Universidade Estadual de Campinas
Instituto de Computação

Luan Cardoso dos Santos

Software implementation of authenticated encryption algorithms on ARM processors

Implementação em software de cifradores autenticados para processadores ARM

CAMPINAS
2018

Luan Cardoso dos Santos

Software implementation of authenticated encryption algorithms on ARM processors

Implementação em software de cifradores autenticados para processadores ARM

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Julio César López Hernández

Este exemplar corresponde à versão final da Dissertação defendida por Luan Cardoso dos Santos e orientada pelo Prof. Dr. Julio César López Hernández.

CAMPINAS
2018

Agência(s) de fomento e nº(s) de processo(s): Não se aplica.

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Sa59s

Santos, Luan Cardoso dos, 1993-
Software implementation of authenticated encryption algorithms on ARM processors / Luan Cardoso dos Santos. – Campinas, SP : [s.n.], 2018.

Orientador: Julio César López Hernández.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Microprocessadores ARM. 2. Arquitetura de computador. 3. Criptografia de dados (Computação). 4. Engenharia de software. 5. Engenharia de software - Medidas de segurança. 6. Software - Medidas de segurança. I. López Hernández, Julio César, 1961-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Implementação em software de cifradores autenticados para processadores ARM
Palavras-chave em inglês:
ARM microprocessors
Computer architecture
Data encryption (Computer science)
Software engineering
Software engineering - Safety measures
Software – Safety measures
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Júlio Cesar López Hernández
Marcos Antonio Simplicio Junior
Diego de Freitas Aranha
Data de defesa: 12-06-2018
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Luan Cardoso dos Santos

Software implementation of authenticated encryption algorithms on ARM processors

Implementação em software de cifradores autenticados para processadores ARM

Banca Examinadora:

- Prof. Dr. Julio César López Hernández
  IC - UNICAMP
- Prof. Dr. Marcos Antônio Simplicio Junior
  LARC - USP
- Prof. Dr. Diego de Freitas Aranha
  IC - UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 12 de junho de 2018

Dedicatória
Dedicated to

I dedicate this work to my father, Devaldite, who will always be in my heart, and will always be my inspiration to walk forward, to be a good man like him.

Eu dedico esse trabalho a meu pai, Devaldite, que sempre está em meu coração, e que sempre será minha inspiração para seguir em frente, ser um homem bom como ele.

The secret to doing anything is believing that you can do it. Anything that you believe you can do strong enough, you can do. Anything. As long as you believe.

– Bob Ross

Acknowledgements

I want to thank all my friends, for their company helped me continue walking forward, even in hard times. My laboratory friends, who helped me with both life and research, always by my side. Some names to cite are Hayato and Renna, Camila and Klairton, Samuel and Fabianna, Amanda, Alisson, Rafael, Marcelo, Armando, Sheila, Renan, so many wonderful people I met in Unicamp –So many that it is impossible to type them all– they will always have a special place in my memory. And I thank my old friends as well, for Guilherme, Victor and Zé Lourenço were always there for me.

I thank my family, for trusting in me, and staying by my side, even if the distance and time make the contact difficult. *Enezina, Gabriel, Bruno, familia é para sempre, e para sempre amo vocês.*

I thank my teachers, Julio and Diego, which whom I worked closely, and learned a lot with. I have to thank you for helping me grow as a student, scientist, and most importantly, as a person. And I also thank all the teachers I had, even if our time was short, every lesson was important.

I also thank LGE, and it's team, for financing the research, and directly helping me, financially and technically with this Master's Degree.

And a very special thanks to Michael Trautsch, a great friend, that adopted me as his little brother, and who helped me and guided me through hard times. And also a very special thanks to his parents, Ute and Winfried, for they received me with open arms, as part of their family. *Mike, Ute, Winfried, Ihr seid meine zweite Familie; mein zweites Zuhause. Ich werde immer dafür dankbar sein, Euch in meinem Leben zu haben.*

And to all those that I may not have mentioned by name, but were with me during these years, my sincere thank you.

Resumo

Algoritmos de cifração autenticada são ferramentas usadas para proteger dados, de forma a garantir tanto sigilo quanto autenticidade e integridade.

Implementações criptográficas não possuem apenas exatidão e eficiência como seus principais objetivos: sistemas computacionais podem vazar informação sobre seu comportamento interno, de forma que uma má implementação pode comprometer a segurança de um bom algoritmo. Dessa forma, esta dissertação tem o objetivo de estudar as formas de implementar corretamente algoritmos criptográficos e os métodos para optimizá-los sem que percam suas características de segurança. Um aspecto importante a ser levado em consideração quando implementando algoritmos é a arquitetura alvo. Nesta dissertação concentra-se na família de processadores ARM. ARM é uma das arquiteturas mais utilizadas no mundo, com mais de 100 bilhões de chips vendidos.

Esta dissertação foca em estudar e implementar duas famílias de cifradores autenticados: NORX e Ascon, especificamente para processadores ARM Cortex-A de 32 e 64 bits. Descrevemos uma técnica de optimização orientada a *pipeline* para NORX que possui desempenho melhor que o atual estado da arte, e discutimos técnicas utilizadas em uma implementação vetorial do NORX. Também analisamos as características de uma implementação vetorial do Ascon, assim como uma implementação vetorial de múltiplas mensagens.

Abstract

Authenticated encryption algorithms are tools used to protect data, in a way that guarantees both its secrecy, authenticity, and integrity.

Cryptographic implementations do not have only correctness and efficiency as its main goals: computer systems can leak information about their internal behavior, and a bad implementation can compromise the security of a good algorithm. Therefore, this dissertation aims to study the forms of correctly and efficiently implementing cryptographic algorithms and the methods of optimizing them without losing security characteristics. One important aspect to take into account during implementation and optimization is the target architecture. In this dissertation, the focus is on the ARM family of processors. ARM is one of the most widespread architectures in the world, with more than 100 billion chips deployed.

This dissertation focus on studying and implementing two different families of authenticated encryption algorithms: NORX and Ascon, targeting 32-bits and 64-bits ARM Cortex-A processors. We show a pipeline oriented technique to implement NORX that's faster than the current state-of-art; and we also discuss the techniques used on a vectorial implementation of NORX. We also describe and analyze the characteristics of a vectorial implementation of Ascon, as well as a multiple message vectorial implementation.

List of Figures

List of Tables

List of abreviations and acronyms

AEAD ..... Authenticated encryption with additional data

AES ..... Advanced Encryption Standard

AE ..... Authenticated encryption

ARM ..... Advanced RISC Machine

ARX ..... Add-Rotate-XOR

CISC ..... Complex instruction set computing

CPB ..... Cycles per byte

DDoS ..... Distributed denial of service

eBAEAD ..... ECRYPT Benchmarking of Authenticated Ciphers

ECRYPT ..... European Network of Excellence for Cryptology

ENISA ..... European Union Agency for Network and Information Security

IoT ..... Internet of Things

LSB ..... Least significant bit

MAC ..... Message authentication code

MSB ..... Most significant bit

NIST ..... National Institute for Standards and Technology

NSA ..... National Security Agency

RISC ..... Reduced instruction set computing

SHA ..... Secure hashing algorithm

SIMD ..... Single Instruction Multiple Data

SPN ..... Substitution-permutation Network

SUPERCOP ..... System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives

VM ..... Virtual Machine

Contents

Chapter 1

Introduction

In the last decade or so, there was a deep change in computation: powerful devices as small as the palm of a hand are carried in our pockets day in and out. These devices connect us with services, other people, with the Internet in general, with almost no downtime. Our televisions, once passive entertainment devices are now full-fledged computers. Light-bulbs are connected, refrigerators, watches, even little buttons that can immediately order goods from an online store with a single touch. This movement, aptly named with the buzzword "Internet of Things" shows no signal of changing. However, these devices hold a large amount of information about themselves and their owners, information that could be leveraged by adversaries, and the devices themselves could be taken and turned into silent zombies, part of a botnet. Far from speculation, such a thing already happened: Mirai was a malware that turned IoT devices into workers of a botnet in 2016, being used on DDoS attacks [1].

With those dangers in mind, cryptography is a tool that can be used to protect data, both at rest and in transit. IoT devices are normally constrained by processing power and even in energy consumption by using batteries, what introduces challenges when using cryptographic algorithms. Historically, cryptography was designed and deployed mostly with security in mind, leaving efficiency as a secondary concern, what can prove to be an issue when deploying crypto to IoT. A solution to this problem is the area of cryptography known as *Lightweight cryptography*, where both security and efficiency are considered in the design and implementation of cryptographic services.

The focus of this work is on Authenticated Encryption, a subset of symmetric key encryption, useful for providing both confidentiality and authenticity to data. Keeping in mind the scenario of constrained devices, this dissertation aims to study AEAD algorithms, how to correctly implementing them in software, and optimize these implementations for ARM processors, the *de facto* standard for IoT devices.

1.1 Contributions of this work

This dissertation discusses software techniques to optimize two sponge-based authenticated encryption algorithms, namely NORX[2] and Ascon[3]. These techniques are based on the characteristics of ARM processors, such as instruction pipeline and vecto-

rial engines, with focus on performance.

Regarding NORX, it is presented a pipeline oriented optimization technique that improves on the state-of-art, being faster than vectorial implementations in some scenarios. NORX's results were published on the XVII SBSEG – Brazilian Symposium in Information and Computational Systems Security – in a paper titled "Pipeline Oriented Implementation of NORX" [4]. The paper's abstract follows:

> **Abstract:** NORX is a family of authenticated encryption algorithms that advanced to the third-round of the ongoing CAESAR competition for authenticated encryption schemes. In this work, we investigate the use of pipeline optimizations on ARM platforms to accelerate the execution of NORX. We also provide benchmarks of our implementation using NEON instructions. The results of our implementation show a speed improvement up to 48% compared to the state-of-art implementation on Cortex-A ARMv8 and ARMv7 processors.

On Ascon, this dissertation shows a multiple message processing technique that uses ARM vector instructions to execute parallel processing of independent payloads, as well as a technique to implement Ascon using 128-bit NEON registers.

1.2 Document structure

This document is divided into three main parts. Chapter 2 briefly introduces the main concepts used throughout this work, such as cryptography, authenticated encryption, cryptographic competitions and the ARM architecture.

Chapter 3 will present the work done on the NORX family of algorithms, and Chapter 4 will present the work done on the Ascon family of algorithms.

Lastly, Chapter 5 will group the closing remarks on this work. Further technical information and references will be in the Appendices.

Throughout this dissertation, we will use the notation shown in Table 1.1 for algorithms and equations, except where pointed otherwise.

Table 1.1: Notations used throughout this dissertation.

Notation	Description
0^n	All-zero bitstring of lenght n
ε	Empty bitstring
$\|x\|$	Legth of x in bits
$\|x\|_n$	Length of x in blocks of n bits
$x \parallel y$	Concatenation of x and y
$x \ll n$, $x \gg n$	Left or Right shift of x by n bits
$x \lll n$, $x \ggg n$	Left or Right rotation of x by n bits
$\neg$, $\wedge$, $\vee$, and $\oplus$	Bitwise negation, `AND`, `OR`, and `XOR`
`~, & ,|, and ^`	Bitwise negation, `AND`, `OR`, and `XOR` (code notation)
$\leftarrow$	Variable assignment
`=`	Variable assignment (code notation)
$\text{left}_l(x)$	Truncation of x to the l leftmost bits
$\text{right}_r(x)$	Truncation of x to the r leftmost bits

Chapter 2

Background

In this chapter, the reader will be introduced to key concepts used throughout this dissertation. General concepts on cryptography will be presented on Section 2.1. On Section 2.2, concepts specific to authenticated encryption will be briefly presented. Section 2.3 will introduce the reader to cryptographic competitions and, lastly, Section 2.4 will present a high level description of the ARM architecture.

2.1 Cryptography

Cryptography, from the Greek words *kruptós gráphō* meaning "hidden" and "writing", is the "study of mathematical techniques for securing digital information, systems, and distributed computations against adversarial attacks" [5].

Historically, cryptography was used mainly by military organizations, as a way to keep secrets. Back then, cryptography was more based on the feelings and intuition of the cryptographer designing the codes, without formal mathematical rigor in their construction. As historical example, one of the first use was the *Caesar cipher*, a very simple cipher, where each letter of a message was exchanged for another a few positions forward on the alphabet. After that, techniques somewhat more advanced started to be employed, such as the Vigenère cipher, a polyalphabetic substitution scheme. It was not until the 20th century that cryptography started to be studied as a proper science and mathematical discipline. From that period, a very prominent use of cryptography was the Enigma machine. The Enigma was a series of electromechanical cipher machines, based on rotors and developed by Arthur Scherbius, a German engineer, near the end of World War I. The machine was adopted by the German Military as a form of protecting sensitive information during the Second World War. A great triumph of the Allies was the breaking of the Enigma cipher: The techniques and technologies created during the analysis of the Enigma eventually led to the construction of the first modern computers, and by itself was one of the factors responsible by the outcome of the war [6].

Nowadays, cryptography is not a military-only tool, but a constant tool used on a day-to-day basis: From authenticating users on our personal computers, to securing financial transactions, to protecting mission-critical systems, cryptography is as ubiquitous as computing itself. Adding to that, there is a trend where the common person is

more and more entangled with computing applications, with sensible data being transmitted back and forth: with that, the need for well designed and secure cryptographic tools is also growing.

One sub-area of modern cryptography is called symmetric cryptography –or secret-key cryptography– and studies the cryptographic schemes where a shared secret is used to , for example, encrypt and decrypt a message or authenticate it. A secret-key encryption scheme is defined by the tuple of probabilistic polynomial-time algorithms $(\text{Gen}, \text{Enc}, \text{Dec})$ with the following characteristics[5]:

- Gen is the *key-generation algorithm*; it takes as input 1^n –the security parameter written as unary– and outputs a key k. Gen is a randomized algorithm, and any key output by $\text{Gen}(1^n)$ satisfies $|k| \geq n$.
- Enc is the encryption algorithm; it takes as inputs a key k and a plaintext message $m \in \{0,1\}^*$, and outputs a ciphertext c.
- Dec is the decryption algorithm; it takes as input a key k and a ciphertext c, and outputs a message m.
- It is required that, for every n, every key k generated by $\text{Gen}(1^n)$ and every message $m \in \{0,1\}^*$, it holds that $\text{Dec}_k(\text{Enc}_k(m)) = m$.

In the following sections, relevant concepts from the area of symmetric cryptography will be briefly described.

2.1.1 Block ciphers

A block cipher is a deterministic function that operates over a fixed-length input –the block– parametrized by a symmetric key. Block ciphers are also an important construction block of various cryptographic services, algorithms, and protocols. According to Katz et.al [5], a block cipher is an efficient keyed permutation $F : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^\ell$, defined as $F_k(x) = F(k,x)$, with the function F_k being a bijection. Furthermore, F_k and F_k^{-1} are efficiently computable given k [5]. Nowadays, the common method of designing block ciphers consists in an iterating transformation that combines data substitution and permutation. The design of block ciphers based on iterating simple transformations was proposed by Shannon in 1949 [7].

One of the first block ciphers to be standardized was the Data Encryption Standard –DES–, which is very influential to modern cryptology. DES was published by the U.S. National Bureau of Standards, currently NIST, in 1977 [8]. It can be said that DES started the development and study of encryption algorithms, opening the field to the general public, and not restricting it to military institutions and military use [9].

The successor of DES is the Advanced Encryption Standard –AES–, published in 2001 by NIST, and based on Rijndael [10]. The algorithm was chosen via a public competition that had the objective of "specifying a public encryption algorithm, capable of protecting sensitive data beyond this century". Cryptographic competitions will be presented in more depth in Section 2.3. AES is standardized by FIPS PUB 197 [11], it

became the official US symmetric encryption algorithm in May 2002, and it was also included in ISO/IEC 18033-3. In relation to security, in July 2003, the US government announced that "All AES key lengths (128, 192, and 256 bits) are adequate to protect information up to SECRET level. TOP SECRET data require the use of 192 or 256-bit keys" [12].

2.1.2 Lightweight cryptography

A somewhat recent concept, lightweight cryptography is area of study driven by the lack of cryptographic primitives capable to run on devices with low computing power. At its core, Lightweight cryptography is the area of classical cryptography that studies algorithms pertinent to constrained devices, such as RFID tags, sensors in wireless networks, small devices and Internet-of-things devices [13]. Another important characteristic of Lightweight cryptography is that it is capable of obtaining the adequate levels of security, without necessarily resorting to security-efficiency trade-offs. The properties of lightweight cryptography have been discussed in ISO/IEC 29192.

The "lightweightness" of a cryptographic primitive can be measured against software and hardware constraints. In software implementations, smaller footprints in RAM and ROM are desirable, and so are factors such as latency, power consumption, and throughput. Latency is especially relevant for real-time applications, where high agility in handling encryption and decryption is desirable. For Hardware implementations, the area needed for the cryptographic primitive is usually as important as its speed. As important as that, many of those constrained devices operate with batteries or even with power harvested from the surroundings, so the energy consumption is an important metric of performance for a lightweight implementation [14].

2.1.3 Sponge Functions

A cryptographic sponge function, introduced as a primitive for authenticated encryption in "Duplexing the sponge" [15] and as a general cryptographic function in "On the Indifferentiability of the Sponge Construction" [16][17], is an algorithm with a finite internal state that receives as input a string of any length and produces as output a string of any desired length. Sponge functions can be used to create various cryptographic primitives, such as hash functions, message authentication codes –MACs–, stream ciphers, pseudorandom number generators and authenticated encryption schemes. A sponge function can be compared as a real-world sponge, where data is absorbed and then squeezed from it.

A sponge is based on three main components: A state S of b bits, subdivided into *rate* and *capacity* sections of respectively r and c bits; a round permutation function F^l of b bits with a round number l defined in terms of a permutation F of b bits as the l-fold iteration $F^l(S) = F(F(...F(S)))$ which is used to transform the state in each round; and a padding rule P for the input. A sponge works by initializing the state value and "absorbing" r bits from the padded input and transforming the state with $F^l(S)$. After that, the sponge is ready to be "squeezed", removing up to r bits before

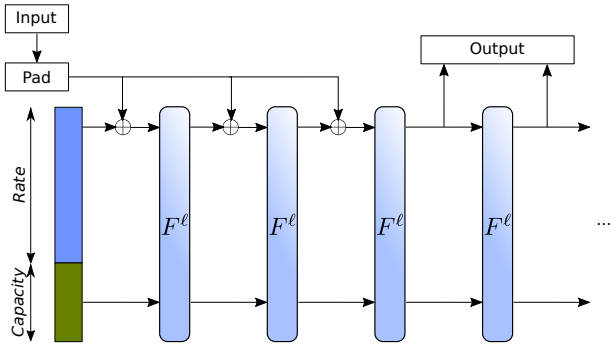

Figure 2.1: The basic design of a sponge function, showing the absorption and squeeze processes [15].

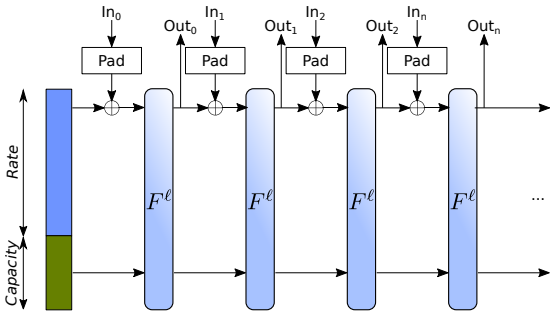

Figure 2.2: The basic design of a duplexed sponge function, showing the absorption and squeeze processes [15].

needing to evaluate $F^l(S)$ again. Figure 2.1 and Figure 2.2 illustrate the operation of a cryptographic sponge [15].

An example of a practical use of sponge functions in cryptographic primitives is the SHA-3[18] hash algorithm, that uses a 1600-bit sponge.

2.2 Authenticated encryption

An authenticated encryption scheme is an algorithm that uses a secret key and a public nonce to process a plaintext and generate a ciphertext and an authentication tag. Furthermore, an authenticated encryption (AE) scheme can also accept as input extra data that is authenticated together with the plaintext. In that mode of operation, this scheme is called Authenticated Encryption with Additional Data (AEAD). Such a scheme is useful, for example, to encrypt the body of a message, while keeping the receiving address in plain form, and authenticating the whole. This way, the recipient of a message can guarantee that public data was not modified by a third party. A basic block diagram of an authenticated encryption algorithm is shown in Figure 2.3.

Formally an AEAD scheme is defined by the tuple $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ and the associated sets $\text{Nonce} = \{0,1\}^n$, $\text{Header} \subset \{0,1\}^*$ and $\text{Message} \subseteq \{0,1\}^*$. The Message set must satisfy the membership test $M \in \text{Message} \Rightarrow M' \in \text{Message}$ for any M' with the same length of M.

The keyspace $\mathcal{K}$ is a non-empty finite set of strings. The encryption algorithm $\mathcal{E}$ is a deterministic algorithm that receives as input the strings $K \in \mathcal{K}$, $N \in \text{Nonce}$, $H \in \text{Header}$ and $M \in \text{Message}$. The encryption algorithm returns a string $\mathcal{C} = \mathcal{E}_K^{N,H}(M) = \mathcal{E}_K(N, H, M)$. The decryption algorithm $\mathcal{D}$ is a deterministic algorithm that receives as input the strings $K \in \mathcal{K}$, $N \in \text{Nonce}$, $H \in \text{Header}$ and $\mathcal{C} \in \{0,1\}^*$ and returns $\mathcal{D}_K^{N,H}(\mathcal{C}) = \mathcal{D}_K(N, H, C)$, that is either a string from the set of possible messages, or a symbol $\perp$ meaning that the set of ciphertext, nonce, and key is invalid.

Beyond that, it is required that $\mathcal{D}_K^N(\mathcal{E}_K^N(M)) = M$ for all $K \in \mathcal{K}$, $N \in \text{Nonce}$, and $M \in \text{Message}$; and that $|\mathcal{E}_K^{N,H}| = l(|M|)$ for some linear-time length function l [19].

In the current state of the art, authenticated encryption is considered the minimum

acceptable in terms of data encryption, as it offers advantages over simple encryption, with very little extra cost. An example of the movement from encryption to authenticated encryption is TSL 1.3, where support for non-AEAD ciphers was removed. Another noteworthy characteristic of authenticated encryption implies that a Chosen Ciphertext Attack cannot be mounted, as no plaintext is returned to the attacker if the authentication fails.

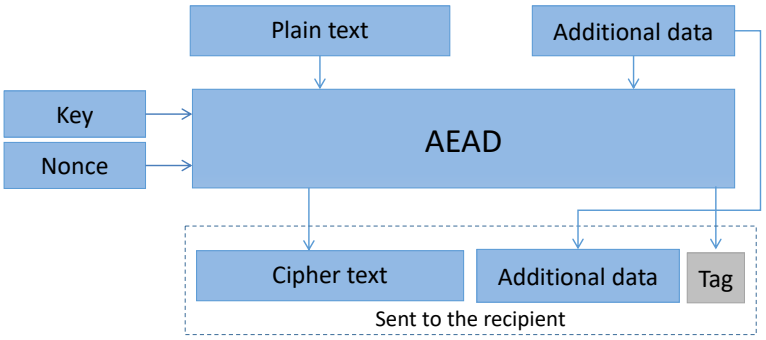

Figure 2.3: Basic block design of an authenticated encryption scheme with additional data, where ciphertext and authentication tag are produced by processing plaintext, additional data, key, and nonce.

2.2.1 Authenticated mode of operations

The first approaches for creating authenticated encryption were based on combining a block cipher and a MAC together in an authenticated mode of operation, also called "generic composition". In 2001, Krawczyk examined the 3 main methods used to create an authenticated mode of operation [20]. They are as following:

- Encrypt-and-MAC (E&M): In this construction, an authentication code is generated from the plaintext, and this value is sent together with the ciphertext to the message recipient. This construction has no strong proofs of being secure against forging attacks[21]. Figure 2.4 illustrates a basic block diagram of an E&M authenticated encryption scheme. It does not provides IND-CCA, NM-CPA, and INT-CTXT.

- Encrypt-then-MAC (EtM): In this design, the plain text is encrypted, then an authentication tag is computed over the ciphertext. This is the method used, for example, in the IPSec protocol, and is the standard according to ISO/IEC 19772:2009. EtM is illustrated in Figure 2.5. This construction provides INT-PTXT and IND-CPA, and it does not provide NM-CPA, IND-CCA and INT-CTXT.

- MAC-then-encrypt (MtE): In this construction, an authentication code is computed over a plaintext, and both plaintext and authentication code are encrypted together. This is used, for example, in SSL/TLS(RFC7366) [20]. Figure 2.6 illustrates a basic block diagram of an EtM authenticated encryption scheme. This

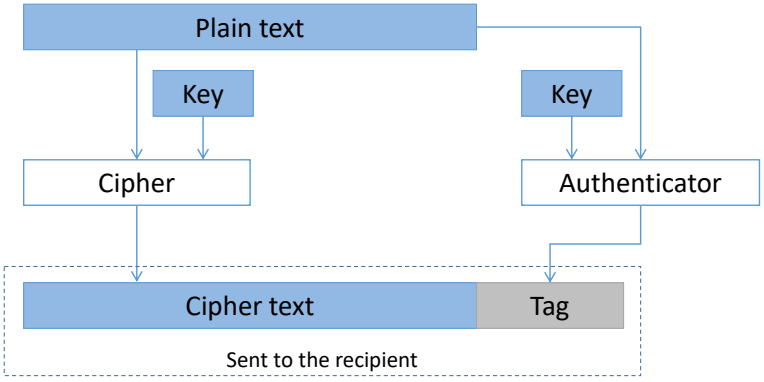

Figure 2.4: Basic block diagram of a E&M authenticated mode of operation.

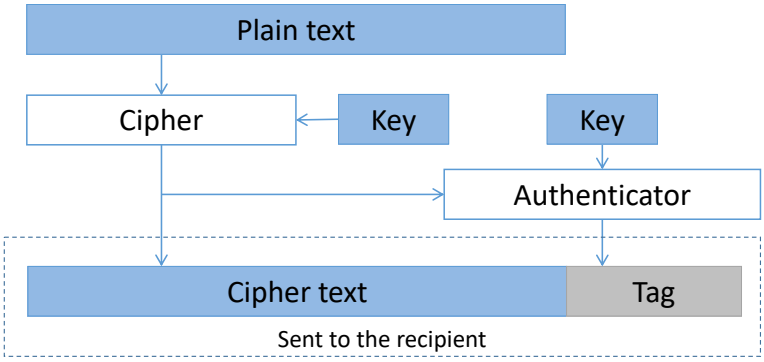

Figure 2.5: Basic block diagram of a EtM authenticated mode of operation.

construction provides IND-CPA and INT-PTXT. It does not provide NM-CPA, IND-CCA, and INT-CTXT.

2.2.2 Dedicated AE(AD) schemes

The methods previously shown to implement authenticated encryption may not be the best solution for all scenarios. One of the characteristics of the "Generic Composition" is the need to use two different algorithms to construct an authenticated encryption algorithm. Given that, is there a way to construct an AE without using two different algorithms, two keys, and two separate passes over the message?

For a long time, cryptographers wanted to find such an operation that achieves AE, and many attempts were broken, until 2000, when Jutla at IBM, created the first correct-proven single-pass AE modes: IAPM and IACBC [22]. Compared to the generic composition, where one needed $2m$ encryption/authenticator calls per message (assuming a value m related to the message and block lengths), these AE schemes needed only $m \log(m)$ calls. With further refinements, this number was almost m, what allows one to achieve authenticated encryption at the same cost of only encryption.

After the announcement of IACBC and IAPM, Rogaway announced the OCB scheme, a follow-up of IAPM, with several improvements [23]. Simultaneously, Gligor and Donescu presented the classes of schemes XCBC and XECB, respectively similar to CBC

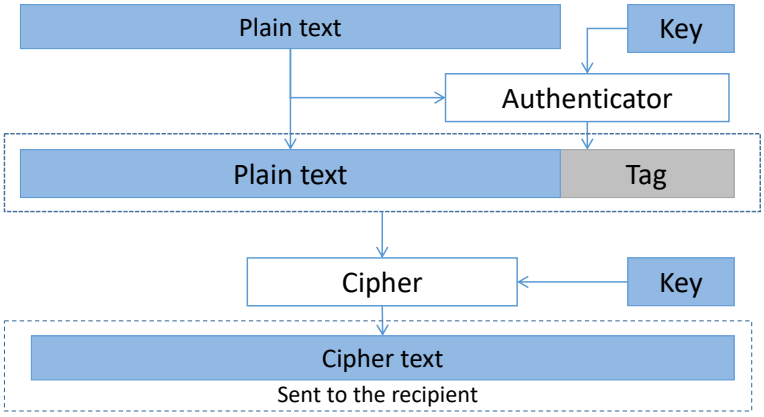

Figure 2.6: Basic block diagram of a MtE authenticated mode of operation.

and ECB modes of operation.

Given the importance of these efficient AE algorithms, patents were filled by the authors. Because of that, many of the probable users of these algorithms were thrown away by the possible legal difficulties being not worth the technical benefits. And, with these complications in mind, new teams of researchers started working on creating further efficient AE algorithms not covered by patents. These schemes include modes such as CCM [24], CWC [25], EAX [26], and newer single-pass dedicated algorithms.

2.2.3 Current Standards

NIST –National Institute of Standards and Technology– standardized the authenticated mode of operations CCM and GCM, through SP 800-38C and SP 80038D standards[24, 27]. Additionally, the European Union Agency for Network and Information Security (ENISA) recommends the use of EtM authenticated modes of operation, namely OCB, CCM, EAX, CWC, and GCM [28]. These modes of operations are briefly presented below:

- CCM: *Counter with CBC-MAC* is an authenticated mode of operation for block ciphers, with the main objective of providing both authentication and secrecy. CCM is defined only for ciphers with a block size of 128-bits and was based on the CTR mode of operation. CCM works as a stream cipher, and as such, is prone to the same issues regarding reuse of initialization vectors [24].

- GCM: *Galois/Counter mode* is an authenticated mode of operation for block ciphers, designed to provide authentication and secrecy. The main characteristic of GCM is the use of 128×128 bits multiplications over a binary finite field, what in turns allows GCM to reach throughputs larger than the other modes of operation. This efficiency is due to the fact that GCM can make use of parallelism, specific instructions, and pipelining on modern processors [27], although the implementation of the Galois operations is a difficult matter to tackle in simpler and more limited architectures that does not feature adequate hardware instruction e.g. 64×64 bit carryless multiplications.

- OCB: *Offset codebook* is a mode of operation based on IAPM [29]. It is an efficient mode of operation, that uses $n+2$ calls to the block cipher to encrypt and authenticate n blocks of the message. The use of this mode of operation is restricted due to two North American patents. Since 2013, this mode of operation is licensed under GNU General Public License, with restrictions for use in military applications [30].

- EAX: EAX is a mode of operation similar to CCM, created with the objective of superseding the latter, by adding resources not present in CCM, and it is a sequential two-pass authenticated encryption mode of operation. The authors of this mode operation made the code available to the public domain [26] and is a NIST recommendation, with a simple design.

- CWC: *Carter-Wegman + Counter* is a mode of operation that combines the Carter-Wergman MAC algorithm with the CTR mode of operation. This mode was considered by NIST for standardization, but GCM was chosen instead [25].

2.3 Cryptographic competitions

Cryptographic competitions are public competitions where proposals for algorithms are submitted and analyzed by academic and private organizations, with the main objective of choosing one or more standard algorithms for widespread use.

The first[1] open cryptographic competition was announced in January of 1997, by NIST, with the objective of selecting a new block cipher and successor to DES: the *Advanced Encryption Standard*, or AES [31]. The competition started with an open call to submit candidate block ciphers. With a total of 15 different algorithms submitted, they were intensively analyzed by the public, members of NIST, and other competitors. In October of 2000, the winning algorithm –Rijndael [10]– was announced amongst the five finalists. According to NIST, any of the five finalists –Rijndael, MARS, RC6, Serpent, and Twofish– would be adequate as the winner, but Rijndael was chosen mostly based on properties such as efficiency and security.

The important characteristic of the AES competition, and by extension of the other competitions, was that any group who submitted an algorithm had a very strong motivation for analyzing and finding attacks on their adversaries submissions. With that in mind, the result was a group formed by the best cryptanalysts and designers focusing their efforts on analyzing the block ciphers submitted to the competition. In a short time, all algorithms in the AES competition were subjected to deep and careful analysis, what only increases the confidence in the security of the winner. A proof of that is, AES is still widely used, and as of 2017, the best key recovery attack has a complexity of $2^{126.1}$ on the 128-bit key AES [2][32].

Following the footsteps of the AES competition, in 2004, ECRYPT announced the eSTREAM –ECRYPT Stream Cipher Project– with the objective of selecting new stream

[1] DES was also a competition, held by NBS, but closed only to invited designers.
[2] This attack has a complexity of $2^{189.7}$ for AES-198, and $2^{254.4}$ for AES-256.

ciphers suitable for widespread adoption. The competition received 34 submissions, and resulted in a portfolio of several stream ciphers announced in 2008: HC128, Rabbit, Salsa20/12, and SOSEMAUK for high-throughput software applications; Grain v1, MICKEY2.0, and Trivium for highly restricted hardware [33].

After eSTREAM, in 2007 NIST announced a new competition, this time to choose a new hash standard –SHA-3. This competition had the objective of specifying a new hash algorithm to augment and revise FIPS 180-2, the NIST standard that specified both SHA-1 and SHA-2. Amongst the 64 initial submissions, Keccak was chosen as the base for the specification of SHA-3, in October of 2012 [18]. SHA-3 was standardized as a subset of the Keccak family of hash functions by FIPS 202, in 2015.

Currently, and following the tradition of previous competitions, CAESAR is a cryptographic competition with the objective of selecting a portfolio of authenticated ciphers that offer advantages over NIST's AEC-GCM, and that is also suitable for widespread adoption. This competition is of special interest to the scope of this work, and it will be presented in more detail in the next Section.

2.3.1 CAESAR

CAESAR was officially announced on January 15th, 2013, at the Early Symmetric Crypto workshop. In March 5th, 2018, the fourth round finalists of CAESAR were announced at the FSE 2018 rump session, a total of 7 competitors out of 55 initial submissions were selected as finalists. The current competitors are listed in Table 2.1 with the third round competitors that were not selected as finalist being listed in Table 2.2, as well as their underlying primitives and main characteristics in relation to parallel encryption and decryption, inverse-free construction, the existence of security proofs, and security against nonce misuse. As of the writing of this dissertation, there was no publically available information on when the final portfólio will be available, or the reasons behind the choice of the current competitors, or disqualification critéria for the ones that did not make into the fourth round.

Table 2.1: Fourth round candidates and finalists of CAESAR. Source: AE zoo [34]

Name	Type	Primitive	Parallel E/D	Online	Inverse-free	Security proof	Nonce-MR
ACORN	SC	LFSR	✓/✓	✓	✓	×	NONE
AEGIS	BC	AES	✓/×	✓	✓	×	NONE
Ascon	Sponge	SPN	×/×	✓	✓	✓	
Deoxys-II	BC	AES	✓/✓	✓	×	✓	
MORUS	SC	LRX	×/×	✓	✓	×	NONE
OCB	BC	AES	✓/✓	✓	×	✓	NONE

CAESAR defines an authenticated encryption algorithm as a function that receives five input arguments in the format of byte strings and outputs a byte string. The inputs are the plaintext $p \in \{0,1\}^*$, the additional data $z \in \{0,1\}^*$, secret number n_{sec}, public number n_{pub}, and key k. Plaintext and additional data have variable length, and the other three inputs have a fixed length for a given algorithm. The output is the cipher-

Table 2.2: Third-round candidates of CAESAR that were not selected to the final round. Source: AE zoo [34]

Name	Type	Primitive	Parallel E/D	Online	Inverse-free	Security proof	Nonce-MR
AES-JAMBU	BC	AES	×/×	✓	✓	×	
AES-OTR	BC	AES	✓/✓	✓	✓	✓	NONE
AEZ	BC	AES	✓/✓	×	✓	✓	OFF-MAX
CLOC	BC	AES,TWINE	×/×	✓	✓	✓	NONE
Ketje	Sponge	Keccak-f	×/×	✓	✓	✓	NONE
Keyak	Sponge	Keccak-f	✓/×	✓	✓	✓	NONE
NORX	Sponge	LRX	✓/✓	✓	✓	✓	NONE
SILC	BC	AES, PRESENT, LED	×/✓	✓	✓	✓	NONE
Tiaoxin	BC	AES	✓/✓	✓	✓	×	NONE

text $c \in \{0,1\}^*$. It must also be possible to recover p and n_{sec} given c, z, n_{pub}, and k. The CAESAR algorithms can specify a limit for the length of plaintext and additional data, as long as it is not smaller than 2^{16} bytes. Beyond that, ciphers are not obligated to support public and secret numbers, by defining their length to 0 bytes. It is expected that the ciphers' security characteristics are not affected by the choice of message numbers or their length. The ciphers also cannot define any rules for the choice of message numbers, although it is allowed that the security characteristics of the cipher be lost if message numbers are reused, as long as this characteristic is documented. It is also allowed that the length of plaintext can be leaked by the length of the ciphertext, and no other information about the inputs be derived from the ciphertext. It is also recommended that the ciphers support the default lengths for keys (80, 128, and 256 bits), public message numbers (96 and 104 bits), and authentication tag (32, 64, 96, 128, and 160 bits). The submissions are not required to support these sizes neither are limited to these sizes.

Amongst the CAESAR candidates, the common constructions are[35]:

- **AES Based:** Many of the CAESAR competitors are designed to use a block cipher as their basic construction block, many of them choosing AES mainly due to the extensive analysis and work put into it by the cryptographic community. Beyond that, many modern processors are outfitted with native instructions for fast computation of AES (such as Intel AES-NI), allowing the candidates to use those native instructions to optimize their ciphers. AEGIS[36], AES-JAMBU [37], AES-OTR [38], AEZ [39], CLOC [40], COLM [41], Deoxys [42], OCB [43], SILC [40], and Tiaoxin [44] are the 3rd round candidates that use AES as a building block. From those, AEGIS, COLM, Deoxys and OCB were selected as finalists.

- **Stream cipher based:** A stream cipher can be seen a symmetric pseudorandom number generator, that use a fixed-length key to generate a variable-length bitstream. In the third round of CAESAR, MORUS [45] and ACORN [46] use stream ciphers as their underlying block. ACORN has a full key recovery attack with complexity of $2^{55,85}$ [47], and MORUS was the only candidate selected as finalist amongst the stream cipher based candidates.

- **Keyless permutation based:** A key-less permutation is a bijection mapped be-

tween fixed-length strings. An example of keyless permutation is the sponge construction [16]. Ascon [3], Ketje [48], Keyak [49], and NORX [50] are the 3rd round candidates that use a cryptographic sponge as their base construction. Amongst those, Ascon was selected as finalist.

- **Hash function based:** A hash function maps strings of arbitrary length into fixed length strings [51]. For a cryptographic hash function, it is computationally difficult to find an output collision, preimage, and second preimage. There are no candidates using hash-based constructions in the third round of CAESAR, and only a candidate in the previous rounds, called OMD [52].
- **Dedicated constructions:** Some candidates of CAESAR are simular to Type-3 Feistel schemes[53], featuring unique constructions that do not fit in the previous categories. The current candidates of such type are AEGIS, MORUS, and Tiaoxin. AEGIS and MORUS were selected as finalists of CAESAR.

2.3.2 CAESAR selection criteria

The selection criteria and evaluation processes used by the CAESAR committee are not publicly available, but it is expected that authenticated encryption algorithms have some specific design characteristics, some of which are common to symmetric cryptography schemes. They are [54]:

1. **Protection against various attacks:** It is expected that authenticated encryption primitives resist to various types of attacks. Each type of attack creates a potential design point for the encryption scheme, with the objective of protecting against said attack. Some of these attacks are:
    - **Corruption:** An authenticated encryption scheme produces as output an authenticated ciphertext. This cyphertext authenticates and encrypts a plaintext, and authenticates additional plaintext data. The main objective of corruption is to forge a combination of plaintext, associated data and message number where the recipient calculates the authentication of this data as valid, whereas the legitimate sender never produced this message.
    - **Ciphertext Corruption:** The objective if this attack is to forge a ciphertext that the recipient accepts as valid, but was never produced by the sender. Notice that the integrity of the plaintext does not imply the integrity of the ciphertext. A practical example would be a function that calculates a MAC as a 100-bit string and pads it to 128 bits using random data. These 28 bits are now malleable, and an attacker can easily forge them, without breaking the plaintext integrity.
    - **Prediction:** The attacker has the objective of being capable of distinguishing ciphertext from random data.
    - **Replay and reordering:** Replay has the objective of convincing a recipient of receiving a valid data more times than it was generated by the sender.

Reordering has the objective of convincing the recipient of receiving valid data in a different order from that in which they were generated.

- **Sabotage:** The objective is disabling the system, violating its disponibility.
- **Espionage:** The attacker simply tries to discover the plaintext or secret message number. Ideally, there should be no better way to discover a secret data than simply trying to guess it by chance.

2. **Protection against resourceful adversaries:** The resources of an attacker must be modeled in a valid way, and the encryption scheme must offer security against various models of adversary computational performance. Each performance model can offer a potential design characteristic for the encryption scheme, in which it resists to adversaries with that capacity.

    - **High-Performance computing:** The computation power of an adversary is one the factors the determines the size of cryptographic keys. Nowadays, the most common key sizes are 80, 128, and 256 bits. The question is, what is an adequate size. Arguments towards using smaller keys are about the cost of breaking such keys being larger than the probable benefits of breaking it, therefore no adversary would engage in an attack that would result in a net loss. Furthermore, smaller keys offer a better performance on cryptographic schemes. Arguments in favor of larger keys include distributed efforts in attacks, and the fact that some adversaries, such as governmental entities can use virtually limitless resources, and can engage in activities economically irrational. Beyond that, larger keys can still be used with adequate performance.
    - **Quantum computers:** It is possible that, in the future, a scalable quantum computer be feasible. That can drastically decrease the number of operations necessary to recover a cryptographic key. For example, the Groover algorithm could be capable of recovering a 128-bit key in only 2^{64} quantum operations.
    - **Multiple messages:** A encryption scheme can have its security level degraded when the number of messages encrypted with the same key increases.
    - **Chosen plaintext, ciphertext, and nonce attacks:** Some designs of encryption schemes can experience a degradation in security level when active adversaries have some control over plaintext and message numbers, or when they can forge ciphertexts.
    - **Multiple users:** Some encryption schemes can have variations in its security level when the number of active keys increases.
    - **Message number reuse:** Many encryption schemes require the use of a unique public number –*nonce*– and they lose all their security characteristics when the same nonce is reused with the same key.
    - **Side channel:** In this case, it is necessary to take extra security measures not only in the project, but also on the implementation of the encryption

schemes. The implementation can leak information through its behavior. For example, cryptographic primitives can leak information on secret data by means of branching, or by execution timing. From a hardware point of view, secret information can be leaked through power consumption or electromagnetic emanations from the system.

- **Fault:** Adversaries can alter bits in a computation, for example by using a laser on a chip, and deduct secret data by analyzing the output.
- **Thief and Monitoring:** It is possible that attacker can steal a copy of the secret key, or even implant a sensor inside a cryptographic device to monitor its computations. While it is not plausible to protect future communications in such a scenario, it is possible to construct cryptographic services capable of securing past messages. This property is known as forward secrecy.

3. **Performance:**Authenticated encryption schemes are expected to have good performance characteristics. The performance itself can be evaluated in various forms:

   - **Low byte energy:** Power consumption is a visible cost to the user, be it on a device's battery or on electricity bills. Normally, power consumption is measured in joules per byte.
   - **Low potency:** One must observe that many devices execute cryptographic operations together with other operations, and the power consumption must be in the device limitations. This also limits the design in terms of parallel operations.
   - **Low area:** For hardware implementations, the cryptographic hardware must fit in the available chip area. Normally, area is measured in Gate Equivalents.
   - **High transfer rates:** Different applications needs different transfer rates, and the encryption scheme must be able to provide data at the right rate.
   - **Low latency:** In the same way, applications also impose limits on the maximum time between providing an input and receiving an output.
   - **Cycles per byte:** From a software implementation point of view, the performance is measured in cycles per byte. Ideally, it is considered that a processor always takes the same number of cycles to execute each kind of instruction, allowing then that time estimates be derived from cycles per byte measures.

4. **Performance in different scenarios:** User activity can affect the performance of cipher under the previously discussed metrics, and result in possible scenarios where ciphers perform well under.

   - **Authentication or Encryption and authentication?** An AE scheme authenticates both ciphertext and plaintext. Normally, the cost per authenticated byte is sensibly smaller than the cost for plaintext byte.

- **Sending and receiving costs:** The cost of sending data is not necessarily the same as receiving data. For example, in the construction of the type *encrypt-then-MAC*, the decryption is not computed if there is a failure in authenticating.
- **Number of inputs:** The encryption scheme can be able to combine the processing of multiple inputs, operating over them in a more efficient form than processing each one.
- **Key reuse:** Some encryption schemes are capable of improving their performance by storing precomputed expanded keys.
- **Input reordering:** The latency of some encryption schemes can be improved by computing data that are related to only the key and nonce before plaintext and additional data are ready. Similar scheduling tactics are applied to the plaintext and ciphertext, where they are received in a gradual way. The encryption scheme can then achieve better latency computing operations over the received plaintext while waiting for the rest of data.
- **Intermediate tags:** In the case of a long plaintext, separated into smaller packages, each one can have their own authentication tag. In this case, it is not necessary to process the whole plaintext in order to detect a falsification.

5. **Suport to cryptanalysis:** Cryptanalysis is the most important tool to evaluate the security of a symmetric key cryptosystem. Many designers announce projects and characteristics to help with the cryptanalysis of the proposed schemes, but those characteristics themselves can be hard to analyze in an objective way. Nonetheless, they are

   - **Simplicity:** Cryptographic schemes without unnecessary complexities and with simple constructions are easier to analyze than more complex designs.
   - **Escalability:** Encryption schemes are normally constructed and structured as a series of similar rounds, providing the cryptanalysis targets by mean of reducing the number of rounds. Some schemes also offer versions with smaller word size or other simplified versions for analysis.
   - **Proofs:** Some encryption schemes offer proofs that some attacks, in specific scenarios, are hard or of similar hardness to a related easier-to-solve problem. It is important to note that, those proofs are not strictly *security proofs*, even though being often presented as so.

2.4 ARM Architecture

ARM –*Advanced RISC Machine*– is an architecture similar to RISC (Reduced instruction set computer), developed by the British company ARM Holdings. A RISC processor, in comparison to CISC (complex instruction set computer), is relatively simpler and requires fewer transistors in the design. in terms of manufacture numbers, ARM is the most used architecture in the world. One of the main reasons behind such popularity is

the large gamut of applications and markets the ARM processor is capable of catering to, from small sensors to real-time applications, from consumer electronics to enterprise servers. The majority of ARM processors support 32-bit fixed-length instructions, and a variable-length 32-bit and 16-bit instruction set for improved code density. The newer ARMv8-A architecture, announced in 2011, adds support for 64-bit address space and arithmetic using a new set of 32-bit fixed-length instructions. Beyond that, some ARM cores also feature the Advanced SIMD extension, also known as NEON: a combined 64- and 128-bit SIMD instruction set, used to provide an acceleration for media, signal processing, and other applications. Further details about the NEON extension will be discussed in Section 2.4.2.

The main features of the ARM instruction set are:

- Load/store architecture, where the instructions are divided into two types: memory access and logic/arithmetic operations. In a simple manner, it means that, differently from *register memory architecture*, all the operands for a logic/arithmetic operation must be previously loaded in registers.
- Uniform 32-bit or 64-bit registers.
- Most instructions can be executed in a single cycle, and most instructions can also be conditionally executed.
- A barrel shifter, with zero performance penalty, that can be used with most of the arithmetic instructions.

In Section 2.7 further details about the specific processors used in this work will be presented, while in the next sections it will be described the main families of ARM processors: Cortex and SecurCore.

2.4.1 Cortex A processors

Cortex-A is a family of processors adequate for a wide range of mobile, consumer, embedded and infrastructure devices. They feature processors using the ARMv7 architecture, with support to 32-bit instruction set and mixed 16/32-bit Thumb2 instructions; and it also features ARMv8 processors, that support AArch32 and AArch64 execution states. Cortex-A processors also offer support for rich Operating Systems, including Linux, Android, and Chrome. Cortex-A processors can also feature architecture extensions, such as SIMD and Advaced SIMD(NEON), VFP, ThrustZone and others [55].

2.4.2 SIMD instructions - NEON

All the processors used in this work have the general-purpose SIMD extension engine called NEON. It is a technology that uses a 128-bit Single Instruction Multiple Data architecture extension, designed to provide acceleration for algorithms such as video encoding and decoding, gaming, audio processing, image processing and cryptography. The NEON technology was introduced on ARMv7-A and ARMv7-R. Each processor

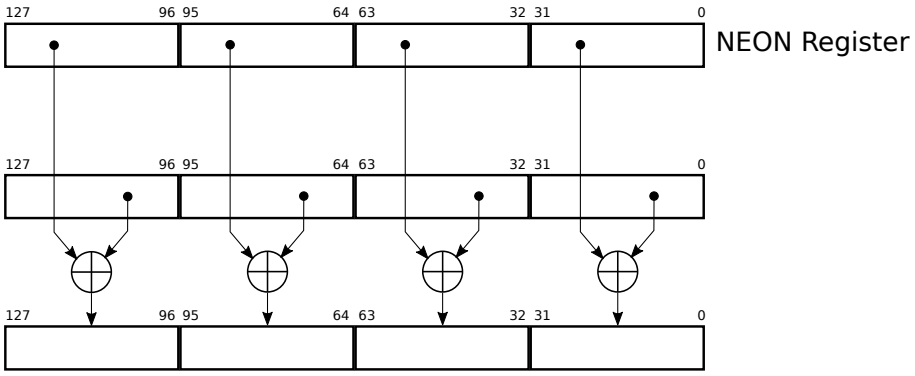

Figure 2.7: Conceptual layout of a NEON instruction, with the same operation being performed on multiple elements stored in a NEON register.

has thirty-two 64-bit wide registers, which can be reinterpreted as sixteen 128-bit wide registers. Those registers can be seen as vectors of elements with the same data type (8-bit, 16-bit, 32-bit, 64-bit signed/unsigned; single precision floats and polynomials).

Each NEON instruction performs the same operation in all elements of the vector, as shown in Figure 2.7. NEON can be used in various ways, such as NEON enabled libraries, auto-vectorization features on compilers, intrinsics, and assembly code.

2.4.3 Cortex M processors

ARM Cortex-M is a family of energy-efficient processors, adequate for use on embedded applications. The Cortex-M line of processors use the Thumb-2 instruction set, that offers advantages over 8-bit, 16-bit, and fixed 32-bit architectures by reducing memory requirements and saving on-chip flash memory. This instruction set supports the 16-bit Thumb instructions and is extended with 32-bit instructions. In many cases, the compiler will optimize code using the smaller 16-bit instructions, only using 32-bit instructions where they are more efficient. The Cortex-M processors are used in a variety of applications, including sensor fusion, environmental, wearable technology, medical instruments, smart cities, and automotive systems [56].

2.4.4 Other lines of processors

ARM also designs and licenses real-time processors –the Cortex-R family of processors–, adequate for embedded systems with characteristics such as reliability, high availability, and fault tolerance. Beyond that, those processors are also adequate for applications where the system functionality is directly responsible to avoid hazardous situations, such as in autonomous and medical systems [57].

Another new addition to the ARM family are the SecurCore processors. Those processors are based on pre-existent ARM cores, coupled together with features specific for security applications. Those are cores designed for tamper-resistant applications, such as smart cards, payment systems, electronic passports, SIM cards and electronic tickets. SecurCore is built over the Cortex-M designs, and as such may share various charac-

teristics with those, but further details about the features are only obtainable under a non-disclosure agreement from ARM [58].

2.5 Algorithm choice

For this work, two algorithms from CAESAR were selected amongst the second round candidates[3]. Characteristics such as documentation, the existence of reference code, cryptanalysis and performance on ARM processors were taken into account. The quality of documentation was a subjective point for choosing the algorithms and was based on personal preferences and perceptions; cryptanalysis was a quantitative one: checking whether a determinate algorithm had been analyzed and no defects found. The two main sources of information for that choice were the Authenticated Encryption Zoo[34] –An website hosted by DTU Compute at the Technical University of Denmark, with the objective of providing an up-to-date overview of the cryptanalysis results of CAESAR– and the CAESAR newsgroup[4] [59]. Regarding performance, the results from SUPERCOP's eBAEAD [60] were used to choose the algorithms.

Initially, an algorithm based on a sponge design and another based on a tweakable block cipher were chosen, to better represent the different constructions. The tweakable block cipher candidate –SCREAM [61]– was not chosen for the third round of CAESAR and was replaced by another sponge-based candidate. The chosen family of ciphers were NORX, with two different algorithms with 128-bit security and two with 256-bit security; and Ascon, with two different algorithms both with 128-bit security.

2.6 Software implementation for cryptographic functions

The author of *Secrets and Lies* said "the weak points [of cryptographic implementations] had nothing to do with mathematics [...] Beautiful pieces of mathematics were made irrelevant through bad programming" [62]. When cryptographic algorithms are implemented, either in software or hardware, correctness and efficiency are not the only goals: Computer systems leak information about their processing through various side-channels. Leakage can occur via execution time, cache operations, power consumption, electromagnetic emanations and others. These leaks of information can be exploited to construct side-channel attacks with the objective of recovering secret data that has been processed. Particularly vulnerable to these attacks are mobile devices and sensor nodes, due to the fact of being usually deployed for work in non-controlled and hostile environments.

Secure implementations in software normally target protection against side channel attacks on execution time, cache operations, and memory access. Timing attacks are based on measuring the time taken by various computations in order to recover se-

[3] At the time of the start of this work, the second round candidates were the latest announcement from the committee.

[4] As of December 2017, the public group was moved to a mailing list at `http://cr.yp.to`. The old messages are still accessible on `http://groups.google.com/forum/#!forum/crypto-competitions`.

cret information. For example, a secret-dependent multiplication can finish in a time related to the bit pattern in the operands. Implementations must then use operations and techniques that allow the code to execute in constant time, independent of its secret inputs. Cache attacks are based on the attacker's ability to monitor cache access. For example, a table lookup can generate cache misses, forcing a new part of the table to be loaded into the cache memory. An attacker can then use this information to infer the table's access pattern and correlate it with secret data [63]. Furthermore, such construction can also lead to time leaks, via the difference in time needed to access cached and uncached information. This type of attack is known to even compromise cross-VM data in Cloud computing platforms [64]. This concept can also be broadened to RAM and disk access. The use of secret-dependent tables should be avoided when implementing secure algorithms, or when ultimately needed, using countermeasures to protect the data, such as Masking [65][66] and Randomization [67]. Another weak point in cryptographic implementations is branching: Different execution paths can also be leveraged by an attacker to recover information about the algorithm execution, and mathematical techniques should be employed when decision structures are needed in the code.

While not relevant to the scope of this work, software can also be tweaked to mitigate some side channels more easily solved using on hardware implementations. For example, power attacks can be mitigated by representing data in a constant-weight code, what in turns can reduce leakage about the hamming weight of secret data [68]. Other side channels, such as magnetic, optic, and acoustic emanations, are hard to protect via only software and need specific countermeasures in a hardware implementation. One example of side channel attack tackling these channels is NSA TEMPEST, a program and NATO certification on spying systems through radio and electrical signals, sounds, and vibrations.

2.7 ARM Architecture: Target processors

In this dissertation we choose to target ARM processors mainly due to their widespread presence in IoT and small devices. Specifically, we choose two 32-bit ARMv7 cores –Cortex A7 and A15–, a 64-bit ARMv8 processor –Cortex A53–, and smaller Cortex M for validation. In this section, some specific characteristics of those cores will be introduced.

Target processors: Cortex A

Cortex A processors are ARMv7 and ARMv8 cores, used on a range of devices capable of undertaking complex tasks, such as hosting rich operating systems. In this work, we focused on two 32-bit ARMv7 cores, and one 64-bit ARmv8 core. The main characteristics of the cores targeted in this work are as follows [55]:

- Cortex-A7: Currently the most power efficient ARMv7-A core, with over a billion shipped units in production. The processor is capable of 40-bit physical

addressing and has an eight-stage in-order pipeline. The A7 core is compatible with higher performance cores such as the Cortex-A15 and A17 for use with the big.LITTLE technology, where high-performance cores are combined with highly efficient cores in a heterogeneous computation approach.

- Cortex-A15: A high-performance ARMv7-A core, well suited to consumer items such as smartphones and embedded applications. As with other processors of the same line, it is capable of 40-bit physical addressing. It also features a 15 stage pipeline for integer calculations.
- Cortex-A53: An ARMv8-A core capable of seamlessly running both 32-bit and 64-bit code, and is made as an efficient 64-bit core for a low area and power footprint. Like the Cortex-A7, it is capable of being deployed together with high-end CPUs for chips with heterogeneous cores. The Cortex-A53 uses an efficient eight-stage in-order pipeline.

Reference processors: Cortex M

The Cortex M is a family of low-power, energy efficient processors, specially designed for embedded applications. In this work, three cores of this family are used:

- Cortex-M0: The M0 is a very small core, with a footprint as low as 12K gates, being the low-cost low-power core made for deeply embedded applications. The entire instruction set of this core is composed of 56 instructions, with a C-friendly construction, with most of them being 16-bit Thumb-2 instructions, what yields good code density and reduced usage of on-chip flash memory. This core is used on cost-sensitive devices, such as analog mixed signal devices, finite state machines, power management, motor controllers, health and environment monitoring, and wearables.
- Cortex-M3: It is considered an optimal System-on-chip processor, with a good balance between power and performance. This core also takes advantage of the Thumb-2 code density, that mixes 16-bit and 32-bit instructions. This core is used in wearable and IoT devices, motor control, domestic appliances, smart homes and connectivity applications.
- Cortex-M4: The M4 core is a high-performance embedded processor, developed with a focus on digital signal control. The M4 provides all the features of the Cortex-M3 plus SIMD instructions and fast, single-cycle MAC operations. It can optionally support IEEE 754 floating point operations. Another key feature of this core is low dynamic power for 32-bit instructions, what delivers a good system energy efficiency. This core is used on IoT applications, sensor fusion, signal processing, audio processing, and smart homes.

2.8 Testing methodology

The implemented algorithms were validated for correctness in the following ways:

- Matching the algorithm outputs with the test vectors published in their respective specifications, when such data is available.
- Matching the algorithm outputs and internal state in various steps –for example, after each sponge permutation– with reference implementations, as to guarantee both have the same behavior.
- Testing sanity, by checking if $\mathcal{D}^h_{k,n}(\mathcal{E}^h_{k,n}(m)) = m$ for random values of key, nonce, additional data and message. The last two, also with variable lengths.

Beyond that, the algorithms are implemented in such a way that benchmarks are only executed after a successful correctness check, in order to guarantee that the optimizations do not change the algorithm specification.

For benchmarking, a series of macros were used to instrument the calls to encrypt and decrypt functions. All measurements are done using the cycle counter of each architecture, and the reported numbers are the average of at least five executions. Since empirically there is a small variance in the reported values (less than 0.1 cycles per byte), the average is a good compromise for the representing value. In cases where there is a larger variance, computing the median of the iterations, or other statistical approaches may be more adequate, but won't be discussed in the scope of this work. All the comparisons are carried between code compiled on the same machine, using the same versions of kernel and compiler, with measurements being taken in succession. Measuring cycles with precision is a fairly difficult problem in various architectures, therefore it is difficult to make claims regarding the absolute precision of the reported figures. On the other hand, tests show that the used method is self consistent across different implementations, and matches published values for known implementations.

For ARM-v7-A architectures, the following code is used to access the contents of the performance counter, that is used to measure elapsed clock cycles:

```
1 asm("mrc p15, 0, %0, c9, c13, 0" : "=r"(value));
```

For ARM-v8 AArch64 processors, the following code is used:

```
1 asm ("mrs %0, pmccntr_el0" : "=r" (r));
```

In order to use these measurements, the performance counters need to be enabled via a loadable kernel module, using the following code:

```
1 asm ("MCR p15, 0, %0, C9, C14, 0\n\t" :: "r"(1));
```

The performance counter can be disabled by executing the same code but with the constant `0x8000000f` instead of `0x01`.

These calls, wrapped with report macros, are done right before and after the measured function, together with a forceful sync instruction. This method is similar to the one used by ECRYPT's SUPERCOP [60] benchmarking platform, albeit simpler. Complete benchmark code is shown in Appendix C

Chapter 3

Software implementation: NORX AEAD

NORX is an authenticated encryption scheme due to Jean-Philippe Aumasson, Philipp Jovanovic and Samuel Neves [50], supporting associated data in the form of both headers and trailers. NORX also supports arbitrary parallelism and is optimized for efficient hardware and software implementations, with a SIMD-friendly construction, no secret array indexing, and only bitwise operations. ARX[1] primitives are thoroughly used, without modular additions, and is based on the monkey-duplex construction. NORX's core permutation function is based on ChaCha's permutation [69], with the integer addition $(a+b)$ replaced by the approximation[2] $a \oplus b \oplus (a \wedge b) \ll 1$, which in turn –according to the design team of NORX– simplifies cryptanalysis and improves hardware efficiency [50].

3.1 Description of NORX family of algorithms

The NORX family of algorithms is parametrized by the word size in bits w; a round number ℓ with $1 \leq \ell \leq 63$; a parallelism degree p with $0 \leq p \leq 255$ (where $p = 0$ defines arbitrary parallelism) and a tag length t. Regarding the key length, NORX32 uses a 128-bit key, NORX64 a 256-bit key, while NORX16 and NORX08 use a 96-bit and an 80-bit key respectively. The 32 and 64-bit versions of NORX also use an $n = 4w$ bits nonce; the 8-bit and 16-bit variants have a nonce of $n = 32$ bits.

On the CAESAR submission, Aumasson et al. [50] propose five instances of NORX for different uses cases. They are listed in Table 3.1, from the highest recommendation at the top to the lowest. The naming convention for a specific instance of the algorithm is NORXw-l-p-t, with w, l, p and t being the instance parameters. When the tag length is the default $t = 4w$, then the notation is shortened as NORXw-l-p.

NORX parametrized with $w = 32$ bits is adequate for lightweight applications and resource-constrained environments, requiring small hardware area and small ROM size for software implementations. On the other hand, the instances with $w = 64$ bits are

[1] Addition, rotation, and exclusive OR.

[2] This approximation is derived from the identity $a + b = (a \oplus b) + (a \wedge b) \ll 1$.

adequate for high-performance and high-security applications, being efficient in both 64-bit and 32-bit CPUs[3]. Requirements for ASIC implementations are about 64 kGE, and at most 64 bytes of ROM for the initialization constants. It is also possible to implement NORX using only one byte plus the sponge size of data in RAM [50].

Table 3.1: The five instances of NORX

Instance name	w	l	p	t	k	n
NORX64-4-1	64	4	1	256	256	128
NORX32-4-1	32	4	1	128	128	64
NORX64-6-1	64	6	1	256	256	128
NORX32-6-1	32	6	1	128	128	64
NORX64-4-4	64	4	4	256	256	128

NORX follows a duplexed sponge layout, as shown in Figure 3.1. NORX's construction allows parallel processing of the payload, def ined by p. For serial processing, with $p = 1$, the layout of NORX is that of a standard duplexed sponge. For a value $p > 1$, the number of parallel processing lanes is given by the value of p; for example, Figure 3.1 illustrates the case of $p = 2$. For $p = 0$, the number of processing lanes is bounded by the size of the payload itself, making the layout of NORX similar to that of the PPAE construction [70]. In this section, we will follow the same convention for variable names as the original NORX specification, summarized in Table 3.2.

Table 3.2: Common Norx variables

Variable	Description
S	The 16-word internal state of the sponge construction
$s_0, \cdots, s_{16}$	The individual words of S
$\bar{S}$	State for the parallel payload processing
F and F^ℓ	The permutation function and ℓ applications of F
$G()$	The column/diagonal transformation function
$r_0, \cdots, r_3$	NORX rotation constants
K, N, T	Respectively key, nonce and authentication tag
$Msg, Cipher$	Respectively plaintext, ciphertext
A, Z	Additional data as header and trailer, respectively
$u_0, \cdots, u_{15}$	Norx initialization constants
v	Domain separation tag
w	Word size, in bits
b, r, c	Size on bits of Sponge, Sponge rate, and Spong capacity

[3]A draft of these use-cases can be found in the CAESAR mailing list at the address `https://groups.google.com/forum/#!topic/crypto-competitions/DLv193SPSDc`.

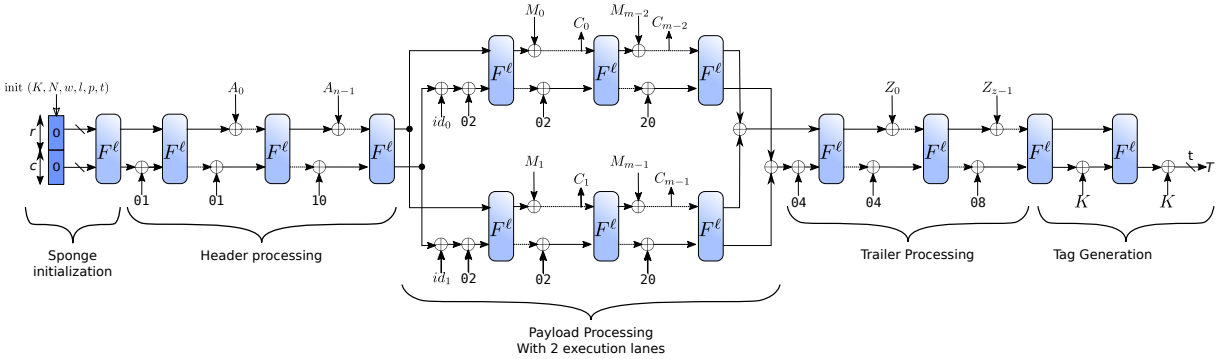

Figure 3.1: The layout of NORX with parallelism degree $p = 2$. Notice that the sponge is divided into multiple execution lanes in the payload processing step. Those lanes can be computed in parallel, as there is no data dependency amongst them. Based on a figure from [50].

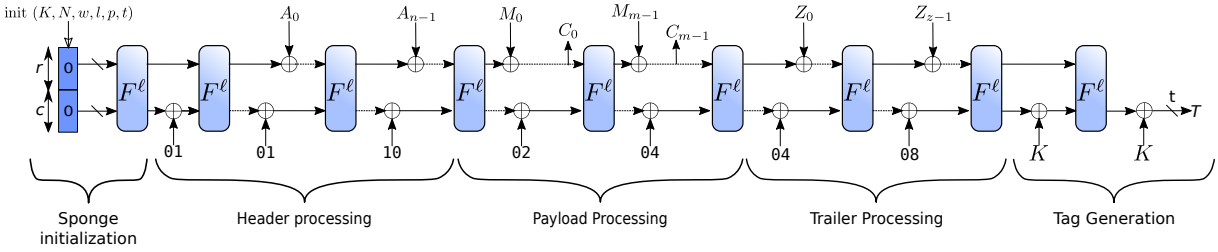

Figure 3.2: The layout of NORX with parallelism degree $p = 1$. Based on a figure from [50].

NORX's core is the permutation function $F^\ell()$, applied to the NORX internal state S, with ℓ being the number of rounds, and defined as the ℓ fold iteration $F^\ell(S) = F(F(\cdots F(S)))$. The state is a concatenation of 16 w-bit words in the form $S = s_0 \parallel \cdots \parallel s_{15}$, where the words $s_0, \cdots, s_{11}$ are called the *rate words*, where data is injected and extracted from, and the remaining words $s_{12}, \cdots, s_{15}$ are called *capacity words*. Conceptually, the state S can be viewed as a 4×4 matrix:

$$S = \begin{pmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{pmatrix}$$

A single permutation $F()$ processes the state S by applying the function $G : (a, b, c, d) \rightarrow (a, b, c, d)$ to the matrix's columns and then diagonals. $G()$ is described in Algorithm 1, and the permutation $F()$ is specified in Algorithm 2. The round permutation $F^\ell(S)$ is also illustrated in Figure 3.3. The rotation constants are shown in Table 3.3

Table 3.3: NORX's rotation constants [50]

w	r_0	r_1	r_2	r_3
32	8	11	16	31
64	8	19	40	63

$F^{\ell}(S) = \ell$ times

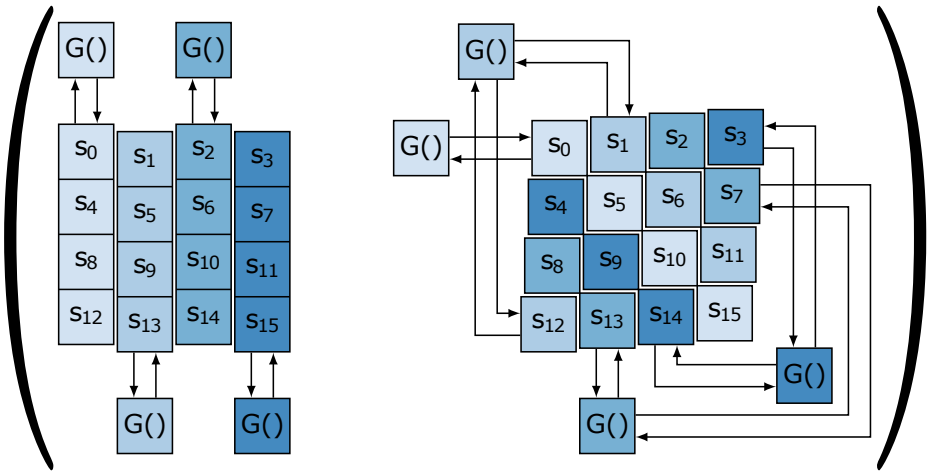

Figure 3.3: Column (left) and diagonal (right) steps of NORX, composing $F^{\ell}(S)$. Adapted from: Norx v3.0 specification [50].

Algorithm 1 Function $G()$ of NORX

Input: a, b, c, d ▷ Four w-bit words of the internal State
Output: a, b, c, d ▷ Four w-bit words of the internal State
Function $G(a, b, c, d)$
$a \leftarrow (a \oplus b) \oplus ((a \wedge b) \ll 1)$
$d \leftarrow (a \oplus d) \ggg r_0$
$c \leftarrow (c \oplus d) \oplus ((c \wedge d) \ll 1)$
$b \leftarrow (c \oplus b) \ggg r_1$
$a \leftarrow (a \oplus b) \oplus ((a \wedge b) \ll 1)$
$d \leftarrow (a \oplus d) \ggg r_2$
$c \leftarrow (c \oplus d) \oplus ((c \wedge d) \ll 1)$
$b \leftarrow (c \oplus b) \ggg r_3$
return a, b, c, d
end Function

Algorithm 2 Round function $F()$ of NORX

Input: $S, G : (a, b, c, d) \rightarrow (a, b, c, d)$ ▷ Norx State $S = \{s_0, \cdots, s_{15}\}$ and function G
Output: S ▷ Norx State after one permutation round
Function F
/* *Processing the columns* */
$s_0, s_4, s_8, s_{12} \leftarrow G(s_0, s_4, s_8, s_{12})$
$s_1, s_5, s_9, s_{13} \leftarrow G(s_1, s_5, s_9, s_{13})$
$s_2, s_6, s_{10}, s_{14} \leftarrow G(s_2, s_6, s_{10}, s_{14})$
$s_3, s_7, s_{11}, s_{15} \leftarrow G(s_3, s_7, s_{11}, s_{15})$
/* *Processing the diagonals* */
$s_0, s_5, s_{10}, s_{15} \leftarrow G(s_0, s_5, s_{10}, s_{15})$
$s_1, s_6, s_{11}, s_{12} \leftarrow G(s_1, s_6, s_{11}, s_{12})$
$s_2, s_7, s_8, s_{13} \leftarrow G(s_2, s_7, s_8, s_{13})$
$s_3, s_4, s_9, s_{14} \leftarrow G(s_3, s_4, s_9, s_{14})$
return $s_0 \cdots s_{15}$
end Function

NORX's encryption and decryption primitives are described in Algorithm 3 and 4, where `header`, `branch`, `payload`, `merge`, `trailer` and `tag` are domain separation constants; K is the key, N is the nonce, A is the additional data to be processed before the plaintext, `Msg` is the plaintext, Z is the additional data to be processed after the plaintext, `Tag` is the authentication tag, and `Cipher` is the ciphertext. For more details on the algorithm's description, the reader is invited to see chapter 2 of [50].

Algorithm 3 NORX AEAD encryption

Input: K, N, A, Msg, Z ▷ Key, Nonce, Additional header data, plaintext, and Additional trailer data
Output: $Cipher, Tag$ ▷ Encrypted plaintext and authentication tag
Function Encrypt(K, N, A, Msg, Z)
$S \leftarrow \text{initialise}(K, N)$
$S \leftarrow \text{absorb}(S, A, \texttt{header})$
$\bar{S} \leftarrow \text{branch}(S, |Msg|, \texttt{branch})$
$\bar{S}, Cipher \leftarrow \text{payloadEncrypt}(\bar{S}, Msg, \texttt{payload})$
$S \leftarrow \text{merge}(\bar{S}, |Msg|, \texttt{merge})$
$S \leftarrow \text{absorb}(S, Z, \texttt{trailer})$
$S, Tag \leftarrow \text{finalise}(S, \texttt{tag})$
return $Cipher, Tag$
end Function

Algorithm 4 NORX AEAD Decryption

Input: $K, N, A, Cipher, Z, T$ ▷ Key, Nonce, Additional header data, ciphertext, Additional trailer data, and tag
Output: Msg, Tag or $\perp$ ▷ Decrypted message and authentication code, or failure symbol
Function Decrypt($K, N, A, Cipher, Z, T$)
  $S \leftarrow \text{initialise}(K, N)$
  $S \leftarrow \text{absorb}(S, A, \texttt{header})$
  $\bar{S} \leftarrow \text{branch}(S, |Cipher|, \texttt{branch})$
  $\bar{S}, Msg \leftarrow \text{payloadDecrypt}(\bar{S}, Cipher, \texttt{payload})$
  $S \leftarrow \text{merge}(\bar{S}, |Cipher|, \texttt{merge})$
  $S \leftarrow \text{absorb}(S, Z, \texttt{trailer})$
  $S, Tag' \leftarrow \text{finalise}(S, \texttt{tag})$
  if $Tag' == T$ **then return** Msg, Tag
  else
    return $\perp$ ▷ Symbol for failed decryption
  end if
end Function

The functions for `absorb`, `branch`, `decryptPayload`, `encryptPayload`, `finalise`, `initialise`, and `merge` are described, in order, on Algorithm 6, 7, 10, 9, 11, 5, and 8. The initialization constants $(u_0, \cdots, u_{15})$ are given in Table 3.5

3.1.1 Padding

NORX uses a *multi-rate padding*, defined by the map $\text{pad}_r : X \to X \parallel 10^u 1$ where X is a bit string and $u = (-|X| - 2 \mod r)$. In the case where $|X|$ and r are divisible by 8 and X is a sequence of bytes, then the padding rule can be written as:

$$\text{pad}_r : \begin{cases} X \to X \parallel \texttt{0x01} \parallel \texttt{0x00}^u \parallel \texttt{0x80} & \text{if} \quad u > 0 \\ X \to X \parallel \texttt{0x81} & \text{if} \quad u = 0 \end{cases}$$

3.1.2 Domain separation constants

NORX features a simple domain separation mechanism, where different constants are XORed to the least significant byte of s_{15} before a transformation of the state by F^ℓ. Table 3.4 specifies the values of those constants, while Figure 3.1 and Figure 3.2 illustrate the constant integration into the mode of operation.

Table 3.4: Norx domain separation constants

`header`	`payload`	`trailer`	`tag`	`branching`	`merging`
01	02	04	08	10	20

3.1.3 Sponge initialization

The initialization function prepares the internal $16w$-bit state by combining the key, nonce, instance parameters, and the initialization constant as described in Algorithm 5. The initializations constants are listed in Table 3.5 and can be calculated as

$$(u_0, \cdots, u_{15}) \leftarrow F^2(0, \cdots, 15)$$

with the parameters of F having the adequate bit-length for the NORX instance. Notice that only the subset of values $\{u_8, \cdots, u_{15}\}$ are used in the sponge initialization method.

Algorithm 5 NORX AEAD initialization

Input: K, N ▷ Key and public nonce
Output: S ▷ Initialized state
Function initialise(K, N)
$k_0, k_1, k_2, k_3 \leftarrow K$, s.t. $|k_i| = w$
$n_0, n_1, n_2, n_3 \leftarrow N$, s.t. $|n_i| = w$
$S \leftarrow (n_0, n_1, n_2, n_3, k_0, k_1, k_2, k_3, u_8, u_9, u_{10}, u_{11}, u_{12}, u_{13}, u_{14}, u15)$
$(s_{12}, s_{13}, s_{14}, s_{15}) \leftarrow (s_{12}, s_{13}, s_{14}, s_{15}) \oplus (w, \ell, p, t)$ ▷ Parameter addition
$S \leftarrow F^{\ell}(S)$ ▷ State permutation
$(s_{12}, s_{13}, s_{14}, s_{15}) \leftarrow (s_{12}, s_{13}, s_{14}, s_{15}) \oplus (k_0, k_1, k_2, k_3)$ ▷ Key addition
return S
end Function

Table 3.5: Norx Initialization constants.

w	32	64
u_0	0454EDAB	E4D324772B91DF79
u_1	AC6851CC	3AEC9ABAAEB02CCB
u_2	B707322F	9DFBA13DB4289311
u_3	A0C7C90D	EF9EB4BF5A97F2C8
u_4	99AB09AC	3F466E92C1532034
u_5	A643466D	E6E986626CC405C1
u_6	21C22362	ACE40F3B549184E1
u_7	1230C950	D9CFD35762614477
u_8	A3D8D930	B15E641748DE5E6B
u_9	3FA8B72C	AA95E955E10F8410
u_{10}	ED84EB49	28D1034441A9DD40
u_{11}	EDCA4787	7F31BBF964E93BF5
u_{12}	335463EB	B5E9E22493DFFB96
u_{13}	F994220B	B980C852479FAFBD
u_{14}	BE0BF5C9	DA24516BF55EAFD4
u_{15}	D7C49104	86026AE8536F1501

3.1.4 Absorption

The absorption method, used on the trailer and header authentication, takes a string X of arbitrary size and "absorbs" it into the internal states as blocks of r bits. If the last block of data is smaller than r, it is extended using the previously described padding rule. The absorption method is skipped if the input is an empty bit string. Algorithm 6 describes the *absorb* function.

Algorithm 6 NORX AEAD additional data absorption

Input: S, X, v ▷ State, data to be authenticated and domain constant.
Output: S ▷ NORX State after data absorption
Function absorb(S, X, v)
  $X_0 \parallel \cdots \parallel X_{m-1} \leftarrow X$, s.t. $|X_i| = r$, $0 \leq |X_{m-1}| < r$
  if $|X| > 0$ **then**
    for $i \in \{0, \cdots, m-2\}$ **do**
      $s_{15} \leftarrow s_{15} \oplus v$
      $S \leftarrow F^{\ell}(S)$
      $S \leftarrow S \oplus (X_i \parallel 0^c)$
    end for
    $s_{15} \leftarrow s_{15} \oplus v$
    $S \leftarrow F^{\ell}(S)$
    $S \leftarrow S \oplus (\text{pad}_r(X_{m-1}) \parallel 0^c)$
  end if
return S
end Function

3.1.5 Branching and Merging

If the parallelism degree $p \neq 1$, the branch method is used to prepare the internal state for parallel payload processing, effectively creating multiple independent sponges by extending the internal state into the multi-state $\bar{S}$. The branching method is skipped if the parallelism degree $p = 1$ or if the message M is empty. Algorithm 7 describes the branching process.

Algorithm 7 NORX AEAD sponge branching

Input: S, m, v ▷ State, message length, and domain constant
Output: $\bar{S}$ ▷ Multistate for parallel payload processing
Function branch(S, m, v)
  $\bar{S} \leftarrow 0^b$
  if $p \neq 1$ and $m > 0$ **then**
    $s \leftarrow p$
    if $p = 0$ **then**
      $s \leftarrow \lceil m/r \rceil$
    end if
    $\bar{S} = (\bar{S}_0, \cdots, \bar{S}_{s-1}) \leftarrow (0^b, \cdots, 0^b)$
    $s_{15} \leftarrow s_{15} \oplus v$

$S \leftarrow F^{\ell}(S)$
for $i \in \{0, \cdots, s-1\}$ **do**
$\bar{S} \leftarrow S \oplus (i, i, i, i, i, i, i, i, i, i, i, i, 0, 0, 0, 0)$
end for
else
$\bar{S} \leftarrow S$
end if
return $\bar{S}$
end Function

After the payload processing, the merge function is executed reduce the Multistate $\bar{S}$ of $16wp$ bits back into the $16w$-bit single sponge internal state S. Algorithm 8 describes the `merge` function.

Algorithm 8 NORX AEAD sponge merge

Input: $\bar{S}, m, v$ ▷ Multistate, message length and domain constant
Output: S ▷ Single $16w$-bit state
Function merge$(\bar{S}, m, v)$
$S \leftarrow 0^b 4$
if $p \neq 1$ and $m > 0$ **then**
for $i \in \{0, \cdots, |\bar{S}_b| - 1\}$ **do**
$\bar{s}_{i,15} \leftarrow \bar{s}_{i,15} \oplus v$
$\bar{S}_i \leftarrow F^{\ell}(\bar{S}_i)$
$S \leftarrow S \oplus \bar{S}_i$
end for
else
$S \leftarrow \bar{S}$
end if
return S
end Function

3.1.6 Encryption and decryption

The payload encryption method takes an arbitrary long bit-string M and encrypts it, producing the encrypted payload C. The message M is absorbed into the sponge during this processes, therefore ensuring its authenticity. Similarly, the payload decryption method takes the ciphertext C and output the decrypted payload. Both functions process their inputs in blocks of r bits, with the last block being padded when it is smaller than r.

Regarding NORX's parallel processing feature, $p = 1$ executes the encryption and decryption as a usual duplexed sponge. In the case where $p > 1$, p lanes are used for data processing, with data blocks of r bits being distributed in a round-robin fashion, e.g. the i-th data block will be processed by the state $i \mod p$. When $p = 0$, each data block is processed by its own separate lane. Since the lane number is integrated into the state during branching, that implies a maximum message size of $2^w r$ bits, what is

approximately $2^{32,58}$ bytes for NORX32 and $2^{64,58}$ bytes for NORX64. Algorithm 9 and Algorithm 10 show the encryption and decryption procedures, respectively.

Algorithm 9 NORX AEAD payload encryption

Input: $\bar{S}, M, v$ ▷ (Multi)State, plaintext and domain constant
Output: $\bar{S}, C$ ▷ (Multi)State and ciphertext
Function encryptPayload$(\bar{S}, M, v)$
  $C \leftarrow \varepsilon$
  $M_0 \parallel \cdots \parallel M_{m-1} \leftarrow M$ s.t. $|M_i| = r$, $0 \leq |M_{m-1}| < r$
  if $|M| > 0$ **then**
    for $i \in \{0, \cdots, m-2\}$ **do**
      $j \leftarrow i \mod |\bar{S}|_b$
      $\bar{s}_{j,15} \leftarrow \bar{s}_{j,15} \oplus v$
      $S_j \leftarrow F^\ell(\bar{S}_j)$
      $M_i \leftarrow \mathsf{left}_r(\bar{S}_j) \oplus M_i$
      $\bar{S}_j \leftarrow C_i \parallel \mathsf{right}_c(\bar{S}_j)$
    end for
    $j \leftarrow (m-1) \mod |\bar{S}|_b$
    $\bar{s}_{j,15} \leftarrow \bar{s}_{j,15} \oplus v$
    $\bar{S}_j \leftarrow F^\ell(\bar{S}_j)$
    $C_{m-1} \leftarrow \mathsf{left}_{|M_{m-1}|}(\bar{S}_j) \oplus M_{m-1}$
    $\bar{S}_j \leftarrow \bar{S}_j \oplus (\mathsf{pad}_r(M_{m-1}) \parallel 0^c)$
    $C \leftarrow C_0 \parallel \cdots \parallel C_{m-1}$
  end if
  return $\bar{S}, C$
end Function

Algorithm 10 NORX AEAD payload decription

Input: $\bar{S}, C, v$ ▷ (Multi)State, ciphertext and domain constant
Output: $\bar{S}, M$ ▷ (Multi)State and plaintext
Function decryptPayload$(\bar{S}, C, v)$
  $M \leftarrow \varepsilon$
  $C_0 \parallel \cdots \parallel C_{m-1} \leftarrow C$ s.t. $|C_i| = r$, $0 \leq |C_{m-1}| < r$
  if $|C| > 0$ **then**
    for $i \in \{0, \cdots, m-2\}$ **do**
      $j \leftarrow i \mod |\bar{S}|_b$
      $\bar{s}_{j,15} \leftarrow \bar{s}_{j,15} \oplus v$
      $S_j \leftarrow F^\ell(\bar{S}_j)$
      $M_i \leftarrow \mathsf{left}_r(\bar{S}_j) \oplus C_i$
      $\bar{S}_j \leftarrow C_i \parallel \mathsf{right}_c(\bar{S}_j)$
    end for
    $j \leftarrow (m-1) \mod |\bar{S}|_b$
    $\bar{s}_{j,15} \leftarrow \bar{s}_{j,15} \oplus v$
    $\bar{S}_j \leftarrow F^\ell(\bar{S}_j)$

$M_{m-1} \leftarrow \text{left}_{|C_{m-1}|}(\bar{S}_j) \oplus C_{m-1}$
$\bar{S}_j \leftarrow \bar{S}_j \oplus (\text{pad}_r(M_{m-1}) \parallel 0^c)$
$M \leftarrow M_0 \parallel \cdots \parallel M_{m-1}$
end if
return $\bar{S}, M$
end Function

3.1.7 Finalization

The finalization function generates the authentication tag by executing two permutations F^ℓ and two key additions. The sponge capacity c is used as the authentication tag. The finalization function is described in Algorithm 11.

Algorithm 11 NORX AEAD finalization and tag generation

Input: S, K, v ▷ State, key and domain constant
Output: S, T ▷ Finalized State and Authentication tag
Function finalise(S, K, v)
$s_{15} \leftarrow s_{15} \oplus v$
$S \leftarrow F^\ell(S)$
$(s_{12}, s_{13}, s_{14}, s_{15}) \leftarrow (s_{12}, s_{13}, s_{14}, s_{15}) \oplus (k_0, k_1, k_2, k_3)$
$S \leftarrow F^\ell(S)$
$(s_{12}, s_{13}, s_{14}, s_{15}) \leftarrow (s_{12}, s_{13}, s_{14}, s_{15}) \oplus (k_0, k_1, k_2, k_3)$
$T \leftarrow \text{right}_t(S)$
return S, T
end Function

3.1.8 Tag verification

The tag verification, used in the decryption algorithm, while not an explicit part of the NORX specification, should be implemented in a secure way. This function verify is the received tag T is equal to the generated tag T'. The tag verification procedure should execute in constant time and should not leak any kind of information regarding the values of the bit-strings being compared. Another important feature of the Tag Verification procedure is that no payload should be returned after a failed verification, but rather a failure symbol $\perp$. Ideally, the "decrypted" bits generated previously should be securely erased from memory after a failed verification. A proposal for implementing this function is given in Algorithm-12, using a notation closer to the C language.

Algorithm 12 NORX AEAD Tag Verification

Input: T, T' ▷ Received tag and generated tag as bytestrings.
Output: r ▷ Return 0 if $T = T'$
Function verify(T, T')
let $|r| \geq$ 16-bit
$r \leftarrow 0$
for $0 < i <$ byte lenght of Tag **do**

$r \leftarrow r \vee (T_i \oplus T'_i)$
end for
$r \leftarrow (((r-1) \gg 8) \wedge 1) - 1$
return r
end Function

3.2 Code profiling

The first step to successfully optimize an algorithm is to identify points of interest, where most of the execution time is spent. In order to do that, after the first implementation of NORX, the Linux `perf` tool was used to determine the hotspots in both NORX32 and NORX64.

Perf is a Linux tool, available on the Linux kernel since version 2.6.31 [71], capable of lightweight profiling. It is included in the Linux kernel and uses the CPU hardware performance counters to detect events such as the number of instructions executed, cache-misses, branch mis-predictions, and others. Performance counters are the basis for profiling applications, and *perf* provides an easy-to-use abstraction over specific hardware capabilities. A small description of the usage of *perf*, as well as links to documentation is given in Appendix D.

For both NORX32 and NORX64, a basic encryption-decryption procedure was used to measure the function overheads. The only change done to code prior to measurement was disabling function inlining, what results in a better granularity and ease of analysis. The main loop of the code executes a call to the encryption function, with randomized inputs, and a 256KB payload length to minimize the impact of initialization overheads.

Figure 3.4 shows the relative overhead values for NORX3261 symbols, running on a 32-bit Cortex-A15. Similarly, Figure 3.5 shows the overhead results for NORX6461 on a 64-bit Cortex-A53. On both base cases, the empirical results match the expected behavior: The permutation function, used in all steps of the algorithm, is responsible for the largest code overhead and is, therefore, a great candidate for optimization. It should be noted that the symbol `sha256_compress` is not a part of NORX, but instead used by the pseudo-random number generator employed to feed the cipher inputs. *Perf* also allows the annotation of individual instruction overhead: Such deeper analysis shows a somewhat evenly distributed instruction overhead, with the presence of some load instructions with four times more overhead contribution than other close instructions. This fact suggests that an improvement in data handling could also result in positive gains. With that in mind, a pipeline-oriented code optimization was chosen as the main optimization strategy for NORX

3.3 Permutation optimization

Some smaller improvements were applied to the code, not strictly restricted to the permutation. The ones with a positive impact on the code performance were:

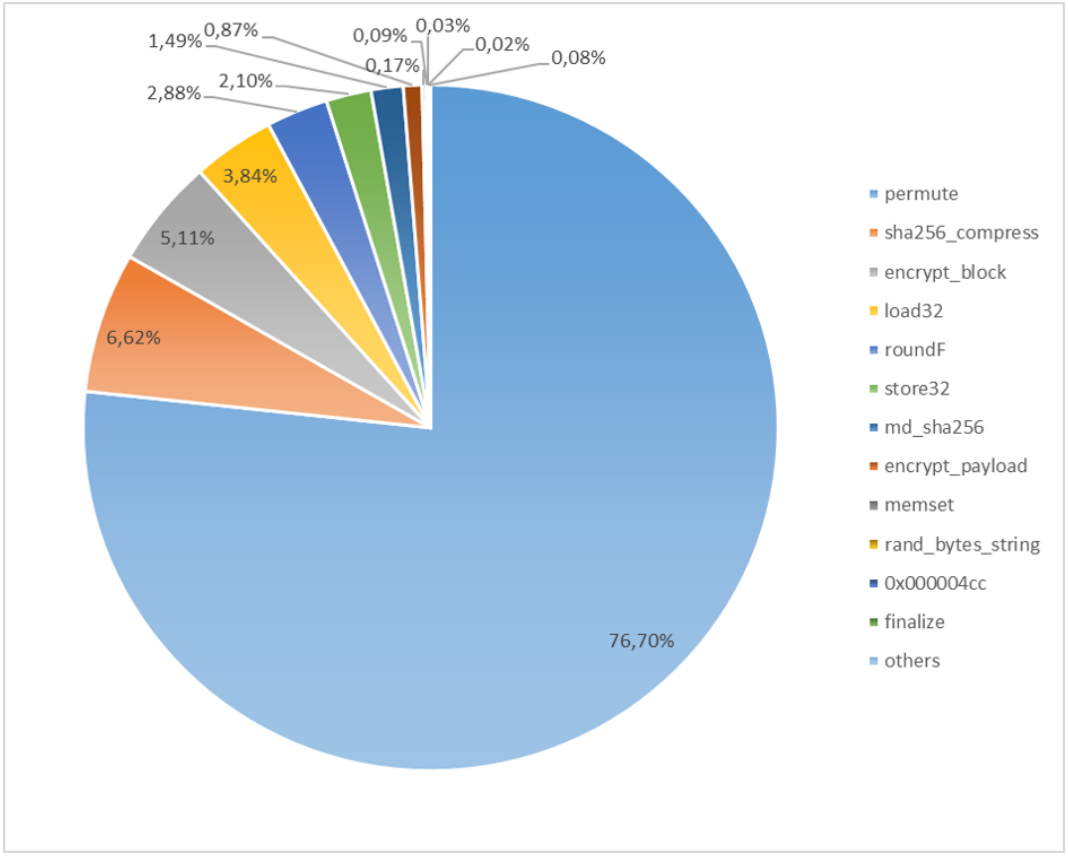

Figure 3.4: Overheads for NORX 3261 running on a Cortex-A15(32-bit processor)

- Extensive use of preprocessor macros and function inlining, which avoids overhead while still keeping code readability.
- Avoid the use of temporary variables whenever possible, performing most of the encryption, decryption, and additional data processing in place.
- Use a prefix operation instead of a postfix one on loop counters yields small improvements, more visible on Cortex-M based processors.
- Initialize the sponge using constants instead of calculating it as $F^2(0 \parallel 1 \parallel \cdots \parallel 15)$, where each number j is represented using w bits.
- Where possible, concatenate shift and rotate operations together with arithmetic operations, in order to use the target processor's barrel shifter, causing the shift operation to be executed at no cost.

Other approaches were tested, such as replacing `memcpy()` calls with loops of byte assignations, manually unrolling loops and changing memory alignment. Those did not impact the performance in any significant way, resulting in negligible variations in cycle count.

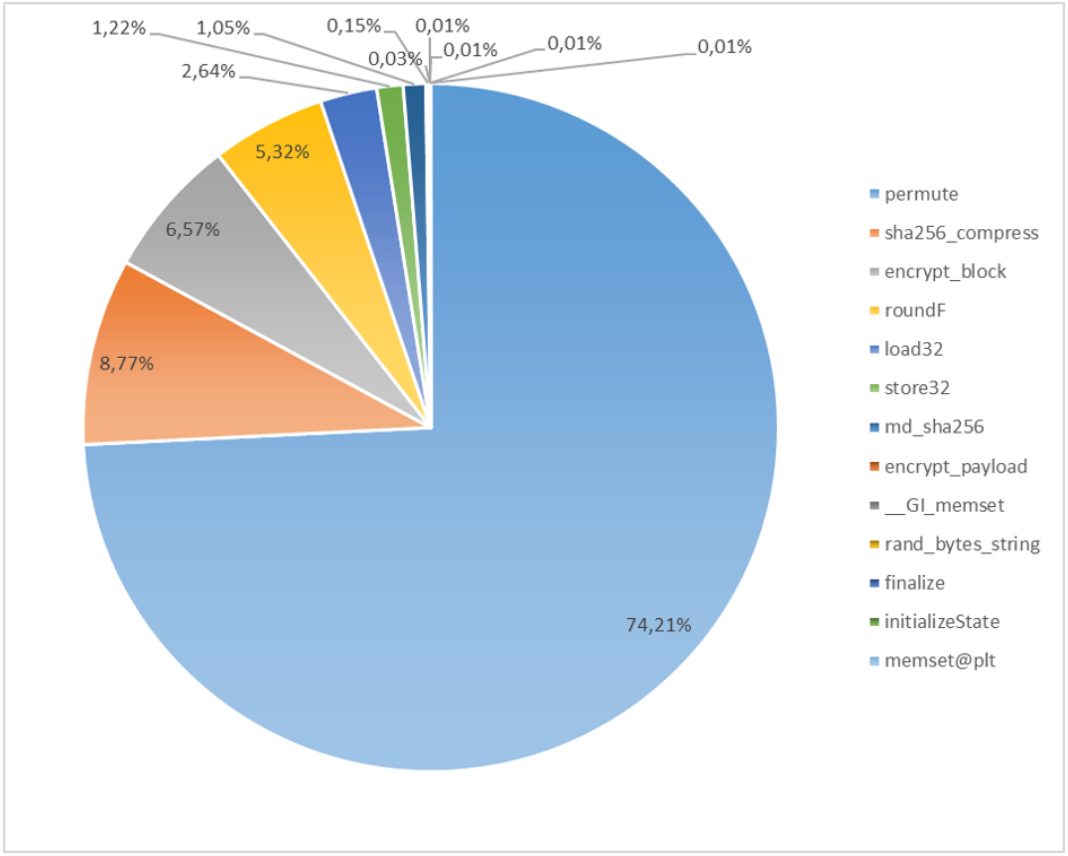

Figure 3.5: Overheads for NORX 6461 running on a Cortex-A53(64-bit processor)

3.4 Pipeline oriented optimization

The function G is executed on the columns and then on the diagonals of the 4×4 matrix representation of NORX State, using Algorithm 1 for each column and diagonal. Since there is no dependency between each column and diagonal, the function G can be rewritten in a way that each step of G is executed right after the other, for each column or diagonal. This way, the execution can be arranged in groups of two columns or four columns. This approach allows a better use of the processor's pipeline, improving the execution time. Figure 3.6 illustrates the idea behind using a pipeline-oriented implementation to optimize an algorithm.

In this way, the optimized code of function G for a 4-way pipeline is described in Algorithm 13 and the same function, optimized for a 2-way pipeline is described in Algorithm 14. In a similar way, Figure 3.8 and Figure 3.7 shows the domains of the new implementations of $G()$.

In those algorithms `ROR(a, r)` is the bitwise right rotation of `a` by `r` bits; `ROR4({a, b, c, d}, r)` is each word in the tuple `{a,b,c,d}` rotated by `r` bits, and `r1`, `r2`, `r3`, `r4` are NORX rotation constants. Notice that, in order to execute the function $F()$, the function $G4()$ must be called twice, with a different argument order in the second call to execute the diagonal step. Similarly, $G2()$ must be called a total of four times in order to execute the column and diagonal steps.

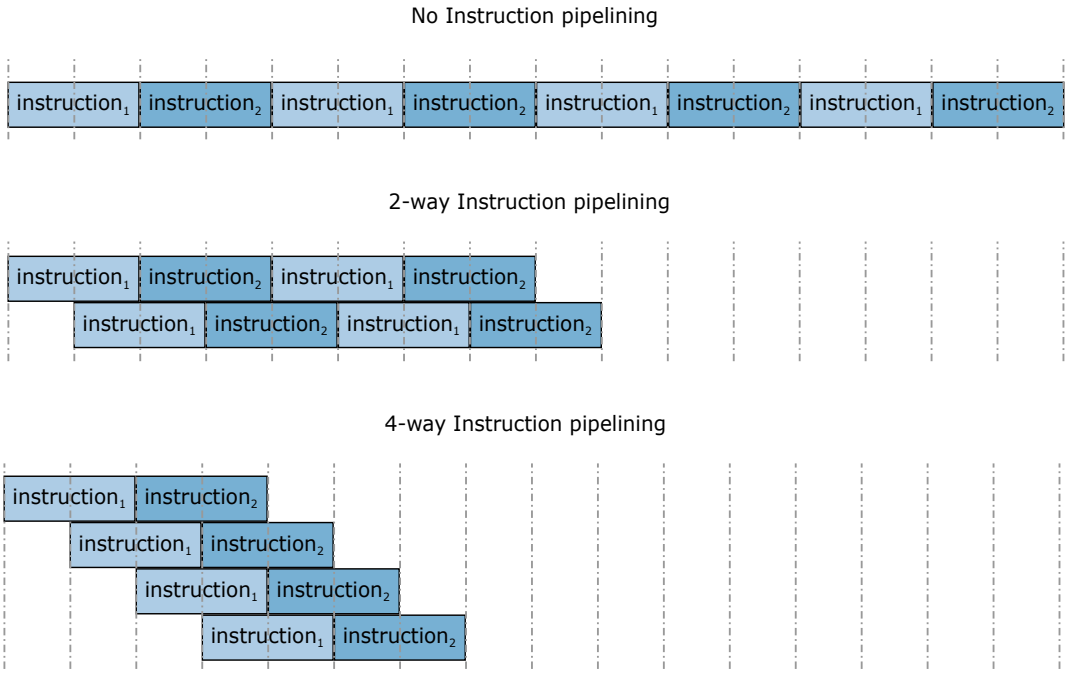

Figure 3.6: Illustration of the basic idea behind a pipeline/instruction parallelism approach of optimization.

Algorithm 13 The function G with 4-way pipeline optimization

Input: $S = (s_0, \cdots, s_{15})$ ▷ Whole Norx State
Output: $S = (s_0, \cdots, s_{15})$ ▷ Whole Norx State
Function G4(S)
  $s_0 = (s_0 \oplus s_4) \oplus ((s_0 \wedge s_4) \ll 1)$
  $s_1 = (s_1 \oplus s_5) \oplus ((s_1 \wedge s_5) \ll 1)$
  $s_2 = (s_2 \oplus s_6) \oplus ((s_2 \wedge s_6) \ll 1)$
  $s_3 = (s_3 \oplus s_7) \oplus ((s_3 \wedge s_7) \ll 1)$
  $s_{12} = s_{12} \oplus s_0$; $s_{13} = s_{13} \oplus s_1$
  $s_{14} = s_{14} \oplus s_2$; $s_{15} = s_{15} \oplus s_3$
  $\{s_{12}, s_{13}, s_{14}, s_{15}\} = \texttt{ROR4}(\{s_{12}, s_{13}, s_{14}, s_{15}\}, \texttt{r0})$
  $s_8 = (s_8 \oplus s_{12}) \oplus ((s_8 \wedge s_{12}) \ll 1)$
  $s_9 = (s_9 \oplus s_{13}) \oplus ((s_9 \wedge s_{13}) \ll 1)$
  $s_{10} = (s_{10} \oplus s_{14}) \oplus ((s_{10} \wedge s_{14}) \ll 1)$
  $s_{11} = (s_{11} \oplus s_{15}) \oplus ((s_{11} \wedge s_{15}) \ll 1)$
  $s_{12} = s_{12} \oplus s_0$; $s_{13} = s_{13} \oplus s_1$
  $s_{14} = s_{14} \oplus s_2$; $s_{15} = s_{15} \oplus s_3$
  $\{s_4, s_5, s_6, s_7\} = \texttt{ROR4}(\{s_4, s_5, s_6, s_7\}, \texttt{r1})$
  $s_0 = (s_0 \oplus s_4) \oplus ((s_0 \wedge s_4) \ll 1)$
  $s_1 = (s_1 \oplus s_5) \oplus ((s_1 \wedge s_5) \ll 1)$
  $s_2 = (s_2 \oplus s_6) \oplus ((s_2 \wedge s_6) \ll 1)$
  $s_3 = (s_3 \oplus s_7) \oplus ((s_3 \wedge s_7) \ll 1)$
  $s_{12} = s_{12} \oplus s_0$; $s_{13} = s_{13} \oplus s_1$

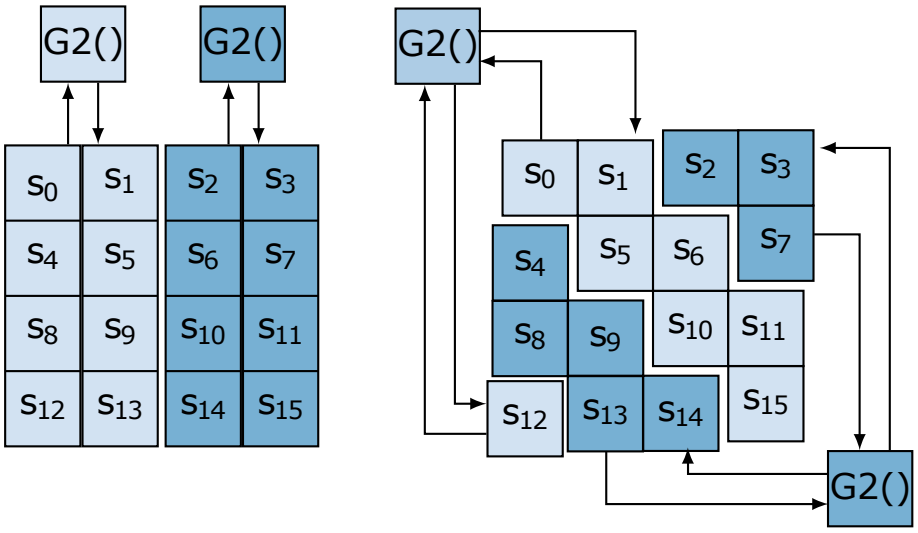

Figure 3.7: $2\times$ optimization of the NORX F function. The function $G2 : \{0,1\}^{8w} \rightarrow \{0,1\}^{8w}$ operates over 8 words of the internal state each time.

$s_{14} = s_{14} \oplus s_2$; $s_{15} = s_{15} \oplus s_3$
$\{s_{12}, s_{13}, s_{14}, s_{15}\} = \texttt{ROR4}(\{s_{12}, s_{13}, s_{14}, s_{15}\}, \texttt{r2})$
$s_8 = (s_8 \oplus s_{12}) \oplus ((s_8 \wedge s_{12}) \ll 1)$
$s_9 = (s_9 \oplus s_{13}) \oplus ((s_9 \wedge s_{13}) \ll 1)$
$s_{10} = (s_{10} \oplus s_{14}) \oplus ((s_{10} \wedge s_{14}) \ll 1)$
$s_{11} = (s_{11} \oplus s_{15}) \oplus ((s_{11} \wedge s_{15}) \ll 1)$
$s_{12} = s_{12} \oplus s_0$; $s_{13} = s_{13} \oplus s_1$
$s_{14} = s_{14} \oplus s_2$; $s_{15} = s_{15} \oplus s_3$
$\{s_4, s_5, s_6, s_7\} = \texttt{ROR4}(\{s_4, s_5, s_6, s_7\}, \texttt{r3})$
return S
end Function

Algorithm 14 The function G with 2-way pipeline optimization

Input: $s_0, s_1, s_4, s_5, s_8, s_9, s_{12}, s_{13}$
output: $s_0, s_1, s_4, s_5, s_8, s_9, s_{12}, s_{13}$
Function G2$(s_0, s_1, s_4, s_5, s_8, s_9, s_{12}, s_{13})$
▷ Either two columns or diagonals of the 16×16 State representation.
$s_0 = (s_0 \oplus s_4) \oplus ((s_0 \wedge s_4) \ll 1)$
$s_1 = (s_1 \oplus s_5) \oplus ((s_1 \wedge s_5) \ll 1)$
$s_{12} = s_{12} \oplus s_0$
$s_{13} = s_{13} \oplus s_1$
$s_{12} = \texttt{ROR}(s_{12}, \texttt{r0})$
$s_{13} = \texttt{ROR}(s_{13}, \texttt{r0})$
$s_8 = (s_8 \oplus s_{12}) \oplus ((s_8 \wedge s_{12}) \ll 1)$
$s_9 = (s_9 \oplus s_{13}) \oplus ((s_9 \wedge s_{13}) \ll 1)$
$s_4 = s_4 \oplus s_8$
$s_5 = s_5 \oplus s_9$

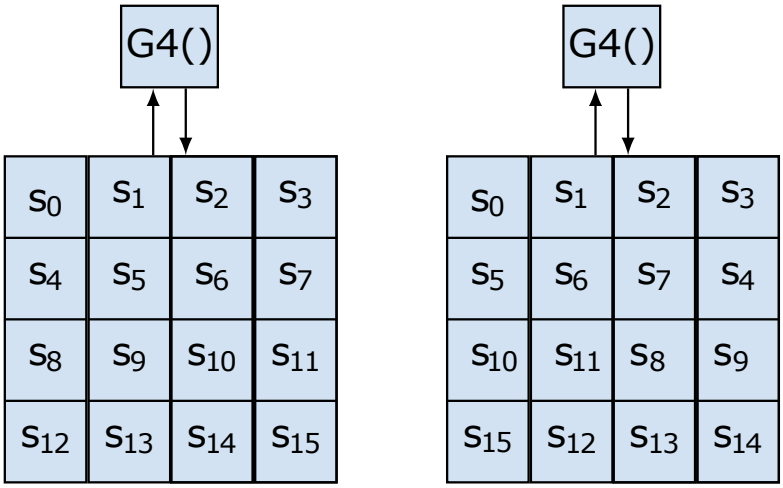

Figure 3.8: $4\times$ optimization of the NORX F function. The function $G4 : \{0,1\}^{16w} \rightarrow \{0,1\}^{16w}$ operates over 16 words of the internal state at once. Notice that the diagonal step is done by simply reordering the inputs of $G4$.

$s_4 = \texttt{ROR}(s_4, \texttt{r1})$
$s_5 = \texttt{ROR}(s_5, \texttt{r1})$
$s_0 = (s_0 \oplus s_4) \oplus ((s_0 \wedge s_4) \ll 1)$
$s_1 = (s_1 \oplus s_5) \oplus ((s_1 \wedge s_5) \ll 1)$
$s_{12} = s_{12} \oplus s_0$
$s_{13} = s_{13} \oplus s_1$
$s_{12} = \texttt{ROR}(s_{12}, \texttt{r2})$
$s_{13} = \texttt{ROR}(s_{13}, \texttt{r2})$
$s_8 = (s_8 \oplus s_{12}) \oplus ((s_8 \wedge s_{12}) \ll 1)$
$s_9 = (s_9 \oplus s_{13}) \oplus ((s_9 \wedge s_{13}) \ll 1)$
$s_4 = s_4 \oplus s_8$
$s_5 = s_5 \oplus s_9$
$s_4 = \texttt{ROR}(s_4, \texttt{r3})$
$s_5 = \texttt{ROR}(s_5, \texttt{r3})$
return $s_0, s_1, s_4, s_5, s_8, s_9, s_{12}, s_{13}$
end Function

Input: $S = (s_0, \cdots, s_{15})$, GH() ▷ Whole Norx State and the half permutation
Output: $S = (s_0, \cdots, s_{15})$ ▷ Norx State after permutation
Function $F_{G2}(S = (s_0, \cdots, s_{15}))$ ▷ Permutation using $G2$
▷ Column step
$(s_0, s_1, s_4, s_5, s_8, s_9, s_{12}, s_{13}) \leftarrow (\text{GH})(s_0, s_1, s_4, s_5, s_8, s_9, s_{12}, s_{13})$
$(s_2, s_3, s_6, s_7, s_{10}, s_{11}, s_{14}, s_{15}) \leftarrow (\text{GH})(s_2, s_3, s_6, s_7, s_{10}, s_{11}, s_{14}, s_{15})$
▷ Diagonal step
$(s_0, s_1, s_5, s_6, s_{10}, s_{11}, s_{15}, s_{12}) \leftarrow (\text{GH})(s_0, s_1, s_5, s_6, s_{10}, s_{11}, s_{15}, s_{12})$
$(s_2, s_3, s_7, s_4, s_8, s_9, s_{13}, s_{14}) \leftarrow (\text{GH})(s_2, s_3, s_7, s_4, s_8, s_9, s_{13}, s_{14})$
return S

42: **end Function**

In Algorithm 14, the even-numbered lines of $G2()$ execute $G() : s^4 \rightarrow s^4$ on a column or diagonal of the internal state, while the odd numbered lines execute the same instructions on a different and independent diagonal or column of the state. This then allows the issue of independent instructions and loads, for example, the `XOR`s in lines 7 and 8. This same idea is applied even further in Algorithm 13, where 4 independent instructions are issued one after another, what makes this implementation better suited for a processor with a deep pipeline.

3.5 NEON implementation

Cortex-A processors usually feature a SIMD engine called NEON. The structure of NORX, with processing being executed in a matrix's columns and diagonals, with no data dependency between these structure makes it a great candidate for a NEON implementation. Furthermore, a single 128-bit NEON register is able to hold a whole line of NORX32 state matrix, and while two of these registers are necessary for each line of NORX64 state, the native NEON instructions are capable of working directly on 64-bit words.

A straightforward way to implement NORX using NEON is defining the internal state as an array of type-Q registers: `uint32x4_t[4]` for NORX32 and `uint64x2_t[8]` for NORX64.

3.5.1 NEON word-wise rotations

NORX $F()$ needs four word-wise right-rotations with different rotation constants, executed inside the NEON register lanes. An implementation for NORX32 rotations is shown in Algorithm 15, and the 64-bit version is shown in Algorithm 16.

Regarding NORX32, the default rotation (`ROTRN`) uses the *vector shift right and insert* (`VSRIQ`) instruction, with a left shift to execute the rotation without need for an exclusive or to combine intermediate shift values. The 16-bit optimized rotation is executed by interpreting the NEON register as 8 16-bit words; using the 32-bit swap endianness intrinsic and then casting the variable back to `uint32x4_t`. While this may look counterproductive in a first analysis, this code is compiled into a single instruction `VREV32.16 q0,q0`. The instruction `VREVn.m` reverses the order of the m-bit lanes within a set that is n bits wide [72]. Regarding NORX64, the 63-bit rotation (`ROTRN`) follows a simple shift-and-XOR approach, while the default rotation is similar to the one used by NORX32.

Algorithm 15 Optimized rotations for NORX32

```
//ROTate Right Neon
#define ROTRN(X, C) vsriq_n_u32((X << (32-C)), (X), (C))

//optmized for(16)r2: reinterprets vector as 16bit and reverse
    every 32bits
#define ROTRNR2(X) ((uint32x4_t)(vrev32q_u16((uint16x8_t)(X))))

//optmized for (31)r3:
#define ROTRNR3(X) ((X>>31)^(X<<1))
```

Algorithm 16 Optimized rotations for NORX64

```
//ROTateRightNeon
#define ROTRN(X, C) vsriq_n_u64(X << (64-C), (X), (C))

//Optmized for (63)r3:
#define ROTRNR3(X) veorq_u64((X >> 63), (X << 1))
```

3.5.2 Register wide rotations

The application of the function $G()$ to the state columns is straightforward in NEON. On the other hand, the diagonal step results in misaligned data. One way to implement the diagonal-step is rotating the state's lines, applying $G()$ to columns –now made of the aligned diagonals, and them rotating the lines back into the initial state, in preparation for the next round of $F()$. These transformations, that will be called `DIAG` and `UNDIAG`, are illustrated in Figure 3.9.

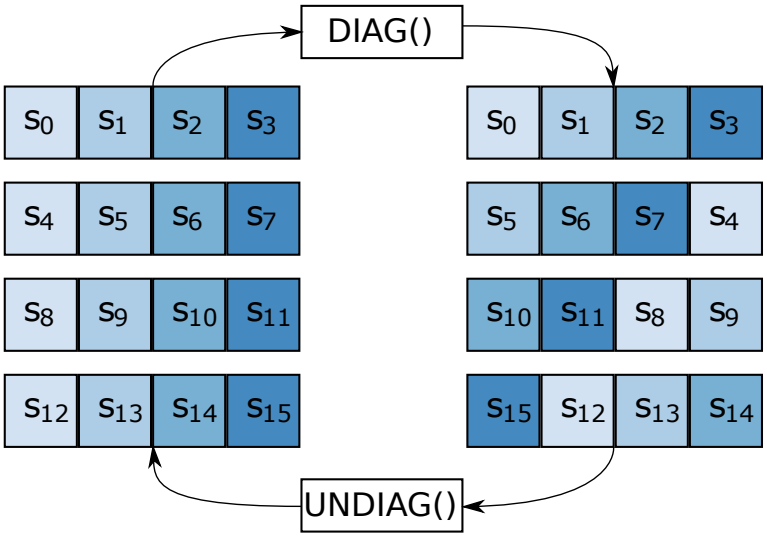

Figure 3.9: Transformations needed for the diagonal step.

For NORX32, the `DIAG` and `UNDIAG` functions are shown in Algorithm 17. The 3-word and single word rotations are executed using the vector extract (`VEXT{cond}.8 {Qd}, Qn, Qm, #imm`) instruction. This instruction extracts 8-bit elements from the bottom of the second operand and from the top of the first operand, concatenates and stores the result in the destination vector. By using the same register as inputs to the function, this function executes a register-wide rotation. While this approach could also be used to implement the 2-word rotation, one can use the fact that physically a type-Q register is a pair of type-D rotation to improve even further: by using the `vcombine`, `vget_high`, and `vget_low` intrinsics, the compilator generates a single `VSWP` instruction operating on the 64-bit type-D registers.

Algorithm 17 Optmized register-wide rotations for NORX32

```
//{A, B, C, D} are of type uint32x4_t
#define DIAG(A, B, C, D) do{                                  \
    D = vextq_u32(D, D, 3);                                   \
    /*C = vextq_u32(C, C, 2); */                              \
    C = vcombine_u32(vget_high_u32(C), vget_low_u32(C));      \
    B = vextq_u32(B, B, 1);                                   \
}while(0)

#define UNDIAG(A, B, C, D) do{                                \
    D = vextq_u32(D, D, 1);                                   \
    /* C = vextq_u32(C, C, 2); */                            \
    C = vcombine_u32(vget_high_u32(C), vget_low_u32(C));      \
    B = vextq_u32(B, B, 3);                                   \
}while(0)
```

For NORX64, the `DIAG` and `UNDIAG` functions are shown in Algorithm 18. Similar to NORX32, the rotations uses the combination of `vcombine`, `vget_high`, and `vget_low` intrinsics to generate the swaps.

Algorithm 18 Optmized register-wide rotations for NORX64

```
//{Al, Ah, Bl, Bh, Cl, Ch, Dl, Dh} are of type uint64x2_t
#define DIAG(Al, Ah, Bl, Bh, Cl, Ch, Dl, Dh) do{              \
    uint64x2_t x, y;                                          \
    x = vcombine_u64(vget_high_u64(Bl), vget_low_u64(Bh));\
    y = vcombine_u64(vget_high_u64(Bh), vget_low_u64(Bl));\
    Bl=x; Bh=y;                                               \
                                                              \
    x = Cl;                                                   \
    Cl = Ch;                                                  \
    Ch = x;                                                   \
                                                              \
    x = vcombine_u64(vget_high_u64(Dl), vget_low_u64(Dh));\
    y = vcombine_u64(vget_high_u64(Dh), vget_low_u64(Dl));\
```

```
    Dl = y; Dh = x;                                                 \
}while(0)

#define UNDIAG(Al, Ah, Bl, Bh, Cl, Ch, Dl, Dh) do{                  \
    uint64x2_t x, y;                                                \
    x = vcombine_u64(vget_high_u64(Bh), vget_low_u64(Bl));          \
    y = vcombine_u64(vget_high_u64(Bl), vget_low_u64(Bh));          \
    Bl=x; Bh=y;                                                     \
                                                                    \
    x = Cl;                                                         \
    Cl = Ch;                                                        \
    Ch = x;                                                         \
                                                                    \
    x = vcombine_u64(vget_high_u64(Dh), vget_low_u64(Dl));          \
    y = vcombine_u64(vget_high_u64(Dl), vget_low_u64(Dh));          \
    Dl = y; Dh = x;                                                 \
}while(0)
```

3.5.3 NEON Permutation

Given the word-wide and register-wide rotations the permutation can then be implemented using the NEON instructions. The function $G()$ is given in Algorithm 19 for NORX32 and Algorithm 20 for NORX64.

Algorithm 19 NEON implementation of the function $G()$ NORX32

```
#define G(A, B, C, D)                                               \
    do{                                                             \
        ALIGN uint32x4_t t, y;                                      \
                                                                    \
        t = A ^ B; y = A & B; y = SHL(y);                           \
        A = t ^ y;                                                  \
        D = D ^ A; D = ROTRN(D, r0);                                \
                                                                    \
        t = C ^ D; y = C & D; y = SHL(y);                           \
        C = t ^ y;                                                  \
        B = B ^ C; B = ROTRN(B, r1);                                \
                                                                    \
        t = A ^ B; y = A & B; y = SHL(y);                           \
        A = t ^ y;                                                  \
        D = D ^ A; D = ROTRNR2(D);                                  \
                                                                    \
        t = C ^ D; y = C & D; y = SHL(y);                           \
        C = t ^ y;                                                  \
        B = B ^ C; B = ROTRNR3(B);                                  \
    }while(0)
```

Algorithm 20 NEON implementation of the function $G()$ NORX64

```
#define G(Al, Ah, Bl, Bh, Cl, Ch, Dl, Dh)                    \
    do{                                                      \
        uint64x2_t tl, th, yl, yh;                           \
                                                             \
        tl = Al ^ Bl;                th = Ah ^ Bh;           \
        yl = Al & Bl;                yh = Ah & Bh;           \
        yl = SHL(yl);                yh = SHL(yh);           \
        Al = tl ^ yl;                Ah = th ^ yh;           \
        Dl = Dl ^ Al;                Dh = Dh ^ Ah;           \
        Dl = ROTRN(Dl, r0);          Dh = ROTRN(Dh, r0);     \
                                                             \
        tl = Cl ^ Dl;                th = Ch ^ Dh;           \
        yl = Cl & Dl;                yh = Ch & Dh;           \
        yl = SHL(yl);                yh = SHL(yh);           \
        Cl = tl ^ yl;                Ch = th ^ yh;           \
        Bl = Bl ^ Cl;                Bh = Bh ^ Ch;           \
        Bl = ROTRN(Bl, r1);          Bh = ROTRN(Bh, r1);     \
                                                             \
        tl = Al ^ Bl;                th = Ah ^ Bh;           \
        yl = Al & Bl;                yh = Ah & Bh;           \
        yl = SHL(yl);                yh = SHL(yh);           \
        Al = tl ^ yl;                Ah = th ^ yh;           \
        Dl = Dl ^ Al;                Dh = Dh ^ Ah;           \
        Dl = ROTRN(Dl, r2);          Dh = ROTRN(Dh, r2);     \
                                                             \
        tl = Cl ^ Dl;                th = Ch ^ Dh;           \
        yl = Cl & Dl;                yh = Ch & Dh;           \
        yl = SHL(yl);                yh = SHL(yh);           \
        Cl = tl ^ yl;                Ch = th ^ yh;           \
        Bl = Bl ^ Cl;                Bh = Bh ^ Ch;           \
        Bl = ROTRNR3(Bl);            Bh = ROTRNR3(Bh);       \
    }while(0)
```

Finally, given these functions, the permutation $F()$ is given in Algorithm 21 for NORX32 and Algorithm 22 for NORX64.

Algorithm 21 NEON implementation of the function $F()$ for NORX32

```
static inline void roundF(uint32x4_t S[4]){
        G(S[0], S[1], S[2], S[3]);
        //Diagonal transformation
        DIAG(S[0], S[1], S[2], S[3]);
        G(S[0], S[1], S[2], S[3]);
        //Back to original layout
        UNDIAG(S[0], S[1], S[2], S[3]);
}
```

Algorithm 22 NEON implementation of the function $F()$ for NORX64

```
static inline void roundF(uint64x2_t S[8]) {
        G(S[0], S[1], S[2], S[3], S[4], S[5], S[6], S[7]);
        //columns transformation
        DIAG(S[0], S[1], S[2], S[3], S[4], S[5], S[6], S[7]);
        G(S[0], S[1], S[2], S[3], S[4], S[5], S[6], S[7]);
        //Back to original layout
        UNDIAG(S[0], S[1], S[2], S[3], S[4], S[5], S[6], S[7]);
}
```

3.6 Other implementations

Some novel ways to implement NORX were thought of, but ultimately had no measurable improvement in comparison with the state-of-art implementations. For completeness, those negative results will be reported in this Section.

For NORX3261, a implementation using `uint64_t` to pack two words of the internal state was tried. Beyond the less legible code, this implementation had impact similar to the $2\times$ pipeline implementation, due to the fact that, internally, the 32-bit processors was transforming back the operations into 32-bit ones. On the 64-bit processor, this code showed, again, no tangible improvements due to the data manipulation overheads.

The original NORX paper [50] defines a permutation with temporary variables and a "low latency" construction. In our tests, this different permutation was either equal or worse than the default permutation. The same behavior was found when the pipeline optimization were applied to the low latency permutation.

The code from Algorithm 23 is a different way to implement the function $G : \{0,1\}^{4w} \rightarrow \{0,1\}^{4w}$. Overall, this method was 23% slower than the optimized code on the 32-bit platforms, and 11% on the 64-bit ones.

Algorithm 23 Different approach of implementing $G()$.

```
#define G(a, b, c, d) do{                                   \
    uint32_t t1, t2;                                        \
                                                            \
    d = d ^ a;                                              \
    t1 = a & b;                                             \
    t2 = b ^ (t1 << 1);                                     \
    b = b ^ c;                                              \
    a = a ^ t2;                                             \
    d = d ^ t2;                                             \
    d = ROR(d, r0);                                         \
                                                            \
    t1 = c & d;                                             \
    t2 = d ^ (t1 << 1);                                     \
    d = d ^ a;                                              \
    c = c ^ t2;                                             \
    b = b ^ t2;                                             \
    b = ROR(b, r1);                                         \
                                                            \
    t1 = a & b;                                             \
    t2 = b ^ (t1 << 1);                                     \
    b = b ^ c;                                              \
    a = a ^ t2;                                             \
    d = d ^ t2;                                             \
    d = ROR(d, r2);                                         \
                                                            \
    t1 = c & d;                                             \
    t2 = d ^ (t1 << 1);                                     \
    c = c ^ t2;                                             \
    b = b ^ t2;                                             \
    b = ROR(b, r3);                                         \
}while(0)
```

On the NEON implementation, a different approach to implement a SIMD function $G : (\{0,1\}^{4w})^4 \rightarrow (\{0,1\}^{4w})^4$ was tried, but it had slightly worse ($\leq 5\%$) performance than the reference NEON implementation. This code is shown in Algorithm 24.

Algorithm 24 Different approach of implementing $G()$ using NEON.

```
#define G(A, B, C, D)                                   \
    do{                                                 \
        ALIGN uint32x4_t t0,                            \
        t0 = A & B;                                     \
        t1 = SHL(t0);                                   \
        A = A ^ t1;                                     \
        A = A ^ B;                                      \
        D = D ^ A;                                      \
        D = ROTRN(D, r0);                               \
                                                        \
        t0 = C & D;                                     \
        t1 = SHL(t0);                                   \
        C = C ^ t1;                                     \
        C = C ^ D;                                      \
        B = B ^ C;                                      \
        B = ROTRN(B, r1);                               \
                                                        \
        t0 = A & B;                                     \
        t1 = SHL(t0);                                   \
        A = A ^ t1;                                     \
        A = A ^ B;                                      \
        D = D ^ A;                                      \
        D = ROTRNR2(D);                                 \
                                                        \
        t0 = C & D;                                     \
        t1 = SHL(t0);                                   \
        C = C ^ t1;                                     \
        C = C ^ D;                                      \
        B = B ^ C;                                      \
        B = ROTRNR3(B);                                 \
}while(0)
```

3.7 Results and considerations

Regarding the security of those implementations, the tool *FlowTracker* [73] was used to analyze the behavior of the algorithm and check if it runs in constant-time. *FlowTracker* is a tool used to detect timing attack vulnerabilities in cryptographic codes written in C/C++, capable of finding data traces or control sequences with dependencies in secret information. Beyond that, NORX is designed to be resistant against side-channel attacks, with no table-lookups, branching or looping dependent on secret data. Furthermore, there are cryptanalysis works regarding differential, algebraic, fixed-point, slide, and rotational attacks [2]. For implementation correctness, test vectors were used to compare outputs with the reference implementation, and internal consistency was ver-

ified using encryption-decryption of random sets of plaintext, nonce, additional data and keys.

Table 3.6 shows the results for the 32-bit processors, namely Cortex A7 and Cortex A15; Table 3.7 shows the results for the 64-bit Cortex A53 processor. Lastly, for completeness, Table 3.8 on the embedded Cortex-M micro-controllers An input length of 256KB was chosen to report the average CPB –cycle per byte– since it better dilutes the initialization overheads, without risking overflow of the performance counter registers, except in the Cortex-M micro-controllers, due to memory constraints. The initialization overhead can be seen in Figures 3.10 and 3.11. In these tables, “reference code" refers to the C code submitted by the NORX authors to CAESAR.

Table 3.6: Cycles per byte for NORX encryption. Plaintext length of 256KiB on the 32-bit processors. Reference code from CAESAR [74]. NORX3261 and NORX3264 use a 128-bit key, and NORX6461 uses a 256-bit key.

		Ref. code	4x pipe	2x pipe	Ref. NEON	Speedup
NORX 3261	Cortex A7	29.45	29.70	**24.72**	26.99	16%
	Cortex A15	17.77	**14.23**	15.16	18.25	20%
NORX 3264	Cortex A7	28.46	33.74	**26.50**	33.27	7%
	Cortex A15	16.88	**15.26**	15.37	18.21	10%
NORX 6461	Cortex A7	48.52	50.09	**46.65**	17.81	4%
	Cortex A15	33.83	**26.76**	28.33	10.90	21%
NORX 6464	Cortex A7	47,86	48,85	**44,98**	38,75	6%
	Cortex A15	35.54	**25.00**	26.73	22,94	25%

Table 3.7: Cycles per byte for NORX encryption on the 64-bit platform Cortex-A53. Plaint text length of 256KiB. Reference code from CAESAR [74]. NORX3261 and NORX3264 uses a 128-bit key, and NORX6461 uses a 256-bit key.

	Ref.	4x pipe	2x pipe	Ref. NEON	Speedup
NORX 3261	19.55	**10.94**	12.27	10.81	44%
NORX 3264	19.42	**12.08**	13.06	9.56	38%
NORX 6461	10.29	**5.84**	6.58	9,54	43%
NORX 6464	9.43	**6.09**	9.62	8,92	35%

For the 32-bit variant of NORX, a 4× pipeline implementation is faster than the reference code in up to 20% on a 32-bit ARM and 44% on the 64-bit Cortex-A53. Our optimized implementation is faster than the reference NEON implementation: the 2× pipeline implementation is 12% faster than the reference code on the Cortex A7 core; the 4× pipeline implementation is 22% faster on the Cortex A15. While NORX has a SIMD friendly construction, with the internal state fitting in four 128-bit NEON registers, there are two extra transformations needed in each application of the function G in order to align the words between the column and diagonal steps. This transformation requires three extra pairs of SIMD load and store instructions, two `vext.8` instructions, and a `vwsp` instruction. We believe that this, together with the extra cost needed to transfer data from the NEON registers back to the ARM registers every round, coupled with

Table 3.8: Perfomance of NORX3261 (cycles per byte) on 32-bit Cortex-M architecture.

Cortex model	Size	No pipeline optimizations	Ref. code	4x pipe	2x pipe
M0	8KiB	99.52	100.12	111.84	99.96
M3	32KiB	49.96	50.49	67.21	66.26
M4	16KiB	49.96	50.49	47.28	66.26

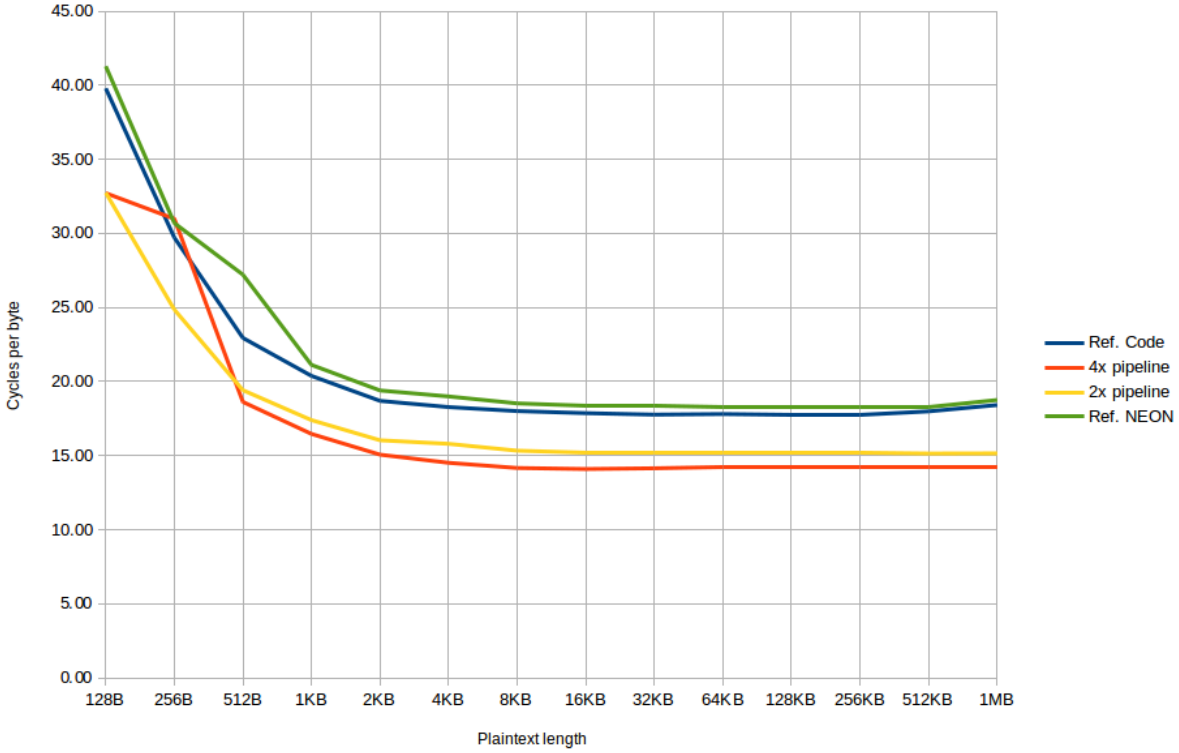

Figure 3.10: Cycles per byte results for NORX3261 on Cortex A15.

the optimal usage of the pipeline makes our solution better than using SIMD instructions for these cores. Notice that the pipeline implementations cannot be implemented in the NEON implementation of the 32-bit variants of NORX: The idea behind these optimizations is to explore the lack of data dependencies between different columns and diagonals of the internal state, and the NEON implementation keeps all these elements inside the register lanes.

For the 64-bit variant of NORX, a 2× pipeline is better suited for the Cortex-A7 processor, and a 4× pipeline for the Cortex-A15 processor, due to the differences in pipeline length. With SIMD instructions being adequate for larger volume of data, NORX6461 on the 32-bit platform shows better performance using SIMD instructions, mainly due to the 64-bit word rotations being expensive using the 32-bit ARM registers, in comparison to the NEON approach, where the rotations of two words can be done at the same time in the 128-bit register. For Cortex A53, both pipeline implementations show satisfactory results, being 43% faster than the reference code. In relation to a NEON

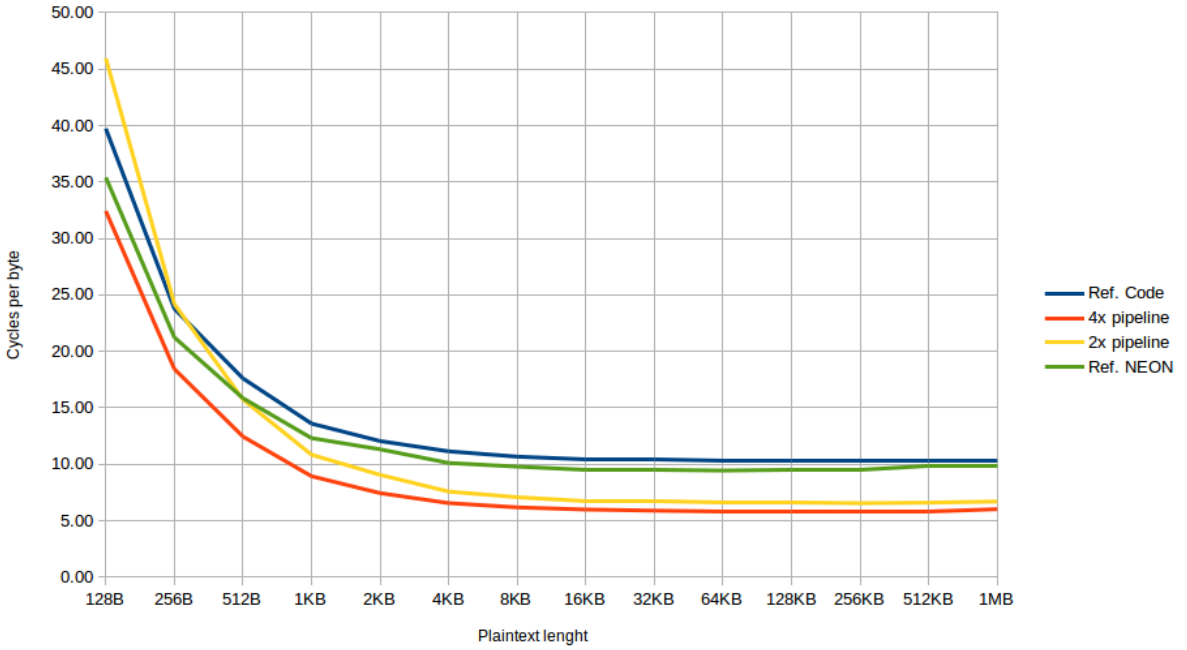

Figure 3.11: Cycles per byte results for NORX6461 on Cortex A53.

implementation, the $4\times$ pipeline implementation is 39% faster and the $2\times$ pipeline implementation is 31% faster. Similarly to NORX3261, the presence of a native 64-bit register and a deep pipeline with 8 stages makes a pipeline oriented approach superior to the SIMD alternative. Notice that an "optimization" similar to the $2\times$ pipeline code is already implemented in the NEON code of NORX6461 and NORX6464, where the NEON instructions operate in a round-robin fashion on the pairs of columns and diagonals of the state, but in our tests, there is no performance difference between this and a serial approach on the operation order on the sponge registers.

The multi-sponge algorithms –NORX3264 and NORX6464– show a similar behavior to their single-sponge counterparts: NORX3264 gets a small speed improvement using the $2\times$ pipeline implementation on Cortex-A7, and a more pronounced speedup using the $4\times$ implementation in both Cortex-A15 and Cortex-A53. The 64-bit multi-sponge algorithm –NORX6464– also shows a small speedup using the $2\times$ implementation on Cortex-A7, and a better speedup using the deeper pipeline implementation on the Cortes-A15 and Cortex-A53 processors. Those tests are referent to a single-thread implementation. The multi-sponge algorithms can be further improved by using multiple threads to execute the parallel sponges.

3.8 Applying the ideas to the BLAKE2 hash algorithm

BLAKE2 [75] is a family of cryptographic hash functions due to the same creators of NORX, specified in RFC 7693, and finalist of the SHA-3 competition. This family of hash functions are, according to the authors, faster than MD5, SHA-1, SHA-2, and SHA-3,

while still being at least as secure as the standard SHA-3. Due to its performance, simplicity, and security, BLAKE2 is used in many projects: WolfSSL [76], OpenSSL [77], Bouncy Castle [78], Noise protocol [79], Sodium, WinRAR, etc. It's also part of password hashing schemes such as Catena [80] and others.

BLAKE2 shares many common traits with NORX:

- BLAKE2 features a 16-word state, organized as a 4×4 matrix, with a permutation function being applied to it's columns and diagonals.
- BLAKE2 has two variations that resemble NORX32 and NORX64, with a similar $G : s^4 \rightarrow s^4$ function, made from XOR, rotations and additions
- A combination of $G()$ functions create a round permutation $F()$, that is then iterated for a given number of rounds on the internal state.

For this work, we modified BLAKE2s, the 32-bit optimized version of BLAKE, with the $2\times$ and $4\times$ optimizations, and executed tests in the same environment as the tests of NORX. Snippets for these optimizations are given in Algorithm 25 and Algorithm 26.

Algorithm 25 $G()$ snippet for the $2\times$ optimization

```
#define G(m,r,i,a,b,c,d,I,A,B,C,D) {           \
    a = a + b + m[blake2s_sigma[r][2*i+0]];    \
    A = A + B + m[blake2s_sigma[r][2*I+0]];    \
    d = ROR(d ^ a, 16);                        \
    D = ROR(D ^ A, 16);                        \
    c = c + d;                                 \
    C = C + D;                                 \
    b = ROR(b ^ c, 12);                        \
    B = ROR(B ^ C, 12);                        \
    a = a + b + m[blake2s_sigma[r][2*i+1]];    \
    A = A + B + m[blake2s_sigma[r][2*I+1]];    \
    d = ROR(d ^ a, 8);                         \
    D = ROR(D ^ A, 8);                         \
    c = c + d;                                 \
    C = C + D;                                 \
    b = ROR(b ^ c, 7);                         \
    B = ROR(B ^ C, 7);                         \
}
```

Algorithm 26 $G()$ snippet for the $4\times$ optimization

```
#define G(m,r,i,a,b,c,d,I,A,B,C,D,j,e,f,g,h,J,E,F,G,H) {\
    a = a + b + m[blake2s_sigma[r][2*i+0]];  \
    e = e + f + m[blake2s_sigma[r][2*j+0]];  \
    A = A + B + m[blake2s_sigma[r][2*I+0]];  \
    E = E + F + m[blake2s_sigma[r][2*J+0]];  \
    d = ROR(d ^ a, 16);                      \
    h = ROR(h ^ e, 16);                      \
    D = ROR(D ^ A, 16);                      \
    H = ROR(H ^ E, 16);                      \
    c = c + d;                               \
    g = g + h;                               \
    C = C + D;                               \
    G = G + H;                               \
    b = ROR(b ^ c, 12);                      \
    f = ROR(f ^ g, 12);                      \
    B = ROR(B ^ C, 12);                      \
    F = ROR(F ^ G, 12);                      \
    a = a + b + m[blake2s_sigma[r][2*i+1]];  \
    e = e + f + m[blake2s_sigma[r][2*j+1]];  \
    A = A + B + m[blake2s_sigma[r][2*I+1]];  \
    E = E + F + m[blake2s_sigma[r][2*J+1]];  \
    d = ROR(d ^ a, 8);                       \
    h = ROR(h ^ e, 8);                       \
    D = ROR(D ^ A, 8);                       \
    H = ROR(H ^ E, 8);                       \
    c = c + d;                               \
    g = g + h;                               \
    C = C + D;                               \
    G = G + H;                               \
    b = ROR(b ^ c, 7);                       \
    f = ROR(f ^ g, 7);                       \
    B = ROR(B ^ C, 7);                       \
    F = ROR(F ^ G, 7);                       \
}
```

The results of our tests are shown in Table 3.9. The overhead dilution behavior can be seen in the chart shown in Figure 3.12.

Table 3.9: Cycles per byte for BLAKE2s digest. Plaintext length of 64KB.

	Ref. Code	4x pipe	2x pipe	Speedup
Cortex A7	123,09	123,07	123,08	0%
Cortex A15	79,91	**45,14**	59,63	44%
Cortex A53	110,81	**97,73**	110,82	12%

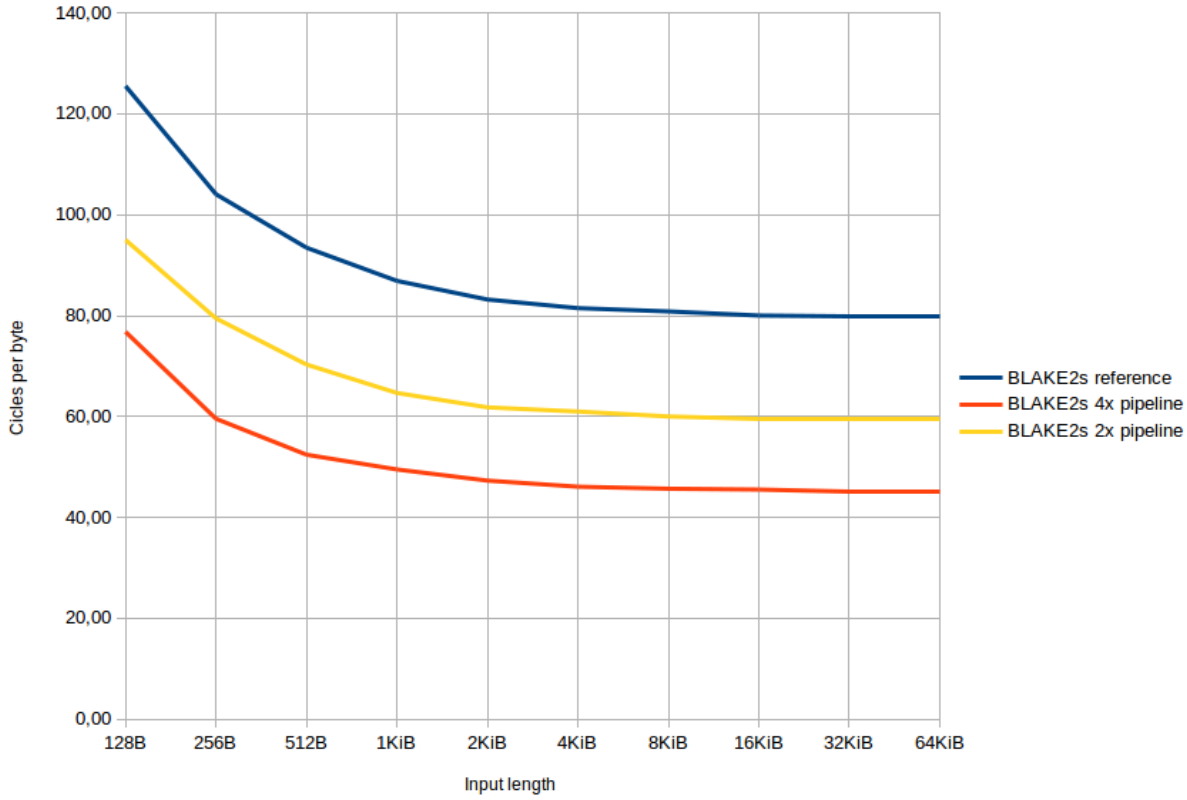

Figure 3.12: Cycles per byte results for BLAKE2s on Cortex A15.

While the optimization had neither a positive or negative effect on the shallow pipeline of Cortex A7, there were positive results on other processors: On the 32-bit Cortex A15, a $4\times$ pipeline resulted in 44% of speedup. On a Cortex A53, 12% of speedup was found.

Chapter 4

Software implementation: Ascon AEAD

Ascon is a family of AEAD algorithms currently in the fourth round of the CAESAR competition. It was designed by Christoph Dobraunig, Maria Eichlseder, Florian Mendel and Martin Schläffer, from Graz University of Technology and Infineon Technologies. The main goal of Ascon is to have a very low memory footprint in both hardware and software implementations, while still providing adequate speed, simplicity of analysis, and good bounds for security. The rationale behind the design of this AEAD scheme is to provide the best combination of security, speed and size on hardware and software, with a focus on the size. Ascon is based on the sponge design, being very similar to the SpongeWrap [15] and MonkeyDuplex [81] designs, but with a stronger keyed initialization and finalization steps. The permutation of Ascon is an iterated SPN, designed to provide fast diffusion at low cost. The Sbox used in the permutation function is an improved version of the χ mapping of Keccak [82], and the linear layer uses a function similar to the Σ functions of SHA-2.

The design of Ascon allows it to have lightweight implementations on both software and hardware, with a good performance, although it was not designed with the intention to compete with very fast parallel AEAD schemes on unconstrained devices. On the other hand, the scheme was designed with good instruction parallelism. Amongst the characteristics of Ascon, we can cite:

- **Light and flexible in Hardware.** Current implementations of Ascon can achieve a throughput of 4.9-7.3 Gbps using less than 10kGE. It is also possible, with trade-offs, to implement Ascon in as low as 2.5 kGE or with throughput as high as 13.2 Gbps [3].

- **Bitsliced software implementation.** The design of Ascon allows for an easy bitsliced implementation in software, thanks to the internal state and permutation being defined in standard operations over 64-bit words. Beyond that, the small size of the internal state(320 bits) allows the whole sponge to be kept in registers, even on constrained architectures.

- **Easy side-channel countermeasures.** The bitsliced Sbox implementation pre-

vents cache-timing attacks, since there are no secret-data dependencies on lookup tables. Furthermore, the Sbox design facilitates side-channel leakage countermeasures such as Masking.

- **Online, single-pass, and inverse-free.** Ascon can encrypt plaintext blocks before subsequent blocks or plaintext length is known. This property is also true for decryption. Both operations can be executed in a single pass, what also allows in-place encryption and decryption. Beyond that, the permutations are only evaluated in a single direction, without the necessity of inverse operations, which in turn significantly reduces area overhead in hardware implementations.
- **Key agility and simplicity**. The scheme does not need a key schedule or expansion, therefore there are no hidden setup costs. The implementation of the scheme is defined over 64-bit words and uses only common bitwise boolean functions, namelly `AND`, `OR`, `NOT`, and bitwise rotation.

The Ascon family of AEAD algorithms will be described in the next section.

4.1 Description of Ascon algorithms

The algorithms in the Ascon family have names in the form ASCON$_{a,b}-k-r$. The algorithms are parametrized by the key length $k \leq 128$ bits, the rate r and the internal round numbers a and b. The Ascon authors define two algorithms named Ascon-128 and Ascon-128a, with their parameters shown in Table 4.1 [3]. There are two main differences between Ascon128 and Ascon-128a: The permutation p^b –executed in all steps except sponge initialization and finalization– of Ascon-128a has two extra rounds in comparison to Ascon-128; and Ascon-128a processes 128 bits of data at a time, versus 64 bits of Ascon-128. The authors of the AEAD scheme define Ascon-128 as their primary recommendation for Ascon parameters and Ascon128a as a secondary recommendation, and is a modification of a version of Ascon with lower security bounds, specified on the first version of their CAESAR submission.

Table 4.1: Recommended instances of Ascon

Name	Algorithm	Bit size of				Rounds	
		Key	Nonce	Tag	rate	p^a	p^b
Ascon-128	ASCON$_{12,6}-128-64$	128	128	128	64	12	6
Ascon-128a	ASCON$_{12,8}-128-128$	128	128	128	128	12	8

Throughout this work, when referring to the Ascon algorithms, we will use the same variable names as in the original work[3]. These variables and a short description of each one is given in Table 4.2.

4.1.1 Ascon Mode of Operation

The Ascon mode of operation is based on duplexed sponges, such as the MonkeyDuplex [81], but with stronger keyed initialization and finalization functions. The core

Table 4.2: Common Ascon variables

Variable	Description
S	The 320-bit internal state of the sponge construction
S_r, $S-c$	The r-bit rate and the c-bit capacity parts of S
$x_0, \cdots, x_4$	The five 64-bit words of the state
K, N, T	Respectively key, nonce and tag (128-bits)
P, C, A	Respectively plaintext, ciphertext and aditional data
$\perp$	Symbol of failed authentication
p, p^n	Single permutation, and n update rounds of p

permutations p^a and p^b operate on a 320-bit Sponge S, with a rate r and capacity c. The rate and capacity of the state S are denoted by, respectively, S_r and S_c. Figure 4.1 illustrates the mode of operation of Ascon.

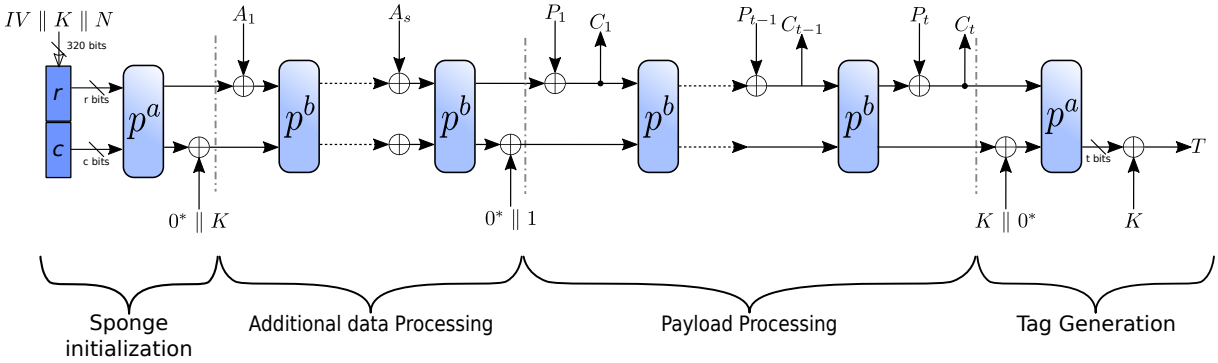

Figure 4.1: Ascon mode of operation. Adapted from [3]

A high-level description of the encryption and decryption algorithms of Ascon is given in Algorithm 27 and Algorithm 28.

Algorithm 27 Ascon AEAD Encryption

Input: Key, nonce, additional data and plaintext
Output: Ciphertext and authentication tag
Function Encryption $\mathcal{E}_{a,b,k,r}\ (K, N, A, P)$
  $c \leftarrow 320 - r$ ▷ Sponge initialization
  $P_1, \cdots, P_t \leftarrow \text{pad}_r(P)$
  $\ell = |P| \mod r$
  $S \leftarrow IV \| K \| N$
  $S \leftarrow p^a(S) \oplus (0^{320-k} \| K)$
  for $i == 1, \cdots, s$ **do** ▷ Additional data processing
    $S \leftarrow p^b((S_r \oplus A_i) \| S_c)$
  end for
  $S \leftarrow S \oplus (0^{319} \| 1)$
  for $i = 1, \cdots, t-1$ **do** ▷ Plaintext processing
    $S_r \leftarrow S_r \oplus P_i$

$C_i \leftarrow S_r$
$S \leftarrow p^b(S)$
end for
$S_r \leftarrow S_r \oplus P_t$
$C_t \leftarrow \lfloor S_r \rfloor$
$S \leftarrow p^a(S \oplus (0^r \parallel K \parallel 0^{c-k}))$ ▷ Finalization
$T \leftarrow \lceil S \rceil^r \oplus K$
return $C_1 \parallel \cdots \parallel C_t, T$
end Function

Algorithm 28 Ascon AEAD Decryption

Input: Key, nonce, additional data, ciphertext, and tag
Output: Plaintext or $\perp$
Function Decryption $\mathcal{D}_{a,b,k,r}$ (K, N, A, C, T)
$c \leftarrow 320 - r$ ▷ Sponge initialization
$\ell = |P| \mod r$
$S \leftarrow IV \parallel K \parallel N$
$S \leftarrow p^a(S) \oplus (0^{320-k} \parallel K)$
for $i == 1, \cdots, s$ **do** ▷ Additional data processing
$S \leftarrow p^b((S_r \oplus A_i) \parallel S_c)$
end for
$S \leftarrow S \oplus (0^{319} \parallel 1)$
for $i = 1, \cdots, t-1$ **do** ▷ Ciphertext processing
$P_i \leftarrow S_r \oplus C_i$
$S \leftarrow C_i \parallel S_c$
$S \leftarrow p^b(S)$
end for
$P_t \leftarrow \lfloor S_r \rfloor_l \oplus C_t$
$S_r \leftarrow C_t \parallel (\lfloor S_r \rfloor^{r-\ell} \oplus (1 \parallel 0^{r-1-\ell}))$
$S \leftarrow p^a(S \oplus (0^r \parallel K \parallel 0^{c-k}))$ ▷ Finalization
$T^* \leftarrow \lceil S \rceil^r \oplus K$
if $T = T^*$ **then**
Return $P_1 \parallel P_t$
else
Return $\perp$
end if
end Function

4.1.2 Padding rule

The padding rule of Ascon appends a singe bit `1` and the smallest number of `0`s to the plaintext P so that the length of the padded plaintext is a multiple of r bits. The same is done for the additional data A, except when $|A| = 0$, where no padding is added. Formally, the padding rule can be described as:

$$P_1, \cdots, P_t \leftarrow \text{pad}_r(P) = r\text{-bit blocks of} P \parallel 1 \parallel 0^{r-1-(|P| \mod r)}$$

$$A_1, \cdots, A_s \leftarrow \text{pad}_r^*(A) = \begin{cases} r\text{-bit blocks of} A \parallel 1 \parallel 0^{r-1-(|A| \mod r)} & \text{if} |A| > 0 \\ \varnothing & \text{if} |A| = 0 \end{cases}$$

4.1.3 Initialization

The state of Ascon is formed by the concatenation of an initialization vector IV, the secret key K and the nonce N, in a total of 320 bits. The IV specifies the parameters of the instance of Ascon and is defined as:

$$IV = k \parallel r \parallel a \parallel b \parallel 0^{288-2k} = \begin{cases} \texttt{80400c0600000000} & \text{for Ascon-128} \\ \texttt{80800c0800000000} & \text{for Ascon-128a} \end{cases}$$

With $S = IV \parallel K \parallel N$, the initialization finishes by calculating a rounds of the permutation p on S, followed by XORing the secret key K:

$$S \leftarrow p^a(S) \oplus (0^{320-k} \parallel K)$$

The initialization function is also described in Algorithm 29.

Algorithm 29 Ascon AEAD sponge initialization

Input: k, n ▷ Key and nonce
Output: S ▷ Initialized sponge
Function initialize(k, n)
  let $S = \{s_0 \parallel \cdots \parallel s_4\}$
  $s_0 \leftarrow \texttt{IV}$
  $s_1, s_2 \leftarrow k_0, k_1$
  $s_3, s_4 \leftarrow n_0, n_1$
  $S \leftarrow p^a(S)$
  $s_3 \leftarrow s_3 \oplus k_0$
  $s_4 \leftarrow s_4 \oplus k_1$
  return S
end Function

4.1.4 Additional data processing

For each padded block of A_i, with $i = 1, \cdots, s$, the following transformation is applied to the state:

$$S \leftarrow p^b((S_r \oplus A_i) \parallel S_c), \qquad \text{for } 1 \leq i \leq s$$

After the last block of additional data is processed, or if the $A = \varnothing$, a single bit domain separation constant is XORed to the state $S \leftarrow S \oplus \texttt{01}$. The Additional data processing function is also described in Algorithm 30

Algorithm 30 Ascon additional data processing

Input: $S = S_r \parallel S_c, A$ ▷ State and additional data
Output: S ▷ Initialized sponge
Function absorb(S, A)
  if $|A| > 0$ **then**
    let $A \leftarrow \text{pad}_r^*(A)$
    let $A = A_0 \parallel \cdots \parallel A_n$ with $|A_i| = |S_r|$
    for all A_i in A **do**
      $S \leftarrow S_r \oplus A_i \parallel S_c$
      $S \leftarrow p^b(S)$
    end for
  end if
  $S \leftarrow S \oplus 01$
return S
end Function

4.1.5 Plaintext processing

For encryption, each iteration XORs a plaintext block P_i with S_r, followed by the extraction of a ciphertext block C_i. For each block, except the last one, the internal state S is transformed by the permutation p^b:

$$C_i \leftarrow S_r \oplus P_i$$
$$S \leftarrow \begin{cases} p^b(C_i \parallel S_c) & \text{if } 1 \leq i < t \\ (C_i \parallel S_c) & \text{if } 1 \leq i = t \end{cases}$$

The last ciphertext block is truncated to the unpadded length of the last plaintext fragment, therefore $C_t \leftarrow \lfloor C_t \rfloor_{|P| \mod r}$. The payload encryption algorithm is described in Algorithm 31.

Algorithm 31 Ascon payload encryption

Input: $S = S_r \parallel S_c, P$ ▷ State and key
Output: S, C ▷ State and Ciphertext
Function encPayload(S, P)
  let $C = C_0 \parallel \cdots \parallel C_n$ with $|C_i| = |S_r|$ and $|C| = |P|$
  let $P \leftarrow \text{pad}_r(P)$
  let $P = P_0 \parallel \cdots \parallel P_n$ and $|P_i| = |S_r|$
  for all P_i from $i = 0$ to $i = n - 1$ **do**
    $S_r \leftarrow S_r \oplus P_i$
    $C_i \leftarrow S_r$
    $S \leftarrow p^b(S)$
  end for ▷ Last block
  $S_r \leftarrow S_r \oplus P_n$
  $C_i \leftarrow \lfloor S_r \rfloor_{|P| \mod r}$

return S, C
end Function

For decryption, in each iteration except the last one, the plaintext block P_i is computed by XORing the ciphertext C_i with S_r. S_r is then replaced with C_i, and the internal state is transformed with the permutation p^b:

$$P_i \leftarrow S_r \oplus C_i$$
$$S \leftarrow p^b(C_i \parallel S_c), \qquad 1 \leq i < t$$

The last truncated ciphertext block with length $0 \leq \ell < r$, is calculated as following:

$$P_t \leftarrow \lfloor S_r \rfloor_\ell \oplus C_t$$
$$S \leftarrow C_t \parallel (\lceil S_r \rceil^{r-\ell} \oplus (1 \parallel 0^{r-1-\ell})) \parallel S_c$$

The payload decryption algorithm is described in Algorithm 32

Algorithm 32 Ascon payload decryption

Input: $S = S_r \parallel S_c, C$ ▷ State and key
Output: S, P ▷ State and Plaintext
Function decPayload(S, C)
  let $P = P_0 \parallel \cdots \parallel P_n$ and $|P_i| = |S_r|$ and $|P| = |C|$
  let $C \leftarrow \text{pad}_r(C)$
  let $C = C_0 \parallel \cdots \parallel C_n$ and $|C_i| = |S_r|$
  for all C_i from $i = 0$ to $i = n - 1$ **do**
    $P_i \leftarrow S_r \oplus C_i$
    $S \leftarrow C_i \parallel S_c$
    $S \leftarrow p^b(S)$
  end for ▷ Last block
  $P_n \leftarrow \lfloor C_n \rfloor_{|P| \mod r} \oplus \lfloor S_r \rfloor_{|P| \mod r}$
  $S \leftarrow \lfloor C_n \rfloor_{|P| \mod r} \parallel (\lceil S_r \rceil^{r-(|P| \mod r)} \oplus (1 \parallel 0^{r-1-(|P| \mod r)})) \parallel S_c$
  return S, P
end Function

4.1.6 Finalization

The finalization step XORs the key K to the internal state, and then applies the permutation p^a to it. The authentication tag is defined as the last k bits of the sponge XORed with K:

$$S \leftarrow p^a(S \oplus (0^r \parallel K \parallel 0^{c-k}))$$
$$T \leftarrow \lceil S \rceil^k \oplus K$$

The finalization algorithm is described in Algorithm 33. The encryption algorithm returns the ciphertext C and the tag T, while the decryption algorithm return the plaintext P only if the calculated tag matches the one received by the function as input.

Algorithm 33 Ascon finalization and tag generation

Input: $S = S_r \parallel S_c, K$ ▷ State and key
Output: S, T ▷ State and authentication tag
Function finalize(S, K)
  let $k = |K|$
  $S \rightarrow S \oplus (0^r \parallel K \parallel 0^{c-k})$ ▷ Key is aligned to the first bits of S_c
  $S \rightarrow p^a(S)$
  $T \rightarrow \lceil S \rceil^k \oplus K$ ▷ Last k bits of S
return S, T
end Function

4.2 Ascon permutation

The main component of Ascon is the 320-bit permutation p, parametrized by the round numbers a and b, that iteratively apply an SPN-like round transformation to the internal state. The permutation p can be subdivided in three steps: p_C, p_S, and p_L. Those steps are, respectively, constant addition, substitution layer, and permutation layer.

For the application of the permutation p, the 320-bit state is split into five 64-bit words x_i such that $S = S_r \parallel S_c = x_0 \parallel x_1 \parallel x_2 \parallel x_3 \parallel x_4$.

Each round of p^a starts with the constant addition operation p_C, which adds the constant c_r to the word x_2. For p^b, the constant added to x_2 is $c - a - b + r$. The constants are defined as $c_i = (\texttt{0x0f} - i) \ll 4 \oplus i$ with $0 \leq i < 16$. This round constant was chosen to avoid slide, rotational, self-similarity and other attacks, and was designed in a simple obvious way, e.g. increasing and decreasing the halves of a byte. More than 16 rounds for the permutation p^a is not expected by the cipher designers [3].

The substitution layer p_S consists in 64 parallel applications of the 5-bit Sbox $\mathcal{S}(x)$, in a bitsliced manner on the five words of the state. For that matter, the bits of x_0 are the MSB and the bits of x_4 are the LSB. Table 4.3 defines the Sbox and Figure 4.2 shows the diagram for the bitsliced implementation of $\mathcal{S}(x)$. The Sbox design is based on the χ mapping of Keccak.

Table 4.3: Ascon Sbox

x	00	01	02	03	04	5	06	07	08	09	0a	0b	0c	0d	0e	0f
$\mathcal{S}(x)$	04	0b	1f	14	1a	15	09	02	1b	05	08	12	1d	03	06	1c
x	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
$\mathcal{S}(x)$	1e	13	07	0e	00	0d	11	18	10	0c	01	19	16	0a	0f	17

The bitsliced Sbox can be implemented with the pipelinable instructions shown in Algorithm 34.

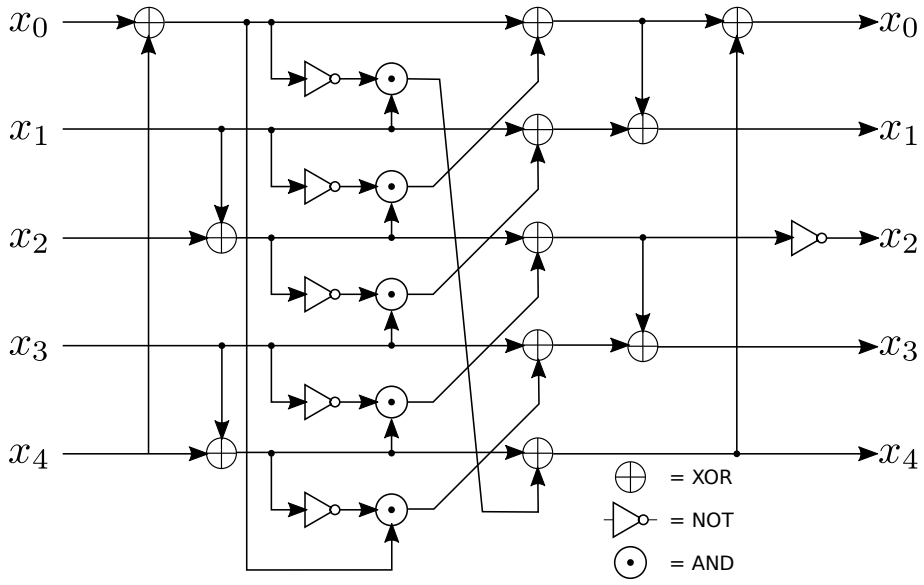

Figure 4.2: Ascon Sbox $\mathcal{S}(x)$. Adapted from [3]

Algorithm 34 Pipelinable C implementation of Ascon Sbox

Input: x_0, x_1, x_2, x_3, x_4 ▷ State words
Output: x_0, x_1, x_2, x_3, x_4
Function sbox$(x_0, x_1, x_2, x_3, x_4)$
Let `t0`, `t1`, `t2`, `t3`, and `t4` be temporary variables

```
x0 ^= x4;   x4 ^= x3;   x2 ^= x1; /*01*/
t0  = x0;   t1  = x1;   t2  = x2;   t3  = x3;   t4  = x4; /*02*/
t0 =~ t0;   t1 =~ t1;   t2 =~ t2;   t3 =~ t3;   t4 =~ t4; /*03*/
t0 &= x1;   t1 &= x2;   t2 &= x3;   t3 &= x4;   t4 &= x0; /*04*/
x0 ^= t1;   x1 ^= t2;   x2 ^= t3;   x3 ^= t4;   x4 ^= t0; /*05*/
x1 ^= x0;   x0 ^= x4;   x3 ^= x2;   x2 =~ x2; /*06*/
```

return x_0, x_1, x_2, x_3, x_4
end Function

Lastly, the linear diffusion layer p_L of Ascon operates over each of the 64-bit register words of the state, and it is defined as:

$$
\begin{aligned}
x_0 &\leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \\
x_1 &\leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \\
x_2 &\leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6) \\
x_3 &\leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \\
x_4 &\leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)
\end{aligned}
$$

The design of the linear diffusion layer and the rotation values are similar to the Σ function in SHA-2. In Figure 4.3, each gray box represents a byte of the internal state of Ascon: p_c modifies only a byte of x_2, the substitution-step p_S operates on each bit

of the five words vertically aligned, and the linear-step p_L operates inside each word independently of each other.

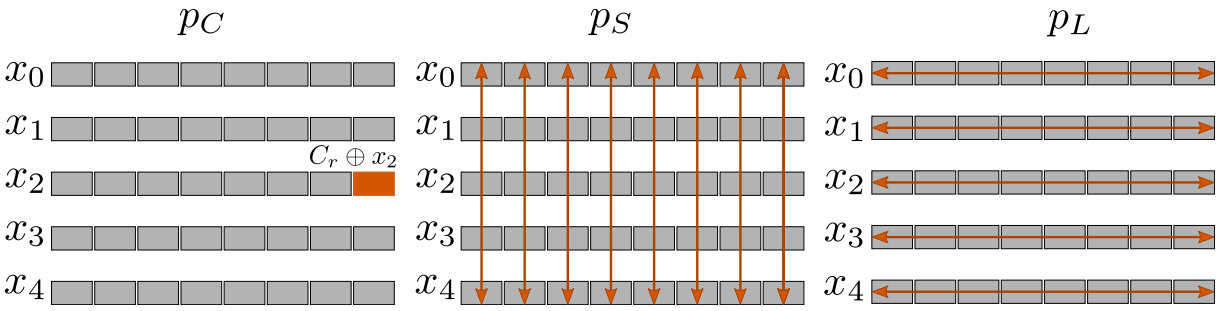

Figure 4.3: Scopes in the internal state for each step of the permutation.

4.3 Code profiling

In a similar manner to the profiling of NORX, Ascon was profiled using a basic encryption-decryption procedure to measure the function overheads. The only change done to code prior to measurements was disabling function inlining, what results in a better granularity and ease of analysis. The main loop of the code executes a call to the encryption function, with randomized inputs, and a 64KB payload length to minimize the impact of initialization overheads. The results for Cortex A15 are shown in Figure 4.4 and Figure 4.5 for Cortex A53. As expected for a sponge-based algorithm, the round permutation function p is the largest overhead, and therefore the best target for algorithm optimization.

4.4 NEON implementation and optimizations

At first, it may not be entirely clear on the CAESAR submission, but Ascon uses a big-endian data layout. This makes an efficient implementation of this procedure an important step in implementing and optimizing the algorithm. In Algorithm 35, it is shown the schoolbook approach to endianness change on a 64-bit word. This procedure uses six different masks, six shifts, six logical AND operations and three logical or operations.

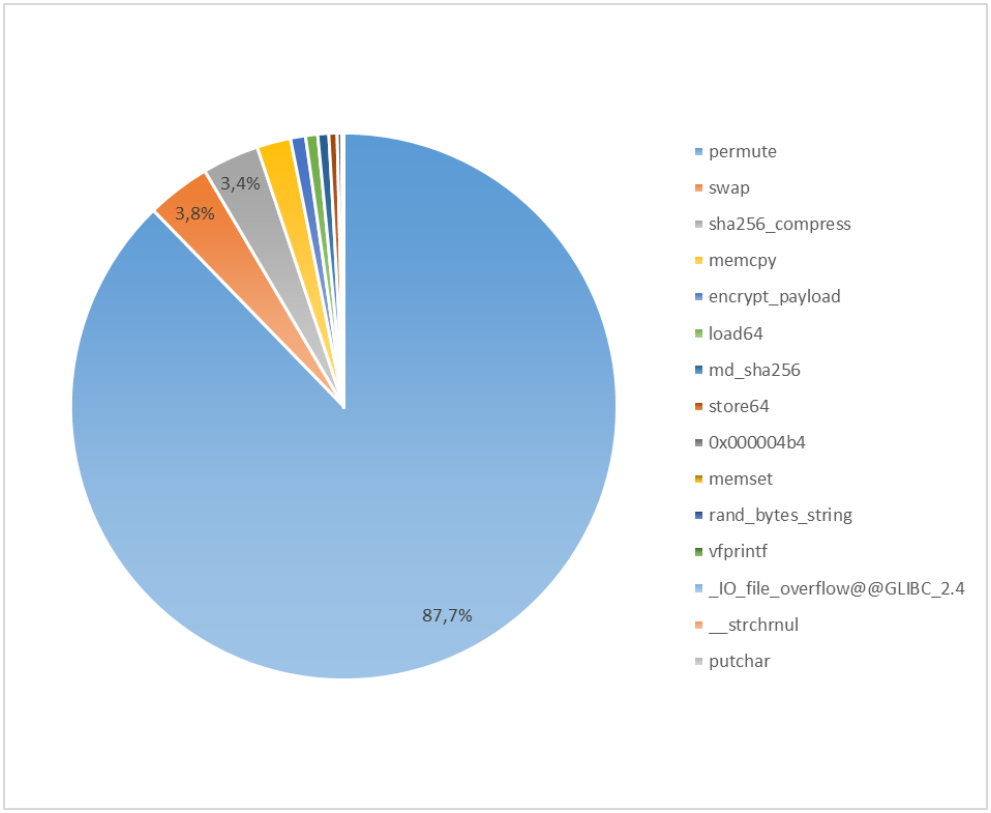

Figure 4.4: Overheads for ASCON128 running on a Cortex-A15 (32-bit processor)

Algorithm 35 Naive approach to an endianness change function.

```
1 static uint64_t swap(uint64_t a){
2     a = (a & 0x00000000FFFFFFFF)<<32 | (a & 0xFFFFFFFF00000000)
          >>32;
3     a = (a & 0x0000FFFF0000FFFF)<<16 | (a & 0xFFFF0000FFFF0000)
          >>16;
4     a = (a & 0x00FF00FF00FF00FF)<<8  | (a & 0xFF00FF00FF00FF00)
          >>8;
5 return a;
6 }
```

On ARMv7-A (32-bit) architectures, the swap can be implemented with some assembler in order to use the `REV` instruction. This approach gets even simpler on ARMv8-A (64-bit), were a single `REV` instruction needs to be issued to change the endianness of a whole 64-bit word. These approaches are shown, respectively, in Algorithm 36 and Algorithm 37. Lastly, a NEON implementation of the swap is then shown in Algorithm 38, and it is done via a simple call to the `vrev\lstinline` intrinsic.

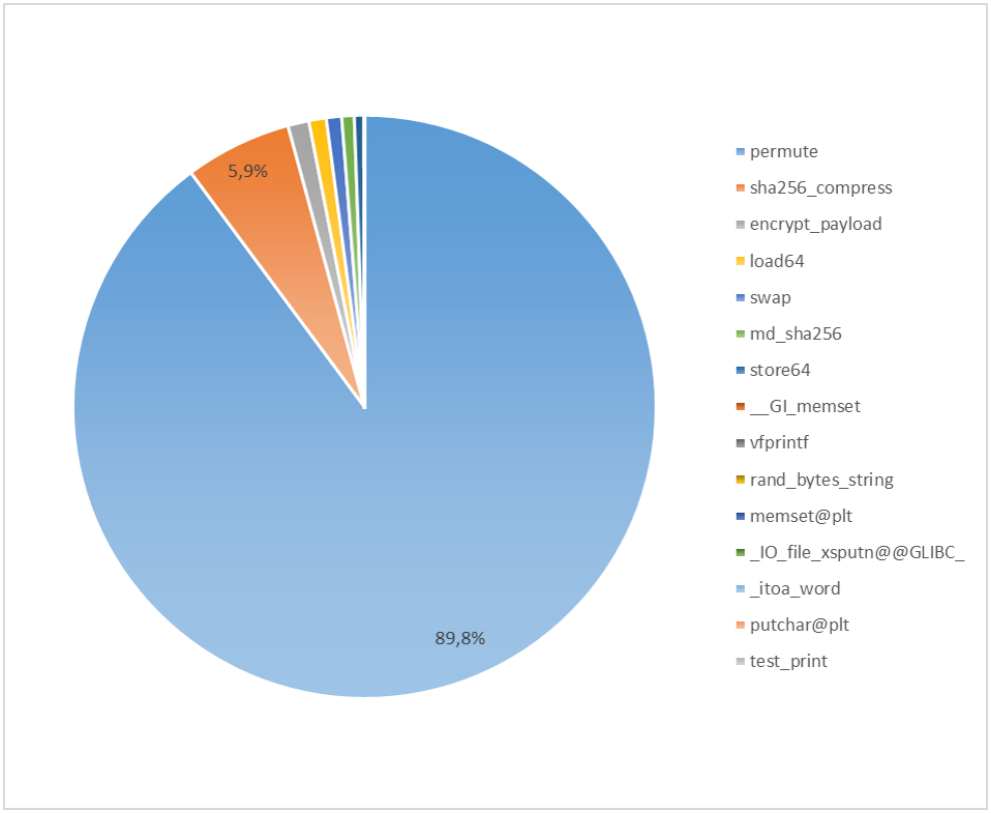

Figure 4.5: Overheads for ASCON128 running on a Cortex-A53 (64-bit processor)

Algorithm 36 Endianness change on ARM32.

```
static uint64_t swap(uint64_t a){
    uint32_t h=(uint32_t)(a>>32);
    uint32_t l=(uint32_t)(a);
    __asm__ (
        "rev %0, %0 \n\t"
        "rev %1, %1 \n\t"
        :"+r"(h), "+r"(l)
        :
        );
    a=((uint64_t)(l)<<32)|(uint64_t)(h);
    return a;
}
```

Algorithm 37 Endianness change on ARM64.

```
static uint64_t swap(uint64_t a){
        __asm__ ("rev %0, %0 \n\t"
            :"+r"(a)
            :
            );
        return a;
    }
```

Algorithm 38 Endianness change on ARM64.

```
#define swap_d(X) (uint64x1_t)(vrev64_u8((uint8x8_t)X))
#define swap_q(X) (uint64x2_t)(vrev64q_u8((uint8x16_t)X))
```

In order to better and easily implement Ascon, the NEON load and store intrinsics were wrapped together with the `vrev` instructions, as shown in Algorithm 39

Algorithm 39 Load and Store instructions for the NEON implementation of Ascon.

```
#define LOAD64x1(in) \
    (uint64x1_t)(vrev64_u8(vld1_u8((uint8_t*)(in))))
#define LOAD64x2(in) \
    (uint64x2_t)(vrev64q_u8(vld1q_u8((uint8_t*)(in))))
void inline static STORE64x1(uint8_t *out, uint64x1_t x){
    vst1_u8(out, vrev64_u8((uint8x8_t)(x)));
}
void inline static STORE64x2(uint8_t *out, uint64x2_t x){
    vst1q_u8(out, vrev64q_u8((uint8x16_t)(x)));
}
```

With those rotations in mind, a good implementation of Ascon can be made using the native 64-bit registers to hold the five words of the internal state. In pure C language, it is done using the `uint64_t` type and a trivial NEON implementation can be done using the 64-bit registers. But in order use the full potential of the NEON SIMD engine, Ascon can be implemented using two 128-bit NEON registers, and an additional 64-bit one, by means of a different state representation. Figure 4.6 illustrates the word layout in the NEON implementation of ASCON.

In this new representation, the SBOX can then be implemented as shown in Algorithm 40. Notice that, as with NORX NEON implementation, calls to `vcombine_u64` with `vget_high_u64` and `vget_low_u64` as arguments compiles into `vmov` and `vswp` instructions.

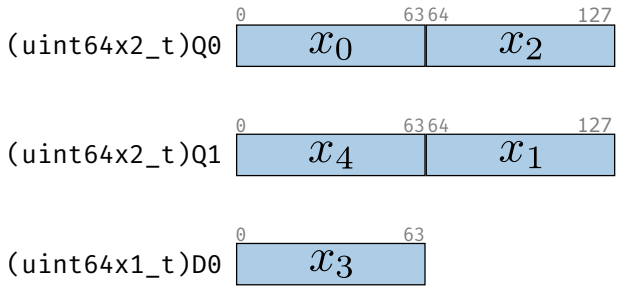

Figure 4.6: Sponge layout for the NEON implementation of ASCON.

Algorithm 40 NEON implementation of Ascon SBOX. This algorithm uses the sponge representation shown in Figure 4.6. The comment numbering inside the code refers to the pipelined steps in Algorithm 34.

Input: `Q0` $= \{x_0 \parallel x_2\}$, `Q1` $= \{x_4 \parallel x_1\}$, `D0` $= \{x_3\}$ ▷ State words
Output: `Q0` $= \{x_0 \parallel x_2\}$, `Q1` $= \{x_4 \parallel x_1\}$, `D0` $= \{x_3\}$
Function sbox(`Q0`, `Q1`, `D0`)

```
#define high(n) (vget_high_u64(n))
#define low(n) (vget_low_u64(n))
#define vcmbn(low,high) (vcombine_u64(low,high))
//vars
uint64x2_t Q0, Q1, Qt0, Qt1; uint64x1_t Dt0={0};
/*01*/
Q0 ^= Q1;
Q1 ^= vcmbn(D0, Dt0);
/*02 and 03*/
Qt0 = ~Q0;
Qt1 = ~Q1;
Dt0 = ~D0;
/*04*/
Qt0 &= vcmbn(high(Q1), D0);
Qt1 &= vcmbn(high(Q0), low(Q0));
Dt0 &= low(Q0);
/*05*/
Q0 ^= vcmbn(high(Qt1), Dt0);
Q1 ^= vcmbn(high(Qt0), low(Qt0));
D0 ^= low(Qt1);
/*06*/
Dt0 = (uint64x1_t){0};
Q1 ^= vcmbn(0, high(Q0));
Q0 ^= vcmbn(low(Q1), Dt0);
D0 ^= low(Q0);
Q0 ^= vcmbn(low(Q0), ~high(Q0));
```

return `Q0`, `Q1`, `D0`
end Function

Keeping the same representation, the Lbox can be rewritten using vector shifts to implement the rotation. The function `vshlq_u64` allows to execute a left shift of the elements inside a 128-bit register by the elements stored as signed integers in a 128-bit register. This instruction also allows to use negative displacements to represent right shifts. The NEON implementation of the Lbox is shown in Algorithm 41. Notice that the 64-bit variable `D0` is rotated using the single element rotation `vshr_n_u64`.

Algorithm 41 NEON implementation of Ascon Lbox. This algorithm uses the sponge representation shown in Figure 4.6.

Input: $\texttt{Q0} = \{x_0 \parallel x_2\}, \texttt{Q1} = \{x_4 \parallel x_1\}, \texttt{D0} = \{x_3\}$ ▷ State words
Output: $\texttt{Q0} = \{x_0 \parallel x_2\}, \texttt{Q1} = \{x_4 \parallel x_1\}, \texttt{D0} = \{x_3\}$
Function lbox(Q0, Q1, D0)

```
#define high(n) (vget_high_u64(n))
#define low(n) (vget_low_u64(n))
#define vcmbn(low,high) (vcombine_u64(low,high))
//Vector rotations
int64x2_t r1={-19, -1};
int64x2_t r2={45, 63};
int64x2_t r3={-28, -6};
int64x2_t r4={36, 58};
int64x2_t r5={-7, -62};
int64x2_t r6={57, 3};
int64x2_t r7={-41, -39};
int64x2_t r8={23, 25};
uint64x1_t t3; //temporary variable

Q0 ^= (vshlq_u64(Q0, r1) | vshlq_u64(Q0, r2))^ \
      (vshlq_u64(Q0, r3) | vshlq_u64(Q0, r4));

Q1 ^= (vshlq_u64(Q1, r5) | vshlq_u64(Q1, r6))^ \
      (vshlq_u64(Q1, r7) | vshlq_u64(Q1, r8));

#define ROTR(x,n) (veor_u64(vshr_n_u64((x),(n)), \
      vshl_n_u64((x),(64-n))))
t3 = veor_u64(ROTR(S[4],  7), ROTR(S[4], 41));
D0 = veor_u64(D0,t3);
```

return Q0, Q1, D0
end Function

Another method that can be used to further improve Ascon in a NEON implementation is data pre-fetching. Since the whole state uses only 5 NEON registers, the payload processing functions can load data for the next n iterations, execute them and only then save those back into the memory. Such an approach allows a better use of the NEON pipeline, which allows the issue of load/store instructions together with logic/arithmetic ones. In our tests, Ascon-128 gets the best performance improvement

with a 4-word pre-fetching, while Ascon-128a shows the best improvement with a 2-word pre-fetching.

A different approach would be using a Multiple Message implementation. The main characteristic this multiple message implementation is to make use of the second 64-bit lane of a NEON type-Q register to perform two parallel instances of Ascon. This allows some parallelization to be achieved, even with the internal data dependencies seen in a single-message implementation. Such approach is useful, for example, when a somewhat simpler device has a higher volume of data to encrypt and authenticate, and the payloads share their length: For example, a sensor that must send data to more than one recipient, and they cannot share key, nonce or authenticated data.

4.5 Results and considerations

Table 4.4: Times in CPB for ASCON128 and ASCON128a on the 32-bit processors, with payload size of 64KB. Speedups given in relation to the reference code.

		Ref. Code	Our code	NEON64	NEON128	NEON64 Speedup	NEON128 Speedup	Own code speedup
ASCON128	Cortex A7	174.10	127.49	120.11	91.45	31.0%	47.5%	26.8%
	Cortex A15	90.14	68.20	55.10	49.32	38.9%	45.3%	24.3%
ASCON128a	Cortex A7	111.75	91.19	81.62	64.13	27.0%	42.6%	18.4%
	Cortex A15	57.62	42.66	35.60	30.29	38.2%	47.4%	26.0%

Table 4.5: Times in CPB for ASCON128 and ASCON128a on the 64-bit processor, with payload size of 64KB. Speedups given in relation to the reference code.

	Ref. code	Our code	NEON64	NEON128	NEON64 Speedup	NEON128 Speedup	Own code speedup
ASCON128	22.28	10.79	10.72	40.65	51.9%	-82.3%	51.57%
ASCON128a	28.83	12.90	14.97	27.63	48.1%	-3.0%	55.25%

Ascon is well rounded-up and well-designed algorithm – a fact acknowledged by its performance on CAESAR –, what makes optimization efforts very challenging. For example, changing the algorithm endianness could result in performance improvements on the target architecture, but there is not a good mapping of the linear permutation to a little-endian equivalent. On the other hand, making good use of the endianness changing instructions present on ARM keeps this overhead at a minimum: Comparing the times of a C implementation using the endianness swap instructions with a non-compatible one without the endianness change instruction, the difference in performance was less than 1%.

Regarding a C implementation, good practices such as using adequate data types, macros, and function inlining yield a good performance gain in relation to the reference code, and match the performance of other state-of-art implementations, without sacrificing code readability. The clever use of preprocessor macros allows one to use the architecture-specific endianness change functions while also keeping code compatibility with other architectures.

The internal structure of Ascon makes it a challenging algorithm to implement using NEON SIMD engine: the algorithm uses simple bit-wise operations, and the internal data dependencies make it difficult to offset the intrinsic overheads of moving data around the NEON register, in comparison with the native registers. This fact makes a NEON implementation attractive only on 32-bit processors, since the native 64-bit registers are superior to the NEON registers.

Ascon shows no speedup on the 64-bit processor Cortex-A53, when compared with an optimized C implementation; mainly due to the design of the linear layer. The Lbox is an operation executed in each line of the internal state, having the form $x_n \leftarrow x_n \oplus (x_n \ggg r_{n1}) \oplus (x_n \ggg r_{n2})$, where x_n is one of the five 64-bit words of Ascon's state, and r_{nm} are the rotation constants. An implementation using the native 64-bit registers of this function can be done as 2 XORs, with the one rotation being executed "for free" using ARM's barrel shifter, as shown in Algorithm 42. The NEON code for the same function is more complicated, due to the lack of a `ror` instruction and the barrel shifter, the rotation must be calculated as shifts and `and` operations[1]. The situation is slightly improved in the case of a 128-bit NEON implementation of Ascon, where using the vector shift operation four of the five rotations can be executed in parallel.

Beyond that, upon analysis of the generated binaries, the GCC compiler will, starting at `-O1`, move as much as possible of the 64-bit operations from the NEON registers back to the native 64-bit registers. The dump of the NEON code for a Lbox, with `-O3` compiler optimizations, is shown in Appendix 48. With that in mind, a 64-bit NEON implementation of Ascon on a ARMv8 core will actually be a native 64-bit implementation, except when using NEON-only instructions. Table 4.6 shows the cycle-per-byte costs for the Lbox and Sbox code, showing that the cost of a NEON 64-bit implementation is very close to a native C implementation, what further confirms that the processing is not being done in the SIMD engine, but instead inside the native 64-bit registers. Another interesting comparison is that of the performance of a 128-bit NEON implementation across different platforms: Comparing with Cortex-A15, the same implementation is faster on the Cortex-A53, what suggests that, in this scenario, the native 64-bit registers and instructions are exceedingly good in performing the computations of `Ascon`.

Algorithm 42 Native assembler of ASCON LBOX. Immediate values are the ones applied to x_0.

```
1 <lbox>:
2 ;*x ^= ROTR(*x, 19) ^ ROTR(*x, 28);
3 ldr     x1, [x0]
4 ror     x2, x1, #28 ;
5 eor     x3, x2, x1, ror #19 ;
6 eor     x4, x1, x3
7 str     x4, [x0]
8 ret
```

[1] A 64-bit NEON rotation of `x` by `n` bits is defined as `veor_u64(vshr_n_u64((x),(n)), vshl_n_u64((x),(64-n)))`.

Table 4.6: Costs of Ascon permutation on Cortex A53

		Ascon128	Ascon128a
NEON 128bit	SBOX	25.87	17.06
	LBOX	18.84	12.41
Native C	SBOX	6.78	4.52
	LBOX	4.39	4.40
NEON 64bit	SBOX	7.65	4.52
	LBOX	5.90	4.09

Comparing the performance of implementations written during this dissertation, both the native and NEON implementations were faster than the reference code from the Ascon team. The speedup was, in relation to this reference code, 27% on the Cortex A7, 24% on the Cortex A15, and 55% on the Cortex A53, with the NEON implementation being 47% faster on the Cortex-A7 and 45% on the Cortex-A15. And in relation to ASCON128a, the native C implementation showed a speedup of 18% in Cortex A7, 26% on Cortex A15, and 51% on Cortex A53. The NEON implementation shows speedups of 47% and 45% on Cortex A7 and A15, respectively. In Table 4.4 it is shown the results for ASCON128 and ASCON128a on the 32-bit processors, and in Table 4.5, the results for the 64-bit Cortex-A53 processor. Figures 4.7 through 4.12 show the charts with cycle-per-byte results for inputs ranging from 128 bytes to 64 Kilobytes.

The Multi-message version of the algorithm shows a speedup of 53% in relation to the single-message NEON implementation con Cortex A15 and A7 (respectively 43,44 and 82,19 cycles per byte). The performance gain is small compared to the 128-bit single-message SIMD implementation mainly due to unaligned load-stores. Similarly to the other NEON implementations, this code is not adequate for the 64-bit processors due to the lack of `ror` instruction and barrel shifter.

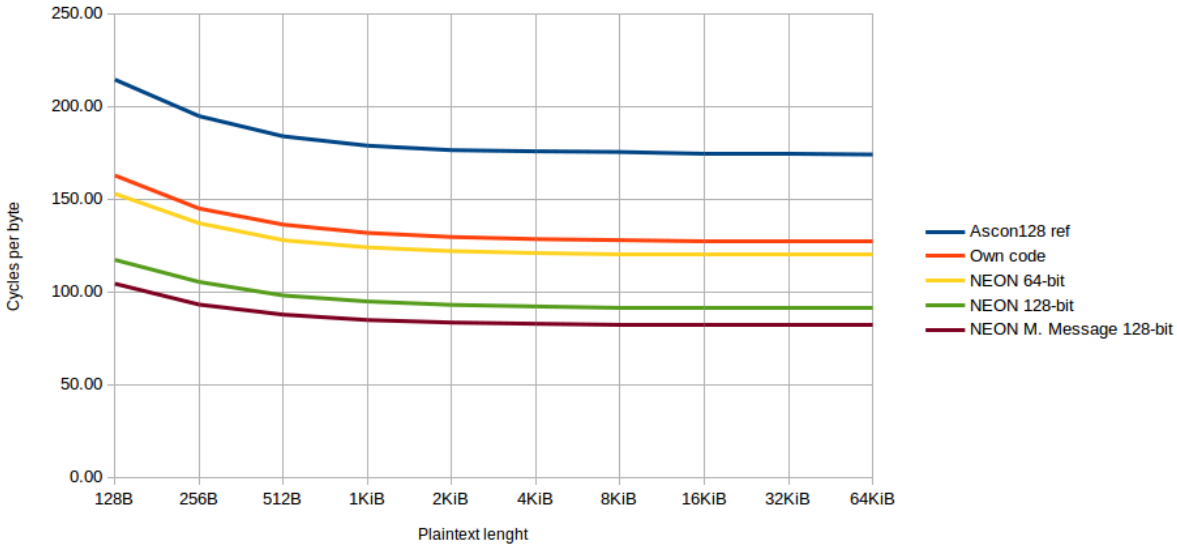

Figure 4.7: CPB results for ASCON128 on Cortex A7.

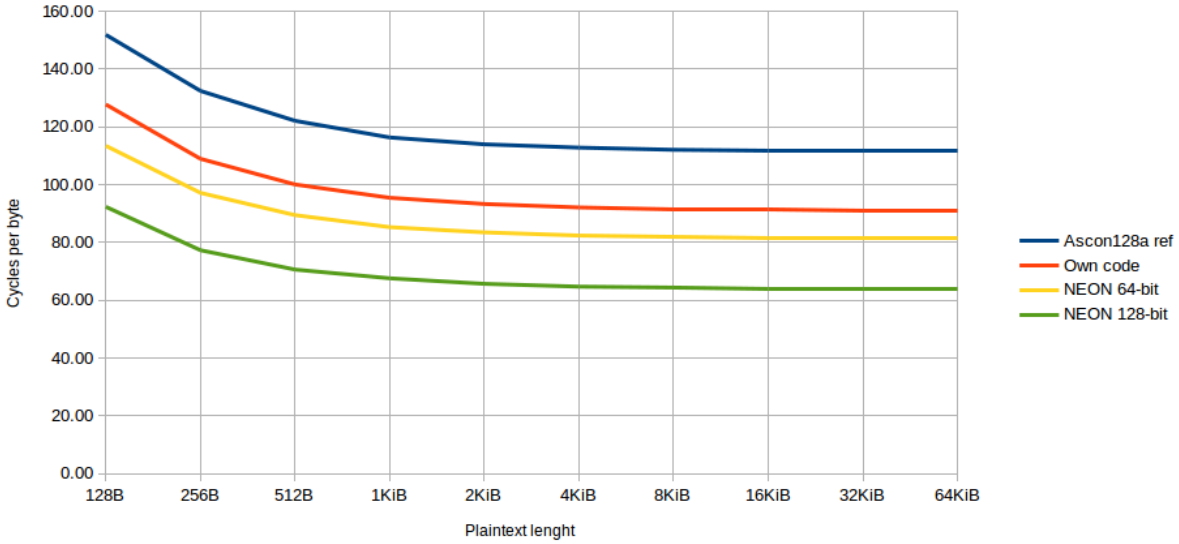

Figure 4.8: CPB results for ASCON128a on Cortex A7.

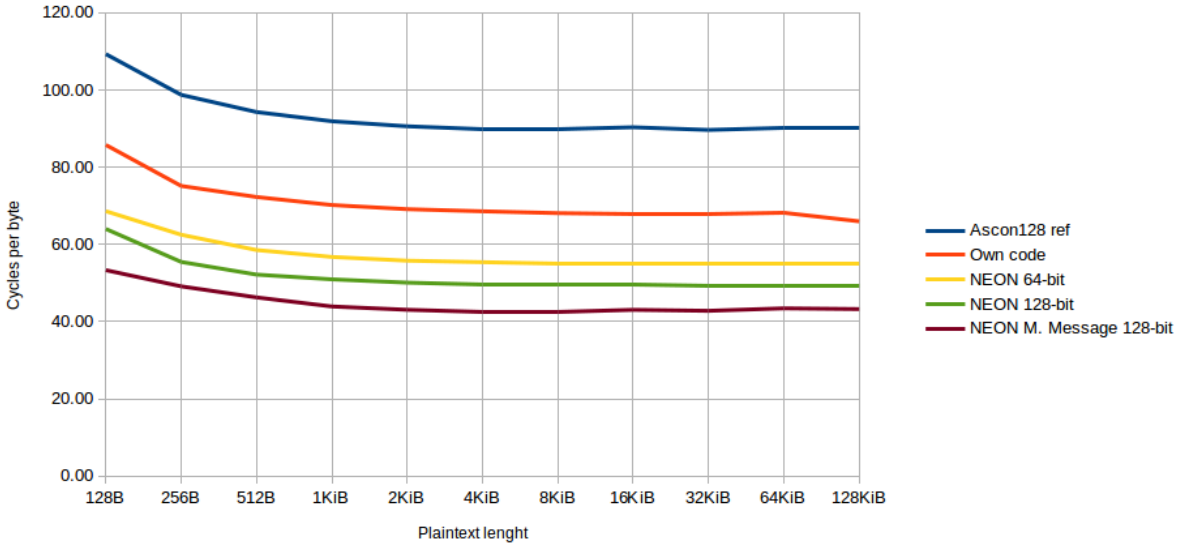

Figure 4.9: CPB results for ASCON128 on Cortex A15.

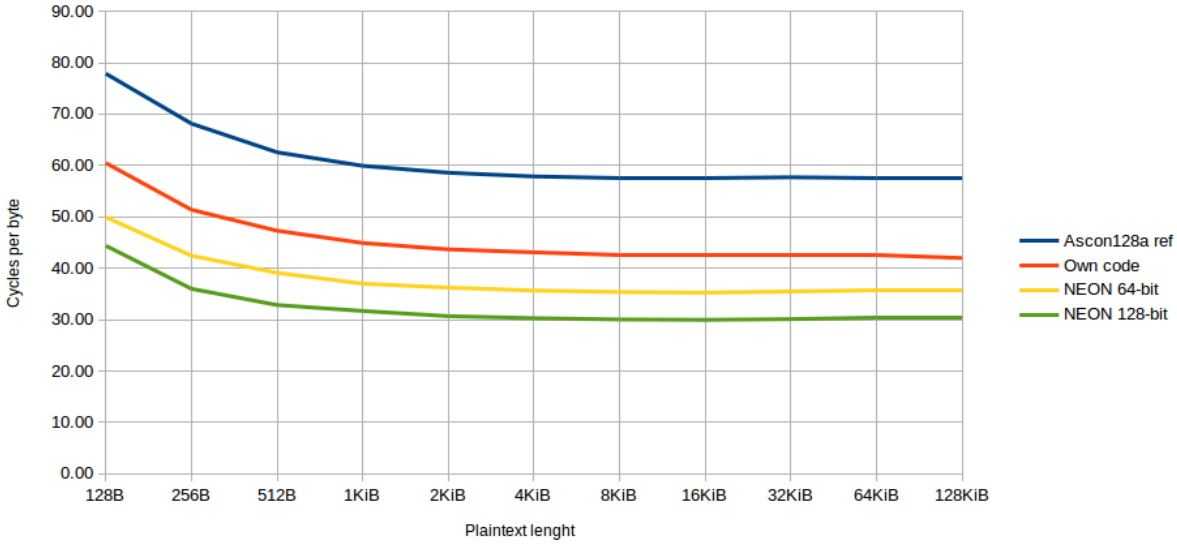

Figure 4.10: CPB results for ASCON128a on Cortex A15.

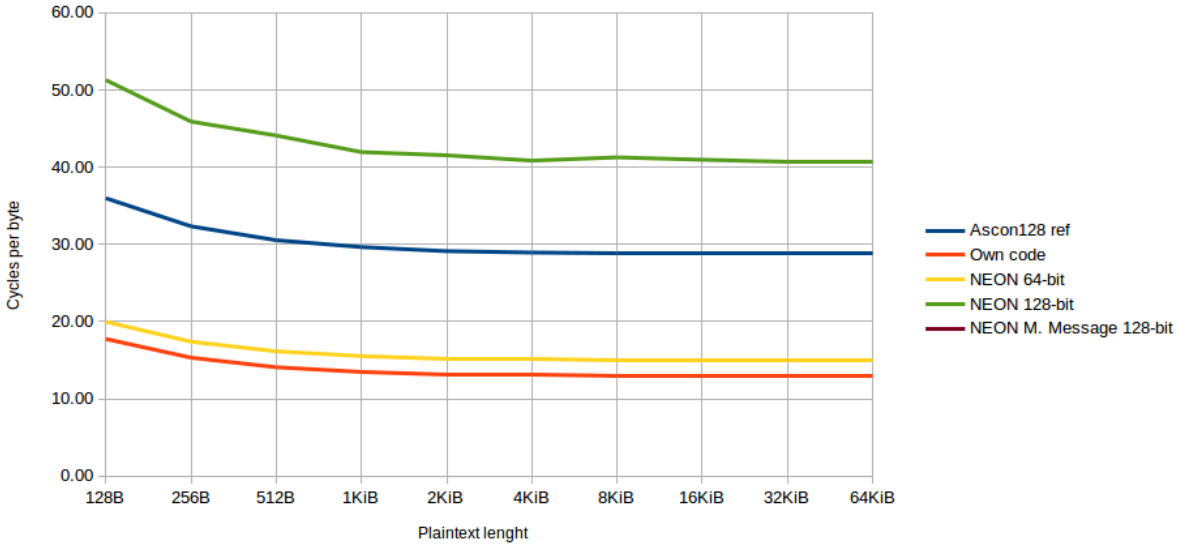

Figure 4.11: CPB results for ASCON128 on Cortex A53.

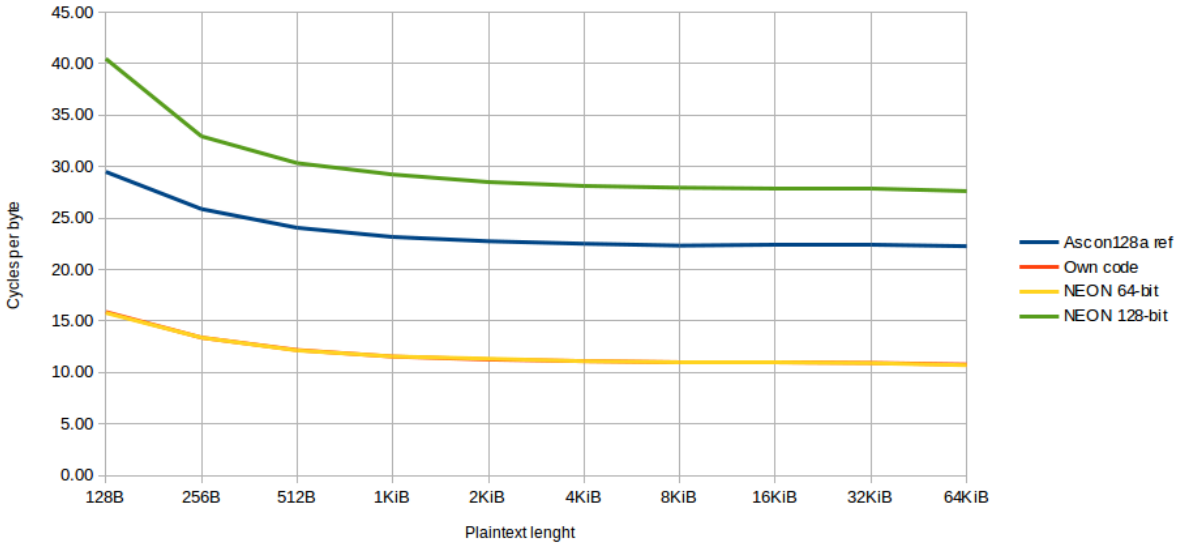

Figure 4.12: CPB results for ASCON128a on Cortex A53.

Chapter 5

Conclusion and final remarks

In this dissertation, we investigate the software implementation of authenticated encryption algorithms on ARM Cortex-A processors. We show techniques to optimize NORX and Ascon, two sponge-based AEAD schemes, using characteristics of the target processors such as instruction pipelines and vectorial instructions. The study of AEAD algorithms allowed a deeper insight into the current design paradigms; for example, these algorithms are built without key-dependent table lookups or branchings as to make them more secure against attacks.

During this dissertation, the structure of AEAD algorithms was studied, what allowed a deeper insight in the current paradigm of construction; not only performance is taken into account when designing an algorithm, as they already have side-channel countermeasures built right in. For example, neither NORX or ASCON have key dependent operations, table lookups, or key-dependent branching. Their construction is simple, feature mostly the ARX operations. With that in mind, the CAESAR finalists are very well thought algorithms, and hard to find purely mathematical optimizations. On the other hand, there is still space for improvement in a more "Cryptographic Engineering" approach, albeit not as pronounced as one would expect from less polished algorithms, but still relevant.

Sponge-based algorithms with a construction similar to NORX can profit from a well-crafted C implementation, when the characteristics of the target architecture, such as instruction pipelining, are taken into account. In some cases, such an implementation can be superior to a SIMD oriented implementation. This can be counter-intuitive at first: One would expect that a vectorized implementation will be faster than a code running on the native registers. The behavior of NORX is explained by the cost of loading and storing data on NEON registers, and extra transformations needed to execute the sponge permutation –Namely, the diagonalization and undiagonalization steps. With this, one conclusion is that vectorization is not a panacea and overheads must be taken into account in order to determine the best solution for each scenario: In this case, speedups of up to 44% can be achieved with a pipeline-oriented implementation.

ASCON proved to be a difficult target to optimization: its structure and construction left little to work with in terms of implementation, and no mathematical improvement was found. It's 64-bit per word internal state makes it a fast algorithm in 64-bit platforms. On the other hand, those same characteristics make it a bad candidate for

a NEON implementation, with overheads offsetting any gains from a SIMD engine, and those overheads get even more pronounced on ASCON128a since it has double the load/store overheads in relation to ASCON128. Even with that, ASCON is a very simple and efficient algorithm and a strong participant of CAESAR. Even with those characteristics in mind, a diligent change of the internal layout of the cipher allows an efficient NEON implementation, with up to 47% of speedup in relation to the CAESAR code on the 32-bit ARM processors.

In a nutshell, we introduced a method of implementing and optimizing two families of AEAD schemes by using well-crafted C implementations, tailored for ARM processors. The work on NORX resulted in performance improvements over the state-of-art software implementations that were published on SBSEG 2017; and we also introduced and analyzed a vectorial implementation of Ascon, that makes full use of the 128-bit NEON SIMD engine, as well as a multiple-message vectorial implementation. A future step in this work would be to apply these optimization techniques to other algorithms of interest, as they allow fast code, with the flexibility of a C implementation. Of great interest are the algorithms of the upcoming NIST "Call for Lightweight Cryptography", that aims to define cryptographic standards for the Internet of Things.

References

[1] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[2] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. NORX v3.0. norx.io/data/norx.pdf, September 2016.

[3] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1. 2. *Submission to the CAESAR Competition*.

[4] L. C. d. Santos and J. López. Pipeline Oriented Implementation of NORX for ARM Processors. In *XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais: SBSEG 2017: Anais*, pages 2–15. Sociedade Brasileira de Computação - SBC, Nov 2017.

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

[6] F.W. Winterbotham. *The Ultra Secret: The Inside Story of Operation Ultra, Bletchley Park and Enigma*. Orion (an Imprint of The Orion Publishing Group Ltd), 2000.

[7] Claude E Shannon. Communication theory of secrecy systems*. *Bell system technical journal*, 28(4):656–715, 1949.

[8] National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. National Institute for Standards and Technology, Gaithersburg, MD, USA, October 1999. supersedes FIPS 46-2.

[9] William E Burr. Data encryption standard. *A Century of Excellence in Measurements, Standards, and Technology–A Chronicle of Selected NBS/NIST Publications*, 2000:250–253, 1901.

[10] Joan Daemen and Vincent Rijmen. Rijndael for AES. In *AES Candidate Conference*, pages 343–348, 2000.

[11] NIST-FIPS Standard. Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197:1–51, 2001.

[12] Lynn Hathaway. National policy on the use of the advanced encryption standard (aes) to protect national security systems and national security information. *National Security Agency*, 23, 2003.

[13] Masanobu Katagi and Shiho Moriai. Lightweight cryptography for the internet of things. *Sony Corporation*, pages 7–10, 2008.

[14] Kerry A McKay, Larry Bassham, Meltem Sönmez Turan, and Nicky Mouha. Report on lightweight cryptography. *NIST DRAFT NISTIR*, 8114, 2016.

[15] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. *IACR Cryptology ePrint Archive*, 2011:499, 2011.

[16] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007. Citeseer, 2007.

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

[18] Bart Preneel. AHS competition/sha-3. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pages 27–29. Springer, 2011.

[19] Phillip Rogaway. Authenticated-encryption with associated-data. In *ACM Conference on Computer and Communications Security*, pages 98–107. ACM, 2002.

[20] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is ssl?). In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer, 2001.

[21] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, 2008.

[22] Charanjit S. Jutla. Encryption modes with almost free message integrity. *IACR Cryptology ePrint Archive*, 2000:39, 2000.

[23] Phillip Rogaway, Mihir Bellare, and John Black. Ocb: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):365–403, 2003.

[24] Morris J. Dworkin. Sp 800-38c. recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality. Technical report, Gaithersburg, MD, United States, 2004.

[25] Tadayoshi Kohno, John Viega, and Doug Whiting. CWC: A high-performance conventional authenticated encryption mode. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 408–426. Springer, 2004.

[26] Mihir Bellare, Phillip Rogaway, and David Wagner. The EAX mode of operation. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 389–407. Springer, 2004.

[27] Morris J. Dworkin. Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Technical report, Gaithersburg, MD, United States, 2007.

[28] Algorithms, key size and parameters report 2014. Technical report, European Network and Information Security Agency (ENISA), November 2014.

[29] Charanjit S. Jutla. Encryption modes with almost free message integrity. Cryptology ePrint Archive, Report 2000/039, 2000. `http://eprint.iacr.org/`.

[30] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, 2003.

[31] National Institute of Standards and Technology. Aes: the advanced encryption standard. http://competitions.cr.yp.to/aes.html, January 2014.

[32] Alex Biryukov and Dmitry Khovratovich. Related-key cryptanalysis of the full AES-192 and AES-256. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2009.

[33] Marion Videau. estream. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pages 426–427. Springer, 2011.

[34] CAESAR Committee. Autheticated encryption zoo. `http://aezoo.compute.dtu.dk/doku.php`, 2016.

[35] Farzaneh Abed, Christian Forler, and Stefan Lucks. General overview of the first-round caesar candidates for authenticated ecryption. Technical report, Cryptology ePrint report 2014/792, 2014.

[36] Hongjun Wu and Bart Preneel. AEGIS: A fast authenticated encryption algorithm. *IACR Cryptology ePrint Archive*, 2013:695, 2013.

[37] Hongjun Wu and Tao Huang. The JAMBU lightweight authentication encryption mode (v2. 1). *CAESAR competition proposal*, 2016.

[38] Kazuhiko Minematsu. AES-OTR v3. *CAESAR competion proposal*, 2016.

[39] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. AEZ v1: Authenticated-encryption by enciphering. *CAESAR 1st Round, competitions. cr. yp. to/round1/aezv1. pdfl*, 2014.

[40] Kazuhiko Minematsu, Jian Guo, and Eita Kobayashi. Cloc and silc. 2016.

[41] Elena Andreeva, Andrey Bogdanov, Nilanjan Datta, Atul Luykx, Bart Mennink, Mridul Nandi, Elmar Tischhauser, and Kan Yasuda. Colm v1 (2016). *Submission to the CAESAR competition*.

[42] Jérémy Jean, Ivica Nikolic, Thomas Peyrin, and Yannick Seurin. Deoxys v1. 41. *CAESAR candidate*, 2016.

[43] Ted Krovetz and Phillip Rogaway. Ocb (v1. 1). submission to caesar (2016), 2016.

[44] Ivica Nikolić. Tiaoxin-346. *Submission to the CAESAR competition*, 2015.

[45] Hongjun Wu and Tao Huang. The authenticated cipher morus (v1). *CAESAR submission*, 2014.

[46] Hongjun Wu. Acorn: a lightweight authenticated cipher (v3). *Candidate for the CAESAR Competition. See also https://competitions. cr. yp. to/round3/acornv3. pdf*, 2016.

[47] Prakash Dey, Raghvendra Singh Rohit, and Avishek Adhikari. Full key recovery of acorn with a single fault. *Journal of Information Security and Applications*, 29:57–64, 2016.

[48] G Bertoni, J Daemen, M Peeters, and GV Assche. Caesar submission: Ketje v2. online at http://ketje. noekeon. org/ketje-1.1. pdf, 2014.

[49] Guido Bertoni, Joan Daemen, Michaël Peeters, GV Assche, and RV Keer. Caesar submission: Keyak v2 (2015), 2016.

[50] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. NORX: parallel and scalable AEAD. In *ESORICS (2)*, volume 8713 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2014.

[51] Douglas R. Stinson. Universal hashing and authentication codes. *Des. Codes Cryptography*, 4(4):369–380, 1994.

[52] Simon Cogliani, DS Maimut, David Naccache, Rodrigo Portella do Canto, Reza Reyhanitabar, Serge Vaudenay, and D Vizźar. Offset merkle-damgård (omd) version 1.0 a caesar proposal. *Proposal in CAESAR competition (March 2014)*, 2014.

[53] Viet Tung Hoang and Phillip Rogaway. On generalized feistel networks. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 613–630. Springer, 2010.

[54] CAESAR committee. Features of various secret-key primitives. http://competitions.cr.yp.to/features.html, January 2014.

[55] ARM Holdings. Processors cortex-a series family. http://www.arm.com/products/processors/cortex-a, March 2017.

[56] ARM Holdings. Processors cortex-m series family. http://www.arm.com/products/processors/cortex-m, March 2017.

[57] ARM Holdings. Processors cortex-r series family. http://www.arm.com/products/processors/cortex-r, March 2017.

[58] ARM Holdings. Processors securcore family. http://www.arm.com/products/processors/securcore, March 2017.

[59] CAESAR Committee. Cryptographic competitions google group. http://groups.google.com/forum/#!forum/crypto-competitions, 2017.

[60] Daniel J Bernstein. Supercop: System for unified performance evaluation related to cryptographic operations and primitives. https://bench.cr.yp.to/supercop.html, 2009.

[61] Vincent Grosso, Gaëtan Leurent, François Durvaux, Lubos Gaspar, and Stéphanie Kerckhof. Caesar candidate scream. In *DIAC 2014*, 2014.

[62] Bruce Schneier. *Secrets and lies: digital security in a networked world*. John Wiley & Sons, 2011.

[63] Daniel J Bernstein. Cache-timing attacks on AES. http://palms.ee.princeton.edu/system/files/Cache-timing+attacks+on+AES.pdf, 2005.

[64] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 368–388. Springer, 2016.

[65] Paul C Kocher, Joshua M Jaffe, and Benjamin C Jun. Cryptographic computation using masking to prevent differential power analysis and other attacks, February 23 2010. US Patent 7,668,310.

[66] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. 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 142–159. Springer, 2013.

[67] Bodo Möller. Securing elliptic curve point multiplication against side-channel attacks. In *International Conference on Information Security*, pages 324–334. Springer, 2001.

[68] Ljiljana Spadavecchia. *A network-based asynchronous architecture for cryptographic devices*. PhD thesis, University of Edinburgh, UK, 2006.

[69] Daniel J Bernstein. Chacha, a variant of salsa20. In *Workshop Record of SASC*, volume 8, 2008.

[70] Alex Biryukov and Dmitry Khovratovich. PAEQ: parallelizable permutation-based authenticated encryption. In *ISC*, volume 8783 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2014.

[71] Jake Edge. Perfcounters added to the mainline, 2009. `https://lwn.net/Articles/339361/`, 2016.

[72] ARM. Arm compiler user guide. `http://infocenter.arm.com/help/topic/com.arm.doc.dui0472m/DUI0472M_armcc_user_guide.pd`.

[73] Bruno R Silva, Leonardo Ribeiro, Diego Aranha, and Fernando MQ Pereira. Flowtracker-detecç ao de código nao isócrono via análise estática de fluxo.

[74] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. Norx reference implementations (software). `https://github.com/norx/norx`, 2015.

[75] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-OHearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[76] SSL WolfSSL-Embedded. Library for applications, devices, iot, and the cloud.

[77] OpenSSL Project.

[78] Bouncy Castle. Bouncy castle crypto apis. *http://www. bouncycastle. org/*, 2007.

[79] Trevor Perrin. The noise protocol framework. `https://noiseprotocol.org/noise.pdf`, 2016.

[80] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password-scrambling framework. Technical report, Citeseer, 2013.

[81] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Permutation-based encryption, authentication and authenticated encryption. *Directions in Authenticated Ciphers*, 2012.

[82] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The keccak reference. Submission to NIST (Round 3), 2011.

[83] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. Norx a parallel and scalable authenticated encryption algorithm. `https://github.com/norx`, 2016.

Appendix A

NORX test vectors

A.1 Computations of F

Given the values of the initialization constants, an implementation of F can be verified by recalculating the constants as:

$$u_0, \cdots, u_{15} = F^2(0, \cdots, 15)$$

Where the elements $\{0, \cdots, 15\}$) are words of adequate bit-length: 32 bits for NORX32 constants and 64 bits for NORX64. The values of the initialization constants are given on table 3.5

A.2 Full AEAD computations

Given the input values in Table A.1, the values of the full AEAD computations are given on the next sections. For a more in depth trace of the full AEAD computations, the reader is invited to see the original NORX paper [2] and the official NORX repository [83].

Table A.1: Inputs for NORX

NORX32					NORX32				
Var	Data			Length	Var	Data			Length
K	00 01	...	0E 0F	16	K	00 01	...	1E 1F	32
N	20 21	...	2E 2F	16	N	20 21	...	3E 3F	32
A	00 01	...	7E 7F	128	A	00 01	...	7E 7F	128
M	00 01	...	7E 7F	128	M	00 01	...	7E 7F	128
Z	00 01	...	7E 7F	128	Z	00 01	...	7E 7F	128

A.2.1 NORX32-4-1

Ciphertext:

```
6C E9 4C B5 48 B2 0F ED 7B 68 C6 AC 60 AC 4C B5
```

```
EB B1 F0 9A EC 5A 75 0E CF 50 EC 0E 64 93 8B F2
40 17 A4 FF 06 84 F8 08 A6 7C 19 6C 31 A0 AF 12
56 9B E5 F7 C5 6A D3 BC AC 88 DA 36 86 57 5F 93
43 96 8D A2 20 77 EE CC E7 D6 63 17 49 08 A3 F7
3C 9E 9A C1 49 B5 CE 6B E6 9C 9E 31 7C D7 E7 E8
0C 85 69 97 74 02 24 41 3A E0 64 A2 5A 81 08 B8
D3 A6 85 92 74 C7 65 86 E2 9C 27 ED 11 FB 71 95
```

Tag:

```
D5 54 E4 BC 6B 5B B7 89 54 77 59 EA CD FF CF 47
```

A.2.2 NORX32-6-1

Ciphertext:

```
20 9B 0B 2A FE 36 2A 83 3B B1 8A CF 03 E1 D0 C2
7C 69 47 52 66 79 47 FC 73 8C 0E 40 E3 D5 97 C2
2D 74 E9 06 E8 C4 73 AD F0 DB 63 61 D3 97 41 C4
26 0F B3 D3 9F 84 22 A3 CF DF 93 0D 2D 17 75 EB
3F 97 0E 52 95 23 07 C9 AA 07 3F C5 E1 19 BA DF
B2 FF 00 9E 69 7C 8E 85 61 4F 44 78 C5 7B D2 B4
AC C5 57 F3 D2 DC E7 11 A5 43 0A 48 8C 16 63 A2
07 67 81 48 9A C7 3A 6B FB 6A FE 39 6A E7 9F 97
```

Tag:

```
B3 B1 1A 8F 9A 94 F1 B1 AC 18 53 E9 4C 43 26 4A
```

A.2.3 NORX64-4-1

Ciphertext:

```
C0 81 6E 50 8A E4 A0 50 0B 93 38 7B BB AB C2 41
AC 42 38 7E F5 E8 BF 0E C3 82 6C ED E1 66 A1 D5
CA A3 E8 D6 2C D6 41 B3 FA F2 AA 2A DD E3 E5 ED
0A 13 BD 8B 96 D5 F0 FB 7F E3 9C A7 80 95 31 75
E2 45 BC 3E 53 4B 80 0E 96 46 77 1F 13 EA 40 85
CB 3E 26 7F 10 6F 5F 17 A0 64 FF 23 4A 02 7C 64
4B E7 86 65 DB 1C 46 A4 B0 1A 4F BF 52 76 DF BD
30 EB BF B8 84 66 F8 DC 89 7A 78 16 D0 D0 70 D8
```

Tag:

```
D1 F2 FA 33 05 A3 23 76 E2 3A 61 D1 C9 89 30 3F
BF BD 93 5A A5 5B 17 E4 E7 25 47 33 C4 73 40 8E
```

A.2.4 NORX64-6-1

Ciphertext:

```
50 CE 69 2C 19 CB 91 02 C6 12 96 6F 0F 62 6B 62
96 DE 89 27 1C 98 29 10 AA C1 C3 55 52 2E 8F A7
13 03 F8 D5 C9 DE 39 04 84 BA 91 A9 94 CF F9 1B
F7 15 D6 CB 22 CC 00 F3 64 02 10 03 17 19 61 68
72 39 DD 94 53 02 9B 87 85 9C 10 93 21 13 59 40
BC 1B C8 1A 55 A9 51 C7 1B 29 42 FF DE BF 8D 13
C4 F3 87 2B 78 D4 50 6F 40 DB 65 3C E3 B8 D2 BE
A7 A2 F9 E9 7F F4 56 B7 F0 DB 8C 92 27 E2 2F 23
```

Tag:

```
A0 D1 0D 28 52 91 BE DB 7B 7C BD C4 7E 0F E2 38
5B F5 5B C5 F0 57 BC AB 2C 57 CC D0 83 D2 9B 2C
```

A.2.5 NORX64-4-4

Ciphertext:

```
B6 5A D4 9D 08 12 87 73 03 76 A0 38 F1 32 B2 0C
33 E5 58 30 20 27 C0 D9 1C 03 0B 9C 7D DA 19 C7
51 1A 4F 02 5A FD 40 FD A2 95 C9 22 29 FA EA 13
A6 14 05 36 44 0B EB FC D3 62 72 5D 9E E9 0F 2C
2A AC 10 6B 5F 49 86 9B 9F E2 2C D9 F1 84 84 FC
70 C2 22 8C 1D A3 07 21 21 97 2C 2B D9 9A 29 2A
15 51 52 B1 67 72 3F F7 CD A5 BB A3 DA 09 E3 69
F2 7B FE 53 88 63 FF 56 18 40 01 28 8C C1 BE EC
```

Tag:

```
01 61 3B 7E 49 80 00 A7 67 F5 D5 35 3F 8F FD 99
78 72 05 7C 1F DC 50 14 CF 82 27 EB B8 A7 5C AC
```

A.2.6 NORX64-6-4

Ciphertext:

```
B3 16 97 9C 8B 60 D2 0E 83 43 B2 A5 AD DB CF 61
68 CF E1 B4 C8 3C 3E C8 5E CE 1B 08 E8 BB 12 1F
A5 D1 08 D5 27 09 5C F5 56 36 26 A9 DD 6D 5F 56
03 DD 94 2C 6E 6B D2 01 96 37 84 18 94 02 21 78
E5 9E 03 6D FD 2C 01 AC 7D 45 D3 17 B2 6F F9 C0
AC AD F7 BD 36 05 B4 54 69 F0 30 79 4F 41 40 65
E8 B9 F7 F0 5E 22 7D 1D 17 93 EF 1E 50 2E 41 B4
45 6D B7 09 A6 67 48 F7 44 4F BB 33 16 03 91 CE
```

Tag:

```
1A 71 F3 76 B4 C5 D2 FD 61 95 D4 84 CD 11 0E 4D
61 A8 03 2F 11 C9 00 9E BF 9D F9 6A F7 52 D2 CD
```

Appendix B

Ascon test vectors

Table B.1: Inputs for Ascon128

Ascon 128		
Var	Data	Length
K	00 01 ... 0E 0F	16
N	20 21 ... 2E 2F	1P6
A	00 01 ... 7E 7F	32
M	00 01 ... 7E 7F	32

Given the input values in Table B.1, the values of the internal state and full AEAD computations are given on the next sections.

B.1 Ascon128

B.1.1 Sponge states

In this section, we show the intermediate states of the processing of data given in Table B.1. Two representations are given, the first being the 64-bit contents of the registers containing the sponge $(x_0 \cdots x_4)$. The second representation is the byte stream, taking into account the big-endian nature of Ascon.

State after initialization

Sponge registers (64-bit):

$$x_0 = \texttt{CE2216A1AB6D5D35}$$
$$x_1 = \texttt{75C69BB8FF3B7667}$$
$$x_2 = \texttt{65571B3F35C46164}$$
$$x_3 = \texttt{00E8BC75DD6CB55A}$$
$$x_4 = \texttt{3B0E5E1E4C7FBE72}$$

Byte representation:

```
35 5D 6D AB A1 16 22 CE 67 76 3B FF B8 9B C6 75
64 61 C4 35 3F 1B 57 65 5A B5 6C DD 75 BC E8 00
72 BE 7F 4C 1E 5E 0E 3B
```

State after additional data processing

Sponge registers (64-bit):

$$x_0 = \texttt{2608D8471412873D}$$
$$x_1 = \texttt{4B168A364AC530F5}$$
$$x_2 = \texttt{E75CED7AD5F522E0}$$
$$x_3 = \texttt{32C364DBE0157218}$$
$$x_4 = \texttt{7862D4BF24BE3B62}$$

Byte representation:

```
3D 87 12 14 47 D8 08 26 F5 30 C5 4A 36 8A 16 4B
E0 22 F5 D5 7A ED 5C E7 18 72 15 E0 DB 64 C3 32
62 3B BE 24 BF D4 62 78
```

State after plaintext data processing

Sponge registers (64-bit):

$$x_0 = \texttt{B10D8442E8CAD002}$$
$$x_1 = \texttt{9AB6FC946C6E5940}$$
$$x_2 = \texttt{FF87A048C3F7B306}$$
$$x_3 = \texttt{3833BBCF92DDF237}$$
$$x_4 = \texttt{8CA249B696C66170}$$

Byte representation:

```
02 D0 CA E8 42 84 0D B1 40 59 6E 6C 94 FC B6 9A
06 B3 F7 C3 48 A0 87 FF 37 F2 DD 92 CF BB 33 38
70 61 C6 96 B6 49 A2 8C
```

State after tag generation

Sponge registers (64-bit):

$$x_0 = \texttt{232BC5A5B01F1AF8}$$
$$x_1 = \texttt{5C07B505E9FB7AD3}$$
$$x_2 = \texttt{1111085EE3DF8933}$$
$$x_3 = \texttt{484751CAADDE9EF2}$$
$$x_4 = \texttt{BA3BBEEA1ED53F1E}$$

Byte representation:

```
F8 1A 1F B0 A5 C5 2B 23 D3 7A FB E9 05 B5 07 5C
33 89 DF E3 5E 08 11 11 F2 9E DE AD CA 51 47 48
1E 3F D5 1E EA BE 3B BA
```

B.1.2 Full AEAD results

Ciphertext:

```
26 09 DA 44 10 17 81 3A 3F 0F B2 0D 9B 0D E8 9C
34 2F 21 81 D5 26 5D 22 8C 8F 2F 07 66 04 77 5F
60 53 7A 29 DC 64 E7 01 C8 B3 39 AA 61 1E FE DE
8C 7F A4 52 04 91 E6 D3 3F 30 73 95 1B FF 0D 31
D0 88 85 D4 F6 35 F9 22 3B EA AA 2B FE 6E 15 12
20 31 58 F9 3C 48 A3 FC 23 7B F6 E8 44 E0 03 B4
E2 1C 43 3A 3F A6 09 91 E0 9E 54 44 D0 CC 05 42
2A EB 8E B5 4B 62 13 AC 96 9C 6F 11 67 DE 92 AE
```

Tag:

```
48 47 51 CA AD DE 9E F2 BA 3B BE EA 1E D5 3F 1E
```

B.2 Ascon128a

B.2.1 Sponge states

In this section, we show the intermediate states of the processing of data given in Table B.1. Two representations are given, the first being the 64-bit contents of the registers containing the sponge $(x_0 \cdots x_4)$. The second representation is the byte stream, taking into account the big-endian nature of Ascon.

State after initialization

Sponge registers (64-bit):

$$x_0 = \texttt{E7429E34CC44D82C}$$
$$x_1 = \texttt{FDBE752EEE81EA4F}$$
$$x_2 = \texttt{8CD33290A4E70044}$$
$$x_3 = \texttt{A16477EAE2FDAD81}$$
$$x_4 = \texttt{D85139D0A93DC5AA}$$

Byte representation:

```
2C D8 44 CC 34 9E 42 E7 4F EA 81 EE 2E 75 BE FD
44 00 E7 A4 90 32 D3 8C 81 AD FD E2 EA 77 64 A1
AA C5 3D A9 D0 39 51 D8
```

State after additional data processing

Sponge registers (64-bit):

$$x_0 = \texttt{73CDC2A420275096}$$
$$x_1 = \texttt{06C1BA39E833DAE0}$$
$$x_2 = \texttt{6724824C1B2D47C8}$$
$$x_3 = \texttt{1924A379931FE37D}$$
$$x_4 = \texttt{294CC6C1657BEA5C}$$

Byte representation:

```
96 50 27 20 A4 C2 CD 73 E0 DA 33 E8 39 BA C1 06
C8 47 2D 1B 4C 82 24 67 7D E3 1F 93 79 A3 24 19
5C EA 7B 65 C1 C6 4C 29
```

State after plaintext data processing

Sponge registers (64-bit):

$$x_0 = \texttt{60B7645344108008}$$
$$x_1 = \texttt{2C6FF01EDF18C653}$$
$$x_2 = \texttt{671190B4AF96F078}$$
$$x_3 = \texttt{DDC17CABDB1A13F8}$$
$$x_4 = \texttt{56F805E5A956926C}$$

Byte representation:

```
08 80 10 44 53 64 B7 60 53 C6 18 DF 1E F0 6F 2C
78 F0 96 AF B4 90 11 67 F8 13 1A DB AB 7C C1 DD
6C 92 56 A9 E5 05 F8 56
```

State after tag generation

Sponge registers (64-bit):

$$x_0 = \texttt{5ED80E70E9ADB5E9}$$
$$x_1 = \texttt{D0F3067A4EF24B9D}$$
$$x_2 = \texttt{96AA97679295684F}$$
$$x_3 = \texttt{86EF4786ADCE94BB}$$
$$x_4 = \texttt{57712DFA87406CAB}$$

Byte representation:

```
E9 B5 AD E9 70 0E D8 5E 9D 4B F2 4E 7A 06 F3 D0
4F 68 95 92 67 97 AA 96 BB 94 CE AD 86 47 EF 86
AB 6C 40 87 FA 2D 71 57
```

B.2.2 Full AEAD results

Ciphertext:

```
73 CC C0 A7 24 22 56 91 0E C8 B0 32 E4 3E D4 EF
17 8D 8B 2D A6 BC 73 71 20 41 B4 2D DD D2 AC 5A
75 EC 46 46 26 5D 50 30 A7 CC E7 A7 58 51 CF 1E
10 9A A4 AB B5 40 D7 45 0B 76 DB 6B 23 4F 15 C6
0B 24 D7 85 E9 21 CD 4E 3A DF BD A7 53 2C EE 82
9B 34 E4 B1 1A D9 71 A8 C7 00 2D D6 77 20 D0 2A
C5 F6 CD 95 D1 8E 41 1D A9 AA AD 3E 06 A7 B7 0F
62 4E 20 25 AF 62 3B B4 A3 B6 24 78 5F 67 03 DC
```

Tag:

```
86 EF 47 86 AD CE 94 BB 57 71 2D FA 87 40 6C AB
```

Appendix C

C code for benchmarking

C.1 Kernel modules

These kernel modules are used to enable the performance counters on ARM processors, used by the benchmark code to calculate elapsed cycles. Algorithm 43 is used for ARM-V7a processors, and Algorithm 44 is used for AArch64 compatible processors.

Algorithm 43 Kernel module for enabling counter on ARM7a

```
#include <linux/module.h>
#include <linux/kernel.h>
MODULE_LICENSE ("Dual BSD/GPL");

#define DEVICE_NAME "enableccnt"

static void
enableccnt_init (void *data)
{
  printk (KERN_INFO DEVICE_NAME " starting\n");
  asm volatile ("mcr p15, 0, %0, c9, c14, 0"::"r" (1));
//  return 0;
}

static void
enableccnt_exit (void *data)
{
  asm volatile ("mcr p15, 0, %0, c9, c14, 0"::"r" (0));
  printk (KERN_INFO DEVICE_NAME " stopping\n");
}
static int
enableccnt_init_counters (void)
{
  on_each_cpu(enableccnt_init, NULL, 1);
  return 0;
}
```

```
27
28 static void
29 enableccnt_exit_counters (void)
30 {
31   on_each_cpu(enableccnt_exit, NULL, 1);
32 }
33
34 module_init (enableccnt_init_counters);
35 module_exit (enableccnt_exit_counters);
```

Algorithm 44 Kernel module for enabling counter on AArch64 processors

```
1  static inline void
2  enable_pmu(uint32_t evtCount)
3  {
4  #if defined(__GNUC__) && defined __aarch64__
5          evtCount &= ARMV8_PMEVTYPER_EVTCOUNT_MASK;
6          asm volatile("isb");
7          /* Just use counter 0 */
8          asm volatile("msr pmevtyper0_el0, %0" : : "r" (evtCount))
               ;
9          /*   Performance Monitors Count Enable Set register bit
               30:1
10         disable, 31,1 enable */
11         uint32_t r = 0;
12
13         asm volatile("mrs %0, pmcntenset_el0" : "=r" (r));
14         asm volatile("msr pmcntenset_el0, %0" : : "r" (r|1));
15 #else
16 #error Unsupported architecture/compiler!
17 #endif
18 }
19
20 static inline void
21 disable_pmu(uint32_t evtCount)
22 {
23 #if defined(__GNUC__) && defined __aarch64__
24         /*   Performance Monitors Count Enable Set register:
               clear bit 0 */
25         uint32_t r = 0;
26
27         asm volatile("mrs %0, pmcntenset_el0" : "=r" (r));
28         asm volatile("msr pmcntenset_el0, %0" : : "r" (r&&0
               xfffffffe));
29 #else
30 #error Unsupported architecture/compiler!
31 #endif
```

```
}
```

C.2 Cycle counter

The code shown in Algorithm 45 returns the value stored in the ARM performance cycle counter.

Algorithm 45 Code for returning the cycle counter in ARM processors

```
static inline uint32_t cycles(void) {
#if defined(__GNUC__)
        uint32_t r = 0;
#if  defined __aarch64__
        asm ("mrs %0, pmccntr_el0" : "=r" (r));
#elif defined(__ARM_ARCH_7A__)
        asm ("mrc p15, 0, %0, c9, c13, 0" : "=r"(r) );
#else
#error Unsupported architecture/compiler!
#endif
        return r;
#endif
}
```

The code shown in Algorithm 46 returns the value stored in cycle counter of x86 processors.

Algorithm 46 Code for returning the cycle counter in ARM processors

```
unsigned long cycles(void) {
        unsigned int hi, lo;
        asm (
                "cpuid\n\t"/*serialize*/
                "rdtsc\n\t"/*read the clock*/
                "mov %%edx, %0\n\t"
                "mov %%eax, %1\n\t"
                : "=r" (hi), "=r" (lo):: "%rax", "%rbx", "%rcx",
                   "%rdx"
        );
        return ((unsigned long long) lo) | (((unsigned long long)
            hi) << 32);
}
```

The code shown in Algorithm 47 is a wrapper for these functions, to facilitate code benchmarking.

Algorithm 47 Wrapper for cycle counter

```
#define BENCH_PRINT  printf("%lu cycles\r\n", bench_total());

#include <time.h>
static clock_t before, after, total;

#define BENCH_ONCE(LABEL, FUNCTION)                                    \
    printf("BENCH: " LABEL " = ");                                     \
    total=0;                                                           \
    before = cycles();                                                 \
    FUNCTION;                                                          \
    after = cycles();                                                  \
    result = (after - before);                                         \
    printf("%lu cycles\r\n", result);
```

Appendix D

Code profilling with Perf

Perf is a set of performance analysis for Linux, that cover both hardware and software performance counters.

D.1 Use

Simple code profiling can be easily done in Perf, with little to no modification of the original code. For better results, one should avoid function inlining, and compile the code with debug information.

To record execution traces on Perf, use the command:

`$ perf-record ./program_to_trace`

An interactive interface for exploring the trace can then be called by executing the following command in the same directory:

`$ perf report`

Alternatively, a static text report can be generated with the command:

`$ perf report -stdio`

By default, the number of clock cycles is used as the event to trace. More events, such as branch mispredictions and cache access can be used instead, passing them with the switch `-e`. A list of the supported events can be retrieved via the command `$ perf list`.

For a larger measurement resolution, the switch `-F` can be used to specify the frequency, in Hertz, of sampling.

Perf is a very powerful tool, and while this work does not aim to list all the characteristics of the tool and scenarios where it can be used, two good sources of information are given below:

`https://perf.wiki.kernel.org/index.php/Tutorial`

`http://www.brendangregg.com/perf.html`

Alternatively, archive links for these pages are given below:

`https://web.archive.org/web/20180225014548/https://perf.wiki.kernel.org/index.php/Main_Page`

`https://web.archive.org/web/20180724055416/http://www.brendangregg.com/perf.html`

Appendix E

NORX full result tables

In this Appendix, it is shown the full results for NORX benchmarks on the target architectures.

Table E.1: Results in cycles per byte for NORX3261 on Cortex-A7

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	60.25	70.58	58.61	59.56
256B	44.57	52.62	43.85	44.56
512B	34.17	41.85	35.50	34.23
1KiB	31.06	36.88	30.08	31.21
2KiB	27.90	32.98	27.57	28.15
4KiB	26.79	32.23	27.03	27.24
8KiB	26.19	31.47	26.23	26.60
16KiB	25.99	31.23	26.00	26.25
32KiB	25.80	31.12	25.99	26.12
64KiB	25.80	30.99	25.98	26.20
128KiB	25.86	31.07	26.03	26.34
256KiB	26.28	31.94	26.87	26.62
512KiB	27.31	33.40	28.32	27.66
1MiB	28.32	33.79	28.78	28.68

Table E.2: Results in CBP for NORX3261 on Cortex-A15

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	38.22	32.72	32.75	45.13
256B	28.44	30.99	24.86	33.39
512B	21.82	18.63	19.42	25.80
1KiB	19.32	16.47	17.40	22.86
2KiB	18.23	15.07	16.04	21.36
4KiB	17.31	14.52	15.80	20.50
8KiB	16.94	14.17	15.34	20.00

Table E.2 – Continued from previous page

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
16KiB	16.75	14.10	15.20	19.71
32KiB	16.59	14.15	15.20	19.67
64KiB	16.59	14.23	15.18	19.61
128KiB	16.60	14.24	15.17	19.58
256KiB	16.58	14.23	15.16	19.57
512KiB	16.57	14.22	15.15	19.56
1MiB	16.60	14.24	15.16	19.60

Table E.3: Results in cycles per byte for NORX3261 on Cortex-A53

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	39.62	25.13	28.34	26.35
256B	29.02	18.68	25.16	19.92
512B	22.21	14.46	16.23	15.57
1KiB	19.57	12.85	14.40	13.93
2KiB	17.86	11.80	13.21	12.84
4KiB	17.42	11.70	12.84	12.43
8KiB	16.78	11.13	12.49	12.16
16KiB	16.64	11.08	12.39	12.11
32KiB	16.54	11.08	12.31	12.05
64KiB	16.50	10.97	12.35	12.10
128KiB	16.52	10.95	12.26	12.00
256KiB	16.50	10.94	12.27	12.00
512KiB	16.50	10.97	12.38	12.07
1MiB	16.64	11.03	12.45	12.13

Table E.4: Results in cycles per byte for NORX3264 on Cortex-A7

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	147.00	173.26	137.18	167.09
256B	93.74	108.30	86.84	101.33
512B	56.85	67.98	53.58	65.48
1KiB	43.13	51.68	40.98	49.08
2KiB	34.84	42.16	33.35	40.65
4KiB	31.12	37.81	29.56	36.63
8KiB	29.00	35.61	27.91	34.12
16KiB	28.19	34.35	26.88	33.07
32KiB	27.65	33.80	26.50	32.51
64KiB	27.46	33.64	26.41	32.33

Table E.4 – Continued from previous page

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128KiB	27.41	33.59	26.30	32.23
256KiB	28.46	33.74	26.50	32.28
512KiB	29.75	34.54	27.35	33.27
1MiB	30.18	35.87	28.57	34.61

Table E.5: Results in cycles per byte for NORX3264 on Cortex-A15

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	105.70	99.04	83.04	99.29
256B	56.97	57.39	49.90	58.68
512B	36.80	33.41	32.43	37.18
1KiB	27.52	25.04	24.42	27.78
2KiB	23.12	20.99	20.65	22.69
4KiB	20.41	18.67	18.40	20.37
8KiB	19.27	17.22	17.31	19.65
16KiB	18.68	16.89	16.61	18.79
32KiB	18.22	16.81	16.42	18.44
64KiB	18.06	16.71	16.33	18.31
128KiB	17.98	16.64	16.26	18.25
256KiB	17.96	16.62	16.25	18.21
512KiB	17.96	16.61	16.22	18.20
1MiB	18.59	16.61	16.23	18.19

Table E.6: Results in cycles per byte for NORX3264 on Cortex-A53

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	233.77	60.99	66.47	74.92
256B	66.59	37.20	40.59	51.13
512B	42.55	24.21	26.43	36.97
1KiB	31.47	18.25	21.25	31.01
2KiB	25.64	15.00	16.42	27.98
4KiB	22.69	13.75	14.80	26.08
8KiB	21.23	12.75	13.94	25.16
16KiB	20.61	12.35	13.53	24.80
32KiB	20.25	12.17	13.32	24.56
64KiB	20.07	12.10	13.23	24.48
128KiB	19.98	12.04	13.17	24.42
256KiB	20.08	12.02	13.15	24.40
512KiB	20.32	12.08	13.17	24.45

Table E.6 – Continued from previous page

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
1MiB	20.38	12.15	13.17	24.47

Table E.7: Results in cycles per byte for NORX6461 on Cortex-A7

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	199.86	237.81	212.02	87.23
256B	121.01	134.45	127.07	50.93
512B	87.63	96.47	89.00	36.23
1KiB	65.20	71.19	65.95	27.13
2KiB	57.96	61.89	57.06	23.84
4KiB	52.55	54.87	50.98	21.36
8KiB	50.52	52.47	48.70	20.53
16KiB	49.20	50.82	47.39	20.06
32KiB	48.77	50.26	46.88	19.75
64KiB	48.55	50.12	46.65	19.66
128KiB	48.53	50.09	46.65	19.67
256KiB	49.01	50.37	46.87	19.72
512KiB	50.15	51.60	48.08	20.55
1MiB	9.94	11.40	8.05	21.96

Table E.8: Results in cycles per byte for NORX6461 on Cortex-A15

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	130.64	115.35	120.12	54.91
256B	78.01	67.79	70.64	32.13
512B	57.83	48.47	54.71	21.96
1KiB	44.65	36.53	38.46	16.26
2KiB	39.58	31.74	33.54	14.03
4KiB	36.89	29.31	30.72	12.64
8KiB	35.47	27.80	29.39	12.36
16KiB	34.64	27.05	28.67	11.94
32KiB	34.42	26.92	28.51	11.88
64KiB	34.19	26.84	28.39	11.78
128KiB	34.14	26.76	28.33	11.76
256KiB	34.08	26.72	28.29	11.73
512KiB	34.05	26.70	28.26	11.72
1MiB	34.04	26.70	28.26	11.71

Table E.9: Results in cycles per byte for NORX6461 on Cortex-A53

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	49.84	32.43	45.96	47.02
256B	28.78	18.44	24.20	31.16
512B	20.15	12.46	15.74	19.92
1KiB	15.56	8.95	10.85	15.02
2KiB	12.70	7.45	9.07	13.01
4KiB	11.38	6.57	7.59	12.05
8KiB	10.83	6.19	7.09	11.35
16KiB	10.53	6.00	6.78	11.07
32KiB	10.39	5.90	6.72	11.00
64KiB	10.34	5.86	6.60	10.94
128KiB	10.31	5.84	6.58	10.90
256KiB	10.29	5.83	6.56	10.90
512KiB	10.31	5.87	6.61	10.96
1MiB	10.37	6.03	6.71	10.96

Table E.10: Results in cycles per byte for NORX6464 on Cortex-A7

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	379.89	504.84	440.57	379.29
256B	203.57	271.13	237.47	204.61
512B	123.18	163.12	143.44	123.45
1KiB	79.71	104.29	92.61	80.00
2KiB	59.46	77.14	69.12	59.75
4KiB	48.62	62.38	56.57	48.80
8KiB	43.65	55.59	50.71	43.77
16KiB	40.82	51.97	47.55	41.13
32KiB	39.70	50.35	46.16	39.81
64KiB	39.11	49.50	45.45	39.18
128KiB	38.82	49.05	45.12	38.89
256KiB	38.78	48.85	44.98	38.75
512KiB	7.49	17.67	13.84	7.45
1MiB	8.70	3.04	14.95	8.79

Table E.11: Results in cycles per byte for NORX6464 on Cortex-A15

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	223.03	245.52	283.33	222.88

Table E.11 – Continued from previous page

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
256B	120.01	130.51	151.44	119.96
512B	72.51	78.94	91.95	72.14
1KiB	47.06	50.94	57.73	46.88
2KiB	34.97	38.40	42.33	35.17
4KiB	28.57	31.33	34.28	28.75
8KiB	25.66	28.08	30.41	25.79
16KiB	24.11	26.35	28.37	24.39
32KiB	23.37	25.66	27.63	23.64
64KiB	23.04	25.37	27.10	23.23
128KiB	22.87	25.11	26.90	23.09
256KiB	22.77	25.00	26.73	22.94
512KiB	22.73	24.95	26.60	22.90
1MiB	6.71	8.93	10.57	6.90

Table E.12: Results in cycles per byte for NORX6464 on Cortex-A53

Input Len	Ref	4xpipe	2xpipe	Ref. NEON
128B	105.83	61.56	94.80	49.10
256B	57.63	33.93	51.01	27.94
512B	34.46	19.82	30.89	18.91
1KiB	22.27	12.67	19.92	13.61
2KiB	16.62	9.42	14.88	11.35
4KiB	13.60	7.58	12.16	10.02
8KiB	12.15	6.77	10.88	9.48
16KiB	11.50	6.34	10.21	9.14
32KiB	11.05	6.20	9.91	9.01
64KiB	10.86	6.09	9.75	8.93
128KiB	10.77	6.04	9.66	8.91
256KiB	10.74	6.09	9.62	8.88
512KiB	10.75	6.12	9.62	8.92
1MiB	10.75	6.02	9.69	8.92

Appendix F

Ascon full result tables

In this Appendix, it is shown the full results for Ascon benchmarks on the target architecture.

Table F.1: Results in cycles per byte for ASCON128 on Cortex-A15

Input Len	Ascon128 ref.	Own code	NEON 64-bit	NEON 128-bit	NEON M. Msg.
128B	109.31	84.48	68.62	64.01	53.34
256B	98.75	74.88	62.52	55.47	49.14
512B	94.30	72.00	58.55	52.19	46.27
1KiB	91.92	68.09	56.77	50.96	43.92
2KiB	90.61	66.91	55.79	50.10	43.07
4KiB	89.83	66.51	55.41	49.69	42.46
8KiB	89.72	66.10	55.15	49.46	42.40
16KiB	90.35	66.00	55.04	49.44	43.05
32KiB	89.67	65.98	55.05	49.32	42.80
64KiB	90.14	66.02	55.10	49.32	43.44

Table F.2: Results in cycles per byte for ASCON128 on Cortex-A7

Input Len	Ascon128 ref.	Own code	NEON 64-bit	NEON 128-bit	NEON M. Msg.
128B	214.48	166.33	152.93	117.26	104.37
256B	194.77	146.47	137.03	105.33	93.10
512B	183.89	137.79	127.85	98.01	87.68
1KiB	178.85	133.24	123.98	94.78	84.78
2KiB	176.43	130.63	121.97	92.96	83.42
4KiB	175.82	129.54	120.95	92.11	82.76
8KiB	175.46	128.87	120.41	91.68	82.41
16KiB	174.44	128.62	120.16	91.47	82.30
32KiB	174.18	128.52	120.08	91.46	82.22
64KiB	174.10	128.50	120.11	91.45	82.19

Table F.3: Results in cycles per byte for ASCON128 on Cortex-A53

Input Len	Ascon128 ref.	Own code	NEON 64-bit	NEON 128-bit	NEON M. Msg.
128B	35.99	17.75	19.98	51.29	29.10
256B	32.35	15.33	17.40	45.91	25.77
512B	30.55	14.09	16.15	44.11	24.36
1KiB	29.66	13.48	15.53	41.96	23.33
2KiB	29.13	13.18	15.21	41.55	22.88
4KiB	28.95	13.06	15.09	40.86	22.67
8KiB	28.81	12.96	15.00	41.28	22.55
16KiB	28.90	12.92	14.98	40.97	22.58
32KiB	28.85	12.92	14.97	40.65	22.50
64KiB	28.83	12.90	14.96	40.62	22.52

Table F.4: Results in cycles per byte for ASCON128a on Cortex-A15

Input Len	Ascon128a ref.	Own code	NEON 64-bit	NEON 128-bit
128B	77.87	60.47	49.94	44.39
256B	68.14	51.39	42.45	36.00
512B	62.56	47.30	39.11	32.84
1KiB	59.93	44.91	36.99	31.68
2KiB	58.59	43.68	36.23	30.68
4KiB	57.87	43.10	35.66	30.28
8KiB	57.50	42.68	35.36	30.03
16KiB	57.60	42.49	35.24	29.96
32KiB	57.70	42.62	35.48	30.10
64KiB	57.62	42.66	35.60	30.29

Table F.5: Results in cycles per byte for ASCON128a on Cortex-A7

Input Len	Ascon128a ref.	Own code	NEON 64-bit	NEON 128-bit
128B	151.88	127.77	113.48	92.37
256B	132.47	108.97	97.19	77.35
512B	122.12	100.10	89.50	70.66
1KiB	116.36	95.49	85.33	67.62
2KiB	113.98	93.34	83.52	65.72
4KiB	112.84	92.15	82.43	64.75
8KiB	112.09	91.53	81.98	64.42
16KiB	112.00	91.28	81.76	64.06
32KiB	111.80	91.18	81.66	64.13

Table F.5 – Continued from previous page

Input Len	Ascon128a ref.	Own code	NEON 64-bit	NEON 128-bit
64KiB	111.75	91.19	81.62	64.13

Table F.6: Results in cycles per byte for ASCON128a on Cortex-A53

Input Len	Ascon128a ref.	Own code	NEON 64-bit	NEON 128-bit
128B	29.50	15.89	15.80	40.48
256B	25.89	13.40	13.40	32.95
512B	24.07	12.19	12.16	30.35
1KiB	23.18	11.56	11.57	29.24
2KiB	22.76	11.27	11.35	28.51
4KiB	22.52	11.12	11.11	28.13
8KiB	22.34	11.03	11.06	27.96
16KiB	22.47	10.98	11.01	27.89
32KiB	22.45	10.94	10.92	27.88
64KiB	22.28	10.79	10.72	27.63

Appendix G

Object dump of Ascon's LBOX.

Algorithm 48 Dump of the NEON code for Ascon LBOX, generated without compiler optimizations in order to force the use of NEON registers. Immediate values are the ones applied to x_0.

```
0000000000000b10 <lbox>:
void lbox(uint64x1_t *x){
     b10:       f8190ffe        str     x30, [sp, #-112]!
     b14:       f9000fe0        str     x0, [sp, #24]
     b18:       900000a0        adrp    x0, 14000 <__FRAME_END__
        +0xf428>
     b1c:       f947e000        ldr     x0, [x0, #4032]
     b20:       f9400001        ldr     x1, [x0]
     b24:       f90037e1        str     x1, [sp, #104]
     b28:       d2800001        mov     x1, #0x0
                                // #0
        ;#define GGG(x,n) (veor_u64(vshr_n_u64((x),(n)),
           vshl_n_u64((x),(64-n))))
        ;*x ^= GGG(*x, 28) ^ GGG(*x, 29);
     b2c:       f9400fe0        ldr     x0, [sp, #24]
     b30:       f9400000        ldr     x0, [x0]
     b34:       f90023e0        str     x0, [sp, #64]
  ; return (uint64x1_t) {__builtin_aarch64_lshr_simddi_uus ( __a
     [0], __b)};
     b38:       f94023e0        ldr     x0, [sp, #64]
     b3c:       d35cfc00        lsr     x0, x0, #28
     b40:       d2800001        mov     x1, #0x0
                                // #0
     b44:       aa0003e1        mov     x1, x0
     b48:       aa0103e0        mov     x0, x1
     b4c:       aa0003e1        mov     x1, x0
     b50:       f9400fe0        ldr     x0, [sp, #24]
     b54:       f9400000        ldr     x0, [x0]
     b58:       f9001fe0        str     x0, [sp, #56]
```

```
25    ; return (uint64x1_t) {__builtin_aarch64_ashldi ((int64_t) __a
          [0], __b)};
26       b5c:       f9401fe0        ldr     x0, [sp, #56]
27       b60:       aa0003e2        mov     x2, x0
28       b64:       52800480        mov     w0, #0x24
                                   // #36
29       b68:       9ac02040        lsl     x0, x2, x0
30       b6c:       aa0003e2        mov     x2, x0
31       b70:       d2800000        mov     x0, #0x0
                                   // #0
32       b74:       aa0203e0        mov     x0, x2
33       b78:       f9002fe1        str     x1, [sp, #88]
34       b7c:       f90033e0        str     x0, [sp, #96]
35    ; return __a ^ __b;
36       b80:       f9402fe1        ldr     x1, [sp, #88]
37       b84:       f94033e0        ldr     x0, [sp, #96]
38       b88:       ca000020        eor     x0, x1, x0
39       b8c:       d2800001        mov     x1, #0x0
                                   // #0
40       b90:       aa0003e1        mov     x1, x0
41       b94:       aa0103e0        mov     x0, x1
42       b98:       aa0003e3        mov     x3, x0
43       b9c:       f9400fe0        ldr     x0, [sp, #24]
44       ba0:       f9400000        ldr     x0, [x0]
45       ba4:       f9001be0        str     x0, [sp, #48]
46    ; return (uint64x1_t) {__builtin_aarch64_lshr_simddi_uus ( __a
          [0], __b)};
47       ba8:       f9401be0        ldr     x0, [sp, #48]
48       bac:       d35dfc00        lsr     x0, x0, #29
49       bb0:       d2800001        mov     x1, #0x0
                                   // #0
50       bb4:       aa0003e1        mov     x1, x0
51       bb8:       aa0103e0        mov     x0, x1
52       bbc:       aa0003e1        mov     x1, x0
53       bc0:       f9400fe0        ldr     x0, [sp, #24]
54       bc4:       f9400000        ldr     x0, [x0]
55       bc8:       f90017e0        str     x0, [sp, #40]
56    ; return (uint64x1_t) {__builtin_aarch64_ashldi ((int64_t) __a
          [0], __b)};
57       bcc:       f94017e0        ldr     x0, [sp, #40]
58       bd0:       aa0003e2        mov     x2, x0
59       bd4:       52800460        mov     w0, #0x23
                                   // #35
60       bd8:       9ac02040        lsl     x0, x2, x0
61       bdc:       aa0003e2        mov     x2, x0
62       be0:       d2800000        mov     x0, #0x0
                                   // #0
```

```
63        be4:        aa0203e0        mov     x0, x2
64        be8:        f90027e1        str     x1, [sp, #72]
65        bec:        f9002be0        str     x0, [sp, #80]
66    ; return __a ^ __b;
67        bf0:        f94027e1        ldr     x1, [sp, #72]
68        bf4:        f9402be0        ldr     x0, [sp, #80]
69        bf8:        ca000020        eor     x0, x1, x0
70        bfc:        d2800001        mov     x1, #0x0
                                    // #0
71        c00:        aa0003e1        mov     x1, x0
72        c04:        aa0103e0        mov     x0, x1
73        c08:        aa0303e1        mov     x1, x3
74        c0c:        ca000020        eor     x0, x1, x0
75        c10:        f9400fe1        ldr     x1, [sp, #24]
76        c14:        f9400021        ldr     x1, [x1]
77        c18:        ca010000        eor     x0, x0, x1
78        c1c:        d2800001        mov     x1, #0x0
                                    // #0
79        c20:        aa0003e1        mov     x1, x0
80        c24:        f9400fe0        ldr     x0, [sp, #24]
81        c28:        f9000001        str     x1, [x0]
82    ; }
83        c2c:        d503201f        nop
84        c30:        900000a0        adrp    x0, 14000 <__FRAME_END__
            +0xf428>
85        c34:        f947e000        ldr     x0, [x0, #4032]
86        c38:        f94037e1        ldr     x1, [sp, #104]
87        c3c:        f9400000        ldr     x0, [x0]
88        c40:        ca000020        eor     x0, x1, x0
89        c44:        f100001f        cmp     x0, #0x0
90        c48:        54000040        b.eq    c50 <lbox+0x140>  //
            b.none
91        c4c:        97ffff39        bl      930 <__stack_chk_fail@plt
            >
92        c50:        f84707fe        ldr     x30, [sp], #112
93        c54:        d65f03c0        ret
```