly matched the implementation strategy. Though, our results show that full key recovery is possible even when the selection function does not perfectly match the attacked implementation. The attacks on the S-box implementation using the T-table selection function needed 204 more traces on average to recover the master key compared to the attacks on the same implementation using the S-box selection function. Similarly, using the S-box selection function instead of the T-table selection function to attack the implementation based on the T-table strategy required 354 more traces on average. For details on the exact number of traces required to recover the master key for each evaluation case and attack scenario we refer the reader to Section 7.6.3.

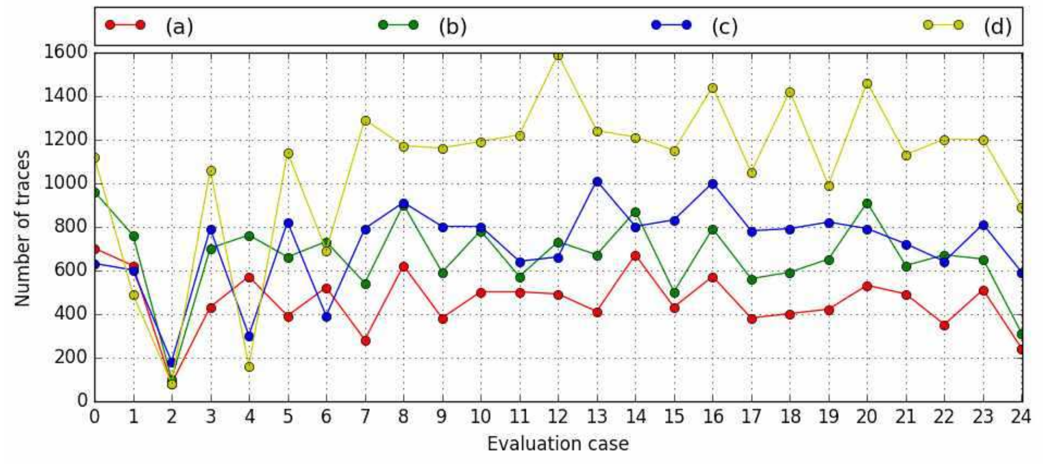


Figure 7.5: The number of EM traces required to fully recover the master key. Scenarios: (a) S-box implementation, S-box selection function; (b) S-box implementation, T-table selection function; (c) T-table implementation, T-table selection function; (d) T-table implementation, S-box selection function.

7.6.2 Simulated Leakage

We averaged the guessing entropy of 100 experiments on simulated traces for each of the 25 evaluation cases. Then we selected the minimum number of traces for which the guessing entropy was zero and remained stable. The results are shown in Figure 7.6.

Comparing the results for the simulated traces (Figure 7.6) with the results for the EM traces acquired from the Cortex-M3 processor (Figure 7.5), we can notice the following differences:

- In general, for simulated traces the attacks against the T-table implementation using the T-table selection function (c) required a similar but slightly smaller number of traces than the attacks on the S-box implementation using the S-box selection function (a). In contrast, the leakage estimation of the two selection functions indicates that the S-box selection function leaks a little bit more than the T-table selection function. But when the leakages of the two selection functions were quantified, they were clearly isolated from the leakages of other operations. As a consequence, the intermediate results of similar neighboring operations have a greater influence on the correlation of the S-box leakage than on the correlation of the T-table leakage. This can be explained by the fact that there are 19 possible Hamming weight values for the T-table output but only 9 possible Hamming weight values for the S-box output. Thus, it is easier to distinguish a T-table output from the neighboring T-table outputs than an S-box output from the neighboring S-box outputs. On the other hand, for the EM traces, the attacks on the S-box implementation using the S-box selection function (a) required less traces than the attacks on the T-table implementation using the T-table selection function (c). For the

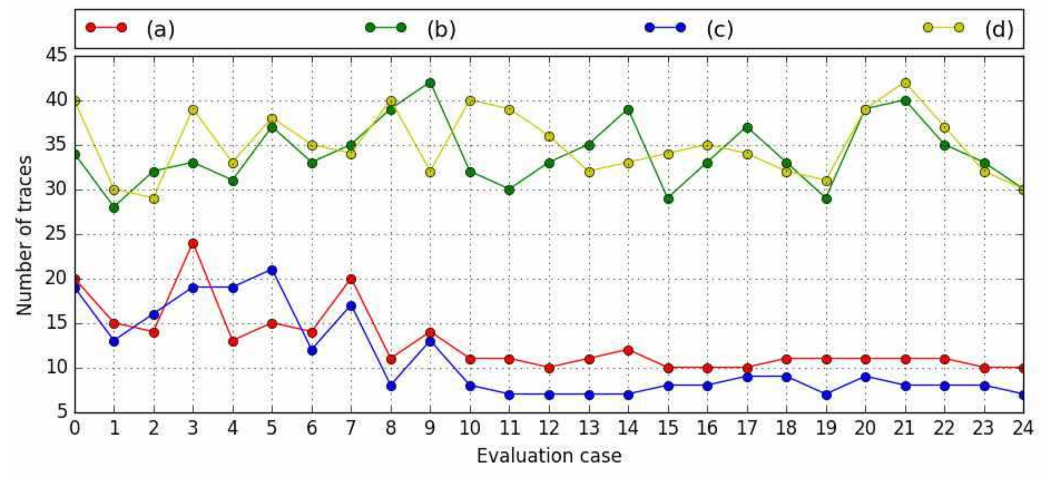


Figure 7.6: The number of simulated traces required to fully recover the master key. Scenarios: (a) S-box implementation, S-box selection function; (b) S-box implementation, T-table selection function; (c) T-table implementation, T-table selection function; (d) T-table implementation, S-box selection function.

EM traces, the attack results are consistent with the leakages of the selection functions as quantified by the correlation coefficient difference.

- The attacks that used the non-matching selection functions (b, d) required a similar number of simulated traces. Contrariwise, the attacks on the T-table implementation using the S-box selection function (d) required more EM traces than the attacks on the S-box implementation using the T-table selection function (b).

- In the case of simulated traces attacked with the matching selection functions (a, c), the number of traces necessary to fully recover the master key when the attacker controlled less than six input bytes was greater than when the attacker controlled more than six input bytes. On the contrary, the number of traces necessary to fully recover the master key for the EM leakage was minimal when the attacker controlled exactly three input bytes.

For details on the exact number of traces required to recover the master key for each evaluation case and attack scenario we refer the reader to Section 7.6.3.

7.6.3 Detailed Results

We give the number of traces required to fully recover the AES master key using simulated and EM traces for all evaluation cases in Table 7.6.

Case	Bytes	Rounds	EM leakage				Simulated leakage			
			(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)
0	1	4	700	960	630	1120	20	34	19	40
1	2	4	620	760	600	490	15	28	13	30
2	3	4	80	100	180	80	14	32	16	29
3	4	3	430	700	790	1060	24	33	19	39
4	4	4	570	760	300	160	13	31	19	33
5	5	3	390	660	820	1140	15	37	21	38
6	5	4	520	730	390	690	14	33	12	35
7	6	3	280	540	790	1290	20	35	17	34
8	6	4	620	900	910	1170	11	39	8	40
9	7	3	280	590	800	1160	14	42	13	32
10	7	4	500	780	800	1190	11	32	8	40
11	8	3	500	570	640	1220	11	30	7	39
12	8	4	490	730	660	1590	10	33	7	36
13	9	3	410	670	1010	1240	11	35	7	32
14	9	4	670	870	800	1210	12	39	7	33
15	10	3	430	500	830	1150	10	29	8	34
16	10	4	570	790	1000	1440	10	33	8	35
17	11	3	380	560	780	1050	10	37	9	34
18	11	4	400	590	790	1420	11	33	9	32
19	12	3	420	650	820	990	11	29	7	31
20	12	4	530	910	790	1460	11	39	9	39
21	13	3	490	620	720	1130	11	40	8	42
22	14	3	350	670	640	1200	11	35	8	37
23	15	3	510	650	810	1200	10	33	8	32
24	16	1	240	310	590	890	10	30	7	30
Average			459	663	716	1070	13	34	11	35

Table 7.6: The number of traces required to fully recover the master key for each evaluation case ("Case"). "Bytes" gives the number of bytes controlled by attacker; "Rounds" gives the number of rounds that have to be attacked to recover the master key. Scenarios: (a) S-box implementation, S-box selection function; (b) S-box implementation, T-table selection function; (c) T-table implementation, T-table selection function; (d) T-table implementation, S-box selection function.

7.7 Countermeasures

Our experimental results show that side-channel countermeasures such as masking must be employed in order to protect the AES implementations based on lookup tables (S-box and T-table implementation strategies) even in a communication protocol scenario, when the adversary has limited control of the input. Masking nonlinear lookup tables is a challenging task since it adds a considerable penalty on execution time and memory usage [368].

Although bit-sliced implementations are not present in many cryptographic libraries due to their limitations (i.e. can not be used in a feedback mode of operation such as CCM), they have a lower CPA leakage than implementations using lookup tables [50]. Nevertheless, they are still vulnerable to DPA attacks [29].

A lightweight algorithm (block cipher or authenticated encryption), particularly one designed for efficient masking, is a good replacement for the AES-CCM when considering side-channel protection.

Other countermeasures, such as a key refreshing mechanism, can support a defense in depth approach. However, any additional countermeasure affects the overall efficiency of an IoT protocol and consequently the most effective ones (i.e. masking) must have priority given the resource constraints.

7.8 Summary

In this chapter, we presented an extensive security analysis of AES software implementations against CPA attacks in the context of network protocols. In this scenario the attacker has control of several input bytes, while the remaining input bytes are fixed. To asses the resilience of AES implementations to all possible input combinations, we presented an algorithm for symbolic processing of the cipher state. Then, we classified all possible inputs into 25 independent evaluation cases depending on the number of input bytes controlled by attacker and the number of rounds that must be attacked to recover the master key. Finally, we described an optimal algorithm that recovers the master key by mounting the minimum number of CPA attacks possible. It makes clever decisions based on the set of key pairs that affects the key byte under attack and the correlation coefficient of possible key candidates to discard impossible keys.

We showed that unprotected implementations of the AES based on the S-box and T-table strategies can be broken even when the attacker controls only one input byte of the cipher with less than 1600 electromagnetic traces acquired from a 32-bit ARM Cortex-M3 processor in about one hour. Knowledge of the implementation strategy does not significantly improve the attack outcome, nor does it reduce the attack complexity. Thus, unprotected implementations of the AES should not be used to secure the communication between end devices in networks. Care must be taken when using implementations of the AES from popular open-source cryptographic libraries since most of them are not protected against side-channel attacks.

Chapter 8

An Electromagnetic Vulnerability Analysis of Thread

Contents

8.1 Introduction

Over the last few years we have seen a huge increase of IoT-enabled devices available on the market. These devices, intended to make our lives easier by collecting, processing and exchanging data, are manufactured by various companies around the world. To foster the development of industry-wide standards for smart devices, companies from different business fields gathered together in various working groups, organizations or consortia. Their aim is to augment the smart objects' capabilities by enhancing the communication and data exchange between devices from different manufacturers. This is a challenging task given the heterogeneous nature of the IoT comprising a vast variety of devices, of which the overwhelming majority is characterized by a multitude of constraints such as energy or power consumption, code size and memory footprint to name a few.

The IoT ecosystem is still in its early inception stages and a lot has to be done until all smart devices can communicate seamlessly with each other. Unfortunately, compared to the current abundance of emerging standards for the IoT, there is little to no effort to thoroughly analyze the security of these proposals. Thus, neither the companies involved in the development of such standards, nor the end users are fully aware of the security and privacy aspects of future connected products that will flood the market in the coming years.

8.1.1 Attack Surface and Threats for Connected Devices

In the connected world, attacks that can be mounted remotely pose a major threat. Software exploitation and network attacks fall into this category. They require low resources (usually just a connected PC), and do not require physical proximity.

Especially in the context of the connected home, most current devices are within the home perimeter and physical access would mean that the attacker is already inside the house (assuming building access as an asset). However, devices like smart locks and cameras are on the edge or outside of the building perimeter, and thus may be physically accessed. In the near future, we will most likely see devices for outdoor lighting or garden sprinklers connected to the smart home ecosystem. An attack on one of such devices may provide an entry point to the ecosystem.

Physical proximity attacks pose a relatively larger threat in a commercial setting, for instance a hotel where rooms are equipped with wireless door locks. If an adversary has access to such a smart lock connected to a mesh network like Thread in her room, she might be able to exploit it in a similar way to what we describe in this chapter to get access to the hotel network. A recent security incident targeting the access control system of a hotel [256] shows that this type of scenario is well possible; see also the physical attack on a door lock presented in [342].

8.1.2 Motivation

With the growing complexity of exploit-mitigation techniques, recent software attacks need to chain several vulnerabilities to succeed (e.g. [147]). Moreover, new network protocols are designed with security in mind.

We are driven by the curiosity to see if extending the attack surface to physical attacks, such as electromagnetic or power analysis (common in the smart card security world), would give additional benefits to the attacker that pay off the need for physical proximity. With the increased availability and reduced cost of both hardware and software tools for side-channel attacks ([252], [76], [323] to list a few) these attacks are becoming familiar and affordable to a wide hacker community. We are interested to evaluate the realistic effort required to apply such an attack in the IoT context.

The network layer of the connectivity stack typically relies on a master key and relatively long-lived network keys to provide the first layer of defense against an attacker in the proximity of an IoT device. A question raised by the designers of IoT hardware is: *Do cryptographic implementations in the network layer need protection against side-channel attacks?* We have not seen a consolidated opinion on this matter, with academic experts in Differential Power Analysis (DPA) claiming attacks are possible (see related work paragraph below), while industry being on the conservative side. In our view, this disagreement is due to the lack of in-depth case studies that could demonstrate the feasibility (or infeasibility) of such attacks.

Numerous articles and marketing campaigns advertise Thread [351] as a new, efficient and secure solution for the connected home with the roadmap to expand into the commercial building and professional sectors. The claim of being *always secure* [1] garnered our attention and made us curious to check ourselves if this claim is true, especially in view of the availability of OpenThread. The lessons learned from side-channel analysis applied to the Thread networking stack could be used to improve the overall security level of current and future protocols designed for the IoT.

8.1.3 Contribution

We perform, to the best of our knowledge, the first public side-channel vulnerability analysis of Thread. Thread is a complex networking stack and an exhaustive analysis

[1]Meanwhile, the claim has been changed to "built-in security". See `http://threadgroup.org/What-is-Thread/Overview`

would require a tremendous effort, especially when multidisciplinary attack vectors are considered as in our work. While providing some coverage, we focus on finding fast and effective ways to get access to a Thread network. Our contributions are:

- We perform a vulnerability analysis of Thread specifically with respect to an adversary capable to mount electromagnetic side-channel attacks. In this context, we outline several attack vectors to bypass the security mechanisms of the networking stack. In particular, we target manipulations of the security material (i.e. cryptographic keys).
- We describe a fully implemented attack that chains the exploitation of network-level mechanisms and electromagnetic side-channel analysis techniques to get unauthorized access to an existing Thread network after several hours of acquisitions in the close proximity of a Thread Router or Router-Eligible End Device that is already in the network.
- We explain that the failure of the full attack is due to a fortunate side-effect of a feature not related to security (packet fragmentation). Therefore, the protocol weaknesses we discovered are relevant.
- We describe a range of countermeasures for the protocol and for the implementation that can be applied whenever side-channel resistance is required.

We believe this case study provides a useful lesson to designers of IoT protocols and devices. Our work comes early in the life cycle of future Thread products. Because of this, we believe that it has a more profound impact, although being less impressive than breaking an off-the-shelf Thread device.

8.1.4 Related Work

In the past years, numerous papers addressed various aspects of IoT security. One notable direction is the analysis of the security and privacy of software frameworks for the IoT [126, 127]. A survey of the security and privacy of implantable medical devices and body area networks is given in [301].

Though, the impact of side-channel attacks on the security of connected objects is far from being completely and clearly understood. A step towards this goal was made by the following works. O'Flynn and Chen attacked the MAC layer encryption of an IEEE 802.15.4 node [259] using side-channel attacks; they describe the approach and implement the basic steps but not the full attack. Their attack builds on previous works of Jaffe [175] and Kizhvatov [190]. Ronen *et al.* [300] exploited popular smart lights to create a worm capable to quickly spread an infection over large areas. Their work used side-channel attacks to recover the global AES-CCM key used to encrypt and authenticate firmware updates.

Compared to the work described in [259], our attack is performed in the context of Thread. It bypasses more complex security mechanisms to affect the full Thread networking stack and not only the standalone MAC layer. By our full implementation, we demonstrate that the threat posed by the recovery of the MAC layer key largely depends on the context, specifically on the upper layers, and that there may be unexpected hurdles. Additionally, we improve the attack on AES-CCM of [259] by increasing the number of ciphertext bytes under our control. As a consequence, we have to attack one AES round less to recover the 16-byte key.

Therefore, to our knowledge, this work is one of the few to demonstrate an EM analysis attack in a complex wireless network setting, and to address the security of an IoT network protocol with respect to adversaries capable to mount side-channel attacks.

8.1.5 Responsible Disclosure

We informed the Thread Group in October 2016 about our findings and proposed countermeasures. We received a confirmation of our findings. Based on our report, the Thread Group decided to elaborate a set of recommendations for implementers in order to enhance the security of Thread products.

8.2 Thread

Supported by more than 200 companies, including most major players in the IoT arena, the Thread Group [351] is a nonprofit organization that promotes Thread's use in connected home solutions. Thread is a network and transport level stack of protocols designed to simplify consumer lifestyles by controlling and connecting products at home. In November 2016, the Thread Group announced the expansion of Thread beyond the connected home to commercial spaces where people work [353]. Nest released an open-source implementation of Thread, called OpenThread, on GitHub [263] in May 2016. As of October 2017, the OpenThread GitHub repository is supported by ten members of the Thread Group and is an important resource for hobbyists and early adopters who cannot afford the membership fee. OpenThread runs on a number of wireless hardware platforms.

Based on well-established technologies, the Thread networking stack is built on top of physical and data link layers of IEEE 802.15.4 [170], operating at 250 kbps in the 2.45 GHz band [351]. Thread uses 6LowPAN to enable IPv6 addressing of up to 250 devices per network. The mesh network topology of Thread accommodates up to 32 routers to create a resilient network with no single point of failure. It provides an efficient way to forward messages between nodes using the RIPng distance vector routing protocol. For its transport layer, Thread uses UDP and DTLS. CoAP is used as application layer for the commissioning of new devices.

Thread devices are classified into two groups (see Table 8.1) based on power requirements and resource characteristics: Full Thread Devices (FTDs) and Minimal Thread Devices (MTDs).

- An *FTD* is usually supplied directly from the power lines, but it can also run on batteries. A FTD can have three different roles in a Thread network: Router, Router-Eligible End Device (REED), and Full End Device (FED).

- An *MTD* runs a lighter version of the Thread stack with reduced capabilities due to its limited resources; it usually runs on batteries. A MTD can have one of the following roles: Minimal End Device (MED) or Sleepy End Device (SED).

A key difference between FTDs and MTDs is that FTDs keep a communication link with neighboring Routers, while MTDs do not. A device that is not a Router is called an End Device (ED) and it is attached to a Parent with whom it communicates through a direct link.

A device attaches to a Thread network as an ED. During the lifetime of a Thread network, a device can have different roles at different moments of time. For example, a REED can become a Router if the network configuration is favorable; similarly, a Router can become a REED. A Thread network is managed by a Router autonomously elected by the network and called Leader. The Leader assigns router addresses, collects and distributes information about the network state to all Routers. If the current Leader becomes unavailable, another Router will replace it. A Thread Router having other network interfaces (Ethernet, Wi-Fi, Bluetooth, etc.) is called a Border Router. It can forward the traffic between the Thread network and other networks.

Type	Role	Description	End Device (ED)
FTD	Router	acts as a router	✗
	REED	can act as a router	✓
	FED	will never act as a router	✓
MTD	MED	always on	✓
	SED	sleeps most of the time	✓

Table 8.1: Device types and roles in a Thread network.

Network security is enforced at the MAC (Media Access Control) and MLE (Mesh Link Establishment) layers using AES in CCM mode [263]. All communication within a Thread Network is secured, except for `MLE Discovery Request` and `MLE Discovery Response` messages. Commissioning security is based on a DTLS tunnel established using elliptic curve J-PAKE and the NIST P-256 elliptic curve.

8.2.1 Security Material

Once successfully commissioned into a Thread network, the connected device gets the 16-byte network master key MK used to secure Thread communication and the Commissioning Key CK used to secure Thread commissioning [263].

Each node keeps its own 4-byte *Sequence* counter in synchronization with neighboring devices through the use of designated fields in the security header of MAC frames (1-byte `Key Index`) and MLE messages (4-byte `Key Source`). The 1-byte

`Key Index` is computed from the 4-byte *Sequence* number:

$$KeyIndex = (Sequence \wedge \texttt{0x7F}) + 1 \tag{8.1}$$

Thread communication is secured using a 16-byte MAC key K_{MAC} or a 16-byte MLE key K_{MLE}. These keys are derived from the 4-byte *Sequence* number concatenated with the ASCII binary representation of the string "Thread" (`0x54 0x68 0x72 0x65 0x61 0x64`) using the keyed-hash message authentication code (HMAC) function $HMAC$ under the network master key MK as described below. The hash function used is SHA-256.

$$K_{MAC} \parallel K_{MLE} = HMAC_{MK}(Sequence \parallel \text{``}Thread\text{''}) \tag{8.2}$$

Fresh MAC and MLE keys are generated when the default key rotation timer (set to 672 hours) expires. The *Sequence* number is incremented by one, the $KeyIndex$ value is updated, the $HMAC_{MK}$ function is executed and the key rotation timer is rearmed. When refreshing the keys, the outgoing MAC and MLE frame counters are reset to zero.

When receiving an MLE message with a different *Sequence* number set in the `Key Source` field, the receiver computes a temporary key using the received sequence as described in Equation 8.2. In the case of MAC frames, if the received $KeyIndex$ is not equal to the computed $KeyIndex$ from Equation 8.1, the receiver will generate a temporary key only when the absolute difference between the two values is one. This temporary key allows a node to synchronize with its Parent after a period of absence from the Thread network.

Each Thread node, regardless of its type and role, stores the security material and network parameters of the Thread network to its non-volatile memory to be able to rejoin the network after a reset without human intervention.

8.2.2 Mesh Link Establishment (MLE)

The Mesh Link Establishment (MLE) protocol facilitates the secure configuration of radio links and exchange of network parameters. The MLE messages are sent inside UDP datagrams with the source and destination ports set to 19788. The security of MLE messages is provided by AES in CCM mode using the MLE key K_{MLE}. The `Auxiliary Security Header`, `Source IP` address and `Destination IP` address are authenticated using a 32-bit message integrity code (MIC), while the payload is encrypted. The `Auxiliary Security Header` of an MLE message includes the 4-byte `Key Source`.

A Thread Router periodically multicasts `MLE Advertisement` messages to advertise its presence. Such a message is sent at an interval between 1 and 32 seconds after the previous advertisement was sent by the same Router. The Thread REEDs advertise their presence by multicasting a similar message every ten minutes on average.

The process of establishing a communication link between two Thread nodes N_1 and N_2 is depicted in Figure 8.1. In this case, the Child node N_1 is creating a communication link with its Parent N_2 in three phases: *Attaching*, *Child Synchronization*, and *Link Synchronization*. This message exchange occurs, for example, when an MTD reconnects to a Thread network.

Before initiating the MLE message exchange shown in Figure 8.1, the Child sends an `MLE Discovery Request` to locate the existing Thread devices. It gets in response an `MLE Discovery Response` message containing the Thread network channel number and its PAN ID.

In the *Attaching* phase, the Child (N_1) multicasts an `MLE Parent Request` message with a randomly generated 8-byte challenge. All Routers and REEDs that receive this request store the received challenge and answer with an `MLE Parent Response` message including the received challenge and a new random 8-byte challenge. N_1 selects one of the answers it receives based on the link quality and unicasts an `MLE Child ID Request` message that includes the received challenge to the corresponding node N_2. The Parent sends an `MLE Child ID Response` which may include the network configuration parameters. Then, the *Child Synchronization* takes place. The Child sends an `MLE Child Update Request` to its Parent and gets in response an `MLE Child Update Response` message.

The communication link is established in the *Link Synchronization phase*. Initially, the Child multicasts an `MLE Link Request` message containing an 8-byte random challenge. The Parent answers with an `MLE Link Accept & Request` message that includes the received challenge and a new randomly generated challenge. The Child confirms the Parent request by sending an `MLE Link Accept` message, which includes the challenge received from the Parent.

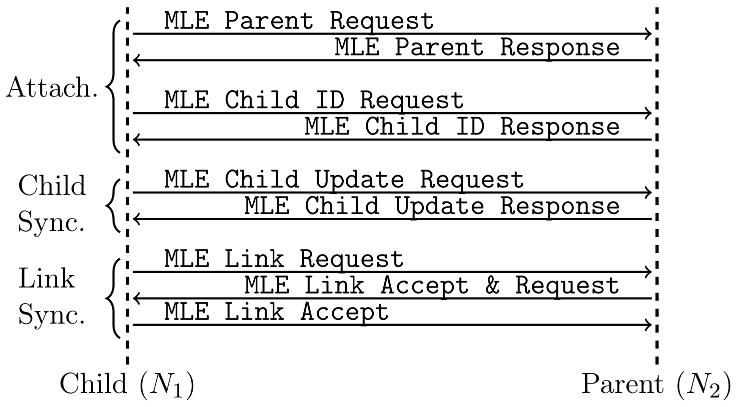


Figure 8.1: Establishing a communication link between two Thread nodes.

8.3 Threat Model

The are numerous avenues an attacker can try to compromise a Thread network. While some threats were well understood and properly mitigated in the design phase of the networking stack, others were less obvious and thus overlooked. The latter ones are harder to eliminate as the protocol becomes more mature and widely used [242]. We make a classification of the possible attack types in the IoT environment, without aiming at a complete coverage of all threats specific for these settings. The threat modeling we performed for Thread can be easily adapted to other IoT protocols and the lessons learned from this study can be employed to protect other IoT solutions as well.

The goal of our threat model is to provide a classification and a better understanding of Thread's attack surface. The attacker attempts to affect one or more of the basic security functions: confidentiality, integrity, and availability. The primary goal of the attacker is to get access into a Thread network in order to intercept and understand the communication or to take control of the network. Other objectives include, but are not limited to, disrupting the normal network operation by performing a DoS attack or altering data sent across the network through a man-in-the-middle attack.

Similar to the work of Atamli and Martin [21], we consider three main entities that can threaten the security of an IoT system: *legitimate user*, *device maker*, and *malicious adversary*. A *legitimate user* poses a threat to the security of an IoT system when, for example, she seeks to bypass the authentication, authorization, and accounting mechanisms used by the target system. In this way, she might unlock restricted features of the device or use the existing ones without paying for them. A *device maker* can threaten the security of an IoT system either accidentally (e.g. poorly implemented security mechanism) or deliberately (e.g. aiming to collect user's data). Finally, a *malicious adversary* is the classical attacker willing to get unauthorized access to a system or to damage that system.

Depending on the location of the attacker with respect to the target system, we distinguish between: *remote*, *proximity*, and *invasive* attacks. The powerful *remote* attacks are well understood from the classical Internet-connected systems. Although, some IoT protocols such as Thread assume that not all IoT devices inside a network are directly accessible from the Internet. Thus the attack surface of *remote* attacks on IoT networks is reduced compared to the attack surface of the same attacks on the classical Internet. The IoT is a highly heterogeneous environment with devices deployed in various, including distant, locations. In such settings, it is hard to enforce the physical security of these systems that become vulnerable to *proximity* and even *invasive* attacks. *Proximity* attacks can be performed without physical access to the target device and thus are harder to detect than *invasive* attacks. A summary of the attack types specific to the IoT is given in Table 8.2.

Proximity attacks are very feasible in the IoT since most protocols use wireless communication means and are deployed in easily accessible spots. Hence, the attacker can easily get in the proximity of a Thread device and observe it performing various operations. Given the ubiquitous nature of the IoT, it is expected that the attacker is able to quickly identify such target devices. We assume the attacker can carry a portable oscilloscope and an EM probe required to perform an EM analysis attack.

We do not restrict the attacker's capabilities in terms of equipment or physical location to accurately capture the current state of security in the IoT with respect to EM analysis attacks.

Attack type	Attack	Mitigation
remote	software exploitation	system hardening
	guessing password	strong security policy
	brute force	
proximity	EM analysis	side-channel countermeasures
invasive	power analysis	
	fault attacks	
	flash read-out	flash read-out protection

Table 8.2: Summary of the attack types specific to the IoT.

8.4 Side-Channel Vulnerability Analysis

The goal of our vulnerability analysis was to investigate attack paths that can provide full access to a Thread network. This allows us to sniff and understand all network traffic, to add new devices into the network, and to take control of the network by changing the security material. In order to achieve this, we explored different attack vectors to recover the security material of the network. We first explore the feasibility of several active attacks paths; these are attacks in which the attacker injects packets to trigger different operations on the target nodes. Then, based on the results of the active attacks, we can estimate how successful a passive attack exploiting the same mechanism could be.

We did not look for implementation-specific issues such as buffer overflow attacks, fragmentation attacks, or improper input sanitization, because they would be meaningful only for a particular software implementation. We did not include attacks that affect the availability of the network such as denial-of-service (DoS) attacks in our scope. Below we present the most promising attack paths. Other attack paths are described in Section 8.8.

8.4.1 Relationship between MK and K_{MLE}

Having the master key MK and *Sequence* number, a node can compute the MLE key K_{MLE} using Equation 8.2. If a node possesses the MLE key K_{MLE} but does not have the master key MK, it can send an `MLE Child ID Request` to ask for the network master key. Its Parent will answer with an `MLE Child ID Response`, which includes the `Master Key TLV` containing the requested master key MK. This means that MK and K_{MLE} are equivalent, in the sense that if a node has one of them, it can easily compute or retrieve the other one. Giving access to the master key to nodes having a key derived from it generates serious security issues as we will describe later.

However, this approach has a major limitation. The `Master Key TLV` is just a small fraction of the data included in the `MLE Child ID Response`. Although the attacker can ask for some specific TLVs in his request, the Parent decides the exact content of the response message. If the answer fits into a single MLE message, then the response is encrypted only at the MLE layer. As the total payload length of the `MLE Child ID Response` message exceeds the maximum transmission unit (MTU) of 127 bytes, the MLE message is fragmented at the 6LowPAN layer. When fragmentation occurs, all resulting fragments are encrypted at the MAC layer using K_{MAC}. Thus, the attacker has to first decrypt the MAC frames, then to reassemble the fragments of the original MLE message, and finally to decrypt the MLE message in order to get the value of the `Master Key TLV`.

Hence, the attacker can get the master key when she knows only the MLE key if the response MLE message is not fragmented and thus not encrypted at the MAC layer. We note that this mechanism is not affected by the `OBTAIN_MASTER_KEY` bit of the `Security Policy TLV`. When set, the `OBTAIN_MASTER_KEY` bit enables a Commissioner to extract the master key for out-of-band commissioning after she was authenticated.

This mechanism does not help a node to reconnect to a Thread network if the network master key was changed while it was sleeping because the node does not possess a valid MLE key to ask for the new master key. Thus its MLE requests will be dropped, and it has to be commissioned again by a human to the Thread network.

8.4.2 Processing of an `MLE Parent Request`

An obvious option for an attacker is to exploit the very first message exchange that allows a Child to connect to a Parent in a Thread network. Next we detail how a Router processes the first message sent by a Child that wishes to establish a communication link with a Parent.

Upon receipt of an `MLE Parent Request` message, the receiving Router extracts the received *Sequence* number from the `Key Source` field. Then, it compares the value of the received *Sequence* with its current internal *Sequence* number. There are two possible cases:

- If the two sequence numbers are equal, then the Router continues by processing the authentication tag of the received MLE message using the current MLE key K_{MLE} of the Thread network.
- If the two sequence numbers are not equal, then the Router derives a temporary key from the received *Sequence* number. The Router uses this temporary MLE key K'_{MLE} to process the authentication tag of the received message.

If the resulting tag is the same as the authentication tag present in the received `MLE Parent Request` message, then the Router prepares a response. Else, it will drop the received MLE message.

Whatever processing path the Router follows, it will perform at least an AES-CCM operation on the received message. Thus, an attacker can easily trigger executions of HMAC-SHA256 or AES-CCM by pretending to be a Child willing to connect to a Parent. If the attacker chooses to trigger HMAC-SHA256 executions on the receiving Router, she has to inject `MLE Parent Request` messages with a different *Sequence* number from the one observed in `MLE Advertisement` messages. On the other hand, if the attacker is interested in observing AES-CCM computations with the current network MLE key K_{MLE}, she has to inject `MLE Parent Request` messages with the same *Sequence* number. Hence, an attacker not yet connected to the target Thread network can take advantage of a normal network mechanism used to establish a communication link between a Child and a Parent Router to trigger executions of the underlying cryptographic algorithms with sensitive key material at her own will.

8.4.3 Attack on Key Generation

An attacker can inject `MLE Parent Request` messages with a chosen *Sequence* number. When receiving an `MLE Parent Request` with a different *Sequence* number from its own *Sequence* number, a Router will derive a temporary MLE key using Equation 8.2.

Although it is possible to trigger executions of the $HMAC$ function, the number of input bytes controlled by attacker is not enough to make the recovery of the master key MK possible as we show next.

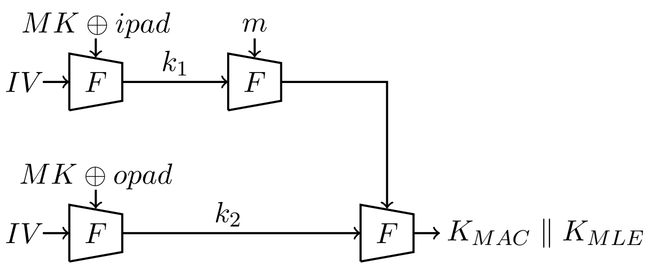


Figure 8.2: Key generation using HMAC.

The key derivation is pictorially shown in Figure 8.2. The one-way compression function of SHA-256 is denoted by F. The input message m to the HMAC function is obtained by concatenating the 4-byte *Sequence* number with the 6-byte representation of the "Thread" string, the 46 bytes of padding, and the 8-byte message length len: $m = Sequence \parallel$ "*Thread*" $\parallel$ `0x80 0x00`... `0x00` $\parallel len$. It is easy to observe that the only variable part of the message m is the *Sequence* number. Thus, the attacker controls exactly four bytes of the input message of the HMAC function.

If the attacker could recover $k_1 = F(IV, MK \oplus ipad)$ and $k_2 = F(IV, MK \oplus opad)$, then she could generate the MLE and MAC keys having only the correct *Sequence* number, but not the master key MK. Though, having the current MAC and MLE keys of the Thread network, the attacker can get the network master key from a Thread node as described in Section 8.4.1. In order to recover k_1 and k_2, the attacker will target executions of the compression function $F(k_1, m)$. The attacker controls four bytes (a 32-bit word) of the input message, which are mixed in the first iteration of the compression function F with constant but unknown bytes of the internal state.

As a consequence, she can learn the relationship between four unknown but constant 32-bit words by attacking a 32-bit modular addition in the first iteration. The attack stops here, because the attacker can not propagate it to the next iterations in absence of known and variable data.

Thus, an attacker can not exploit executions of the *HMAC* function in unprotected implementations using a CPA attack due to the limited control she has over the input. Though, the attacker still has the option to perform a template attack. Due to the complexity of the profiling phase required for mounting a template attack, we decided to stop our investigation at this point and to explore other attack paths instead.

8.4.4 Attack on the AES in CCM Mode

By injecting `MLE Parent Request` messages with the same *Sequence* number as the one used by the target Thread network, an attacker triggers executions of the AES in CCM mode with the current MLE key on the receiving Routers and REEDs.

A typical input block for the two stages (AES-CBC and AES-CTR) of the AES-CCM is shown in Figure 8.3. The constant (fixed) input bytes are given in hexadecimal notation, while the variable bytes are colored in gray. The first input block of the AES in CBC mode is very similar to the first input block of the AES in CTR mode. The first input byte is used for flags and thus is fixed. The last three input bytes are also fixed. The antepenultimate byte specifies the security level (`0x05`). As in the MAC layer of IEEE 802.15.4, this value indicates the use of encryption and authentication with a 4-byte message integrity code (MIC). In the case of AES-CBC the last two bytes are used to indicate the input plaintext length, while in the case of AES-CTR they represent the counter value. The counter value starts from one because the all-zero counter value is used for the computation of the authentication tag [115]. The variable bytes for both input blocks are the 8-byte source MAC address and the 4-byte frame counter.

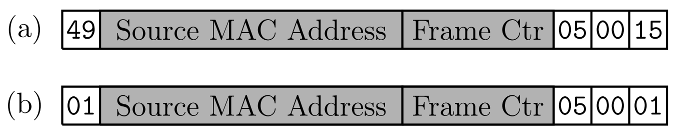


Figure 8.3: Input format for the first block of (a) AES-CBC and (b) AES-CTR.

An attacker can choose to craft `MLE Parent Request` messages having different payload lengths. As a consequence, the last byte of the AES-CBC is variable. This additional variable byte does not improve the attack outcome, but it is very likely to trigger an alert for abnormal network traffic in an intrusion detection/prevention system (IDS/IPS), if available.

To successfully mount a CPA attack, the attacker needs to vary a part of the input of the AES-CCM executions. As shown, an attacker can control up to 12 bytes of the input for AES executions. Thus, she can target either the first execution of the AES in CBC mode or the first execution of the AES in CTR mode.

Since the attacker does not control all input bytes, she has to attack three rounds of the AES in order to recover the 16-byte key. To propagate the CPA attack to the later rounds, the attacker must use the method described by Jaffe [175] to compute temporary keys that incorporate the constant input bytes. We found this MLE key recovery attack vector very promising and we chose to exploit it. Details of the full attack are given in Section 8.5.

8.5 Implementation of the Most Feasible Attack

In this section we describe each individual step of an attack path against a Thread network that chains the vulnerabilities presented above in the most feasible way. Then, we present the experimental setup we used to perform the attack. We show and analyze the results of the attack. Finally, we discuss the cost and complexity of the full attack.

The attack consists of the following four steps:

1. The attacker eavesdrops on the network traffic and records the *Sequence* number present in the `MLE Advertisement` messages sent by Thread Routers and REEDs.

2. The attacker observes and records the EM emanations of a target Router or REED while she injects `MLE Parent Request` messages. The injected messages use the observed *Sequence* number to trigger executions of the AES with the current network MLE key. The variable and known inputs necessary to perform the CPA attack are the source MAC address and frame counter fields. They are randomly generated for each injected message. We stress that the side-channel attack is applicable only to Thread Routers and REEDs because they process the `MLE Parent Request` messages. This step requires the attacker to be in the proximity of the target device such that she is able to reliably measure the EM emissions. After the attacker has recorded enough traces, she continues with the next step of the attack.

3. The attacker correlates the observed EM leakages with a hypothetical model of a key-dependent sensitive intermediate variable in order to determine the unknown key. In practice, CPA is an efficient technique and thus it is employed to recover the MLE key used during the observed computations.

4. Having the current MLE key of the network, the attacker attaches to a Thread Router. She asks the Router for the network configuration parameters including the master key by sending an `MLE Child ID Request` message. The Router will give the attacker the requested information in an `MLE Child ID Response` message.

It is essential for the success of the attack that all the above-mentioned steps succeed. Failure of any of these steps will render the full attack infeasible. The success of the last step highly depends on how the Parent handles the `MLE Child ID Request` messages and on the length of the `MLE Child ID Response` message as discussed in Section 8.4.1. If the message is fragmented and therefore additionally encrypted with the MAC key, the attacker will need to recover K_{MAC}. For example, the MAC key can be recovered by mounting a CPA attack on the AES-CCM executions that use K_{MAC}. Though this is a possible path, we believe that it makes the attack even more complex and we do not analyze it further.

An important factor that can be controlled to a certain extent by the attacker is the quality of the side-channel acquisition. The better the signal-to-noise ratio of the recorded traces, the fewer traces the attacker will need. The number of traces required to recover the MLE key influences the duration of the acquisition. Thus, the risk of the attacker being noticed increases as she needs more EM traces. The reason is twofold. Firstly, the attacker has to spend more time in the vicinity of the target Router or REED. Secondly, she has to inject more messages to trigger executions of the AES.

Once the attacker has the network configuration parameters, including the security material, she has all the rights of a genuine member of the Thread network. Hence, she is able to communicate with other nodes and can understand the communication between other nodes. The attacker can commission new devices to the Thread network. She can become a Router and then change the network parameters so that the owner of the Thread network loses the control over his network.

8.5.1 Experimental Setup

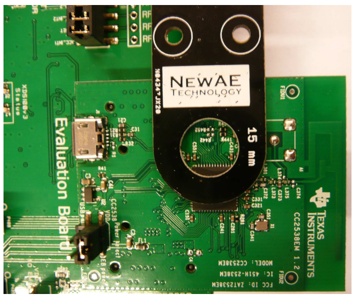
8.5.1.1 Thread Network

We created our own Thread network consisting of two CC2538EM wireless microcontrollers [349]. These devices were an obvious choice since they were the first to support the OpenThread implementation [263]. Moreover, our experimental network uses one of the seven products (device and software stack bundle) certified by the Thread Group [352]. The CC2538EM microcontroller has an ARM Cortex-M3 processor clocked at 32 MHz, up to 512 KB of flash memory and an IEEE 802.15.4 radio transceiver. The purchasing cost of this hardware was much smaller than that of similar devices shipped with proprietary implementations of Thread.

Initially, we experimented using the source code of Open-Thread to understand the communication and the network mechanisms. Then, we modified the source code such that we were able to generate various sequences of messages. Following this approach we were able to better understand the source code and we inferred information about part of the Thread specifications. We emphasize that OpenThread's codebase is very complex. Thus, understanding the relevant network mechanisms to our analysis was a challenging task that required a tremendous effort. Finally, we modified the state machine of the OpenThread implementation to make an attacker able to inject custom-crafted messages.

8.5.1.2 Measurement Setup

For the acquisition of the EM emissions from the target device, we used a Teledyne LeCroy WaveRunner 625Zi [347] oscilloscope. Initially, we ran AES encryptions in a loop on the target device and we used an EM probe to locate the spots that leak information during these executions. To hasten the process we firstly checked the area around the chip and then several decoupling capacitors we could identify using the schematic and layout of the board [349].


(a) H-field probe.

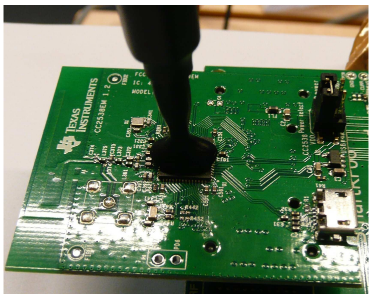

(b) Near field probe.

Figure 8.4: The EM probes used for measurement of the EM leakage.

We started to capture the EM signal with a relatively cheap NewAE H-field probe having a 15 mm coil as shown in Figure 8.4a. Because the signal-to-noise ratio was not satisfactory, we switched to a more precise (coil sizes of about 1 mm) and relatively more expensive set of near field probes from Langer (see Figure 8.4b). We set the probe a few millimeters above the target board. For the results reported in this chapter we fixed the sampling rate to 1 GS/s. A lower sampling rate (500 MS/s) significantly affected the attack outcome, while a higher sampling rate (5 GS/s) did not significantly improve the results.

The attacker's board running the custom implementation of OpenThread generated a trigger signal on an output port each time an injected message was sent. This signal was used by the oscilloscope to record the EM emissions. The injected messages and the corresponding EM traces acquired by the oscilloscope were saved on a personal computer. The target board ran a genuine implementation of OpenThread and acted as a Router in our Thread network. We stress that no dedicated trigger signal was provided from the target board. We chose to power the target board from a regulated power supply rather than from a PC USB port to reduce the noise. These settings accurately replicate a real usage scenario of a Thread device. Admittedly, the network traffic may not be similar to the one of a real Thread network with very active data transfers.

8.5.2 Alignment of the Electromagnetic Traces

Once the side-channel acquisition is over, the attacker has to align the EM traces. The timing of different events that occur during the acquisition of a trace are shown in Figure 8.5. The injected message is sent at t_0 and the oscilloscope is triggered. The oscilloscope records the sampled signal between δ_{min} and $\tau = \delta_{max} + t_{AES}$, while the relevant part for the attacker t_{AES} is between t_1 and t_2.

The attacker has to determine experimentally the interval in which the relevant part of the AES execution is very likely to start at $\delta \in [\delta_{min}, \delta_{max}]$. Besides, she can

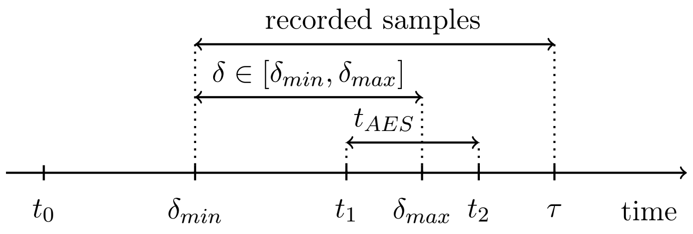


Figure 8.5: Timing of various events that occur during the acquisition of an EM trace.

record a sufficiently long interval that includes the relevant samples for most of the measurements. For our experiments, the value of δ was in the interval $[555, 564]$ μs, while the relevant part of the AES execution t_{AES} took 14.656 μs.

As illustrated by Figure 8.5, the relevant AES execution occurs at different locations in the recorded traces. For the CPA attack to work, the attacker has to extract and align the relevant samples from the recorded traces. To this end, the attacker builds a pattern consisting of interesting samples by precisely identifying the relevant computations within a trace. We were able to visually identify the first round of the AES in the first trace and thus we used this part as the alignment pattern.

We explored two methods for alignment of the EM traces: Sum of Absolute Differences (SAD) and Cross-Correlation (CC). For the sake of accuracy we quantified the precision of the alignment for each of the two methods. We raised a signal when the relevant part of a trace starts and we compared the corresponding sample number to the determined sample number by the SAD and CC methods. The comparison showed that a difference between the two values of more than three samples occurs for less than 1% of the traces. Therefore, both methods are very efficient. We chose to use the SAD method because it was a little bit faster while discarding slightly less traces than the CC method.

8.5.3 Attack Results

The most difficult step of the attack was the side-channel acquisition. Indeed, we spent about half of the total time devoted to mount the full attack on improving the side-channel attack outcome. This was an iterative process, which required a good understanding of the EM leakage of the target and fine adjustments of the attack parameters. The type and position of the probe are crucial for the quality of the side-channel acquisition.

The number and quality of the EM traces required to recover the MLE key determine the cost of the full attack. Our experimental results showed that 10,000 EM traces are sufficient. We note that two key bytes were much more difficult to recover than the rest. We tried different side-channel techniques, including linear regression attacks [218], but we did not see an improvement. This behaviour is determined by the hardware characteristics of the target device, the clock frequency, and the sequence of instructions executed by the target device. It does not depend on the value of the attacked key byte, but on its location. Similar results were reported in the side-channel literature [219].

The acquisition of 10,000 EM traces took about three hours. Given the cost of the active attack, a passive attack scenario is rendered almost impossible. It is very likely that the temporary keys are changed before the attacker has observed enough executions for different input messages.

The last step of the attack appeared to be impossible in a recent version of the OpenThread implementation we experimented with (commit `11c1b49`), because the fragmentation of the `MLE Child ID Response` messages (and therefore their additional encryption with K_{MAC}) can not be avoided. We do not exclude the possibility to get non-fragmented answers from other stacks depending on the Parent implementation.

8.5.4 Improving the Attack

Although the attacker can control up to 12 bytes of the input as shown in Section 8.4.4, she might want to fix the two input bytes corresponding to the key bytes that are difficult to recover. This requires some understanding of the target's leakage, but it is by far easier to perform and much more reliable than a template attack. For example, the attacker can use a similar device running OpenThread to learn which key bytes are more difficult to attack and then adjust the variable input bytes accordingly.

To further optimize the attack, one can search an input configuration that minimizes the number of attacked bytes while considering the input constraints. The number of AES rounds that have to be attacked in order to recover the full key must be kept to the minimum value of three, since attacking more rounds requires more EM samples and thus increases the cost of the measurement equipment as well as the duration of the offline attack.

In our case, there are 45 out of the $2^{10} - 1$ possible input configurations that minimize the number of attacked bytes. Compared to the straightforward attack, the improved attack reduces the number of individual CPA attacks from 44 to 37, which results in a 16% improvement.

The novelty of the improved attack stems from the fact that the attacker can adjust her attack strategy depending on the leakage of the target by fixing some input bytes she can control. To the best of our knowledge, we are the first to use this attack techniques to improve the outcome of a CPA attack. The total number of traces required to recover the full key can be decreased by about 50%, to no more than 5,000 EM traces. Another advantage of this approach is that the attack may pass undetected when it uses fewer variable input bytes because the injected packets resemble normal network traffic. A detailed description of the technique used to improve this attack is provided in Chapter 7.

8.6 Feasibility and Limitations

The main question arising is perhaps the relative complexity of the attack and the realism of the setting where the attacker needs to be in a very close proximity of a Thread Router or REED with a digital oscilloscope. This is a reasonable question, especially in the setting where Thread brings IPv6 to the end nodes and opens up the remote attack surface. We are realistic to state that the attacks we outlined are currently beyond the reach of an average hacker familiar just with software and networking techniques (and absolutely not for "script kiddies"), and apply only in particular settings. Section 8.7 shows a formalized quantification of the attack effort common to the smart card world.

8.6.1 Equipment Cost

The need of specialized equipment (i.e. a digital oscilloscope, an EM probe, and if needed an amplifier) hinders fast widespread application of the attack. In our experiments, we have used a high-end digital oscilloscope because we just had one available. Our experiments suggest that perhaps the cheapest available side-channel analysis hardware, ChipWhisperer [252], would not be sufficient to succeed on most of the targets due to its relatively low sampling rate. However, low- to mid-range digital oscilloscopes such as the Picoscope [275] should be sufficient. Combined with the increased availability of tooling to perform the analysis part of the side-channel attack, starting from the software tooling and tutorials of [252] to higher-performance toolkits like [323] and [76], this makes us claim that the attack is well feasible. With side-channel techniques and expertise becoming more mainstream in the hacker community, the threat of such attacks increases.

8.6.2 Portability

Our attack is moderate in portability. Namely, on another target family (a different hardware or software implementation) the attacker would most likely need to tune the side-channel attack to that target in terms of probe position, alignment parameters, etc. Hence, she has to invest into the identification phase of the attack. Due to the physical nature of side-channel attacks, our complexity estimate is based on one particular implementation we analyzed. For other implementations the complexity may be lower or higher; the attack may require a less or more expensive oscilloscope; however, for an unprotected cryptographic implementation we expect the same order of magnitude in terms of the amount of traces (and therefore time).

8.6.3 Other Attacks

Though we considered in our analysis a side-channel capable adversary, we were not excluding attack paths that do not require the use of side-channel techniques and therefore specialized equipment. However, we did not discover any paths that do not require specialized equipment. They may still exist, though. We did not consider other implementation attacks such as fault injection attacks or timing

Factors	Identification		Exploitation	
	Rating	Score	Rating	Score
Elapsed time	more than 1 month	5	less than 1 day	3
Expertise	expert	5	proficient	2
Knowledge of TOE	public	0	public	0
Access to TOE	less than 10	0	less than 10	0
Equipment	specialized	3	specialized – standard	3
Open samples	public/not required	0	–	–
Sum		13		8
Total	21 (enhanced-basic)			

Table 8.3: Attack rating using an adaptation of the rating for smart cards from Joint Interpretation Library [330].

attacks. They may be applicable and there may be settings where they are a realistic threat, especially timing attacks that can be performed remotely without specialized equipment [3].

The advantage of our attack is that it would circumvent IP protocol protections such as firewalls, akin to the recent ZigBee worm [300], thus may serve as a more feasible entry-point to the system. A limitation of our attack is that it does not address the application layer security mechanisms that would normally be deployed on top of the Thread networking stack. However, such mechanisms are not addressed by Thread.

8.7 Quantification of the Attack Effort

There is no standard procedure to quantify the attacker's potential to perform an attack on an IoT device. Thus, we use an adaptation of the rating for smart cards from the Joint Interpretation Library [330] to rate our attack. The rating procedure interprets the Common Criteria methodology based on smart card evaluation experience gained by the industry. It is used in practice by testing laboratories to quantify the resistance of smart cards to various attacks, including protocol and side-channel attacks. The aforementioned procedure distinguishes two independent phases for an attack: identification and exploitation. The identification phase refers to the demonstration of the attack, while the exploitation phase considers the impact of the attack when all necessary tools are readily available from the identification phase.

Next, we briefly introduce the factors considered by the rating methodology. The *elapsed time* defines the time required by the attacker to mount the attack from the moment she has access to the target. The *expertise* reflects the knowledge the attacker should have to mount the attack. The *knowledge of the target of evaluation (TOE)* indicates the level of access to the specifications. In the case of our attack, although the access to the official specifications is restricted, relevant information can be inferred from the open-source implementation placed in the public domain.

The number of different devices on which the attacker needs to perform the attack is captured by the *access to TOE* factor. The technical resources required for the attack are comprised in the *equipment* factor. Finally, the *open samples* factor measures to which extent the attacker is able to modify the software running on the target device. For more details, we refer the reader to [330].

The individual scores for each factor are given in Table 8.3 for an implementation where the last step of the attack is feasible. The final score of 21 classifies our attack as an enhanced-basic attack. The rating places our attack in between basic attacks that are easy to perform and enhanced attacks that require an advanced effort.

8.8 Additional Attack Paths

8.8.1 Attack on Loading the Security Material

Template attacks are powerful side-channel attacks that can recover sensitive values using very few traces. Thus, an attacker can purchase a device similar to the one to be attacked to create EM profiles. Then, she can force the targeted Thread device to reset such that she can observe the EM emanations corresponding to the loading of the network parameters from non-volatile memory. In particular, the attacker is interested to capture the loading of the network master key MK and commissioning key CK. Though powerful, template attacks depend on the quality of the templates made in the profiling phase. Thus, we did not investigate this attack vector further.

8.8.2 Elliptic Curve Implementations

The execution of elliptic curve computations might be vulnerable to timing [198] or Simple Power Analysis (SPA) [200] attacks if not properly implemented. The OpenThread implementation of the Thread networking stack relies on mbed TLS [18] for cryptographic services. Recently, Dugardin *et al.* showed that the point blinding countermeasure must be activated in mbed TLS for elliptic curve implementations to prevent horizontal and vertical template attacks [113]. We did not investigate into this direction further.

8.9 Countermeasures

Although it is desirable to achieve a defense in depth for a Thread network by employing redundant security mechanisms, other factors such as manufacturing costs or usability pose major constraints. Thus, a trade-off between these contradicting requirements should be sought to ensure an appropriate level of security. Though inspired by our case study of Thread, the countermeasures laid out next are applicable to other IoT protocols and devices as well. They are valuable for both protocol designers and engineers of IoT products.

8.9.1 Tamper Resistance

We suggest the use of shielded and tamper resistant components and cases. A trade-off between cost and product dimensions would be to insert small air gaps between the circuitry and the external case. This would most likely require device disassembly to enable EM analysis, making the attack more cumbersome to perform.

8.9.2 Protected Cryptographic Implementations

We stress that in scenarios where side-channel attacks pose a threat, Thread implementations should employ side-channel protection mechanisms for the manipulation of the security material. Consequently, the loading of the security material as well as all cryptographic algorithm implementations should use countermeasures, such as masking and hiding [223]. If the cost of the countermeasures is prohibitive, offloading the cryptographic algorithms to hardware cryptographic engines might be a good trade-off. In general, it is harder (but still feasible) to attack a hardware implementation than a software implementation. Protected hardware implementations should be considered where both high security and high performance are necessary.

8.9.3 Fresh Re-keying

When the cost of protecting the cryptographic implementations of a block cipher is too high, fresh re-keying schemes can be used to prevent side-channel attacks [236, 235, 110]. These schemes make use of a re-keying function to generate new session keys based on the secret master key and random nonces for every block of message to be encrypted. Although fresh re-keying has the benefit that the re-keying function can be protected against side-channel attacks at a much lower cost than the block cipher [109], it involves significant changes in the protocol.

8.9.4 Protocol-level Mitigations

We suggest to consider ways to mitigate the presented attack paths at the protocol level, changing the network mechanisms that facilitate the attacks. Most importantly, disabling or limiting the message exchange that allows a node to get the network master key by sending an `MLE Child ID Request` message to its Parent should be considered. Rate limiting the incoming `MLE Parent Request` messages processed by a Parent (Router or REED) significantly slows the attacker, but care should be taken not to expose the network to DoS attacks. A more complex solution would be to design a mechanism for tracking valid commissioned devices in a Thread network. Such a mechanism would have the benefit that it allows a Router to treat incoming messages differently depending on the node status. This is a rough idea that needs further investigation.

8.9.5 Security Certification Scheme

We recommend enforcing a security certification scheme for Thread products in addition to the functional certification scheme currently in place. Although a certification scheme can not prevent new attacks, the benefits for the security of the whole ecosystems are obvious. A security certification seal increases consumer awareness of possible security issues and attests resistance to known attacks. The major drawbacks are an increase of the overall price and an additional delay before the certified products are available on the market.

8.10 Summary

We conducted the first electromagnetic side-channel vulnerability analysis of the Thread networking stack. We described how different network mechanisms that can be learned by the attacker from the published OpenThread code can be used to create attack vectors for side-channel attacks. We showed that some of the side-channel attack paths are hard or impossible to exploit in the context of Thread. We implemented the most promising attack path that provides complete access to the Thread network. It exploits a chain of network mechanism and side-channel attacks on executions of unprotected implementations of cryptographic algorithms.

The full attack did not succeed in an experimental network of OpenThread nodes. We consider the setting where the last attack step is indirectly prevented by the `MLE Child ID Response` payload size to be insufficient to rely upon. Firstly, it is not in the design of the protocol that the master key is protected by both K_{MLE} and K_{MAC}. Additional protection by the K_{MAC} is a side effect (though, of course, a fortunate one). Secondly, a possibility to request the master key having the derived key(s) is questionable security-wise as it subverts the essence of key derivation using HMAC. Hence, we suppose that the full attack may succeed with moderate effort in other implementations of the stack.

The possibility of an arbitrary Thread device to trigger cryptographic operations and responses from a commissioned Thread device at unlimited rate presents a standalone risk of a denial-of-service attack.

The lessons learned from our work can be applied to other IoT systems and protocols as well. Our threat model can be used to better shape the attack surface of future IoT products and prevent issues such as: processing of invalid injected messages, EM leakage, converting temporary keys into master key, and using a single network master key to secure the whole network. In light of our results, designers of future protocols for the IoT should carefully consider the threat of side-channel attacks from the early inception.

In general, we demonstrated that in the context of a modern IoT network protocol mounting a side-channel attack is not trivial. Similar to a modern software exploit, it requires chaining multiple vulnerabilities. Nevertheless, such attacks are feasible. Being perhaps too expensive for settings like smart homes, they pose a relatively higher threat to commercial settings.

Part III

Side-Channel Countermeasures

Chapter 9

Optimal First-Order Boolean Masking

Contents

9.1 Introduction

The Internet of Things (IoT) is one of the technical revolutions of our time, with vast amounts of IoT devices being deployed every day to create a global network of smart objects. According to Gartner, 8.4 billion connected things will be in use worldwide by the end of 2017 [136]. From 2018 onwards, Gartner forecasts that devices to be used in smart buildings (LED lighting, HVAC, and physical security systems) will have the biggest market share [136]. In light of the very recent security vulnerabilities [84, 300] discovered in such devices, immediate action is required to prevent large-scale security incidents similar to the Mirai botnet [14].

The attack surface of IoT devices is considerably larger than the attack surface of classical Internet-connected systems due to the various use cases these gadgets, sensors, and actuators are built for. Most of the IoT systems are characterized by low physical security, with devices being deployed in easily accessible places. As a consequence, attack vectors that exploit these weaknesses came to light. Side-channel attacks, such as EM and power analysis attacks, fall into this category of attack vectors that require physical proximity to the target system. If the target system uses an unprotected implementation of a cryptographic algorithm, the adversary can determine the secret key used by the system from the leakage generated during the execution of the algorithm. Hence, countermeasures against side-channel attacks are mandatory for the security of IoT devices.

There are two main requirements an implementation of a cryptographic algorithm to be deployed in the IoT has to satisfy. On the one hand, the implementation must be *lightweight* (i.e. consume few resources) considering the limited computational resources of embedded devices for the IoT. On the other hand, the implementation must be *secure against side-channel attacks* given the attack surface specific to the IoT. Most implementations of the existing lightweight ciphers do not satisfy the second requirement, either because the ciphers were not designed to facilitate the integration of countermeasures, or because the best existing countermeasures add significant performance penalties to the unprotected implementations of the ciphers. Therefore, there is a need for more efficient countermeasures, especially DPA countermeasures like masking. Any improvement of the existing masking schemes brings us closer to the ultimate goal of a secure IoT.

9.1.1 Boolean Masking

Conceptually, Boolean masking of a block cipher is done by replacing each unprotected operation by its masked counterpart. The most common operations used by lightweight block ciphers are logical operations (NOT, AND, OR, XOR), rotations, and modular addition/subtraction. Masked NOT is equivalent to the negation of a single share, while masked XOR and rotations can be realized by simply applying the operation to each pair of shares independently.

To our knowledge, the best known expression for first-order masking of bitwise AND is based on the Trichina AND gate [358]. The same expression of the masked AND was latter used by Coron *et al.* in their algorithm for masked addition on Boolean shares [85]. The sequence of operations used to compute AND on Boolean shares (i.e. SecAnd) is provably secure thanks to a random value that is mixed with the input shares. While the expression of masked AND initially proposed by Trichina can be easily found in the literature, there is almost no reference to a similar expression for masked OR. Hence, one might try to derive such an expression by applying De Morgan's laws to the masked AND expression. The resulting expression requires

Secure operation	Cost	Randoms
SecNot	1	0
SecXor	2	0
SecShift	4	1
SecShiftFill	6	1
SecAnd	8	1
SecOr	8	1
SecAdd	$28 \cdot n + 4$	2
SecSub	$41 \cdot n + 4$	2

Table 9.1: The cost (in number of elementary operations) and the number of randoms of different secure operations; $n = max\big(\lceil \log_2(k-1)\rceil, 1\big)$, where k is the size of the shares in bits.

eleven elementary operations and a random value. However, Baek and Noh [25] gave a better expression for a masked OR gate that requires only six elementary operations and no random value. They also described how to compute an AND gate using eight elementary operations and no random value. Their AND gate can be seen as a version of the Trichina AND gate in which the random value is replaced by one of the input shares. The best known algorithm for secure addition on Boolean shares is based on the Kogge-Stone adder [85]. This algorithm is provably secure because it uses provably secure operations and all its intermediates are uniformly distributed.

In this chapter, we consider the Trichina AND gate [358] to be the reference expression for performing a bitwise AND on Boolean shares. We consider the OR gate proposed by Baek and Noh [25] with an injected random value as the reference expression for bitwise OR. We selected these two expressions because they are provably secure and hence they facilitate security proofs for more complex algorithms such as secure addition on Boolean shares [85].

We divide the secure operations on Boolean shares into three classes according to their computational cost. The first class includes all secure operations with a cost of at most six instructions (e.g. SecXor, SecShift). Then, the second class contains operations that can be masked using up to a dozen instructions (e.g. SecAnd, SecOr). Finally, the third class is represented by operations that need more than 12 instructions. Secure algorithms for modular addition/subtraction on Boolean shares (SecAdd, SecSub) belong to this latter class since they rely on secure operations from the first two classes.

A detailed description of SecXor, SecShift, SecAnd, and SecAdd can be found in [85, 367]. SecShiftFill shifts a sensitive value represented by Boolean shares n bits to the left and sets the $n-1$ least significant bits to 1. We briefly describe the algorithms for secure addition (SecAdd) and subtraction (SecSub) on Boolean shares in Section 9.3.

In this chapter, we study the efficiency of Boolean masking for embedded IoT devices. Although our work is not limited to a specific microprocessor architecture, we evaluate our implementations on a 32-bit ARM Cortex-M3 since these microcontrollers are widely used for IoT applications [284].

9.1.2 Contributions

Firstly, we present an algorithm for efficient search of Boolean masking expressions (Section 9.2). Thanks to several algorithmic optimizations, the search is very fast. As a second contribution, we propose concrete expressions for Boolean masking of the AND and OR operations (Section 9.2.7). We describe an expression of bitwise AND on Boolean shares using fewer operations than the best known expression in the literature. At the same time, our expression for secure AND does not require any randomness. Thirdly, we improve the Kogge-Stone algorithm for addition/subtraction on Boolean shares [85] by using our masking expressions and by processing the shares in a clever way that does not require any randomness (Section 9.3.1). When implemented on an ARM Cortex-M3 processor (Section 9.4.1), the addition/subtraction of 32-bit values using the new algorithm is between 18% and 25% faster than similar implementations using the original algorithm [85]. Finally, we use our expressions for Boolean masking of AND and OR to develop first-order masked implementations of three lightweight block ciphers, namely Simon, Speck, and RECTANGLE (Section 9.4.2). By comparing the performance figures of the masked and unmasked implementations of the three ciphers, we learn which design strategies facilitate efficient masked implementations.

9.2 Search Algorithm

9.2.1 Description

Our search algorithm (Algorithm 6) uses a breadth-first approach to determine the optimal masked representations of a given Boolean function. Its inputs are the number of shares, the target function that has to be masked, a set of sensitive functions that leak some information about the sensitive values, and a set of operations that can be used to build new expressions. The search algorithm returns a set of pairs, where each pair describes how to compute the shares of the target function in a way that does not leak. For instance, we used the following input parameters to search first-order masked expressions of bitwise AND:

- **Number of shares:** $n = 2$. Hence, the two input sensitive variables x and y are split into two shares each such that $x = x_1 \oplus x_2$ and $y = y_1 \oplus y_2$.
- **Target function:** $t(x_1, x_2, y_1, y_2) = (x_1 \oplus x_2) \wedge (y_1 \oplus y_2)$. The output of the search algorithm is a set of pairs (z_1, z_2) such that $z_1 \oplus z_2 = t(x_1, x_2, y_1, y_2)$.
- **Sensitive functions:** $S = \{s_0, s_1, s_0 \wedge s_1, \neg s_0 \wedge s_1, s_0 \wedge \neg s_1, \neg s_0 \wedge \neg s_1\}$, where $s_0 = x_1 \oplus x_2$ and $s_1 = y_1 \oplus y_2$. Each sensitive function $s \in S$ leaks some information about the values that have to be masked.

- **Operations:** $O = \{\neg, \oplus, \wedge, \vee\}$. The algorithm builds expressions using operators from the set O.

The search algorithm explores sequences of expressions (or terms) until some terms of a sequence can be used to compute the target function. The term of a sequence which requires the highest number of operations determines the cost of that sequence. The search starts with an empty sequence, which is extended using the input shares and operators into sequences of cost 1. In subsequent steps, all sequences of cost k ($k \geq 1$) are extended into sequences of cost $k + 1$. A sequence is extended using its previous terms, the input shares, and the input set of operators.

An expression is identified by its truth table, which is stored as a stream of bits for efficiency reasons. The main role of a sequence is to avoid computing several times the repeating terms of an expression. Hence, the search algorithm is faster, but the memory requirement increases. The search space is greatly reduced by three cut-off conditions:

- **Leakage test.** A sequence is extended only if the new expression e to be added to that sequence does not leak information about the input sensitive functions $s \in S$. In other words, the following relation must hold to consider the extended sequence:

$$\frac{\mathsf{HW}(s \wedge e)}{\mathsf{HW}(e)} = \frac{\mathsf{HW}(s \wedge \neg e)}{\mathsf{HW}(\neg e)},$$

  where $\mathsf{HW}(f)$ denotes the Hamming weight of the truth table of function f. In addition to this leakage test, which is a fast initial filter, the leakage of the expressions returned by the search algorithm is assessed using Welch's t-test.

- **Ignoring the order of operations.** If two or more sequences are equal (i.e. they compute the same intermediate expressions), the algorithm explores only the sequence that is reached first.

- **Exploiting the symmetries of shares.** The input and output shares of a symmetric operation (e.g. AND) can be swapped without affecting the result. This property allows the search algorithm to explore just one representative sequence for all equivalent sequences.

9.2.2 Optimality

We stress that the algorithm is designed to find *optimal* expressions. Any optimal expression can be computed using at least one non-leaking sequence of operations. The search algorithm explores all possible sequences, except for those that are discarded by the cut-off conditions. The first cut-off condition reduces the search to non-leaking sequences, while the other two conditions limit the exploration to a single representative per equivalence class of sequences. This representative has minimum cost thanks to the breadth-first nature of the algorithm. Hence, the algorithm returns pairs of optimal expressions that can be used to securely compute the target function.

Algorithm 6 Searching for optimal expression

```
Input:
    number of input shares: $n$                    ▷ Input shares: $x_i$ and $y_i$ for $1 \leq i \leq n$
    target function: $t : \mathbb{F}_2^{2n} \to \mathbb{F}_2$
    set of sensitive functions: $S = \{s_i\}, s_i : \mathbb{F}_2^{2n} \to \mathbb{F}_2$
    set of operations: $O = \{o_i\}, o_i : \mathbb{F}_2^2 \to \mathbb{F}_2$
Output:
    set of $n$ functions $Z = \{z_i\}, z_i : \mathbb{F}_2^{2n} \to \mathbb{F}_2$ such that $z_1 \oplus z_2 \oplus \ldots \oplus z_n = t$
 1: $opt\_cost \leftarrow -1$
 2: $seq_0 \leftarrow \emptyset$
 3: $cost \leftarrow 0$
 4: while $cost \neq opt\_cost$ do
 5:     $seq_{cost} \leftarrow \emptyset$
 6:     $cost \leftarrow cost + 1$
 7:     for all $seq \in seq_{cost-1}$ do
 8:         $seq' \leftarrow$ EXTEND$(S, seq)$
 9:         $seq_{cost} \leftarrow seq_{cost} \cup seq'$
10:         if VALIDEXPRESSION$(t, seq')$ then
11:             $opt\_cost \leftarrow cost$
12:             yield $seq'$
13:         end if
14:     end for
15: end while
16: function EXTEND$(S, seq)$
17:     for all $a, b \in seq \cup \{x_1, x_2, \ldots, x_n, y_1, y_2, \ldots y_n\}$ do
18:         for all $o \in O$ do
19:             $seq' \leftarrow o(a, b)$
20:             if EXPLORE$(S, seq')$ then              ▷ Check the cut-off conditions
21:                 yield $seq'$
22:             end if
23:         end for
24:     end for
25: end function
26: function VALIDEXPRESSION$(t, seq)$
27:     for all $z_1, z_2, \ldots, z_n \in seq$ do
28:         if $t == z_1 \oplus z_2 \oplus \ldots \oplus z_n$ then
29:             return $(z_1, z_2, \ldots, z_n)$
30:         end if
31:     end for
32: end function
```

9.2.3 Instruction Set Architecture (ISA)

We distinguish between two classes of IoT devices depending on the operations supported by the instruction set architecture (ISA): *basic* and *enhanced* devices. Most IoT devices have instructions only for the following bitwise logical operations: NOT, AND, OR, and XOR. We call these architectures *basic* ISAs. In addition to these operations, the *enhanced* ISAs have dedicated instructions for other bitwise logical operations, such as AND NOT or OR NOT. For example, the instruction set of ARM Cortex-M3 includes the `bic` (AND NOT) and `orn` (OR NOT) instructions that perform two basic bitwise logical operations in a single clock cycle instead of two clock cycles. Most microcontrollers execute all logical instructions in a single clock cycle.

9.2.4 Leakage Model

The power consumption of most microcontrollers is proportional to the number of bits that are set in the processed sensitive value [223]. Therefore, the Hamming weight power model is a reliable method for modeling the leakage of a sensitive variable. In addition to the bit-level leakage verification performed by the search algorithm, we performed a t-test leakage assessment [148] for each valid expression returned by the algorithm to confirm the absence of any leakage. We used fixed-vs.-random t-test evaluations and two leakage models: Hamming weight and Hamming distance.

9.2.5 Extension to Higher-Order Masking

Our algorithm can naturally be extended to search expressions for higher-order masking. However, further optimizations are required to ensure that the algorithm scales well for a higher number of shares. The main limiting factor of our algorithm is the amount memory required to store the valid sequence of expressions, although both computational and memory complexity increase with the number of shares (i.e. masking order).

9.2.6 Other Improvements

Our search algorithm might benefit from the approach proposed by Groß [152] to tackle a similar problem. Namely, instead of searching for an optimal sequence of instructions, one can try all combinations of truth tables that give a target function and then convert them to a circuit using a tool such as Logic Friday [217].

9.2.7 Results

The optimal expressions for masked SecOr use 6 instructions on both platforms, while the optimal expressions for SecAnd have a cost of 7 on a basic device and 6 on ARM. The expressions for SecOr and SecAnd using basic instructions are unique up to symmetries of the shares, whereas for ARM there are 48 different optimal expressions for SecAnd and 50 different optimal expressions for SecOr. The unique optimal expressions for a basic architecture are actually included in the optimal

Source	Operation	Expression	Rand	Cost	
				Basic	ARM
reference	SecAnd	$z_1 = r$ $z_2 = z_1 \oplus (x_1 \wedge y_1) \oplus (x_1 \wedge y_2) \oplus (x_2 \wedge y_1) \oplus (x_2 \wedge y_2)$	1	8	8
	SecOr	$z_1 = r$ $z_2 = z_1 \oplus (x_1 \vee y_1) \oplus (x_1 \wedge y_2) \oplus (x_2 \vee y_2) \oplus (x_2 \wedge y_1)$	1	8	8
our	SecAnd	$z_1 = (x_1 \wedge y_1) \oplus (x_1 \vee \neg y_2)$ $z_2 = (x_2 \wedge y_1) \oplus (x_2 \vee \neg y_2)$	0	7	6
	SecOr	$z_1 = (x_1 \wedge y_1) \oplus (x_1 \vee y_2)$ $z_2 = (x_2 \vee y_1) \oplus (x_2 \wedge y_2)$	0	6	6

Table 9.2: Expressions, number of randoms ("Rand") and number of operations ("Cost") for different secure operations. Basic cost gives the number of elementary operations, while the ARM cost gives the number of instructions. Expressions in parentheses have priority and operations are executed from left to right.

expressions for the ARM architecture, which makes them universal. A comparison of these two expressions with the reference expressions is given in Table 9.2. Our results show that the expression of the masked OR gate proposed by Baek and Noh [25] is optimal. On the other hand, our expression for bitwise AND uses less instructions than the masked AND gate of Baek and Noh [25] and the Trichina AND gate [358]. Besides using less operations than the reference expressions, our optimal expressions do not require a random value. Thanks to these two properties, our expressions have a significant performance advantage over the reference ones.

9.3 Applications

9.3.1 Modular Addition and Subtraction

Coron *et al.* [85] proposed a logarithmic-time algorithm for modular addition on Boolean shares based on the Kogge-Stone adder. Their algorithm for modular addition uses the following three secure operations: SecAnd, SecXor, and SecShift. The expression of SecAnd uses 8 elementary operations, the one of SecXor needs 2 elementary operations, while SecShift can be performed using 4 elementary operations. Algorithms for all these operations are presented in [85, 367]. Although not described in the original paper [85], the algorithm for modular subtraction can be obtained from the algorithm for modular addition on Boolean shares by making several changes. Namely, the SecShift operations from lines 7 and 15 of [85, Algorithm 6] have to be replaced by SecShiftFill (secure operation for shift to the left by n bits followed by OR of $2^n - 1$). Similarly, SecXor operations from lines 9 and 17 of [85, Algorithm 6] must be replaced by SecOr. These changes affect the performance of the modular subtraction algorithm since operations with a lower cost are replaced by operations with a higher cost.

One can improve the algorithms for modular addition/subtraction based on the Kogge-Stone adder by simply replacing the original expressions for SecAnd and SecOr with our optimal expressions. Yet, the algorithm can be improved further by replacing the expression of the SecShift operation, which requires a random variable, by a more efficient expression that does not require any randomness. Hence, the new versions of the algorithm do not require any randomness at all. The improved algorithm for addition on Boolean shares is described in Algorithm 7, while the analogous algorithm for subtraction is presented in Algorithm 8. It is important to note that lines 3, 10, and 12 of Algorithm 7 are required to prevent composition of operations that otherwise will leak. Similarly, lines 4, 10, 12, 14, and 19 of Algorithm 8 avoid composing operations that leak.

Algorithm 7 Improved Kogge-Stone masked addition

```
Input: $x_1, x_2, y_1, y_2 \in \{0,1\}^k$ such that $x = x_1 \oplus x_2$ and $y = y_1 \oplus y_2$
Output: $z_1, z_2$ such that $z = z_1 \oplus z_2 = (x + y) \bmod 2^k$
 1: $p_1, p_2 \leftarrow \mathsf{SecXor}(x_1, x_2, y_1, y_2)$
 2: $g_1, g_2 \leftarrow \mathsf{SecAnd}(x_1, x_2, y_1, y_2)$
 3: $g_1 \leftarrow (g_1 \oplus x_2) \oplus g_2$                          ▷ $g_2 = x_2$
 4: $n \leftarrow \max\big(\lceil \log_2(k-1) \rceil, 1\big)$
 5: for $i := 1$ to $n - 1$ do
 6:     $h_1, h_2 \leftarrow \mathsf{SecShift}(g_1, g_2, 2^{i-1})$
 7:     $u_1, u_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
 8:     $g_1, g_2 \leftarrow \mathsf{SecXor}(g_1, g_2, u_1, u_2)$
 9:     $h_1, h_2 \leftarrow \mathsf{SecShift}(p_1, p_2, 2^{i-1})$
10:     $h_1 \leftarrow (h_1 \oplus x_2) \oplus h_2$                      ▷ $h_2 = x_2$
11:     $p_1, p_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
12:     $p_1 \leftarrow (p_1 \oplus y_2) \oplus p_2$                      ▷ $p_2 = y_2$
13: end for
14: $h_1, h_2 \leftarrow \mathsf{SecShift}(g_1, g_2, 2^{n-1})$
15: $u_1, u_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
16: $g_1, g_2 \leftarrow \mathsf{SecXor}(g_1, g_2, u_1, u_2)$
17: $z_1, z_2 \leftarrow \mathsf{SecXor}(y_1, y_2, x_1, x_2)$
18: $z_1 \leftarrow \big(z_1 \oplus (g_1 \ll 1)\big) \oplus (x_2 \ll 1)$        ▷ $z_2 = y_2$
```

9.3.1.1 Masking Cost

A comparison between the cost of the secure expressions used by the original version of the algorithm and the new expressions used by the improved version of the algorithm is provided in Table 9.3. Based on these values, one can compute the total cost of these algorithms for different architectures and make an estimation of their performance for different values of the operand size k.

The original version of the algorithm for modular addition on Boolean shares requires 2 SecShift operations, 2 SecAnd operations, 1 SecXor operation and 2 other

Algorithm 8 Improved Kogge-Stone masked subtraction

```
Input: $x_1, x_2, y_1, y_2 \in \{0,1\}^k$ such that $x = x_1 \oplus x_2$ and $y = y_1 \oplus y_2$
Output: $z_1, z_2$ such that $z = z_1 \oplus z_2 = (x - y) \bmod 2^k$
 1: $y_1, y_2 \leftarrow \mathsf{SecNot}(y_1, y_2)$
 2: $p_1, p_2 \leftarrow \mathsf{SecXor}(y_1, y_2, x_1, x_2)$
 3: $g_1, g_2 \leftarrow \mathsf{SecAnd}(x_1, x_2, y_1, y_2)$
 4: $g_1 \leftarrow (g_1 \oplus x_2) \oplus g_2$                    $\triangleright\ g_2 = x_2$
 5: $n \leftarrow \max\left(\lceil \log_2(k-1) \rceil, 1\right)$
 6: for $i := 1$ to $n - 1$ do
 7:     $h_1, h_2 \leftarrow \mathsf{SecShiftFill}(g_1, g_2, 2^{i-1})$
 8:     $u_1, u_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
 9:     $g_1, g_2 \leftarrow \mathsf{SecOr}(g_1, g_2, u_1, u_2)$
10:     $g_1 \leftarrow (g_1 \oplus x_2) \oplus g_2$                $\triangleright\ g_2 = x_2$
11:     $h_1, h_2 \leftarrow \mathsf{SecShift}(p_1, p_2, 2^{i-1})$
12:     $h_1 \leftarrow (h_1 \oplus x_2) \oplus h_2$                $\triangleright\ h_2 = x_2$
13:     $p_1, p_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
14:     $p_1 \leftarrow (p_1 \oplus y_2) \oplus p_2$                $\triangleright\ p_2 = y_2$
15: end for
16: $h_1, h_2 \leftarrow \mathsf{SecShiftFill}(g_1, g_2, 2^{n-1})$
17: $u_1, u_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
18: $g_1, g_2 \leftarrow \mathsf{SecOr}(g_1, g_2, u_1, u_2)$
19: $g_1 \leftarrow (g_1 \oplus x_2) \oplus g_2$                    $\triangleright\ g_2 = x_2$
20: $z_1, z_2 \leftarrow \mathsf{SecXor}(y_1, y_2, x_1, x_2)$
21: $z_1 \leftarrow \Big(z_1 \oplus \big((g_1 \ll 1) \vee 1\big)\Big) \oplus (x_2 \ll 1)$    $\triangleright\ z_2 = y_2$
```

elementary operations in the main loop, hence $28 \cdot (\log_2 k - 1)$ elementary operations. Outside the main loop, it requires 1 SecShift operation, 2 SecAnd operations, 3 SecXor operations, and 4 other elementary operations. Therefore, the total cost is $28 \cdot \log_2 k + 4$ for both basic and enhanced architectures.

The cost of the improved algorithm for addition on Boolean shares can be computed similarly. The main loop consists of 2 SecShit operations, 2 SecAnd operations, 1 SecXor operation, and 4 elementary operations. The rest of the algorithm uses 1 SecShift operation, 2 SecAnd operations, 3 SecXor operations, and 6 elementary operations. In other words, the cost of the improved algorithm for modular addition on Boolean shares is $22 \cdot \log_2 k + 4$ on ARM and $22 \cdot \log_2 k + 6$ on basic architectures.

In the same way, one can compute the cost of the original and improved algorithms for modular subtraction on Boolean shares. All these values are summarized in Table 9.4 alongside the gain of the improved algorithms over the original ones for common values of the operand size k. We see that the improved algorithms outperform the original algorithms on both platforms by at least $6 \cdot \log_2 k$ elementary operations, where k is the operand size.

Platform	Source	Cost					
		SecNot	SecXor	SecAnd	SecOr	SecShift	SecShiftFill
Basic	reference	1	2	8	8	4	6
	our	1	2	7	6	2	4
	Improvement	0	0	1	2	2	2
ARM	reference	1	2	8	8	4	6
	our	1	2	6	6	2	4
	Improvement	0	0	2	2	2	2

Table 9.3: Comparison of the number of instructions required to perform different secure operations.

9.3.1.2 Leakage Assessment

We evaluated the secure operations presented in this section, including the two improved algorithms for addition and subtraction on Boolean shares, against first-order attacks using Welch's t-test [148]. Welch's t-test is a fast and robust way to verify the soundness of a masking scheme [103, 308]. To determine if there is any leakage in our first-order implementations, we used a simple tool similar to the ones described in [233, 271, 291]. Firstly, we validated the correctness of our tool by performing evaluations against a set of masking schemes known to be either secure or broken. Then, we carefully applied the t-test to avoid false negatives [333]. All our secure implementations passed a set of fixed-vs.-random evaluations with up to 10^6 traces using both Hamming weight and Hamming distance models for the simulated leakage.

9.3.2 Other Applications

The optimal expressions for secure computation of AND and OR can be used to mask more complex structures such as S-boxes. They can also be used to efficiently mask ciphers that use only logical bitwise operations such as SIMON [38], as well as bit-sliced designs such as NOEKEON [97], RECTANGLE [391], or RoadRunneR [35]. In Section 9.4, we evaluate how these expressions can be applied to unprotected implementations of several lightweight block ciphers and we determine the performance penalty of the resulting first-order protected implementations.

9.4 Implementations

In this section we describe our efficient implementations of several first-order secure algorithms and block ciphers. All our implementations are written in assembly language for a Cortex-M3 processor for two reasons. Firstly, we wanted to avoid accidental leakages introduced by the transformations made by the GCC compiler which is not optimized for masked implementations, but only for efficiency [28]. On the other hand, when coding in assembly language, the implementer has full

Operation	Platform	Expressions	Rand	Operand size				
				k	8	16	32	64
SecAdd	Basic	reference	2	$28 \cdot \log_2 k + 4$	88	116	144	172
		our	0	$22 \cdot \log_2 k + 6$	72	94	116	138
		Improvement	2	$6 \cdot \log_2 k - 2$	16	22	28	34
	ARM	reference	2	$28 \cdot \log_2 k + 4$	88	116	144	172
		our	0	$22 \cdot \log_2 k + 4$	70	92	114	136
		Improvement	2	$6 \cdot \log_2 k$	18	24	30	36
SecSub	Basic	reference	2	$38 \cdot \log_2 k + 4$	118	156	194	232
		our	0	$32 \cdot \log_2 k + 6$	102	134	166	198
		Improvement	2	$6 \cdot \log_2 k - 2$	16	22	28	34
	ARM	reference	2	$38 \cdot \log_2 k + 4$	118	156	194	232
		our	0	$30 \cdot \log_2 k + 6$	96	126	156	186
		Improvement	2	$8 \cdot \log_2 k - 2$	22	30	38	46

Table 9.4: Cost and random numbers ("Rand") required for Kogge-Stone addition/subtraction on Boolean shares for different values of the operand size k. Basic cost gives the number of elementary operations, while the ARM cost gives the number of instructions.

control of the register allocation and the sequence of instructions executed by the microcontroller. Hence, she can avoid combining instructions and registers in a way that leaks [28, 271]. Secondly, we wanted to get a clear picture of the performance figures of our implementations in order to conduct a fair comparison of the first-order implementations. Hence, the effort spent by a programmer on a more demanding assembly implementation is paid off in the end by a better (i.e. more secure and efficient) implementation.

In line with previous work, we do not include the cost of random number generation for the implementations that need randomness since the cost of random number generation is different from one device to the other and we want a device-independent comparison.

9.4.1 Masked Addition

We implemented the original algorithms for addition and subtraction on Boolean shares as well as the improved algorithms presented in this chapter. For each algorithm we wrote a straightforward implementation and an implementation that unrolls the main loop of the Kogge-Stone adder. The execution time and code size of our implementations are given in Table 9.5.

The improved algorithms are between 18% and 25% faster than the original algorithms. At the same time, the code size of the improved algorithms is between 22% and 37% smaller than the code size of the original ones. Unlike the original algorithms, which require two random values, the improved algorithms do not require

Impl.	Expressions	Rand	Time (cycles)		Code size (bytes)	
			Addition	Subtraction	Addition	Subtraction
rolled	reference	2	336	452	380	492
	our	0	252	372	252	380
	Improvement	2	84	80	128	112
	%		25%	17.69%	33.68%	22.76%
unrolled	reference	2	274	359	764	1048
	our	0	205	281	584	816
	Improvement	2	69	78	180	232
	%		25.18%	21.72%	23.56%	22.13%

Table 9.5: Execution time and code size for secure addition and subtraction on Boolean shares using the Kogge-Stone adder.

any random value. The generation of a 32-bit random number takes between 37 cycles for a XorShift RNG [225] and 85 cycles for the built-in TRNG [16]. Hence, the improved algorithms for addition and subtraction on Boolean shares outperform the original algorithms in all categories: execution time, code size, and required randomness.

9.4.2 Lightweight Block Ciphers

We selected the top-3 block ciphers that use a 64-bit block from the performance evaluation conducted using the FELICS benchmarking framework [105] and we protected them against first-order attacks using the best known algorithms for secure operations on Boolean shares as well as the ones introduced in this chapter. Besides their very lightweight software implementations, these three ciphers (Speck, Simon, and RECTANGLE) have different design strategies. Hence, they facilitate an analysis of the relationship between their design strategies and the performance figures of their masked implementations.

9.4.2.1 Speck

Speck [38] is an ARX-based family of lightweight block ciphers designed for performance in software. Nevertheless, all ciphers of this family perform very well in hardware also. Speck-64/128 refers to the version of Speck characterized by a 64-bit block, a 128-bit key, and 27 rounds. The round function of Speck-64/128 uses only bitwise XOR, addition modulo 2^{32}, and rotations:

$$R_k(x, y) = \Big(\big((x \ggg 8) \boxplus y\big) \oplus k, (y \lll 3) \oplus \big((x \ggg 8) \boxplus y\big) \oplus k\Big),$$

where x and y are the two 32-bit branches of a Feistel network.

While the unprotected implementation of Speck requires only four registers in order to process the cipher's state, the protected implementations need all 13

Impl./Expr.	Rand	Time (cycles)		Code size (bytes)		Penalty factor	
		Enc	Dec	Enc	Dec	Enc	Dec
unprotected	0	318	530	44	52	1	1
rolled KSA/reference	2	8994	12018	428	564	28.28	22.67
rolled KSA/our	0	6583	9342	308	452	20.70	17.62
Improvement	2	2411	2676	120	112		
%		26.80%	22.26%	28.03%	19.85%		
unrolled KSA/reference	2	6890	9430	808	1108	21.66	17.79
unrolled KSA/our	0	5334	7305	612	844	16.77	13.78
Improvement	2	1556	2125	196	264		
%		22.58%	22.53%	24.25%	23.82%		

Table 9.6: Execution time, code size and performance penalty factor for different secure implementations of Speck-64/128. For each set of expressions (best known, our) we wrote two implementations that correspond to the two implementation strategies of the Kogge-Stone adder (KSA): rolled/unrolled KSA.

general-purpose registers of the Cortex-M3 microcontroller. Moreover, the rolled implementations have to save the content of a register on the stack at the beginning of the secure addition/subtraction. The initial value of this register is recovered at the end of the addition/subtraction operation. A pair of stack operations (i.e. `push` and `pop`) adds 4 cycles to the total execution time of the algorithm.

The implementations of Speck based on the improved algorithms for modular addition and subtraction on Boolean shares are faster and use less code space than the implementations of Speck based on the original versions of the same algorithms as can be seen in Table 9.6. The gain of the improved algorithms over the reference ones is at least 28% for both rolled and unrolled implementations. On the other hand, the improvement in code size varies between 20% and 28%.

9.4.2.2 Simon

Simon [38] is a family of lightweight block ciphers designed primarily for optimal performance in hardware, but its instances perform very good in software as well. The round function of Simon uses only bitwise XOR, bitwise AND, and rotations:

$$R_k(x, y) = \big(y \oplus f(x) \oplus k, x\big),$$

where $f(x) = (x \lll 1) \wedge (x \lll 8) \oplus (x \lll 2)$. Simon-64/128 is the instance of Simon that processes a 64-bit block using a 128-bit key in 44 rounds.

The two protected implementations of Simon are very efficient since the operations used by the cipher can be masked with a little impact on the execution time and code size. The most costly operation is secure bitwise AND which, depending on its expression, can be evaluated using 6 or 8 instructions. The other secure operations require only 2 instructions each. The unprotected implementation of

Impl./Expr.	Rand	Time (cycles)		Code size (bytes)		Penalty factor	
		Enc	Dec	Enc	Dec	Enc	Dec
unprotected	0	1068	1113	60	64	1	1
reference	1	1956	1904	160	164	1.83	1.71
our	0	1888	1670	140	144	1.76	1.50
Improvement	1	68	234	20	20		
%		3.47%	12.28%	12.5%	12.19%		

Table 9.7: Execution time, code size and performance penalty factor for different secure implementations of SIMON-64/128.

SIMON needs only four registers. The first-order protected implementation based on the reference expression of AND requires ten registers, while the one based on our optimal expression of AND takes nine registers.

The gain in execution time of the implementation based on the improved expression of AND over the implementation based on the reference expression of AND is modest for encryption (i.e. 3%) but significant for decryption (i.e. 12%). The improvement in code size is about 12%. The results of these implementations are presented in Table 9.7.

9.4.2.3 RECTANGLE

RECTANGLE [391] is a block cipher designed to facilitate lightweight and fast implementations, both in hardware and software, using bit slicing. RECTANGLE processes a 64-bit block in 25 rounds and supports keys of 80 and 128 bits. We refer to the 128-bit version of RECTANGLE as RECTANGLE-64/128. The cipher's state is represented as a matrix of 4×16 bits. Each round of RECTANGLE uses three transformations: `AddRoundKey` (bitwise XOR), `SubColumn` (application of a 4-bit S-box to the state columns), and `ShiftRow` (rotations of the state rows by 1, 12 and 13 bits). The S-box of RECTANGLE can be described using a sequence of 12 basic logical instructions and hence the `SubColumn` transformation can be implemented in a bit-sliced fashion.

The unprotected implementation of RECTANGLE requires seven registers for encryption and eight for decryption. The protected implementations use all available registers of the microcontroller and several pairs of stack operations (i.e. `push` and `pop`). The protected implementation based on the reference expressions uses five pairs of stack operations, while the one based on our optimal expressions uses only three pairs for encryption and four pairs for decryption. The stack operations are necessary because the protected implementations have to keep track of more intermediate variables than they can fit into the registers of the ARM microcontroller.

The performance figures given in Table 9.8 show that the encryption based on our expressions is 15% faster than encryption based on the reference expressions. On the other hand, decryption takes roughly the same time for both reference and improved expressions. The improvement in code size is modest for both encryption

Impl./Expr.	Rand	Time (cycles)		Code size (bytes)		Penalty factor	
		Enc	Dec	Enc	Dec	Enc	Dec
unprotected	0	945	994	200	160	1	1
reference	1	3470	3326	640	460	3.67	3.46
our	0	2937	3315	620	436	3.10	3.33
Improvement	1	533	11	20	24		
%		15.36%	1.10%	3.12%	5.21%		

Table 9.8: Execution time, code size and performance penalty factor for different secure implementations of RECTANGLE-64/128.

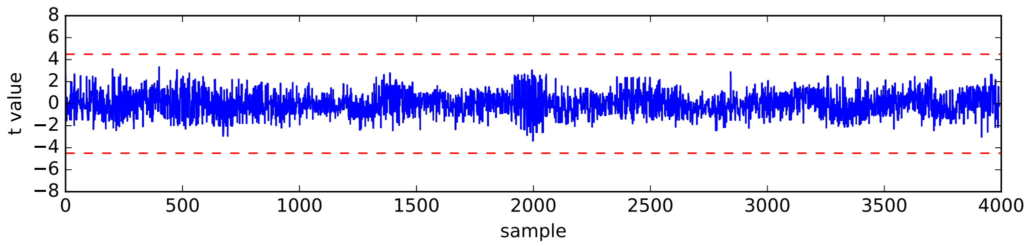


Figure 9.1: The result of the t-test applied to our implementation of Speck.

(i.e. 3%) and decryption (i.e. 5%).

9.4.2.4 Leakage Assessment

The tool we used to assess the security of our implementations against first-order attacks is inspired from similar tools such as ELMO [233], ASCOLD [271], and the one described in [291]. The simulated leakages are computed as follows. For each register r_i we store its previous value r_i^{j-1} and its current value r_i^j. At each step j we dump two leakages $\mathsf{HW}(r_i^j)$ and $\mathsf{HD}(r_i^{j-1}, r_i^j) = \mathsf{HW}(r_i^{j-1} \oplus r_i^j)$, where $\mathsf{HW}(r)$ is the Hamming weight of r.

The result of the t-test applied to 10^6 simulated traces from our first-order protected implementation of Speck is exemplarily shown in Figure 9.1. Similar results for Simon and RECTANGLE are given in Figure 9.2 and Figure 9.3, respectively. All results are based on implementations that use our expressions for secure computation of AND and OR on Boolean shares. We can see that the value of the t-statistic is inside the ± 4.5 interval for each point in time, which implies that the protected implementations are secure against first-order attacks.

9.4.2.5 Comparison

When comparing the performance results of the unprotected implementations of the three ciphers (see Figure 9.4), one can see that Speck is the fastest, followed by RECTANGLE and Simon; each of them takes about three times more cycles

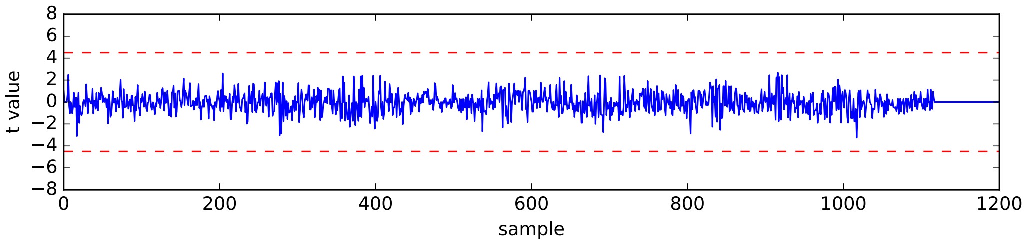


Figure 9.2: The result of the t-test applied to our implementation of SIMON.

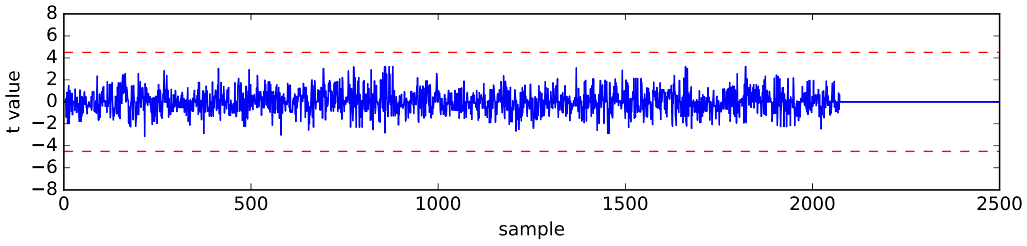


Figure 9.3: The result of the t-test applied to our implementation of RECTANGLE.

than SPECK. On the other hand, when comparing first-order protected implementations, the implementations of SIMON and RECTANGLE take the lead, while the implementation of SPECK is the last one. The performance degradation of the first-order protected implementation of SPECK stems from the high overhead associated with masking modular addition (see Table 9.5). The protected implementation of RECTANGLE is roughly three times slower than its unprotected implementation. Finally, the protected implementation of SIMON is less than twice slower than its unprotected implementation.

From this analysis, we learn that lightweight block ciphers that are very fast in unprotected software implementations (e.g. SPECK), might not be the most suitable ones for first-order masking in software. A second key remark is that a cipher that uses only bitwise operations can have an efficient first-order masked implementation only if it has a small number of intermediate variables.

9.4.2.6 Discussion

Our implementations explored how far one can push the optimization level in Boolean masking of various algorithms and ciphers. However, we lost the benefit of being able to provide strong security proofs for our implementations. But, one can simply insert a random value in our expressions for masked AND and OR to obtain provably secure expressions similar to the reference expressions.

Our first-order implementations did not require fresh random values, but there are situations where the composition of two expressions leaks. In such situations, one can inject a fresh random value to preserve the security of the masking scheme. Another option is to use our search algorithm to find an expression that computes

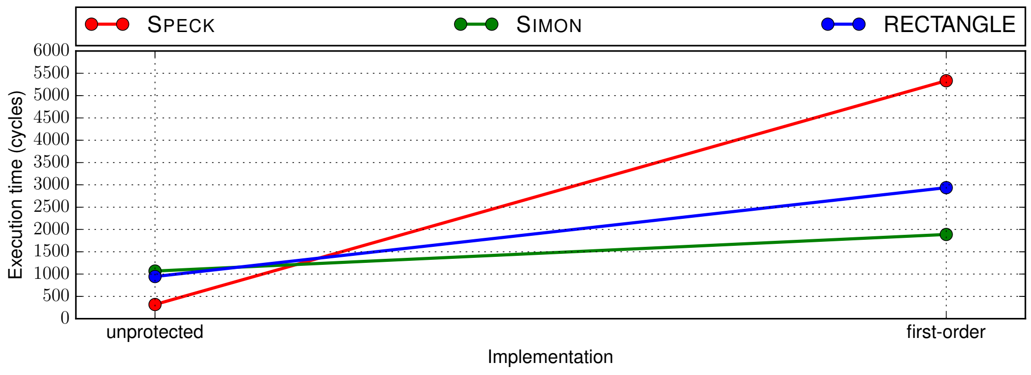


Figure 9.4: Performance comparison of unprotected and first-order protected implementations of SPECK, SIMON, and RECTANGLE.

the composed operations in a secure way.

9.5 Summary

We described an efficient algorithm for searching of optimal Boolean masking expressions. Then, we proposed optimal expressions for the first-order masking of bitwise AND and OR. They require less elementary operations and no random values compared to the reference expressions in the literature. Based on these optimal expressions, we presented an improved version of the algorithm for modular addition on Boolean shares proposed by Coron *et al.* [85]. We implemented the original and improved algorithms for modular addition/subtraction of 32-bit values on an ARM Cortex-M3. Our results show that the improved algorithm is between 18% and 25% faster than the original algorithm of Coron *et al.* [85]. Finally, we used our optimal Boolean masking expressions to write first-order protected implementations of three lightweight block ciphers, namely SIMON, SPECK, and RECTANGLE. The evaluation of these implementations revealed that ciphers with simple structure, based solely on bitwise logical operations and rotations, facilitate efficient software implementations of first-order masking.

The work presented in this chapter can be improved in several ways. First, we need to get a better understanding of what operations can be securely composed in order to prevent sequences of operations that leak sensitive values. Second, the security of the proposed implementations against first-order attacks should be validated using traces measured from an actual device instead of simulated leakages.

Chapter 10

Conclusion

In this thesis, we studied efficient and secure implementations of lightweight symmetric cryptographic primitives for resource-constrained devices that are widely used in the IoT. In this context, our results provide a better understanding of how to design, implement, and protect lightweight symmetric cryptographic algorithms for software applications on various microcontrollers.

A major part of this work was devoted to efficient software implementations. At its core sits the FELICS benchmarking framework, which addresses the need for fair and consistent evaluation of software implementations of lightweight symmetric algorithms in a transparent way. Namely, all implementations use the same programming interface and are placed in the public domain together with the benchmarking framework. FELICS extracts accurate values for three metrics (code size, RAM consumption, and execution time) from three different microcontrollers (8-bit AVR, 16-bit MSP, and 32-bit ARM) in various usage scenarios specific to the IoT. Since its initial release, the framework has become a reference point for assessing the efficiency of software implementations of lightweight cryptographic algorithms. The endorsement of the community is a clear confirmation that the project achieved its initial design goals. We used FELICS to evaluate how suitable implementations of 19 lightweight block ciphers are for resource-constrained applications on the three aforementioned platforms. The performance figures revealed that designs based on simple operations (addition/AND, rotation, and XOR) yield the most efficient implementations. The top performers are Chaskey, Speck, Simon, RECTANGLE, LEA, and Sparx. The implementations of these ciphers have small code and RAM requirements, while being very fast on all three platforms. FELICS facilitated informed decision-making based on software efficiency in the design phase of the Sparx family of lightweight block ciphers. As a result, Sparx is very fast in software and provably secure against simple differential and linear cryptanalysis. The benchmarking results of Sparx place it among the most efficient lightweight block ciphers evaluated using FELICS. Thanks to its flexible structure, the execution time of Sparx reaches the top 3 on MSP and the top 5 on AVR. Moreover, its implementations broke the previous minimum RAM consumption records on AVR and MSP. Finally, we employed FELICS to determine the cost of the main building blocks used in lightweight symmetric algorithms. The results of this comprehensive study are directly applicable to the design process of new symmetric ciphers intended for efficient software implementations. The best building blocks use simple operations on 32-bit values such as bitwise logical operations, modular addition/subtraction, and rotations by carefully chosen amounts.

In the second part of this thesis, we evaluated the security of lightweight cryptographic implementations from the viewpoint of an attacker. Our proactive approach was geared towards security against side-channel attacks that exploit the power consumption or the electromagnetic emanations of devices that execute a cryptographic algorithm. We analyzed the efficiency of different selection functions commonly used in correlation power analysis (CPA) attacks to identify the best operations an adversary should target to mount an effective attack. Our results show that lightweight block ciphers can be divided into two classes according to their resilience against CPA attacks. The first class contains ciphers that are implemented using lookup tables, while the second class comprises designs whose operations (modular addition/subtraction, bitwise logical operations) generally leak less than table lookups. Then, we showed that unprotected implementations of the AES, such as those found in many open-source cryptographic libraries, are vulnerable to side-channel attacks even when the attacker has limited control of the input, which is the case in network communication protocols. For this attack scenario, we introduced an attack algorithm that can recover the master key using an optimal number of CPA attacks. We broke unprotected implementations of the AES based on the S-box and T-table strategies by controlling a single byte of the input with less than 1600 electromagnetic traces acquired from a 32-bit ARM Cortex-M3 processor. Knowledge of the implementation strategy does not significantly improve the attack outcome, nor does it reduce the attack complexity. Finally, we presented a side-channel vulnerability analysis of the Thread networking stack. We identified an attack vector that combines network-specific mechanisms with differential electromagnetic analysis (DEMA) to get full access into a Thread network. The full attack did not succeed against a TI CC2538 system on chip that runs OpenThread, a certified open-source implementation of the stack, due to a fortunate packet fragmentation that is unrelated to security. The possibility to request the master key having the derived key(s) is questionable security-wise as it subverts the essence of key derivation using HMAC. We demonstrated that mounting a side-channel attack in the context of a modern IoT network protocol is not trivial. Being perhaps too expensive for settings like smart homes, such attacks may pose a relatively higher threat to the commercial setting. However, the security problems we identified give a useful lesson to designers of IoT systems.

The third part of this thesis covered the defensive side of security against side-channel attacks. We proposed an algorithm for efficient search of masked Boolean expressions that use an optimal number of elementary operations. In the case of first-order Boolean masking, the optimal expression of bitwise AND can be performed using less operations than the best known expression and does not require fresh random values. On the other hand, the best know expression for bitwise OR is optimal. The protected implementations of SPECK, SIMON, and RECTANGLE revealed that ciphers that have a simple structure, based solely on bitwise logical operations and rotations, facilitate efficient software implementations of first-order masking.

10.1 Impact

Two of our research projects already have impact in the research community or in industry.

First, the FELICS benchmarking framework is well known in the research community. Many people contributed optimized implementations and the evaluation results are becoming a common reference in the literature. Moreover, NIST is interested in using FELICS for a fair comparison of candidates submitted to their portfolio of lightweight algorithms recommended for the IoT.

Second, our vulnerability analysis of the Thread networking stack determined the Thread group to elaborate a set of recommendations for implementers in order to enhance the security of Thread products. In light of our results, designers of future protocols for the IoT should carefully consider the threat of side-channel attacks from the early inception.

10.2 Future Directions

Improving FELICS. FELICS can be improved in several ways. For example, the framework can be extended to support more target devices, especially ultra-low-power microcontrollers. At the same time, energy consumption of actual devices should be measured in order to get a clear picture of the energy requirements of various primitives. Such data is currently missing from the evaluation framework. Moreover, energy requirement can not be reliably estimated since it is difficult to model the power consumption of actual devices. Approximations based on execution time are prone to errors because not all instructions have the same energy requirements. Usually memory instructions take more energy than register-only instructions. Based on energy figures measured from actual devices, one can determine which devices and ciphers are more suitable for a given use case. These two improvements of FELICS are very relevant for applications of cryptography on devices that run only on harvested energy.

Another way of improving FELICS is to add an interface for benchmarking implementations protected against side-channel attacks. We showed that side-channel countermeasures such as Boolean masking influence the performance figures of protected software implementations of lightweight ciphers differently when compared to unprotected implementations. In other words, some ciphers favour efficient masked implementations more than others. Hence, this improvement of FELICS is useful for real-world applications of lightweight symmetric cryptography considering the need for side-channel countermeasures that stems from the IoT threat model.

FELICS supports only symmetric cryptography, but there is also a need for fair comparative results obtained from implementations of public key algorithms; and yet there is no tool to satisfy this need. Therefore, FELICS can be extended to benchmark post-quantum public key cryptography for example.

Side-Channel Attacks. In this work, we focused on side-channel attacks that exploit the power consumption or the electromagnetic emissions of a target device. Yet, there are many other interesting side-channel attack techniques such as timing attacks, cache attacks, and fault attacks that can be applied against embedded systems that perform lightweight cryptographic algorithms.

Energy-Efficient and Secure Communication Protocols. Most of the current communication standards for the IoT use the AES to secure and authenticate communication between end nodes. This is not ideal for several reasons. First, the AES is less suitable for very constrained devices such as those that run on harvested energy than many lightweight block ciphers. Second, the cost of protecting the AES against side-channel attacks is much higher than the cost of protecting most lightweight block ciphers due to its large S-box.

Bibliography

[1] Masayuki Abe, editor. *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, volume 4377 of *Lecture Notes in Computer Science*. Springer, 2006. (Cited on pages 201 and 227.)

[2] Dennis G. Abraham, George M. Dolan, Glen P. Double, and James V. Stevens. Transaction Security System. *IBM Systems Journal*, 30(2):206–229, 1991. (Cited on page 36.)

[3] Onur Aciiçmez, Werner Schindler, and Çetin Kaya Koç. Cache Based Remote Timing Attack on the AES. In Abe [1], pages 271–286. (Cited on page 172.)

[4] Wim Aerts, Eli Biham, Dieter De Moitie, Elke De Mulder, Orr Dunkelman, Sebastiaan Indesteege, Nathan Keller, Bart Preneel, Guy A. E. Vandenbosch, and Ingrid Verbauwhede. A Practical Attack on KeeLoq. *Journal of Cryptology*, 25(1):136–157, 2012. (Cited on page 5.)

[5] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM Side-Channel(s). In Jr. et al. [180], pages 29–45. (Cited on page 20.)

[6] Dakshi Agrawal, Josyula R. Rao, and Pankaj Rohatgi. Multi-channel Attacks. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2003. (Cited on page 27.)

[7] Dakshi Agrawal, Josyula R. Rao, Pankaj Rohatgi, and Kai Schramm. Templates as Master Keys. In Rao and Sunar [289], pages 15–29. (Cited on page 28.)

[8] Mehdi-Laurent Akkar, Régis Bevan, Paul Dischamp, and Didier Moyart. Power Analysis, What Is Now Possible... In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 489–502. Springer, 2000. (Cited on page 26.)

[9] Martin R. Albrecht, Benedikt Driessen, Elif Bilge Kavun, Gregor Leander, Christof Paar, and Tolga Yalçin. Block Ciphers - Focus on the Linear Layer

(feat. PRIDE). In Garay and Gennaro [135], pages 57–76. (Cited on pages 65, 72, and 108.)

[10] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the Security of RC4 in TLS. In King [187], pages 305–320. (Cited on page 5.)

[11] Jude Ambrose, Alexandar Ignjatovic, and Sri Parameswaran. *Power Analysis Side Channel Attacks: The Processor Design-level Context*. VDM Publishing, 2010. (Cited on page 18.)

[12] Frédéric Amiel, Karine Villegas, Benoit Feix, and Louis Marcel. Passive and Active Combined Attacks: Combining Fault Attacks and Side Channel Analysis. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*, pages 92–102. IEEE Computer Society, 2007. (Cited on page 19.)

[13] Ross J. Anderson. *Security Engineering – A Guide to Building Dependable Distributed Systems (Second Edition)*. Wiley, 2008. (Cited on pages 16 and 18.)

[14] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In Kirda and Ristenpart [188], pages 1093–1110. (Cited on page 180.)

[15] Arduino. Arduino Due. `https://store.arduino.cc/arduino-due`. Accessed: September 2017. (Cited on pages 59 and 61.)

[16] Random Number Generator (TRNG) API. `https://forum.arduino.cc/index.php?topic=129083.0`, October 2012. Accessed: September 2017. (Cited on page 191.)

[17] ARM. Cortex-M3 Devices Generic User Guide. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/CHDBIBGJ.html`. Accessed: September 2017. (Cited on page 57.)

[18] ARM. mbed TLS. Available at `https://tls.mbed.org/`. Accessed: April 2017. (Cited on pages 133, 137, and 173.)

[19] Scherbius Arthur. Ciphering Machine, January 1928. US Patent 1,657,411. (Cited on page 3.)

[20] Dmitri Asonov and Rakesh Agrawal. Keyboard Acoustic Emanations. In *2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA*, pages 3–11. IEEE Computer Society, 2004. (Cited on page 18.)

[21] Ahmad Atamli and Andrew P. Martin. Threat-Based Security Analysis for the Internet of Things. In Gabriel Ghinita, Razvan Rughinis, and Ahmad-Reza Sadeghi, editors, *2014 International Workshop on Secure Internet of Things, SIoT 2014, Wroclaw, Poland, September 10, 2014*, pages 35–43. IEEE Computer Society, 2014. (Cited on pages 36, 37, and 161.)

[22] Atmel. ATmega128. `http://www.atmel.com/images/doc2467.pdf`. Accessed: September 2017. (Cited on pages 57 and 58.)

[23] Michael Backes, Markus Dürmuth, Sebastian Gerling, Manfred Pinkal, and Caroline Sporleder. Acoustic Side-Channel Attacks on Printers. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 307–322. USENIX Association, 2010. (Cited on page 18.)

[24] Stéphane Badel, Nilay Dagtekin, Jorge Nakahara Jr., Khaled Ouafi, Nicolas Reffé, Pouyan Sepehrdad, Petr Susil, and Serge Vaudenay. ARMADILLO: A Multi-purpose Cryptographic Primitive Dedicated to Hardware. In Mangard and Standaert [224], pages 398–412. (Cited on page 13.)

[25] Yoo-Jin Baek and Mi-Jung Noh. Differential Power Attack and Masking Method. *Trends in Mathematics*, 8(1):1–15, June 2005. (Cited on pages 181 and 186.)

[26] 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, *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2012. (Cited on pages 14 and 48.)

[27] 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. Implementations of Hash Functions in Atmel AVR Devices. Available at `http://perso.uclouvain.be/fstandae/source_codes/hash_atmel/`. Accessed: September 2017. (Cited on page 48.)

[28] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the Cost of Lazy Engineering for Masked Software Implementations. In Joye and Moradi [177], pages 64–81. (Cited on pages 32, 189, and 190.)

[29] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. DPA, Bitslicing and Masking at 1 GHz. In Güneysu and Handschuh [156], pages 599–619. (Cited on pages 21 and 152.)

[30] Valentina Banciu, Elisabeth Oswald, and Carolyn Whitnall. Exploring the Resilience of Some Lightweight Ciphers Against Profiled Single Trace Attacks. In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*, volume 9064 of *Lecture Notes in Computer Science*, pages 51–63. Springer, 2015. (Cited on page 114.)

[31] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A Block Cipher for Low Energy. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 411–436. Springer, 2015. (Cited on page 14.)

[32] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006. (Cited on page 18.)

[33] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong Non-Interference and Type-Directed Higher-Order Masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129. ACM, 2016. (Cited on page 32.)

[34] Arthur O Bauer. Some Aspects of Military Line Communications as Deployed by the German Armed Forces prior to 1945, December 2004. Available at `http://www.cdvandt.org/Wirecomm99.pdf`. Accessed: September 2017. (Cited on page 16.)

[35] Adnan Baysal and Sühap Sahin. RoadRunneR: A Small and Fast Bitslice Block Cipher for Low Cost 8-Bit Processors. In Tim Güneysu, Gregor Leander, and Amir Moradi, editors, *Lightweight Cryptography for Security and Privacy - 4th International Workshop, LightSec 2015, Bochum, Germany, September 10-11, 2015, Revised Selected Papers*, volume 9542 of *Lecture Notes in Computer Science*, pages 58–76. Springer, 2015. (Cited on pages 65, 73, 108, 109, and 189.)

[36] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight

Block Ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013. (Cited on pages 65, 68, 73, 74, and 109.)

[37] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. SIMON and SPECK: Block Ciphers for the Internet of Things. *IACR Cryptology ePrint Archive*, 2015:585, 2015. (Cited on page 114.)

[38] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Lightweight Block Ciphers. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 175:1–175:6. ACM, 2015. (Cited on pages 108, 189, 191, and 192.)

[39] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. Notes on the Design and Analysis of SIMON and SPECK. *IACR Cryptology ePrint Archive*, 2017:560, 2017. (Cited on page 99.)

[40] GT Becker, J Cooper, E DeMulder, G Goodwill, J Jaffe, G Kenworthy, T Kouzminov, A Leiserson, M Marson, P Rohatgi, and S Saab. Test Vector Leakage Assessment (TVLA) Methodology in Practice. In *International Cryptographic Module Conference*, 2013. (Cited on page 33.)

[41] Arthur Beckers, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Design and Implementation of a Waveform-Matching Based Triggering System. In Standaert and Oswald [337], pages 184–198. (Cited on pages 22 and 25.)

[42] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In Robshaw and Katz [298], pages 123–153. (Cited on page 105.)

[43] Janine Bennett, Ray W. Grout, Philippe P. Pébay, Diana C. Roe, and David C. Thompson. Numerically Stable, Single-Pass, Parallel Statistics Algorithms. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*, pages 1–8. IEEE Computer Society, 2009. (Cited on page 30.)

[44] Olivier Benoît and Thomas Peyrin. Side-Channel Analysis of Six SHA-3 Candidates. In Mangard and Standaert [224], pages 140–157. (Cited on page 115.)

[45] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. Available at `http://bench.cr.yp.to/`. Accessed: September 2017. (Cited on pages 46, 48, and 63.)

[46] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and*

Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013. (Cited on page 8.)

[47] Régis Bevan and Erik Knudsen. Ways to Enhance Differential Power Analysis. In Lee and Lim [211], pages 327–342. (Cited on page 27.)

[48] Shivam Bhasin, Tarik Graba, Jean-Luc Danger, and Zakaria Najm. A Look into SIMON from a Side-Channel Perspective. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2014, Arlington, VA, USA, May 6-7, 2014*, pages 56–59. IEEE Computer Society, 2014. (Cited on page 114.)

[49] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997. (Cited on page 17.)

[50] Alex Biryukov, Daniel Dinu, and Johann Großschädl. Correlation Power Analysis of Lightweight Block Ciphers: From Theory to Practice. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, volume 9696 of *Lecture Notes in Computer Science*, pages 537–557. Springer, 2016. (Cited on pages 138 and 152.)

[51] Alex Biryukov and Eyal Kushilevitz. Improved Cryptanalysis of RC5. In Kaisa Nyberg, editor, *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding*, volume 1403 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 1998. (Cited on page 72.)

[52] Alex Biryukov, Gaëtan Leurent, and Arnab Roy. Cryptanalysis of the "Kindle" Cipher. In Knudsen and Wu [195], pages 86–103. (Cited on page 5.)

[53] Alex Biryukov and Léo Perrin. State of the Art in Lightweight Symmetric Cryptography. *IACR Cryptology ePrint Archive*, 2017:511, 2017. (Cited on page 5.)

[54] Alex Biryukov and David A. Wagner. Slide Attacks. In Lars R. Knudsen, editor, *Fast Software Encryption, 6th International Workshop, FSE '99, Rome, Italy, March 24-26, 1999, Proceedings*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 1999. (Cited on page 87.)

[55] George Robert Blakley. Safeguarding Cryptographic Keys. In *Proceedings of AFIPS 1979 National Computer Conference*, pages 313–317, 1979. (Cited on page 31.)

[56] Céline Blondeau and Kaisa Nyberg. Links between Truncated Differential and Multidimensional Linear Properties of Block Ciphers and Underlying Attack Complexities. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 165–182. Springer, 2014. (Cited on page 72.)

[57] David G. Boak. *A History of U.S. Communications Security. The David G. Boak Lectures*, volume I. National Security Agency, July 1973. Declassified: October 2015. Available at http://www.cryptomuseum.com/intel/nsa/files/nsa_history_comsec_1.pdf, Accessed: September 2017. (Cited on page 16.)

[58] 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 [270], pages 450–466. (Cited on pages 65, 68, and 71.)

[59] Andrey Bogdanov, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, and Yannick Seurin. Hash Functions and RFID Tags: Mind the Gap. In Oswald and Rohatgi [267], pages 283–299. (Cited on page 13.)

[60] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997. (Cited on page 17.)

[61] Steve Bono, Matthew Green, Adam Stubblefield, Ari Juels, Aviel D. Rubin, and Michael Szydlo. Security Analysis of a Cryptographically-Enabled RFID Device. In Patrick D. McDaniel, editor, *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005.* USENIX Association, 2005. (Cited on page 5.)

[62] 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 [375], pages 208–225. (Cited on pages 65, 68, 72, and 114.)

[63] Carsten Bormann and Zach Shelby. Block-Wise Transfers in the Constrained Application Protocol (CoAP). RFC 7959, Internet Engineering Task Force, September 2016. Available at https://tools.ietf.org/html/rfc7959. Accessed: September 2017. (Cited on page 56.)

[64] Paul Bottinelli and Joppe W. Bos. Computational Aspects of Correlation Power Analysis. *Journal of Cryptographic Engineering*, 7(3):167–181, 2017. (Cited on page 30.)

[65] Christina Boura, María Naya-Plasencia, and Valentin Suder. Scrutinizing and Improving Impossible Differential Attacks: Applications to CLEFIA, Camellia, LBlock and Simon. In Sarkar and Iwata [304], pages 179–199. (Cited on page 71.)

[66] Walter H Brattain and Bardeen John. Three-Electrode Circuit Element utilizing Semiconductive Materials, October 1950. US Patent 2,524,035. (Cited on page 3.)

[67] Eric Brier, Christophe Clavier, and Francis Olivier. Optimal Statistical Power Analysis. *IACR Cryptology ePrint Archive*, 2003:152, 2003. (Cited on page 27.)

[68] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Joye and Quisquater [179], pages 16–29. (Cited on pages 27, 116, and 132.)

[69] Stephen D Brown, Robert J Francis, Jonathan Rose, and Zvonko G Vranesic. *Field-Programmable Gate Arrays*, volume 180. Springer Science & Business Media, 2012. (Cited on page 11.)

[70] Frederick J Bruwer, Willem Smit, and Gideon J Kuhn. Microchips and Remote Control Devices Comprising same, May 1996. US Patent 5,517,187. (Cited on page 5.)

[71] CAESAR Competition. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. Available at `http://competitions.cr.yp.to/caesar.html`. Accessed: September 2017. (Cited on pages 47, 48, 55, and 56.)

[72] Anne Canteaut, Thomas Fuhr, Henri Gilbert, María Naya-Plasencia, and Jean-René Reinhard. Multiple Differential Cryptanalysis of Round-Reduced PRINCE. In Cid and Rechberger [82], pages 591–610. (Cited on page 72.)

[73] Claude Carlet. On Highly Nonlinear S-Boxes and Their Inability to Thwart DPA Attacks. In Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan, editors, *Progress in Cryptology - INDOCRYPT 2005, 6th International Conference on Cryptology in India, Bangalore, India, December 10-12, 2005, Proceedings*, volume 3797 of *Lecture Notes in Computer Science*, pages 49–62. Springer, 2005. (Cited on page 115.)

[74] Mickaël Cazorla, Sylvain Gourgeon, Kevin Marquet, and Marine Minier. Implementations of Lightweight Block Ciphers on a WSN430 Sensor. Available at `http://bloc.project.citi-lab.fr/library.html`. Accessed: September 2017. (Cited on pages 45, 72, and 81.)

[75] Mickaël Cazorla, Kevin Marquet, and Marine Minier. Survey and Benchmark of Lightweight Block Ciphers for Wireless Sensor Networks. In Pierangela Samarati, editor, *SECRYPT 2013 - Proceedings of the 10th International Conference on Security and Cryptography, Reykjavík, Iceland, 29-31 July, 2013*, pages 543–548. SciTePress, 2013. (Cited on pages 45 and 46.)

[76] Cees-Bart Breunesse and Ilya Kizhvatov. Jlsca: Side-channel Toolkit in Julia. Available at `https://github.com/Riscure/Jlsca`. Accessed: September 2017. (Cited on pages 30, 155, and 171.)

[77] Kaushik Chakraborty, Sumanta Sarkar, Subhamoy Maitra, Bodhisatwa Mazumdar, Debdeep Mukhopadhyay, and Emmanuel Prouff. Redefining the Transparency Order. *Designs, Codes and Cryptography*, 82(1-2):95–115, 2017. (Cited on page 117.)

[78] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Wiener [382], pages 398–412. (Cited on pages 27 and 31.)

[79] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In Jr. et al. [180], pages 13–28. (Cited on pages 21 and 27.)

[80] Huaifeng Chen and Xiaoyun Wang. Improved Linear Hull Attack on Round-Reduced Simon with Dynamic Key-Guessing Techniques. In Peyrin [274], pages 428–449. (Cited on page 73.)

[81] Omar Choudary and Markus G. Kuhn. Template Attacks on Different Devices. In Prouff [281], pages 179–198. (Cited on page 28.)

[82] Carlos Cid and Christian Rechberger, editors. *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*. Springer, 2015. (Cited on pages 208 and 216.)

[83] Christophe Clavier, Benoit Feix, Georges Gagnerot, and Mylène Roussellet. Passive and Active Combined Attacks on AES: Combining Fault Attacks and Side Channel Analysis. In Luca Breveglieri, Marc Joye, Israel Koren, David Naccache, and Ingrid Verbauwhede, editors, *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2010, Santa Barbara, California, USA, 21 August 2010*, pages 10–19. IEEE Computer Society, 2010. (Cited on page 19.)

[84] Lucian Constantin. Hackers Found 47 New Vulnerabilities in 23 IoT Devices at DEF CON. `http://www.csoonline.com/article/3119765/security/hackers-found-47-new-vulnerabilities-in-23-iot-devices-at-def-con.html`, September 2016. Accessed: September 2017. (Cited on page 180.)

[85] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In Gregor Leander, editor, *Fast Software Encryption -*

22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers, volume 9054 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2015. (Cited on pages 32, 180, 181, 182, 186, and 196.)

[86] Jean-Sébastien Coron and Ilya Kizhvatov. Analysis and Improvement of the Random Delay Countermeasure of CHES 2009. In Mangard and Standaert [224], pages 95–109. (Cited on pages 24 and 30.)

[87] Jean-Sébastien Coron, Paul C. Kocher, and David Naccache. Statistics and Secret Leakage. In Yair Frankel, editor, *Financial Cryptography, 4th International Conference, FC 2000 Anguilla, British West Indies, February 20-24, 2000, Proceedings*, volume 1962 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2000. (Cited on pages 26, 27, and 33.)

[88] Jean-Sébastien Coron, David Naccache, and Paul C. Kocher. Statistics and Secret Leakage. *ACM Transactions on Embedded Computer Systems*, 3(3):492–508, 2004. (Cited on pages 27 and 33.)

[89] Common Criteria. Publications. Available at http://www.commoncriteriaportal.org/cc/. Accessed: September 2017. (Cited on page 29.)

[90] cryptlib. The cryptlib Security Software Development Toolkit. Available at http://www.cryptlib.com/. Accessed: April 2017. (Cited on page 137.)

[91] Crypto++. Crypto++: a Free C++ Class Library of Cryptographic Schemes. Available at https://www.cryptopp.com/. Accessed: April 2017. (Cited on page 137.)

[92] CryptoLUX. Block Ciphers Brief Results. Available at https://www.cryptolux.org/index.php/FELICS_Block_Ciphers_Brief_Results. Accessed: September 2017. (Cited on pages 92 and 94.)

[93] CryptoLUX. FELICS – Fair Evaluation of Lightweight Cryptographic Systems. Available at https://www.cryptolux.org/index.php/FELICS. Accessed: September 2017. (Cited on pages 45, 52, 57, 61, 62, 66, 75, 80, 81, 93, 114, and 118.)

[94] CryptoLUX. SPARX. Available at https://www.cryptolux.org/index.php/SPARX. Accessed: September 2017. (Cited on page 93.)

[95] CryptoLUX. SPARX – The SPARX Family of Lightweight Block Ciphers. Available at https://github.com/cryptolu/SPARX. Accessed: September 2017. (Cited on page 93.)

[96] Guillaume Dabosville, Julien Doget, and Emmanuel Prouff. A New Second-Order Side Channel Attack Based on Linear Regression. *IEEE Transactions on Computers*, 62(8):1629–1640, 2013. (Cited on page 28.)

[97] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie Proposal: NOEKEON. In *First Open NESSIE Workshop*, pages 213–230, 2000. (Cited on pages 74, 86, and 189.)

[98] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Information Security and Cryptography. Springer, 2002. (Cited on pages 7, 68, 69, 134, and 136.)

[99] Damian Gryski. go-sparx: SPARX Lightweight Cipher. Available at `https://github.com/dgryski/go-sparx`. Accessed: September 2017. (Cited on page 94.)

[100] Patrick Derbez and Pierre-Alain Fouque. Exhausting Demirci-Selçuk Meet-in-the-Middle Attacks Against Reduced-Round AES. In Moriai [243], pages 541–560. (Cited on page 69.)

[101] Whitfield Diffie and Martin Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. (Cited on page 3.)

[102] Whitfield Diffie and Martin E Hellman. Privacy and Authentication: An introduction to Cryptography. *Proceedings of the IEEE*, 67(3):397–427, 1979. (Cited on page 7.)

[103] A. Adam Ding, Cong Chen, and Thomas Eisenbarth. Simpler, Faster, and More Robust T-Test Based Leakage Detection. In Standaert and Oswald [337], pages 163–183. (Cited on page 189.)

[104] Daniel Dinu, Alex Biryukov, Johann Großschädl, Dmitry Khovratovich, YL Corre, and Léo Perrin. FELICS–Fair Evaluation of Lightweight Cryptographic Systems. In *NIST Workshop on Lightweight Cryptography*, 2015. (Cited on page 97.)

[105] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. Triathlon of Lightweight Block Ciphers for the Internet of Things. *IACR Cryptology ePrint Archive*, 2015:209, 2015. (Cited on pages 97, 114, 125, and 191.)

[106] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design Strategies for ARX with Provable Bounds: Sparx and LAX. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 484–513, 2016. (Cited on pages 65, 68, 74, 83, 99, and 108.)

[107] Itai Dinur, Orr Dunkelman, Nathan Keller, and Adi Shamir. Key Recovery Attacks on 3-round Even-Mansour, 8-step LED-128, and Full AES2. In Sako and Sarkar [303], pages 337–356. (Cited on page 71.)

[108] dlbeer Engineering. MSPDebug. http://dlbeer.co.nz/mspdebug/. Accessed: September 2017. (Cited on page 61.)

[109] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, and Florian Mendel. On the Security of Fresh Re-keying to Counteract Side-Channel and Fault Attacks. In Joye and Moradi [177], pages 233–244. (Cited on page 174.)

[110] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, and Thomas Unterluggauer. ISAP - Towards Side-Channel Secure Authenticated Encryption. *IACR Transactions on Symmetric Cryptology*, 2017(1):80–105, 2017. (Cited on page 174.)

[111] Christoph Dobraunig, François Koeune, Stefan Mangard, Florian Mendel, and François-Xavier Standaert. Towards Fresh and Hybrid Re-Keying Schemes with Beyond Birthday Security. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, volume 9514 of *Lecture Notes in Computer Science*, pages 225–241. Springer, 2015. (Cited on page 31.)

[112] Julien Doget, Emmanuel Prouff, Matthieu Rivain, and François-Xavier Standaert. Univariate Side Channel Attacks and Leakage Modeling. *Journal of Cryptographic Engineering*, 1(2):123–144, 2011. (Cited on page 28.)

[113] Margaux Dugardin, Louiza Papachristodoulou, Zakaria Najm, Lejla Batina, Jean-Luc Danger, and Sylvain Guilley. Dismantling Real-World ECC with Horizontal and Vertical Template Attacks. In Standaert and Oswald [337], pages 88–108. (Cited on page 173.)

[114] Orr Dunkelman and Liam Keliher, editors. *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, volume 9566 of *Lecture Notes in Computer Science*. Springer, 2016. (Cited on page 14.)

[115] Morris J Dworkin. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. NIST Special Publication 800-38C, 2007. (Cited on pages 132 and 165.)

[116] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-Resilient Cryptography. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 293–302. IEEE Computer Society, 2008. (Cited on page 31.)

[117] William F Ehrsam, Carl HW Meyer, John L Smith, and Walter L Tuchman. Message Verification and Transmission Error Detection by Block Chaining, February 1978. US Patent 4,074,066. (Cited on page 7.)

[118] 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, *Progress in Cryptology - AFRICACRYPT 2012 - 5th International Conference on Cryptology in Africa, Ifrance, Morocco, July 10-12, 2012. Proceedings*, volume 7374 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2012. (Cited on pages 14 and 48.)

[119] 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. Implementations of Low Cost Block Ciphers in Atmel AVR Devices. Available at http://perso.uclouvain.be/fstandae/lightweight_ciphers/. Accessed : September 2017. (Cited on pages 48 and 81.)

[120] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T. Manzuri Shalmani. On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoqCode Hopping Scheme. In David A. 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*, pages 203–220. Springer, 2008. (Cited on page 5.)

[121] 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. (Cited on pages 45 and 81.)

[122] Dave Evans. The Internet of Things: How the Next Evolution of the Internet is Changing Everything, April 2011. Cisco IBSG white paper, available for download at http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf. Accessed: September 2017. (Cited on page 36.)

[123] Martin Feldhofer, Sandra Dominikus, and Johannes Wolkerstorfer. Strong Authentication for RFID Systems Using the AES Algorithm. In Joye and Quisquater [179], pages 357–370. (Cited on page 68.)

[124] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein Hash Function Family. *Submission to NIST (round 3)*, 7(7.5):3, 2010. (Cited on page 85.)

[125] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. John Wiley & Sons, 2011. (Cited on page 4.)

[126] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security Analysis of Emerging Smart Home Applications. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 636–654. IEEE Computer Society, 2016. (Cited on page 156.)

[127] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In Holz and Savage [164], pages 531–548. (Cited on page 156.)

[128] Julie Ferrigno and Martin Hlavác. When AES blinks: introducing optical side channel. *IET Information Security*, 2(3):94–98, 2008. (Cited on page 20.)

[129] Microchip (former Atmel). ARM Cortex-M3 Datasheet. Available at http://www.microchip.com/wwwproducts/en/ATSAM3X8E. Accessed: September 2017. (Cited on page 59.)

[130] Frank Denis. rust-sparx: SPARX Block Ciphers Implementations for Rust. Available at https://github.com/jedisct1/rust-sparx. Accessed: September 2017. (Cited on page 94.)

[131] Jeffrey Friedman. TEMPEST: A Signal Problem. *NSA Cryptologic Spectrum*, 1972. Declassified: September 2007. Available at https://www.nsa.gov/news-features/declassified-documents/cryptologic-spectrum/assets/files/tempest.pdf. Accessed: September 2017. (Cited on pages 16 and 17.)

[132] 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 [224], pages 264–278. (Cited on page 44.)

[133] Kris Gaj, Jens-Peter Kaps, Venkata Amirineni, Marcin Rogawski, Ekawat Homsirikamol, and Benjamin Y. Brewster. ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs. In *International Conference on Field Programmable Logic and Applications, FPL 2010, August 31 2010 - September 2, 2010, Milano, Italy*, pages 414–421. IEEE Computer Society, 2010. (Cited on page 48.)

[134] Taher El Gamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1984. (Cited on page 9.)

[135] Juan A. Garay and Rosario Gennaro, editors. *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*. Springer, 2014. (Cited on pages 202 and 215.)

[136] Gartner. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. http://www.gartner.com/newsroom/id/3598917, February 2017. Accessed: September 2017. (Cited on pages 36, 179, and 180.)

[137] Gemalto. What are the differences between contactless smart cards and RFID?, June 2017. Available at https://www.justaskgemalto.com/us/what-are-differences-between-contactless-smart-cards-and-rfid/. Accessed: September 2017. (Cited on page 35.)

[138] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In Garay and Gennaro [135], pages 444–461. (Cited on pages 18 and 20.)

[139] Daniel Genkin, Adi Shamir, and Eran Tromer. Acoustic Cryptanalysis. *Journal of Cryptology*, 30(2):392–443, 2017. (Cited on page 20.)

[140] Benoît Gérard, Vincent Grosso, María Naya-Plasencia, and François-Xavier Standaert. Block Ciphers That Are Easier to Mask: How Far Can We Go? In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 383–399. Springer, 2013. (Cited on page 114.)

[141] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual Information Analysis. In Oswald and Rohatgi [267], pages 426–442. (Cited on pages 27 and 30.)

[142] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. Stochastic Methods. In Goubin and Matsui [150], pages 15–29. (Cited on page 28.)

[143] GitHub. libtomcrypt: A Fairly Comprehensive, Modular and Portable Cryptographic Toolkit. Available at https://github.com/libtom/libtomcrypt. Accessed: April 2017. (Cited on page 137.)

[144] GitHub. mbed TLS – An Open Source, Portable, Easy to Use, Readable and Flexible SSL Library. Avialable at https://github.com/ARMmbed/mbedtls/blob/development/library/aes.c. Accessed: April 2017. (Cited on page 137.)

[145] Virgil D. Gligor. Light-Weight Cryptography – How Light is Light? Keynote presentation at the Information Security Summer School, Florida State University. Available at http://www.sait.fsu.edu/conferences/2005/is3/resources/slides/gligorv-cryptolite.ppt, May 2005. Accessed: September 2017. (Cited on page 34.)

[146] GnuPG. Libgcrypt: A General Purpose Cryptographic Library Based on the Code from GnuPG. Available at https://gnupg.org/software/libgcrypt/index.html. Accessed: April 2017. (Cited on page 137.)

[147] Dan Goodin. Actively Exploited iOS Flaws that Hijack iPhones Patched by Apple, August 2016. Available at https://arstechnica.com/security/2016/08/actively-exploited-ios-flaws-that-hijack-iphones-likely-spread-for-years/. Accessed: September 2017. (Cited on page 155.)

[148] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A Testing Methodology for Side-Channel Resistance Validation. In *NIST Non-Invasive Attack Testing Workshop*, pages 158–172, 2011. (Cited on pages 33, 185, and 189.)

[149] Louis Goubin. A Sound Method for Switching between Boolean and Arithmetic Masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001. (Cited on page 32.)

[150] 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 *Lecture Notes in Computer Science*. Springer, 2006. (Cited on pages 215, 218, and 235.)

[151] Louis Goubin and Jacques Patarin. DES and Differential Power Analysis (The "Duplication" Method). In Koç and Paar [196], pages 158–172. (Cited on page 31.)

[152] Hannes Groß. Sharing is Caring - On the Protection of Arithmetic Logic Units against Passive Physical Attacks. In Stefan Mangard and Patrick Schaumont, editors, *Radio Frequency Identification. Security and Privacy Issues - 11th International Workshop, RFIDsec 2015, New York, NY, USA, June 23-24, 2015, Revised Selected Papers*, volume 9440 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2015. (Cited on page 185.)

[153] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations. In Cid and Rechberger [82], pages 18–37. (Cited on pages 65, 68, 70, 73, 105, and 114.)

[154] Sylvain Guilley, Philippe Hoogvorst, and Renaud Pacalet. Differential Power Analysis Model and Some Results. In Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas Abou El Kalam, editors, *Smart Card Research and Advanced Applications VI, IFIP 18th World Computer Congress, TC8/WG8.8 & TC11/WG11.2 Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS), 22-27 August 2004, Toulouse, France*, volume 153 of *IFIP*, pages 127–142. Kluwer/Springer, 2004. (Cited on page 118.)

[155] Sylvain Guilley, Philippe Hoogvorst, Renaud Pacalet, and Johannes Schmidt. Improving Side-Channel Attacks by Exploiting Substitution Boxes Properties. In *International Workshop on Boolean Functions: Cryptography and Applications*, pages 1–25, 2007. (Cited on page 118.)

[156] Tim Güneysu and Helena Handschuh, editors. *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*. Springer, 2015. (Cited on pages 204, 223, and 232.)

[157] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED Block Cipher. In Preneel and Takagi [278], pages 326–341. (Cited on pages 65, 68, and 71.)

[158] Peter Gutmann. Data Remanence in Semiconductor Devices. In Dan S. Wallach, editor, *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*. USENIX, 2001. (Cited on page 19.)

[159] Byoungjin Han, Hwanjin Lee, Hyuncheol Jeong, and Yoojae Won. The HIGHT Encryption Algorithm. Internet-Draft draft-kisa-hight-00, Internet Engineering Task Force (IETF), June 2011. `https://tools.ietf.org/id/draft-kisa-hight-00.txt`. (Cited on page 70.)

[160] Helena Handschuh, Pascal Paillier, and Jacques Stern. Probing Attacks on Tamper-Resistant Devices. In Koç and Paar [196], pages 303–315. (Cited on page 19.)

[161] 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, *Applied Cryptography and Network Security, 4th International Conference, ACNS 2006, Singapore, June 6-9, 2006, Proceedings*, volume 3989 of *Lecture Notes in Computer Science*, pages 239–252, 2006. (Cited on pages 30 and 32.)

[162] Thomas Hobbes. *Leviathan*. A&C Black, 2006. (Cited on page 1.)

[163] Gael Hofemeier and Robert Chesebrough. Introduction to Intel AES-NI and Intel Secure Key Instructions. Technical report available at `https://software.intel.com/sites/default/files/m/d/4/1/d/8/Introduction_to_Intel_Secure_Key_Instructions.pdf`. Accessed: April 2017. (Cited on pages 133 and 137.)

[164] Thorsten Holz and Stefan Savage, editors. *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. USENIX Association, 2016. (Cited on pages 214 and 222.)

[165] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs. *IACR Cryptology ePrint Archive*, 2010:445, 2010. (Cited on page 48.)

[166] Deukjo Hong, Jung-Keun Lee, Dong-Chan Kim, Daesung Kwon, Kwon Ho Ryu, and Donggeon Lee. LEA: A 128-Bit Block Cipher for Fast Encryption on Common Processors. In Yongdae Kim, Heejo Lee, and Adrian Perrig, editors, *Information Security Applications - 14th International Workshop, WISA 2013,*

Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers, volume 8267 of *Lecture Notes in Computer Science*, pages 3–27. Springer, 2013. (Cited on pages 65 and 71.)

[167] Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bonseok Koo, Changhoon Lee, Donghoon Chang, Jesang Lee, Kitae Jeong, Hyun Kim, Jongsung Kim, and Seongtaek Chee. HIGHT: A New Block Cipher Suitable for Low-Resource Device. In Goubin and Matsui [150], pages 46–59. (Cited on pages 65, 68, and 70.)

[168] Russell Housley. Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP). RFC 4309, Internet Engineering Task Force, December 2005. Available at `https://tools.ietf.org/html/rfc4309`. Accessed: September 2017. (Cited on page 132.)

[169] Michael Hutter and Jörn-Marc Schmidt. The Temperature Side Channel and Heating Fault Attacks. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 2013. (Cited on page 20.)

[170] IEEE. IEEE Standard for Low-Rate Wireless Networks. Available at `https://standards.ieee.org/about/get/802/802.15.html`. Accessed: September 2017. (Cited on pages xvii, 54, 67, 132, and 157.)

[171] Sebastiaan Indesteege, Nathan Keller, Orr Dunkelman, Eli Biham, and Bart Preneel. A Practical Attack on KeeLoq. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008. (Cited on page 5.)

[172] International Organization for Standardization. ISO/IEC 19772:2009. *Information Technology – Security Techniques – Authenticated Encryption*, February 2009. Available at `https://www.iso.org/standard/46345.html`. Accessed: September 2017. (Cited on page 8.)

[173] International Organization for Standardization. ISO/IEC 19772:2009. *Information Technology – Security Techniques – Lightweight Cryptography – Part 3: Stream Ciphers*, October 2012. Available at `https://www.iso.org/standard/56426.html`. Accessed: September 2017. (Cited on page 8.)

[174] Yuval Ishai, Amit Sahai, and David A. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003. (Cited on page 32.)

[175] Joshua Jaffe. A First-Order DPA Attack Against AES in Counter Mode with Unknown Initial Counter. In Paillier and Verbauwhede [270], pages 1–13. (Cited on pages 132, 134, 156, and 166.)

[176] Anthony Journault, François-Xavier Standaert, and Kerem Varici. Improving the Security and Efficiency of Block Ciphers Based on LS-Designs. *Designs, Codes and Cryptography*, 82(1-2):495–509, 2017. (Cited on pages 73 and 105.)

[177] Marc Joye and Amir Moradi, editors. *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*. Springer, 2015. (Cited on pages 203 and 212.)

[178] Marc Joye and Francis Olivier. Side-Channel Analysis. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 1198–1204. Springer, 2011. (Cited on page 20.)

[179] Marc Joye and Jean-Jacques Quisquater, editors. *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*. Springer, 2004. (Cited on pages 208, 213, and 222.)

[180] Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*. Springer, 2003. (Cited on pages 201 and 209.)

[181] David Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet.* Simon and Schuster, 1996. (Cited on pages 2 and 3.)

[182] Mohamed Karroumi, Benjamin Richard, and Marc Joye. Addition with Blinded Operands. In Prouff [281], pages 41–55. (Cited on page 32.)

[183] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* CRC Press, 2014. (Cited on pages 3, 4, and 5.)

[184] 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 [283], pages 390–407. (Cited on page 45.)

[185] Auguste Kerckhoffs. La Cryptographie Militaire. *Journal des Sciences Militaires*, IX:5–83, January 1883. Available at `http://www.petitcolas.net/kerckhoffs/crypto_militaire_1.pdf`. Accessed: September 2017. (Cited on page 4.)

[186] Khoongming Khoo, Thomas Peyrin, Axel York Poschmann, and Huihui Yap. FOAM: Searching for Hardware-Optimal SPN Structures and Components with a Fair Comparison. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 433–450. Springer, 2014. (Cited on pages 13 and 76.)

[187] Samuel T. King, editor. *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. USENIX Association, 2013. (Cited on pages 202 and 238.)

[188] Engin Kirda and Thomas Ristenpart, editors. *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 2017. (Cited on pages 202 and 224.)

[189] Ilya Kizhvatov. pysca: Toolbox for Advanced Differential Power Analysis of Symmetric Key Cryptographic Algorithm Implementations. Available at `https://github.com/ikizhvatov/pysca`. Accessed: September 2017. (Cited on page 30.)

[190] Ilya Kizhvatov. Side Channel Analysis of AVR XMEGA Crypto Engine. In Dimitrios N. Serpanos and Wayne H. Wolf, editors, *Proceedings of the 4th Workshop on Embedded Systems Security, WESS 2009, Grenoble, France, October 15, 2009*. ACM, 2009. (Cited on page 156.)

[191] Ilya Kizhvatov. *Physical Security of Cryptographic Algorithm Implementations*. PhD thesis, University of Luxembourg, 2011. (Cited on page 22.)

[192] kmarquet. BLOC – Source Code Developed in the BLOC Project. `https://github.com/kmarquet/bloc`. Accessed: September 2017. (Cited on page 46.)

[193] Miroslav Knezevic. *Efficient Hardware Implementations of Cryptographic Primitives (Efficiënte hardware implementaties van cryptografische primitieven)*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2011. (Cited on page 34.)

[194] Miroslav Knezevic, Ventzislav Nikov, and Peter Rombouts. Low-Latency Encryption - Is "Lightweight = Light + Wait"? In Prouff and Schaumont [283], pages 426–446. (Cited on pages 15 and 45.)

[195] Lars R. Knudsen and Huapeng Wu, editors. *Selected Areas in Cryptography, 19th International Conference, SAC 2012, Windsor, ON, Canada, August 15-16, 2012, Revised Selected Papers*, volume 7707 of *Lecture Notes in Computer Science*. Springer, 2013. (Cited on pages 206 and 235.)

[196] Çetin Kaya Koç and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*. Springer, 1999. (Cited on pages 216 and 217.)

[197] Çetin Kaya Koç and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*. Springer, 2000. (Cited on pages 224 and 232.)

[198] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. (Cited on pages 17, 20, and 173.)

[199] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Introduction to Differential Power Analysis and Related Attacks, 1998. Technical Report. Available at `https://www.rambus.com/introduction-to-differential-power-analysis-and-related-attacks/`. Accessed: September 2017. (Cited on pages 18, 20, 22, and 25.)

[200] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Wiener [382], pages 388–397. (Cited on pages 18, 20, 25, 26, 31, and 173.)

[201] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to Differential Power Analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011. (Cited on pages 22 and 25.)

[202] Paul C Kocher, Joshua M Jaffe, and Benjamin C Jun. Using Unpredictable Information to Minimize Leakage from Smartcards and Other Cryptosystems, December 2001. US Patent 6,327,661. (Cited on page 31.)

[203] Boris Köpf and David A. Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 286–296. ACM, 2007. (Cited on page 29.)

[204] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, Internet Engineering Task Force, February 1997. Available at `https://tools.ietf.org/html/rfc2104`. Accessed: September 2017. (Cited on page 8.)

[205] Markus G. Kuhn. Optical Time-Domain Eavesdropping Risks of CRT Displays. In *2002 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 12-15, 2002*, pages 3–18. IEEE Computer Society, 2002. (Cited on page 18.)

[206] Markus G. Kuhn. Compromising Emanations: Eavesdropping Risks of Computer Displays. Technical Report UCAM-CL-TR-577, University of Cambridge, Computer Laboratory, December 2003. Available at `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-577.pdf`. Accessed: September 2017. (Cited on pages 16, 17, and 18.)

[207] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 26(2):203–215, 2007. (Cited on page 11.)

[208] Das Labor. XBX Embedded Hashing. Available at `https://github.com/das-labor/xbx/`. Accessed: September 2017. (Cited on page 47.)

[209] Yee Wei Law, Jeroen Doumen, and Pieter H. Hartel. Survey and Benchmark of Block Ciphers for Wireless Sensor Networks. *TOSN*, 2(1):65–93, 2006. (Cited on page 45.)

[210] Gregor Leander, Brice Minaud, and Sondre Rønjom. A Generic Approach to Invariant Subspace Attacks: Cryptanalysis of Robin, iSCREAM and Zorro. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 254–283. Springer, 2015. (Cited on page 73.)

[211] Pil Joong Lee and Chae Hoon Lim, editors. *Information Security and Cryptology - ICISC 2002, 5th International Conference Seoul, Korea, November 28-29, 2002, Revised Papers*, volume 2587 of *Lecture Notes in Computer Science*. Springer, 2003. (Cited on pages 206 and 223.)

[212] Kerstin Lemke, Kai Schramm, and Christof Paar. DPA on n-Bit Sized Boolean and Arithmetic Operations and Its Application to IDEA, RC6, and the HMAC-Construction. In Joye and Quisquater [179], pages 205–219. (Cited on page 124.)

[213] Kerstin Lemke-Rust and Christof Paar. Gaussian Mixture Models for Higher-Order Side Channel Analysis. In Paillier and Verbauwhede [270], pages 14–27. (Cited on page 28.)

[214] Gaëtan Leurent. Improved Differential-Linear Cryptanalysis of 7-Round Chaskey with Partitioning. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 344–371. Springer, 2016. (Cited on page 70.)

[215] libsodium. The Sodium Crypto Library (libsodium). Available at `https://download.libsodium.org/doc/`. Accessed: April 2017. (Cited on page 137.)

[216] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In Holz and Savage [164], pages 549–564. (Cited on page 137.)

[217] Logic Friday. *Free Software for Boolean Logic Optimization, Analysis, and Synthesis*. Available at `https://sontrak.com/`. Accessed: September 2017. (Cited on page 185.)

[218] Victor Lomné, Emmanuel Prouff, and Thomas Roche. Behind the Scene of Side Channel Attacks. In Sako and Sarkar [303], pages 506–525. (Cited on pages 25, 30, and 169.)

[219] Jake Longo, Elke De Mulder, Dan Page, and Michael Tunstall. SoC It to EM: ElectroMagnetic Side-Channel Attacks on a Complex System-on-Chip. In Güneysu and Handschuh [156], pages 620–640. (Cited on pages 21 and 170.)

[220] LoRa Alliance. Wide Area Networks for IoT. Available at `https://www.lora-alliance.org/`. Accessed: April 2017. (Cited on page 132.)

[221] Joe Loughry and David A. Umphress. Information Leakage from Optical Emanations. *ACM Transactions on Information and System Security*, 5(3):262–289, 2002. (Cited on page 18.)

[222] Stefan Mangard. A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion. In Lee and Lim [211], pages 343–358. (Cited on page 25.)

[223] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007. (Cited on pages 18, 19, 21, 22, 23, 24, 25, 29, 174, and 185.)

[224] 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 *Lecture Notes in Computer Science*. Springer, 2010. (Cited on pages 203, 205, 210, 214, and 239.)

[225] George Marsaglia et al. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003. (Cited on page 191.)

[226] Keith Martin. *Everyday Cryptography: Fundamental Principles and Applications*. Oxford University Press, 2012. (Cited on page 6.)

[227] James L Massey. Guessing and Entropy. In *Proceedings of 1994 IEEE International Symposium on Information Theory*, page 204. IEEE, 1994. (Cited on page 29.)

[228] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal Covert Channels on Multi-core Platforms. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 865–880. USENIX Association, 2015. (Cited on page 18.)

[229] Luke Mather, Elisabeth Oswald, and Carolyn Whitnall. Multi-target DPA Attacks: Pushing DPA Beyond the Limits of a Desktop Computer. In Sarkar and Iwata [304], pages 243–261. (Cited on page 128.)

[230] Mitsuru Matsui. Linear Cryptanalysis Method for DES Cipher. In Tor Helleseth, editor, *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer, 1993. (Cited on page 117.)

[231] Mitsuru Matsui and Yumiko Murakami. Minimalism of Software Implementation - Extensive Performance Analysis of Symmetric Primitives on the RL78 Microcontroller. In Moriai [243], pages 393–409. (Cited on page 45.)

[232] Rita Mayer-Sommer. Smartly Analyzing the Simplicity and the Power of Simple Power Analysis on Smartcards. In Koç and Paar [197], pages 78–92. (Cited on pages 26 and 27.)

[233] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages. In Kirda and Ristenpart [188], pages 199–216. (Cited on pages 189 and 194.)

[234] David A. McGrew and John Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In Anne Canteaut and Kapalee Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer, 2004. (Cited on page 7.)

[235] Marcel Medwed, Christophe Petit, Francesco Regazzoni, Mathieu Renauld, and François-Xavier Standaert. Fresh Re-keying II: Securing Multiple Parties against Side-Channel and Fault Attacks. In Prouff [280], pages 115–132. (Cited on page 174.)

[236] 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, *Progress in Cryptology - AFRICACRYPT 2010, Third International Conference on Cryptology in Africa, Stellenbosch, South Africa, May 3-6, 2010. Proceedings*, volume 6055 of *Lecture Notes in Computer Science*, pages 279–296. Springer, 2010. (Cited on pages 31 and 174.)

[237] Florian Mendel, Vincent Rijmen, Deniz Toz, and Kerem Varici. Differential Analysis of the LED Block Cipher. In Wang and Sako [375], pages 190–207. (Cited on page 71.)

[238] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. (Cited on page 4.)

[239] Thomas S. Messerges. Securing the AES Finalists Against Power Analysis Attacks. In Bruce Schneier, editor, *Fast Software Encryption, 7th International*

Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings, volume 1978 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2000. (Cited on page 32.)

[240] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Investigations of Power Analysis Attacks on Smartcards. In Scott B. Guthery and Peter Honeyman, editors, *Proceedings of the 1st Workshop on Smartcard Technology, Smartcard 1999, Chicago, Illinois, USA, May 10-11, 1999.* USENIX Association, 1999. (Cited on page 26.)

[241] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Examining Smart-Card Security under the Threat of Power Analysis Attacks. *IEEE Transactions on Computers*, 51(5):541–552, 2002. (Cited on page 26.)

[242] Microsoft. Internet of Things Security Architecture, July 2017. Available at `https://docs.microsoft.com/en-us/azure/iot-suite/iot-security-architecture`. Accessed: September 2017. (Cited on page 161.)

[243] Shiho Moriai, editor. *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*. Springer, 2014. (Cited on pages 211 and 224.)

[244] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers. In Antoine Joux and Amr M. Youssef, editors, *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, volume 8781 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2014. (Cited on pages 65, 70, and 109.)

[245] Steven J. Murdoch. Hot or Not: Revealing Hidden Services by their Clock Skew. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 27–36. ACM, 2006. (Cited on page 18.)

[246] National Institute of Standards and Technology (NIST). Hash Functions. Available at `http://csrc.nist.gov/groups/ST/hash/sha-3/`. Accessed: September 2017. (Cited on pages 44, 47, 48, and 64.)

[247] National Institute of Standards and Technology (NIST). Lightweight Cryptography. Available at `https://csrc.nist.gov/Projects/Lightweight-Cryptography`. Accessed: September 2017. (Cited on pages 37, 44, and 64.)

[248] National Institute of Standards and Technology (NIST). Lightweight Cryptography Workshop 2015. Available at `https://www.nist.gov/news-events/events/2015/07/lightweight-cryptography-workshop-2015`. Accessed: September 2017. (Cited on pages 37 and 44.)

[249] National Institute of Standards and Technology (NIST). Lightweight Cryptography Workshop 2016. Available at https://www.nist.gov/news-events/events/2016/10/lightweight-cryptography-workshop-2016. Accessed: September 2017. (Cited on pages 37 and 44.)

[250] National Institute of Standards and Technology (NIST). Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication (FIPS) 197, 2001. (Cited on pages 44, 64, 65, 68, 69, 132, and 134.)

[251] Nettle. Nettle – A Low-Level Cryptographic Library. Available at http://www.lysator.liu.se/~nisse/nettle/. Accessed: April 2017. (Cited on page 137.)

[252] NewAE. ChipWhisperer. Available at https://newae.com/tools/chipwhisperer/. Accessed: September 2017. (Cited on pages 29, 155, and 171.)

[253] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006. (Cited on page 32.)

[254] Thank you Bob Anderson, September 1994. Email sent to cypherpunk mailing list from nobody@jpunix.com. Available at https://web.archive.org/web/20010722163902/http://cypherpunks.venona.com/date/1994/09/msg00304.html. Accessed: September 2017. (Cited on page 5.)

[255] Karsten Nohl, David Evans, Starbug, and Henryk Plötz. Reverse-Engineering a Cryptographic RFID Tag. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 185–194. USENIX Association, 2008. (Cited on page 19.)

[256] Matt Novak. Hackers Shut Down The Key Card Machine In This Hotel Until a Bitcoin Ransom Was Paid [Corrected], January 2017. Available at http://gizmodo.com/hackers-locked-every-room-in-this-hotel-until-a-bitcoin-1791769502. Accessed: September 2017. (Cited on page 155.)

[257] The Editors of Encyclopædia Britannica. Personal Computer (PC). *Encyclopædia Britannica*, November 2016. Available at https://www.britannica.com/technology/personal-computer. Accessed: September 2017. (Cited on page 3.)

[258] Colin O'Flynn and Zhizhang Chen. A Case Study of Side-Channel Analysis Using Decoupling Capacitor Power Measurement with the OpenADC. In Joaquín García-Alfaro, Frédéric Cuppens, Nora Cuppens-Boulahia, Ali Miri,

and Nadia Tawbi, editors, *Foundations and Practice of Security - 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers*, volume 7743 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2012. (Cited on page 21.)

[259] Colin O'Flynn and Zhizhang Chen. Power Analysis Attacks Against IEEE 802.15.4 Nodes. In Standaert and Oswald [337], pages 55–70. (Cited on pages 132, 133, 156, and 157.)

[260] Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research. In Prouff [281], pages 243–260. (Cited on page 29.)

[261] OpenSSL. Cryptography and SSL/TLS Toolkit. Available at `https://www.openssl.org/`. Accessed: April 2017. (Cited on pages 133 and 137.)

[262] OpenSSL. OpenSSL – TLS/SSL and Crypto Library. Available at `https://github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c`. Accessed: April 2017. (Cited on page 137.)

[263] OpenThread. OpenThread: An Open-Source Implementation of the Thread Networking Protocol. Available at `https://github.com/openthread/openthread`. Accessed: September 2017, 2016. (Cited on pages 157, 158, and 167.)

[264] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006. (Cited on page 137.)

[265] David Oswald, Bastian Richter, and Christof Paar. Side-Channel Attacks on the Yubikey 2 One-Time Password Generator. In Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright, editors, *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings*, volume 8145 of *Lecture Notes in Computer Science*, pages 204–222. Springer, 2013. (Cited on page 21.)

[266] Elisabeth Oswald and Stefan Mangard. Template Attacks on Masking - Resistance Is Futile. In Abe [1], pages 243–256. (Cited on page 28.)

[267] 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. (Cited on pages 207 and 215.)

[268] Onur Özen, Kerem Varici, Cihangir Tezcan, and Çelebi Kocair. Lightweight Block Ciphers Revisited: Cryptanalysis of Reduced Round PRESENT and

HIGHT. In Colin Boyd and Juan Manuel González Nieto, editors, *Information Security and Privacy, 14th Australasian Conference, ACISP 2009, Brisbane, Australia, July 1-3, 2009, Proceedings*, volume 5594 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 2009. (Cited on page 70.)

[269] Daniel Page. *A Practical Introduction to Computer Architecture*. Springer Science & Business Media, 2009. (Cited on page 15.)

[270] 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 *Lecture Notes in Computer Science*. Springer, 2007. (Cited on pages 207, 219, and 222.)

[271] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the Gap: Towards Secure 1st-Order Masking in Software. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017. (Cited on pages 189, 190, and 194.)

[272] Eric Peeters. *Advanced DPA Theory and Practice*. Springer, 2013. (Cited on page 18.)

[273] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. SPINS: Security Protocols for Sensor Networks. *Wireless Networks*, 8(5):521–534, 2002. (Cited on page 72.)

[274] Thomas Peyrin, editor. *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*. Springer, 2016. (Cited on pages 85, 209, and 230.)

[275] Pico Technology. Pico Oscilloscope Range. Available at `https://www.picotech.com/products/oscilloscope`. Accessed: September 2017. (Cited on page 171.)

[276] Gilles Piret, Thomas Roche, and Claude Carlet. PICARO - A Block Cipher Allowing Efficient Higher-Order Side-Channel Resistance. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings*, volume 7341 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 2012. (Cited on page 114.)

[277] Axel York Poschmann. *Lightweight Cryptography: Cryptographic Engineering for a Pervasive World*. PhD thesis, Ruhr University Bochum, 2009. (Cited on pages 13 and 34.)

[278] 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. (Cited on pages 217 and 233.)

[279] Emmanuel Prouff. DPA Attacks and S-Boxes. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 2005. (Cited on page 117.)

[280] Emmanuel Prouff, editor. *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*, volume 7079 of *Lecture Notes in Computer Science*. Springer, 2011. (Cited on pages 224 and 231.)

[281] Emmanuel Prouff, editor. *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, volume 8622 of *Lecture Notes in Computer Science*. Springer, 2014. (Cited on pages 209, 219, 227, and 230.)

[282] Emmanuel Prouff and Matthieu Rivain. Theoretical and Practical Aspects of Mutual Information Based Side Channel Analysis. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*, volume 5536 of *Lecture Notes in Computer Science*, pages 499–518, 2009. (Cited on page 27.)

[283] 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. (Cited on pages 219 and 220.)

[284] Public Comments Received on "Profiles for the Lightweight Cryptography Standardization Process". https://www.nist.gov/sites/default/files/documents/2017/06/20/public-comments-profiles-i-ii-june2017.pdf, June 2017. Accessed: September 2017. (Cited on page 182.)

[285] Quininer Kel. sparx-cipher: Another SPARX Block Cipher Implementation for Rust. Available at https://github.com/quininer/sparx-cipher. Accessed: September 2017. (Cited on page 94.)

[286] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001. (Cited on page 20.)

[287] Rambus. DPA Workstation Analysis Platform. Available at https://www.rambus.com/security/dpa-countermeasures/dpa-workstation-platform/. Accessed: September 2017. (Cited on page 29.)

[288] Randombit. mbed TLS. Available at https://botan.randombit.net/. Accessed: April 2017. (Cited on page 137.)

[289] 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 *Lecture Notes in Computer Science*. Springer, 2005. (Cited on pages 201 and 231.)

[290] Christian Rechberger and Elisabeth Oswald. Practical Template Attacks. In Chae Hoon Lim and Moti Yung, editors, *Information Security Applications, 5th International Workshop, WISA 2004, Jeju Island, Korea, August 23-25, 2004, Revised Selected Papers*, volume 3325 of *Lecture Notes in Computer Science*, pages 440–456. Springer, 2004. (Cited on page 21.)

[291] Oscar Reparaz. Detecting Flawed Masking Schemes with Leakage Detection Tests. In Peyrin [274], pages 204–222. (Cited on pages 189 and 194.)

[292] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating Masking Schemes. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015. (Cited on page 32.)

[293] Oscar Reparaz, Benedikt Gierlichs, and Ingrid Verbauwhede. A Note on the Use of Margins to Compare Distinguishers. In Prouff [281], pages 1–8. (Cited on page 120.)

[294] Riscure. Inspector SCA. Available at https://www.riscure.com/security-tools/inspector-sca/. Accessed: September 2017. (Cited on page 29.)

[295] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In 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 *Lecture Notes in Computer Science*, pages 171–188. Springer, 2009. (Cited on page 30.)

[296] Ronald L. Rivest. The RC5 Encryption Algorithm. In Bart Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96. Springer, 1994. (Cited on pages 65, 68, 72, and 114.)

[297] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. (Cited on page 9.)

[298] Matthew Robshaw and Jonathan Katz, editors. *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*. Springer, 2016. (Cited on pages 205 and 232.)

[299] Thomas Roche, Victor Lomné, and Karim Khalfallah. Combined Fault and Side-Channel Attack on Protected Implementations of AES. In Prouff [280], pages 65–83. (Cited on page 19.)

[300] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 195–212. IEEE Computer Society, 2017. (Cited on pages 156, 172, and 180.)

[301] Michael Rushanan, Aviel D. Rubin, Denis Foo Kune, and Colleen M. Swanson. SoK: Security and Privacy in Implantable Medical Devices and Body Area Networks. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 524–539. IEEE Computer Society, 2014. (Cited on page 156.)

[302] Sami Saab, Pankaj Rohatgi, and Craig Hampel. Side-Channel Protections for Cryptographic Instruction Set Extensions. *IACR Cryptology ePrint Archive*, 2016:700, 2016. (Cited on page 137.)

[303] Kazue Sako and Palash Sarkar, editors. *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part I*, volume 8269 of *Lecture Notes in Computer Science*. Springer, 2013. (Cited on pages 211 and 223.)

[304] Palash Sarkar and Tetsu Iwata, editors. *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*. Springer, 2014. (Cited on pages 208 and 223.)

[305] Naveen Sastry and David Wagner. Security Considerations for IEEE 802.15.4 Networks. In Markus Jakobsson and Adrian Perrig, editors, *Proceedings of the 2004 ACM Workshop on Wireless Security, Philadelphia, PA, USA, October 1, 2004*, pages 32–42. ACM, 2004. (Cited on page 132.)

[306] Patrick Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010. (Cited on page 10.)

[307] Werner Schindler, Kerstin Lemke, and Christof Paar. A Stochastic Model for Differential Side Channel Cryptanalysis. In Rao and Sunar [289], pages 30–46. (Cited on pages 28 and 30.)

[308] Tobias Schneider and Amir Moradi. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In Güneysu and Handschuh [156], pages 495–513. (Cited on page 189.)

[309] Tobias Schneider, Amir Moradi, and Tim Güneysu. ParTI - Towards Combined Hardware Countermeasures Against Side-Channel and Fault-Injection Attacks. In Robshaw and Katz [298], pages 302–332. (Cited on page 30.)

[310] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2007. (Cited on page 3.)

[311] Bruce Schneier. The Discovery of TEMPEST, January 2009. Available at `https://www.schneier.com/blog/archives/2009/01/the_discovery_o.html`. Accessed: September 2017. (Cited on page 16.)

[312] Kai Schramm, Thomas J. Wollinger, and Christof Paar. A New Class of Collision Attacks and Its Application to DES. In Thomas Johansson, editor, *Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers*, volume 2887 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 2003. (Cited on page 28.)

[313] Torsten Schütze. Side-Channel Analysis (SCA) – A Comparative Approach on Smart Cards, Embedded Systems, and High Security Solutions, July 2010. Presented at Workshop on Applied Cryptography, Nanyang Technological University, Singapore. Available at `http://www1.spms.ntu.edu.sg/~ccrg/WAC2010/slides/session_3/3_1_Schuetze_SCA-ComparativeApproach.pdf`. Accessed: September 2017. (Cited on page 18.)

[314] Peter Schwabe and Ko Stoffelen. All the AES you need on cortex-m3 and M4. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2016. (Cited on pages 81 and 136.)

[315] Ravikumar Selvam, Dillibabu Shanmugam, and Suganya Annadurai. Vulnerability analysis of PRINCE and RECTANGLE using CPA. In Jianying Zhou and Douglas Jones, editors, *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security, CPSS 2015, Singapore, Republic of Singapore, April 14 - March 14, 2015*, pages 81–87. ACM, 2015. (Cited on page 114.)

[316] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979. (Cited on page 31.)

[317] Adi Shamir. Protecting Smart Cards from Passive Power Analysis with Detached Power Supplies. In Koç and Paar [197], pages 71–77. (Cited on page 31.)

[318] Dillibabu Shanmugam, Ravikumar Selvam, and Suganya Annadurai. Differential Power Analysis Attack on SIMON and LED Block Ciphers. In Rajat Subhra Chakraborty, Vashek Matyas, and Patrick Schaumont, editors, *Security, Privacy, and Applied Cryptography Engineering - 4th International Conference, SPACE 2014, Pune, India, October 18-22, 2014. Proceedings*, volume 8804 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2014. (Cited on page 114.)

[319] Claude E Shannon. A Mathematical Theory of Communication. *Bell Systems Technical Journal*, 27:623–656, 1948. (Cited on page 3.)

[320] Claude E Shannon. Communication Theory of Secrecy Systems. *Bell Labs Technical Journal*, 28(4):656–715, 1949. (Cited on pages 3 and 7.)

[321] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: An Ultra-Lightweight Blockcipher. In Preneel and Takagi [278], pages 342–357. (Cited on pages 65, 68, 71, and 114.)

[322] Robert Shirey. Internet Security Glossary. RFC 2828, Internet Engineering Task Force, May 2000. Available at `http://www.rfc-editor.org/rfc/rfc2828.txt`. Accessed: September 2017. (Cited on page 1.)

[323] SideChannelMarvels. Daredevil. Available at `https://github.com/SideChannelMarvels/Daredevil`. Accessed: September 2017. (Cited on pages 30, 155, and 171.)

[324] Gustavus J Simmons. Cryptology. *Encyclopædia Britannica*, August 1988. Available at `https://www.britannica.com/topic/cryptology`. Accessed: August 2017. (Cited on page 1.)

[325] Simon Singh. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor, 2000. (Cited on page 2.)

[326] Dmitry Sklyarov and Andy Malyshev. eBooks Security – Theory and Practice, July 2001. Presented at DEF CON Nine, Las Vegas, Nevada, USA. Available at `https://www.defcon.org/html/defcon-9/defcon-9-speakers.html#Dmitry%20Sklyarov`. Accessed: September 2017. (Cited on page 5.)

[327] Sergei Skorobogatov. Low Temperature Data Remanence in Static RAM. Technical Report UCAM-CL-TR-536, University of Cambridge, Computer Laboratory, June 2002. Available at `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf`. Accessed: September 2017. (Cited on page 19.)

[328] Sergei P. Skorobogatov. Using Optical Emission Analysis for Estimating Contribution to Power Analysis. In Luca Breveglieri, Israel Koren, David Naccache, Elisabeth Oswald, and Jean-Pierre Seifert, editors, *Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009,*

Lausanne, Switzerland, 6 September 2009, pages 111–119. IEEE Computer Society, 2009. (Cited on page 20.)

[329] Michael John Sebastian Smith. *Application-Specific Integrated Circuits*. Addison-Wesley Professional, 2008. (Cited on page 11.)

[330] SOG-IS. Joint Interpretation Library – Application of Attack Potential to Smartcards. Version 2.9, January 2013. Available at `http://www.sogis.org/documents/cc/domains/sc/JIL-Application-of-Attack-Potential-to-Smartcards-v2-9.pdf`. Accessed: September 2017. (Cited on pages xx, 172, and 173.)

[331] Junhyuk Song, Radha Poovendran, Jicheol Lee, and Tetsu Iwata. The AES-CMAC Algorithm. RFC 4493, Internet Engineering Task Force, June 2006. Available at `https://tools.ietf.org/html/rfc4493`. Accessed: September 2017. (Cited on page 132.)

[332] Ling Song, Zhangjie Huang, and Qianqian Yang. Automatic Differential Analysis of ARX Block Ciphers with Application to SPECK and LEA. In Joseph K. Liu and Ron Steinfeld, editors, *Information Security and Privacy - 21st Australasian Conference, ACISP 2016, Melbourne, VIC, Australia, July 4-6, 2016, Proceedings, Part II*, volume 9723 of *Lecture Notes in Computer Science*, pages 379–394. Springer, 2016. (Cited on page 74.)

[333] François-Xavier Standaert. How (not) to Use Welch's T-test in Side-Channel Security Evaluations. *IACR Cryptology ePrint Archive*, 2017:138, 2017. (Cited on page 189.)

[334] François-Xavier Standaert, Lejla Batina, Elke De Mulder, Kerstin Lemke, Stefan Mangard, Elisabeth Oswald, and Gilles Piret. Electromagnetic Analysis and Fault Attacks: State of the Art. ECRYPT deliverable D.VAM.4. Revision 3, May 2005. Available at `http://www.ecrypt.eu.org/ecrypt1/documents/D.VAM.4-3.pdf`. Accessed: September 2017. (Cited on pages 18, 23, and 24.)

[335] François-Xavier Standaert, Lejla Batina, Elke De Mulder, Kerstin Lemke, Nele Mentens, Elisabeth Oswald, and Eric Peeters. Report on DPA and EMA attacks on FPGAs. ECRYPT deliverable D.VAM.5. Revision 1, July 2005. Available at `http://www.ecrypt.eu.org/ecrypt1/documents/D.VAM.5-1.pdf`. Accessed: September 2017. (Cited on page 24.)

[336] François-Xavier Standaert, Tal Malkin, and Moti Yung. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 443–461. Springer, 2009. (Cited on pages 28 and 119.)

[337] François-Xavier Standaert and Elisabeth Oswald, editors. *Constructive Side-Channel Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers*, volume 9689 of *Lecture Notes in Computer Science*. Springer, 2016. (Cited on pages 205, 211, 212, and 227.)

[338] 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 - Foundations and Practice*, Information Security and Cryptography, pages 99–134. Springer, 2010. (Cited on page 31.)

[339] Didier Stevens. ROT13 is used in Windows? You're joking!, July 2006. Available at `https://blog.didierstevens.com/2006/07/24/rot13-is-used-in-windows-you%E2%80%99re-joking/`. Accessed: September 2017. (Cited on page 5.)

[340] STMicroelectronics. STM32 MCU Nucleo. Available at `http://www.st.com/en/evaluation-tools/stm32-mcu-nucleo.html`. Accessed: April 2017. (Cited on pages 133 and 137.)

[341] Ko Stoffelen. Intrinsic Side-Channel Analysis Resistance and Efficient Masking, 2015. Master's thesis, Radboud University. (Cited on page 118.)

[342] Daehyun Strobel, Benedikt Driessen, Timo Kasper, Gregor Leander, David Oswald, Falk Schellenberg, and Christof Paar. Fuming Acid and Cryptanalysis: Handy Tools for Overcoming a Digital Locking and Access Control System. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 2013. (Cited on page 155.)

[343] Student. The Probable Error of a Mean. *Biometrika*, pages 1–25, 1908. Available at `http://www.jstor.org/stable/2331554`. Accessed: September 2017. (Cited on page 33.)

[344] Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi. TWINE: A Lightweight Block Cipher for Multiple Platforms. In Knudsen and Wu [195], pages 339–354. (Cited on pages 65, 68, and 74.)

[345] Daisuke Suzuki and Minoru Saeki. Security Evaluation of DPA Countermeasures Using Dual-Rail Pre-charge Logic Style. In Goubin and Matsui [150], pages 255–269. (Cited on page 31.)

[346] Jürgen Teich. Hardware/Software Codesign: The Past, the Present, and Predicting the Future. *Proceedings of the IEEE*, 100(Centennial-Issue):1411–1430, 2012. (Cited on page 10.)

[347] Teledyne LeCroy. WaveRunner 625Zi. Available at http://teledynelecroy.com/oscilloscope/oscilloscopemodel.aspx?modelid=4779. Accessed: September 2017. (Cited on page 167.)

[348] Texas Instruments. MSP430F1611. http://www.ti.com/lit/ds/symlink/msp430f1611.pdf. Accessed: September 2017. (Cited on pages 45, 46, 57, and 58.)

[349] Texas Instruments. CC2538 Powerful Wireless Microcontroller System-On-Chip for 2.4-GHz IEEE 802.15.4, 6LoWPAN, and ZigBee Applications, April 2015. Available at http://www.ti.com/lit/ds/symlink/cc2538.pdf. Accessed: September 2017. (Cited on pages 167 and 168.)

[350] James Thrasher. RFID vs. NFC: What's the Difference?, October 2013. Available at http://blog.atlasrfidstore.com/rfid-vs-nfc. Accessed: September 2017. (Cited on page 35.)

[351] Thread Group. Thread. Available at https://www.threadgroup.org/. Accessed: September 2017. (Cited on pages 155 and 157.)

[352] Thread Group. Thread Certified Products. Available at http://threadgroup.org/technology/ourtechnology#certifiedproducts. Accessed: September 2017. (Cited on page 167.)

[353] Thread Group. Thread Group Broadens Focus to Encompass the Places Where People Live and Work with Expansion Into Commercial Building Space, November 2016. Available at http://threadgroup.org/news-events/press-releases/ID/124/Thread-Group-Broadens-Focus-to-Encompass-the-Places-Where-People-Live-and-Work-with-Expansion-Into-Commercial-Building-Space. Accessed: September 2017. (Cited on page 157.)

[354] Kris Tiri, Moonmoon Akmal, and Ingrid Verbauwhede. A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards. In *Proceedings of the 28th European Solid-State Circuits Conference, 2002. ESSCIRC 2002, Florence, Italy, September, 24 - 26, 2002*, pages 403–406. IEEE, 2002. (Cited on page 31.)

[355] Kris Tiri and Ingrid Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, pages 246–251. IEEE Computer Society, 2004. (Cited on page 31.)

[356] Ben L Titzer, Daniel K Lee, and Jens Palsberg. Avrora - The AVR Simulation and Analysis Framework. Available at http://compilers.cs.ucla.edu/avrora/. Accessed: September 2017. (Cited on page 61.)

[357] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks, IPSN 2005, April 25-27, 2005, UCLA, Los Angeles, California, USA*, pages 477–482. IEEE, 2005. (Cited on page 61.)

[358] Elena Trichina. Combinational Logic Design for AES SubByte Transformation on Masked Data. *IACR Cryptology ePrint Archive*, 2003:236, 2003. (Cited on pages 32, 180, 181, and 186.)

[359] George Mason University. ATHENa - Automated Tool for Hardware EvaluatioN. Available at `http://cryptography.gmu.edu/athena/`. Accessed: September 2017. (Cited on page 48.)

[360] U.S. Department Of Commerce/National Institute of Standards and Technology. Data Encryption Standard (DES). *Federal Information Processing Standards Publication. FIPS PUB 46-3*, pages 1–27, January 1977. Available at `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`. Accessed: September 2017. (Cited on page 3.)

[361] U.S. Department Of Commerce/National Institute of Standards and Technology. Advanced Encryption Standard (AES). *Federal Information Processing Standards Publication. FIPS PUB 197*, pages 1–51, November 2001. Available at `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`. Accessed: September 2017. (Cited on page 7.)

[362] U.S. Department Of Commerce/National Institute of Standards and Technology. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. *NIST Special Publication 800-38C*, pages 1–27, May 2004. Available at `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38c.pdf`. Accessed: September 2017. (Cited on page 7.)

[363] U.S. Department Of Commerce/National Institute of Standards and Technology. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. *NIST Special Publication 800-38D*, pages 1–39, November 2007. Available at `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf`. Accessed: September 2017. (Cited on page 7.)

[364] U.S. Department Of Commerce/National Institute of Standards and Technology. Secure Hash Standard (SHS). *Federal Information Processing Standards Publication. FIPS PUB 180-4*, pages 1–39, March 2012. Available at `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`. Accessed: September 2017. (Cited on page 8.)

[365] U.S. Department Of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). *NIST Federal Information*

Processing Standard. FIPS PUB 186-4, pages 1–130, July 2013. Available at http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf. Accessed: September 2017. (Cited on page 9.)

[366] U.S. Department Of Commerce/National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. *Federal Information Processing Standards Publication. FIPS PUB 202*, pages 1–37, August 2015. Available at http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. Accessed: September 2017. (Cited on page 8.)

[367] Praveen Kumar Vadnala. *Provably Secure Countermeasures against Side-channel Attacks*. PhD thesis, University of Luxembourg, 2015. (Cited on pages 181 and 186.)

[368] Praveen Kumar Vadnala. Time-Memory Trade-Offs for Side-Channel Resistant Implementations of Block Ciphers. In Helena Handschuh, editor, *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*, volume 10159 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2017. (Cited on page 152.)

[369] Wim van Eck. Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk? *Computers & Security*, 4(4):269–286, 1985. (Cited on page 17.)

[370] Joel VanLaven, Mark Brehob, and Kevin J. Compton. Side Channel Analysis, Fault Injection and Applications - A Computationally Feasible SPA Attack on AES via Optimized Search. In Ryôichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *Security and Privacy in the Age of Ubiquitous Computing, IFIP TC11 20th International Conference on Information Security (SEC 2005), May 30 - June 1, 2005, Chiba, Japan*, volume 181 of *IFIP*, pages 577–588. Springer, 2005. (Cited on page 25.)

[371] Serge Vaudenay. *A Classical Introduction to Cryptography: Applications for Communications Security*. Springer Science & Business Media, 2006. (Cited on pages 2 and 5.)

[372] Roel Verdult, Flavio D. Garcia, and Baris Ege. Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer. In King [187], pages 703–718. (Cited on page 5.)

[373] 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 [375], pages 740–757. (Cited on pages 24 and 31.)

[374] John Von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. (Cited on page 3.)

[375] 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. (Cited on pages 207, 224, and 238.)

[376] Yanfeng Wang and Wenling Wu. Improved Multidimensional Zero-Correlation Linear Cryptanalysis and Applications to LBlock and TWINE. In Willy Susilo and Yi Mu, editors, *Information Security and Privacy - 19th Australasian Conference, ACISP 2014, Wollongong, NSW, Australia, July 7-9, 2014. Proceedings*, volume 8544 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2014. (Cited on page 74.)

[377] Bernard L Welch. The Generalization of Student's Problem when Several Different Population Variances are Involved. *Biometrika*, 34(1/2):28–35, 1947. Available at `http://www.jstor.org/stable/2332510`. Accessed: September 2017. (Cited on page 33.)

[378] Christian Wenzel-Benner and Jens Gräf. XBX: eXternal Benchmarking eXtension for the SUPERCOP Crypto Benchmarking Framework. In Mangard and Standaert [224], pages 294–305. (Cited on page 47.)

[379] Christian Wenzel-Benner, Jens Gräf, John Pham, and Jens-Peter Kaps. XBX Benchmarking Results January 2012. In *Third SHA-3 Candidate Conference (March 2012)*, 2012. (Cited on page 47.)

[380] Doug Whiting, Russell Housley, and Niels Ferguson. Counter with CBC-MAC (CCM). RFC 3610, Internet Engineering Task Force, September 2003. Available at `https://tools.ietf.org/html/rfc3610`. Accessed: September 2017. (Cited on pages 7 and 132.)

[381] Carolyn Whitnall and Elisabeth Oswald. A Comprehensive Evaluation of Mutual Information Analysis Using a Fair Evaluation Framework. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 316–334. Springer, 2011. (Cited on page 27.)

[382] 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. (Cited on pages 209 and 221.)

[383] wolfSSL. wolfCrypt Embedded Crypto Engine. Available at `https://www.wolfssl.com/wolfSSL/Products-wolfcrypt.html`. Accessed: April 2017. (Cited on page 137.)

[384] Peter Wright. *Spycatcher: The Candid Autobiography of a Senior Intelligence Officer*. Heinemann Publishers Australia, 1987. (Cited on page 17.)

[385] Fred B. Wrixon. *Codes, Ciphers & Other Cryptic & Clandestine Communication: Making and Breaking Secret Messages from Hieroglyphs to the Internet.* Black Dog & Leventhal Pub, 1998. (Cited on page 2.)

[386] Wenling Wu and Lei Zhang. LBlock: A Lightweight Block Cipher. In Javier Lopez and Gene Tsudik, editors, *Applied Cryptography and Network Security - 9th International Conference, ACNS 2011, Nerja, Spain, June 7-10, 2011. Proceedings*, volume 6715 of *Lecture Notes in Computer Science*, pages 327–344, 2011. (Cited on pages 65, 68, 71, and 114.)

[387] Qianqian Yang, Lei Hu, Siwei Sun, Kexin Qiao, Ling Song, Jinyong Shan, and Xiaoshuang Ma. Improved Differential Analysis of Block Cipher PRIDE. In Javier Lopez and Yongdong Wu, editors, *Information Security Practice and Experience - 11th International Conference, ISPEC 2015, Beijing, China, May 5-8, 2015. Proceedings*, volume 9065 of *Lecture Notes in Computer Science*, pages 209–219. Springer, 2015. (Cited on page 72.)

[388] Qianqian Yang, Lei Hu, Siwei Sun, and Ling Song. Extension of Meet-in-the-Middle Technique for Truncated Differential and Its Application to RoadRunneR. In Jiageng Chen, Vincenzo Piuri, Chunhua Su, and Moti Yung, editors, *Network and System Security - 10th International Conference, NSS 2016, Taipei, Taiwan, September 28-30, 2016, Proceedings*, volume 9955 of *Lecture Notes in Computer Science*, pages 398–411. Springer, 2016. (Cited on page 73.)

[389] John L. Young. NSA TEMPEST Documents. Available at `https://cryptome.org/nsa-tempest.htm`. Accessed: September 2017. (Cited on page 18.)

[390] John L. Young. TEMPEST Timeline. Available at `https://cryptome.org/tempest-time.htm`. Accessed: September 2017. (Cited on page 18.)

[391] Wentao Zhang, Zhenzhen Bao, Dongdai Lin, Vincent Rijmen, Bohan Yang, and Ingrid Verbauwhede. RECTANGLE: A Bit-slice Lightweight Block Cipher Suitable for Multiple Platforms. *SCIENCE CHINA Information Sciences*, 58(12):1–15, 2015. (Cited on pages 65, 73, 108, 109, 189, and 193.)

[392] Li Zhuang, Feng Zhou, and J. D. Tygar. Keyboard Acoustic Emanations Revisited. In Vijay Atluri, Catherine A. Meadows, and Ari Juels, editors, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 373–382. ACM, 2005. (Cited on page 18.)

[393] ZigBee Alliance. ZigBee Wireless Standard. Available at `http://www.zigbee.org/`. Accessed: September 2017. (Cited on pages 54, 56, and 67.)

[394] Michael Zohner, Michael Kasper, and Marc Stöttinger. Butterfly-Attack on Skein's Modular Addition. In Werner Schindler and Sorin A. Huss, editors,

Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings, volume 7275 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 2012. (Cited on page 128.)

[395] Michael Zohner, Michael Kasper, Marc Stöttinger, and Sorin A. Huss. Side Channel Analysis of the SHA-3 Finalists. In Wolfgang Rosenstiel and Lothar Thiele, editors, *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, pages 1012–1017. IEEE, 2012. (Cited on page 115.)

Publications

- Daniel Dinu, Alex Biryukov, Johann Großschädl, Dmitry Khovratovich, Yann Le Corre, and Léo Perrin. FELICS – Fair Evaluation of Lightweight Cryptographic Systems. *NIST Workshop on Lightweight Cryptography.* 2015.

  Artifacts: https://www.cryptolux.org/index.php/FELICS

- Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. *IEEE European Symposium on Security and Privacy – EuroS&P 2016.* pages 292–302, IEEE, 2016.

  Internet Engineering Task Force (IETF), Active Internet-Draft: https://datatracker.ietf.org/doc/draft-irtf-cfrg-argon2/

  Artifacts: https://github.com/P-H-C/phc-winner-argon2

- Alex Biryukov, Daniel Dinu, and Johann Großschädl. Correlation Power Analysis of Lightweight Block Ciphers: From Theory to Practice. *Applied Cryptography and Network Security – ACNS 2016.* Lecture Notes in Computer Science, volume 9696, pages 537–557, Springer, 2016.

- Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design Strategies for ARX with Provable Bounds: Sparx and LAX. *Advances in Cryptology – ASIACRYPT 2016.* Lecture Notes in Computer Science, volume 10031, pages 484–513, Springer, 2016.

  Artifacts: https://www.cryptolux.org/index.php/SPARX

- Alex Biryukov, Daniel Dinu, and Yann Le Corre. Side-Channel Attacks Meet Secure Network Protocols. *Applied Cryptography and Network Security – ACNS 2017.* Lecture Notes in Computer Science, volume 10355, pages 435–454, Springer, 2017.

  Artifacts: https://github.com/cryptolu/aes-cpa

- Daniel Dinu, Johann Großschädl, and Yann Le Corre. Efficient Masking of ARX-Based Block Ciphers Using Carry-Save Addition on Boolean Shares. *Information Security Conference – ISC 2017.* Lecture Notes in Computer Science, volume 10599, pages 39–57, Springer, 2017.

- Alex Biryukov, Daniel Dinu, Yann Le Corre, and Aleksei Udovenko. Optimal First-Order Boolean Masking for Embedded IoT Devices. *Smart Card Research and Advanced Application Conference – CARDIS 2017*. To appear.

  Artifacts: https://github.com/cryptolu/ofom

- Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. Triathlon of Lightweight Block Ciphers for the Internet of Things. *Journal of Cryptographic Engineering – JCEN*. To appear.

- Daniel Dinu. SoK: Efficient and Secure Lightweight Symmetric Cryptography for Embedded IoT Systems. In submission.

- Daniel Dinu and Ilya Kizhvatov. EM Analysis in the IoT Context: Lessons Learned from an Attack on Thread. In submission.

Appendices

A Assembly Code for Basic 8-bit Rotations

MCU	**AVR**	**MSP**	**ARM**
Data	**a**	**a**	**a**
$\lll 1$	1. **lsl a** 2. **adc a, R1**	1. **rla.b a** 2. **adc.b a**	1. **bfi a, #8, #7** 2. **ror a, a, #7**
$\lll 4$	1. **swap a**	–	–
$\ggg 1$	1. **bst a, 0** 2. **ror a** 3. **bld a, 7**	1. **bit #1, a** 2. **rrc.b a**	1. **bfi a, #8, #1** 2. **ror a, a, #1**
$\ggg 4$	1. **swap a**	–	–

B Assembly Code for Basic 16-bit Rotations

MCU	**AVR**	**MSP**	**ARM**
Data	a, b	a	a
$\lll 1$	1. **lsl b** 2. **rol a** 3. **adc b, R1**	1. **rla a** 2. **adc a**	
$\lll 4$	1. **swap b** 2. **swap a** 3. **mov c, b** 4. **eor c, a** 5. **andi c, 0x0F** 6. **eor b, c** 7. **eor a, c**	–	–
$\lll 8$	1. **eor a, b** 2. **eor b, a** 3. **eor a, b**	1. **swpb a**	1. **rev16 a, a**
$\ggg 1$	1. **bst b, 0** 2. **ror a** 3. **ror b** 4. **bld a, 7**	1. **bit #1, a** 2. **rrc a**	1. **bfi a, #16, #1** 2. **ror a, a, #1**
$\ggg 4$	1. **swap b** 2. **swap a** 3. **mov c, b** 4. **eor c, a** 5. **andi c, 0xF0** 6. **eor b, c** 7. **eor a, c**	–	–
$\ggg 8$	1. **eor a, b** 2. **eor b, a** 3. **eor a, b**	1. **swpb a**	1. **rev16 a, a**

C Assembly Code for Basic 32-bit Rotations

MCU	AVR	MSP	ARM
Data	a, b, c, d	a, b	a
$\lll 1$	1. lsl d 2. rol c 3. rol b 4. rol a 5. adc d, R1	1. rla b 2. rlc a 3. adc b	1. ror a, a, #31
$\lll 5$	1. push R1 2. ldi e, 32 3. mov f, c 4. mov g, a 5. mul d, e 6. movw d, R1 7. mul b, e 8. movw b, R1 9. mul f, e 10. eor c, R1 11. eor b, R1 12. mul g, e 13. eor a, R1 14. eor d, R1 15. pop R1	–	–
$\lll 8$	1. mov e, d 2. mov d, a 3. mov a, b 4. mov b, c 5. mov c, e	1. swpb a 2. swpb b 3. mov.b a, c 4. xor.b b, c 5. xor c, a 6. xor c, b	1. ror a, a, #24
$\lll 16$	1. movw f, d 2. movw d, b 3. movw b, f	1. xor b, a 2. xor a, b 3. xor b, a	1. ror a, a, #16
$\ggg 1$	1. bst d, 0 2. ror a 3. ror b 4. ror c 5. ror d 6. bld a, 7	1. bit #1, b 2. rrc a 3. rrc b	1. ror a, a, #1

⋙ 4	1. swap d 2. swap c 3. swap b 4. swap a 5. mov f, d 6. andi f, 0xF0 7. andi d, 0x0F 8. mov e, c 9. andi e, 0xF0 10. eor d, e 11. andi c, 0x0F 12. mov e, b 13. andi e, 0xF0 14. eor c, e 15. andi b, 0x0F 16. mov e, a 17. andi e, 0xF0 18. eor b, e 19. andi a, 0x0F 20. eor a, f	–	–
⋙ 8	1. mov e, b 2. mov b, a 3. mov a, d 4. mov d, c 5. mov c, e	1. mov.b b, c 2. xor.b a, c 3. swpb b 4. swpb a 5. swpb c 6. xor c, b 7. xor c, a	1. ror a, a, #8
⋙ 16	1. movw f, d 2. movw d, b 3. movw b, f	1. xor b, a 2. xor a, b 3. xor b, a	1. ror a, a, #16