TUM School of Computation, Information and Technology
Technische Universität München

Secure Lightweight Authenticated Encryption for Critical Infrastructures in the Internet of Things

Sebastian Renner

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:

Prof. Dr. Sebastian Steinhorst

Prüfer der Dissertation:

1. Prof. Dr.-Ing. Georg Sigl
2. Prof. Dr. Jürgen Mottok

Die Dissertation wurde am 29.03.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 03.08.2023 angenommen.

Abstract

Due to the digital transformation, networking is also steadily increasing. More and more devices can be controlled remotely or connected to the Internet. This development has many advantages, but also brings dangers. Due to the additional interfaces and functions, the attack surface of the systems increases. Since smart sensors, for example, are also used in critical infrastructures, these devices and their communications have to be specially secured. The standardization authority NIST therefore launched a project in 2018 with the aim of converting a new cryptographic algorithm into a national standard. This lightweight encryption method should be particularly suitable for securing embedded systems, which have little computing power.

In this thesis, the algorithms available for selection are evaluated with respect to various factors. To this end, a test environment that was developed specifically for this purpose is first presented. Then, the test platforms and test cases, which are used to evaluate the performance of the algorithms, are shown. Lastly, the results of the tests are discussed and interpreted.

In the second part, hardening measures that protect cryptographic methods against implementation side-channel attacks are introduced. Different hardening strategies are discussed here. It is analyzed which performance penalties originate from the additional protection measures. Furthermore, new strategies to efficiently protect a specific group of algorithms are introduced and applied. Then, we compare the results of our hardening techniques to related work, and investigate how much hardening affects the performance of different lightweight algorithms.

Finally, we evaluate the novel cryptographic methods in an industry-oriented use case. For this purpose, a test environment that represents a possible use case for lightweight cryptography is built. In this scenario, the use of different encryption schemes in a standardized wireless protocol is tested. In particular, the performance of the new algorithms is compared to the current industry standard (AES-GCM) and it is analyzed whether they bring significant advantages in the specific use case.

In general, this work summarizes our contributions to the NIST lightweight cryptography standardization process. From starting with the design and implementation of the environment for performance testing, through the analysis of (new) different hardening measures and their effects, to the evaluation of the algorithms in a practical use case, we try to present a detailed picture of the candidates and their applicability.

Kurzfassung

Durch den digitalen Wandel nimmt auch die Vernetzung stetig zu. Immer mehr Geräte können aus der Ferne gesteuert oder mit dem Internet verbunden werden. Diese Entwicklung hat viele Vorteile, bringt aber auch Gefahren mit sich. Aufgrund der zusätzlichen Schnittstellen und Funktionen vergrößert sich die Angriffsfläche der Systeme. Da z.B. intelligente Sensoren auch in kritischen Infrastrukturen eingesetzt werden, müssen diese Geräte und deren Kommunikation besonders abgesichert werden. Die Standardisierungsbehörde NIST hat daher 2018 ein Projekt ausgerufen, das zum Ziel hat, einen neuen kryptographischen Algorithmus in einen nationalen Standard zu überführen. Dieses leichtgewichtige Verschlüsselungsverfahren soll besonders zur Absicherung eingebetteter Systeme geeignet sein, welche über wenig Rechenleistung verfügen.

In dieser Arbeit werden die zur Auswahl stehenden Algorithmen hinsichtlich verschiedener Faktoren evaluiert. Dazu wird zunächst eine Testumgebung vorgestellt, die speziell für diesen Zweck entwickelt wurde. Anschließend werden die Testplattformen und Testfälle dargestellt, mit Hilfe derer die Performanz der Verfahren beurteilt wird. Dann werden die Ergebnisse der Tests diskutiert und eingeordnet.

Im zweiten Teil werden Härtungsmaßnahmen eingeführt, die kryptographische Verfahren zusätzlich gegenüber Seitenkanalangriffen auf deren Implementierung schützen. Hier werden unterschiedliche Strategien zur Härtung diskutiert. Es wird analysiert, welche Einbußen hinsichtlich der Performanz aus den zusätzlichen Schutzmaßnahmen resultieren. Weiterhin werden neue Strategien zur effizienten Absicherung einer bestimmten Algorithmengruppe eingeführt und angewendet. Dann vergleichen wir die Ergebnisse unserer Härtungsverfahren mit verwandten Arbeiten und untersuchen, wie stark sich die Härtung auf die Performanz unterschiedlicher leichtgewichtiger Algorithmen auswirkt.

Abschließend werden die neuartigen kryptographischen Verfahren in einem industrienahen Anwendungsfall evaluiert. Dazu wird eine Testumgebung aufgebaut, die einem möglichen Anwendungsfall für leichtgewichtige Kryptographie entspricht. In diesem Szenario wird die Verwendung unterschiedlicher Verschlüsselungsverfahren in einem standardisiertem Funkprotokoll erprobt. Es wird insbesondere die Performanz der neuen Algorithmen mit dem aktuellen Industriestandard (AES-GCM) verglichen und analysiert, ob diese in dem konkreten Anwendungsfall signifikante Vorteile mit sich bringen.

Im Allgemeinen fasst diese Arbeit unsere Beiträge zum NIST Standardisierungsverfahren für leichtgewichtige Kryptographie zusammen. Beginnend mit dem Design und der Implementierung der Umgebebung für Performanztests, über die Analyse (neuer) unterschiedlicher Härtungsmaßnahmen und deren Wirkung, hin zu der Evaluation der Algorithmen in einem praxisnahem Anwendungsfall, wird versucht ein detailliertes Bild der Kandidaten und deren Anwendbarkeit darzustellen.

Contents

List of Figures

List of Tables

Acronyms

AD	Associated Data.
AEAD	Authenticated Encryption with Associated Data.
AES	Advanced Encryption Standard.
AFL	Authentication and Fragmentation Layer.
API	Application Programming Interface.
APL	Application Layer.
ARX	Add-Rotate-`XOR`.
CAESAR	Competition for Authenticated Encryption: Security, Applicability, and Robustness.
CBC	Cipher Block Chaining.
CFB	Ciphertext FeedBack.
COFB	COmbined FeedBack.
CPA	Correlation Power Analysis.
CRC	Cyclic Redundancy Check.
CSA	Carry-Save Adder.
CTR	Counter Mode.
DDoS	Distributed Denial of Service.
DES	Data Encryption Standard.
DIF	Data Information Field.
DLL	Data Link Layer.
DOM	Domain-Oriented Masking.
DPA	Differential Power Analysis.
DR	Data Record.
Dual-EC-DRBG	Dual Elliptic Curve Deterministic Random Bit Generator.
ECB	Electronic Codebook Mode.
ELL	Extended Link Layer.
FIPS	Federal Information Processing Standard.
GCM	Galois Counter Mode.
IoT	Internet of Things.

IV	Initialization Vector.
KSA	Kogge-Stone Adder.
LFSR	Linear Feedback Shift Register.
LoRa	Long Range.
LSB	Least Significant Bit.
LWC	Lightweight Cryptography.
LWE	Learning with Errors.
M-Bus	Meter-Bus.
MAC	Message Authentication Code.
MAR	Memory Address Register.
MCU	Microcontroller Unit.
MDR	Memory Data Register.
MDS	Maximum Distance Separable.
MEM	Masked Even-Mansour.
MIA	Mutual Information Analysis.
MOO	Multi-Objective Optimization.
NEAT	NeuroEvolution of Augmenting Topologies.
NIST	National Institute of Standards and Technology.
NLFSR	Non-Linear Feedback Shift Register.
NSA	National Security Agency.
NSGA-II	Nondominated Sorting Genentic Algorithm II.
OCB	Offset Codebook Mode.
OFB	Output Feedback.
OTR	Offset Two Rounds.
PAEF	Parallel AEAD from a Forkcipher.
PANORAMA	PArallelisable NOnce Rotating Authenticated Encryption with Assocaited Data.
PFB	Plaintext FeedBack.
PHY	Physical Layer.
PMAC	Parallelizable Message Authentication Code.
PPA	Parallel Prefix Adder.
PQC	Post-Quantum Cryptography.
PRNG	Pseudo-Random Number Generator.
RAM	Random Access Memory.
RNG	Random Number Generator.
ROM	Read Only Memory.

SAEF	Sequential AEAD from a Forkcipher.
SCA	Side-Channel Analysis.
SCT	Synthetic Counter in Tweak mode.
SIV	Synthetic Initialization Vector.
SNR	Signal-to-Noise Ratio.
SPA	Simple Power Analysis.
SPI	Serial Peripheral Interface.
SPN	Substitution-Permutation Network.
TI	Threshold Implementation.
TPL	Transport Layer.
TWEANN	Topology and Weight Evolving Artificial Neural Network.
UART	Universal Asynchronous Receiver/Transmitter.
VIF	Value Information Field.
wM-Bus	Wireless Meter-Bus.

Acknowledgements

First, I would like to thank my supervisors Prof. Dr. Jürgen Mottok and Prof. Dr.-Ing. Georg Sigl for setting up an opportunity for me to pursue a PhD in this cooperative setting – at the Laboratory for Safe and Secure Systems at OTH Regensburg and the Chair of Security in Information Technology at TU München. Thank you for the academic guidance when needed and for allowing me to freely explore interesting research paths on my own.

Also, I would like to thank all my colleagues at the LaS³ for helping me out in various situations. Especially, I would like to mention Enrico Pozzobon, Dr. Nils Weiß, Tobias Frauenschläger and Peter Heller.

Moreover, I would like to thank my family and my girlfriend for supporting me throughout this whole journey, specifically when things were not going as planned.

Lastly, I would like to thank the Bavarian Research Institute for Digital Transformation (bidt) and the BayWISS Consortium Digitization for supporting my research financially.

This thesis was supported by

BayWISS Bayerisches Wissenschaftsforum

LaS³ Laboratory for Safe and Secure Systems a discipline of software engineering

OTH Ostbayerische Technische Hochschule Regensburg

bidt Bayerisches Forschungsinstitut für Digitale Transformation
Ein Institut der Bayerischen Akademie der Wissenschaften

1 Introduction

According to Auguste Kerckhoffs, a Dutch cryptographer living in the 19th century, "[a cipher] should not require secrecy, and it should not be a problem if it falls into enemy hands" [5]. This axiom was published in a journal article in 1883 and is now rather known as "Kerckhoffs's principle". While he referred this axiom only to secret communication in wartimes, it still applies to modern cryptography today. In essence, it means that the security goals of a cryptographic system shall be met, even if an attacker knows the details about its design principle. Therefore, the specification of any modern cipher is published and can be analyzed by any interested party.

This transparency in the design of cryptography should as well help to avoid another incident similar to the recommendation of the Dual Elliptic Curve Deterministic Random Bit Generator (Dual-EC-DRBG). In 2006, the National Institute of Standards and Technology (NIST) issued a recommendation for random number generation for cryptography, containing four different Random Number Generators (RNGs), with Dual-EC-DRBG being one of them [6]. However, already during that time, the existence of a potential backdoor integrated by the National Security Agency (NSA) and the security of the RNG in general has been heavily discussed in the research community [7, 8, 9]. In 2013, the New York Times published an article referring to the leaked internal memos of NSA contractor Edward Snowden. In the article, the author states that "the N.S.A. had inserted a back door into a 2006 standard adopted by N.I.S.T" [10]. Later in the text, it is clarified that this claim concerns the Dual-EC-DRBG. As a response to these events, NIST removed Dual-EC-DRBG from its recommendation (NIST SP 800-90A) in 2014 [11].

Kerckhoffs's principle and the issues within drafting NIST SP 800-90A motivates an as-transparent-as-possible evaluation during the selection process and standardization of a novel cryptographic primitive. In the past, NIST already asked researchers and representatives from industry to take part in the design, analysis and evaluation of cipher candidates which were later to be standardized in a Federal Information Processing Standard (FIPS). A popular example for a winner of such a competition is the Advanced Encryption Standard (AES), which has been selected and standardized by NIST in 2001 and remains one of the most prominent symmetric ciphers until today [12]. AES, which entered this NIST competition under the name Rijndael originally, has been found to be the most suitable candidate among the 15 submitted algorithms and was later renamed after the project name, Advanced Encryption Standard.

Since then, several similar-structured competitions have been held by NIST and other initiatives. From 2004 to 2008, the eSTREAM project focused on evaluating efficient stream ciphers for the use in both software and hardware applications. eSTREAM has

been carried out by the ECRYPT consortium and in the end selected a portfolio of eight algorithms [13].

In 2007, NIST initiated a project for finding a new hash algorithm as an alternative or backup for SHA-2. Five years have been filled with evaluations, analysis and preselections of the candidates, before Keccak was announced as the winner. The hash algorithm was then finally standardized in FIPS 202 [14].

Another cryptographic competition held by NIST is the Post-Quantum Cryptography (PQC) project launched in 2016. The PQC initiative aims to find alternatives to currently used asymmetric cryptography, which can withstand attacks in the presence of general quantum computers. NIST identified the algorithms for standardization in July 2022, including one cipher for public-key encryption and three algorithms for digital signing [15].

The Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) was started in 2013. The idea of this evaluation effort was to select lightweight cryptographic algorithms that support authenticated encryption. Mainly, the goal was to analyze if the submitted candidates outperform the current standard for that use case, AES in Galois Counter Mode (GCM), in certain categories. The CAESAR committee received 58 submissions initially, of which a portfolio of six ciphers has been selected as winners for three different use cases (lightweight applications, high-performance applications, defense in depth) [16].

Further research regarding cryptography for resource-constraint environments is again organized by NIST. With initiating the Lightweight Cryptography (LWC) project in 2018, NIST kicked off the future standardization process of a symmetric cipher supporting authenticated encryption. In its call for submissions, NIST asked cryptographers to hand in cryptosystems that could be used as an alternative to AES-GCM in low-performing systems and optionally are capable of hashing. Originally, 56 candidates have been admitted to round 1. After two selection rounds supported by public evaluation, now ten finalist ciphers remain in the last round 3. NIST expects to conclude the competition with the choice of a winner in early 2023 [17].

The participation of research and industry in the CAESAR and NIST LWC project shows that there is a noticeable interest in standardizing symmetric cryptography for the use in constrained environments. According to a recent report from Business Insider, the number of Internet of Things (IoT) devices will rise from 8 billion in 2019 to 41 billion in 2027 [18]. With the constant growth of IoT devices in the field, the potential attack surface also increases. What can happen if attackers find vulnerabilities in popular IoT systems has been demonstrated, e.g. with the powerful Distributed Denial of Service (DDoS) attacks guided by the so-called *mirai* botnet [19]. Hardening IoT systems also includes securing their communication according to their particular use case. Depending on the purpose of the device, confidential and authenticated data transmission can be a requirement. Especially if these systems are embedded in critical infrastructures like the smart grid or the water supply, a secure setup is of high importance. Small IoT devices like intelligent sensors can lack the computing power to carry out well-established cryptographic primitives, however, still security requirements need to be fulfilled. In these use cases LWC poses a considerable alternative to e.g. AES. Specifically, where perfor-

mance constraints meet security protection goals is where LWC could be an interesting solution. The NIST LWC project wants to analyze under which circumstances novel LWC algorithms can be of use in such scenarios and will establish a new standard to be applied in industrial developments in the future.

This thesis focuses on the evaluation of software implementations of NIST LWC candidates. With 56 round 1 submissions and only one year of initial analysis time, a fair assessment of all algorithms constitutes a big challenge. Obviously, the performance of the candidates is one of the primary interests. But also e.g. the evaluation of the security of the cipher design needs to be taken into account. Moreover, the question arises, how and if a fair performance benchmarking of both soft- and hardware implementations can be reached. In this work, we benchmark the NIST LWC candidates on the bases of later discussed metrics and test software implementations in different disciplines and use cases.

In detail, this thesis contributes to the NIST evaluation process with the following efforts:

CB1 First, we will introduce the design and setup of our new test framework, specifically created for benchmarking NIST LWC software implementations. We will explain our design choices and give an overview over our standard test cases. It is also motivated why we choose to build our own framework instead of relying on existing software.

CB2 We will discuss what challenges and issues one has to face when assessing implementations of (cryptographic) algorithms. We will then elaborate on what we think is required for a fair performance comparison of individual implementations.

CB3 Using our before explained custom framework, we will present obtained performance figures for various implementations of all NIST LWC submissions. In our specific comparisons, we will later focus on 3rd round candidates that form the final group in the ongoing competition. We compare benchmarking results in three different categories and show how the LWC algorithms perform compared to AES in our test cases. We will provide an interpretation of our results and give insights regarding different peculiarities in the test data.

CB4 We study the efficient protection of implementations against basic Side-Channel Analysis (SCA) and investigate how high the performance penalties are for selected attack countermeasures. Moreover, we present novel methods to protect a specific cipher category – Add-Rotate-`XOR` (ARX) ciphers – and compare the performance of our guarded implementations to related work and non-protected variants [20].

CB5 Lastly, we evaluate the performance of the LWC candidates in an industrial use case: We show how we built up a test setup out of hardware components applied in industry and emulate a battery-powered water meter transmitting customer data. We measure the energy consumption of the device running different LWC algorithms and give an estimation if the use of LWC has a significant advantage over the state-of-the art in that use case.

The thesis structure is roughly aligned with the mentioned contributions. In an introductory chapter, the preliminaries for this work are clarified. Especially, the evaluation process within the NIST LWC project is described, together with the specified submission requirements for the algorithms. To conclude this chapter, an overview of the 1st round candidates and their properties is given.

In the following chapter, our LWC setup for benchmarking software implementations is presented. We describe the reasons for building the framework in our chosen way and provide some details regarding how the different test cases are conducted and which hardware elements have been used.

Chapter 4 deals with how a fair evaluation of the candidates can be realized and presents our benchmarking results for many different implementations in the individual test cases. Moreover, we discuss hardened implementations of some ciphers and collect performance figures for these variants. We show how big the performance penalty is for certain protection mechanisms to support the interpretation of the trade-off between implementation security and performance with suitable data.

In chapter 5, we introduce relevant SCA countermeasures for software implementations. Later, we focus on the efficient protection of ARX ciphers, a design principle some novel symmetric ciphers incorporate. We present two different new ways of searching efficiently protected ARX implementations and show our findings with both methods. We then translate our results to a suitable lightweight cipher and compare the performance of our protected implementations to related work.

Chapter 6 deals with the use case of applying LWC in energy-constrained infrastructures. We build up a hardware setup to represent the industrial use case of a battery-powered smart water meter that sends data to the manufacturer wirelessly in a periodic manner. We explain which components we use in the setup and how our energy-efficient firmware and our test case is structured. We then describe how we collect power consumption data from the hardware running different LWC algorithms. We present results for many implementations and judge if (some) LWC ciphers have an advantage over AES in this use case.

In chapter 7, this thesis is concluded and possible future work paths are introduced.

The related work relevant for a specific research field is distributed and discussed in the dedicated chapters.

2 Preliminaries

In this chapter, the details of the NIST LWC competition are introduced, alongside a brief description of the submission requirements for candidates published by NIST. Finally, we give an overview over each initially accepted cipher. With this analysis, we create a common base of knowledge, which can be referred back to over the course of this thesis.

The NIST LWC project was initiated in 2018 with the goal to find a lightweight alternative to AES-GCM. This winning candidate should be standardized and later used in IoT systems or any other environments where the computing resources are low and still some security requirements need to be fulfilled. The call for algorithms resulted in 57 submitted ciphers, of which 56 have been accepted as 1st round candidates. The project is still ongoing and expected to yield a winner in the first half of 2023.

2.1 Evaluation Process

NIST organized the evaluation process transparently, similarly to the AES and SHA-3 initiatives in the past. There exists a public forum in which anyone can publish comments regarding the project or promote relevant research [21]. Moreover, NIST regularly hosts Lightweight Cryptography Workshops, where researchers or representatives from industry can present and share their latest findings. Since the beginning of the LWC project, three of these workshops have taken place, two virtually and one directly at the NIST headquarters. Furthermore, the evaluation process is split in three rounds. After the initial analysis of the 1st round ciphers, NIST announced 32 2nd round candidates in 2019. After another public evaluation period, NIST picked ten ciphers to advance into the final 3rd round. As of now, it is expected that there will be only one winning candidate in the end, in contrast to e.g. CAESAR, which selected a final portfolio of algorithms.

2.2 Submission Requirements

NIST stated various minimal requirements for cipher packages submitted to the LWC contest. Besides obvious deliverables like the algorithm specification and intellectual property statements, also a reference implementation of the cipher and a set of certain test vectors had to be included in the submission. Furthermore, some design and security goals had to be reached by each contender. NIST required the algorithms to support Authenticated Encryption with Associated Data (AEAD). An authenticated encryption E is defined as a function $E(K, N, A, P) = (C, T)$ where K is the secret key, N is the

nonce, A is Associated Data (AD), P is the plaintext, C is the ciphertext and T is an authentication tag. The corresponding decryption-verification function D is defined as $D(K, N, A, C, T) = P$ [22]. In case the ciphertext is invalid, the decryption-verification function shall not return the plaintext. There exist three main approaches to design an AEAD mode, that differ in how the encryption and authentication processes are interleaved. In an encrypt-then-MAC mode, the Message Authentication Code (MAC) is calculated on the previously encrypted message. Here, the encryption happens prior to the MAC calculation, hence the name. Authenticated encryption algorithms can also be built using a MAC-then-encrypt scheme, where the message is authenticated and the plaintext is encrypted together with the appended MAC. A third variation of the AEAD mode consists of MAC-and-encrypt, here the plaintext is encrypted and authenticated separately, the MAC is appended to the ciphertext. Generally, the encrypt-then-MAC approach is preferred in AEAD designs due to providing integrity of the ciphertext in contrast to the other methods [23].

AEAD algorithms usually rely on the nonce being unique while the same secret key is in use. However, there are certain cipher designs that do not lose (all of their) security in case of nonce-misuse, i.e. the repetition of nonces. NIST requires the submitters to document the behavior of the ciphers in a non-unique nonce scenario [24]. NIST also specified the minimum key length to be 128 bits and that "cryptoanalytical attacks shall require at least 2^{112} computations on a classical computer" [24]. In addition, submitters were allowed to submit multiple variants with different parameter settings of the same cipher family. Still, one cipher variant was to be named as the *primary* variant that represents the main parameter set most suitable for the LWC use cases. Optionally, the LWC candidates could also support hashing functionality. However, not every (final) candidate supports it and in this thesis, we only focus on the AEAD variants of the algorithms.

Apart from the requirements, NIST also defined the most important evaluation criteria. These include cost metrics (area, memory and energy consumption) as well as performance metrics (latency, throughput and power consumption). Moreover, the (efficient) protection against side-channel and fault attacks is stated as a secondary evaluation criterion [24].

2.3 Overview of Cipher Candidates

In the upcoming section, we will provide a categorization and some details of the initial NIST LWC candidates. All of these candidates have been evaluated in our benchmarking framework. While we introduce all 1st round LWC ciphers now, we will later focus on the 3rd round candidates that are still part of the competition. We categorize the cryptosystems according to which primitive they are based on. Note that we format the name of the algorithm in **bold** face if the candidate is a member of the final selection round.

This section represents a brief summary of the LWC candidates and their design features. It is mainly meant to be used as a cohesive cipher reference, including links to

more detailed resources. When we discuss the performance and other factors of LWC algorithms later in our work, one can refer back to this preliminary part.

Moreover, we condense important properties of each primary LWC candidate in appendix A. Our table summarizes key, state, block/rate and tag sizes. Also, the algorithms are further distinguished by their number of rounds, security margin, underlying primitive, the cipher type and mode. This allows for a quick classification of each candidate at a glance.

2.3.1 (Tweakable) Block Ciphers

A block cipher describes an encryption/decryption algorithm that operates on a fixed number of bits. Typically, it takes a message block of n bits and a fixed-length key K as inputs. The decryption function D represents the inverse of the encryption function E. Tweakable block ciphers are algorithms that complement the input by a third component, the tweak T. This introduces more variability to the cipher without sacrificing cryptographic strength. The tweak T does not have to be secret. The main goal of tweakable block ciphers is to introduce an additional input value that can be easily exchanged if that is required by the use case. Since changing keys in traditional block ciphers is often complex and not efficient, an easy-to-replace tweak can be beneficial. A suiting use case for tweakable block ciphers is disk encryption, in which each memory block is encrypted with the same key but an ascending counter tweak [25].

Most block cipher designs within the NIST LWC project rely on one of the two following design principles to achieve secure encryption: For many LWC ciphers, Shannon's idea of Substitution-Permutation Networks (SPNs) to achieve confusion and diffusion in a cryptographic algorithm is still relevant today. In a SPN block cipher such as AES, confusion is typically reached through substitution networks (S-boxes), while diffusion is introduced via a permutation layer. A SPN generally consists of a series of non-linear and linear operations that are carried out a fixed number of times (rounds) [26]. The second popular design pattern is a Feistel network. Contrary to SPN-based ciphers, in Feistel networks the input block is divided in two halves L_0 and R_0. Then, R_0 is encrypted using a subkey within the round function. The encrypted output is later `XOR`ed with the input L_0. In the following, the result of the first `XOR` operation is fed into the round function and its output is again `XOR`ed with the other input half, this time R_0 [27].

Block ciphers can be operated in different modes. The mode of operation defines how the block cipher is applied to an input greater than the block size. Depending on the security goals, certain ciphers can be instantiated with modes that provide authentication, confidentiality or both. Cryptographic modes are separated in parallel and feedback-based variants. With the former, one can process multiple message blocks in parallel, while the latter requires some feedback data from the previous iteration in order to process the next block. The most straightforward parallel mode is called Electronic Codebook Mode (ECB). Here, every block is encrypted separately and the same plaintext is always translated into the same ciphertext. This constitutes a weakness since it allows for the recognition of patterns in the ciphertext. Like ECB, Counter Mode (CTR) mode allows for parallelization and provides confidentiality only. It is commonly used to-

gether with AES in encryption-only scenarios. Effectively, in this mode, the block cipher is operated as a stream cipher. First, a nonce is combined with a counter. This value is then encrypted, before the result is `XOR`ed with the plaintext [28]. Another popular parallel mode is Offset Codebook Mode (OCB). It offers authenticated encryption by integrating a MAC into the encryption routine of a block cipher. There exist three versions of this mode, of which only OCB1 and OCB3 are still considered to be secure [29, 30]. A parallel mode used in the NIST LWC project is Offset Two Rounds (OTR). This mode also combines authentication and encryption. However, it only requires a call to the encryption function of the underlying block cipher to execute both encryption and decryption. Moreover, OTR represents a one-pass mode, meaning the input data needs to be processed only once to achieve both confidentiality and authenticity.

Besides parallel modes, cipher modes based on feedback are heavily used in many designs of the NIST LWC competition. These approaches always use some information from the current block to process the next input. The selection of the particular feedback value determines the corresponding mode of operation. When using Output Feedback (OFB) mode, the output of the current encryption is used in the upcoming encryption operation [28]. Similarly, by choosing Plaintext FeedBack (PFB) or Ciphertext FeedBack (CFB), the plaintext or ciphertext block is fed to the following step as an input, respectively. Another popular method is to combine two or more feedback sources. For example, COmbined FeedBack (COFB) mode suggests mixing at least two values from the previously mentioned triplet (output, plain- and ciphertext) to form the final feedback [31].

The last operation mode described here is Synthetic Initialization Vector (SIV) mode. It represents a MAC-and-encrypt scheme that incorporates cipher-based message authentication and uses CTR mode to ensure confidentiality. SIV provides better security against nonce-misuse than e.g. GCM. In SIV mode, the produced authentication tag is reused as a "synthetic initialization vector" (or nonce) input for the CTR mode encryption. In this authenticated encryption mode, the plaintext has to be processed twice, which makes SIV a two-pass approach compared to e.g. OCB [32].

After the most commonly used operation modes within the NIST LWC project have been introduced, in the following every algorithm candidate will be described briefly. Each cipher profile acts as a short reference for the candidate and provides references to more specialized, in-depth details.

- COMET [33], where the CO stands for CTR, operates on a mixture of the Beetle and CTR modes. In its original design, Beetle combines a sponge construction (see 2.3.2) with a feedback function. This increases the security bound compared to traditional sponge structures and allows for a smaller state size [34]. The developers take the design paradigm from Beetle, but replace the internal sponge permutation with a call to a block cipher and make use of CTR mode. COMET offers different variants with the underlying block ciphers AES, CHAM and Speck [35, 36]. Depending on the variant, the block size of the primitive is set to either 64 or 128 bits.

- **Elephant** [37] is based on an encrypt-then-MAC mode. The exploited primitive in Elephant is a simplified version of the tweakable Masked Even-Mansour (MEM) block cipher [38], encryption is done in CTR mode and message authentication is handled by a variation of the Wegman-Carter-Shoup principle [39]. Elephant provides different instances relying on either SPONGENT [40] or KECCAK [41] as a permutation, depending on the configuration of the MEM cipher.

- ESTATE [42] features a MAC-then-encrypt mode, with OFB [28] applied for encryption and the optimized Cipher Block Chaining (CBC) MAC variant FCBC for tag generation [43]. There exist two instances of ESTATE, one using a tweakable variant of the GIFT cipher [44] and another one using tweakable AES in its core. ESTATE is built to be specifically energy-efficient in lightweight applications.

- FlexAEAD [45] describes a modified version of the FlexAD cipher using the Even-Mansour construction [46]. The submission supports block and state sizes ranging from 64 to 256 bits. Diffusion is achieved with exclusive-or (`XOR`) and shuffle operations, the 8-bit AES S-Box is utilized for confusion.

- ForkAE [47] is constructed out of three main building blocks: First, a forkcipher as introduced by Andreeva et al. is implemented as a symmetric key design component. A forkcipher is a construction that receives a plaintext message and a tweak as an input and generates two ciphertexts as an output [48]. As a second component, the tweakable block cipher SKINNY is selected and used as the underlying primitive. The last building block is formed by two specific operation modes, Sequential AEAD from a Forkcipher (SAEF) and Parallel AEAD from a Forkcipher (PAEF). In short, ForkAE is a primitive exploiting SKINNY in the forked instantiation FORKSKINNY, which is operating in the defined modes SAEF and PAEF. Since it is based on SKINNY, the ForkAE shares many properties with this primitive. The candidate is optimized for handling short messages particularly efficient, and is comparatively easy to protect against basic side-channel attacks.

- **GIFT-COFB** [49] utilizes the COFB AEAD mode for block ciphers [31]. This was initially designed to perform best regarding hardware implementation size. The designers combine the COFB mode with the GIFT block cipher [44]. GIFT was developed to deliver excellent performance in hardware and seemed to be complicated to implement efficiently in software. However, the authors of the submission presented a new a method called *fixslicing* to express the algorithm in a more performant way in software [50]. Considering its novelty, the underlying GIFT cipher presents a well-researched and fitting building block for LWC.

- Similar to GIFT-COFB, HyENA [51] is instantiated with the GIFT cipher. Its input is put together from PFB and CFB, thus Hyena uses a hybrid feedback mode [31]. HyENA focuses on efficiency in hardware designs. Moreover, it only needs one block cipher call per data call, which is commonly referred to as the single-pass property.

- LAEM [52] adopts the SIMON block cipher as its main primitive [36]. The block and state size of the candidate is 128 bits, independent of the algorithm variant. The dominant changing parameter is the key size, with 128, 192 or 256 bits, depending on the instance. LAEM operates in ECB mode and is built such that its implementations can be easily parallelizable.

- LILLIPUT-AE [53] is based on the tweakable version of the block cipher LILLIPUT presented in [54]. It features the two operation modes OCB and Synthetic Counter in Tweak mode (SCT) [55]. The permutation consists of an extended generalized Feistel network [56]. With the two different modes, LILLIPUT-AE offers one nonce-respecting and one none-misuse resistant variant, which allows for flexibility depending on the use case.

- Limdolen [57] operates in CTR mode with the round function inspired by the Parallelizable Message Authentication Code (PMAC) SIV construction [58, 32]. The permutation features a Feistel structure and confusion is achieved through ARX operations. Due to its design, Limdolen is inherently resistant to timing side-channel and fault attacks.

- LOTUS-AEAD and LOCUS-AEAD [59] are instantiated with a tweakable variant of the GIFT cipher [44]. The mode of operation of LOTUS relies on OTR, while LOCUS features an OCB mode which has already been used in a variant of AES submitted to CAESAR [60, 61]. Due to the nature of their modes, LOTUS and LOCUS are easy to parallelize. The primary variant provides one of the smallest state sizes (64-bit) of all NIST LWC candidates.

- mixFeed [62], short for MInimally Xored FEEDback, describes an AEAD algorithm with a hybrid mode using a combined feedback from ciphertext and plaintext. The designers choose AES with a key and nonce size of 128 bits as an underlying primitive. However, in their AES-based operation, they carry out the *MixColumns* step also in the final round. The secret subkey for each encryption is computed using the input nonce. Through use of the nonce-dependent key, the master key remains secret even in case the key corresponding to the nonce is leaked. Technically, mixFeed can also work with other block ciphers incorporated in its core.

- Pyjamask [63] focuses on side-channel resistance while still being lightweight in its design. The primitive is internally using its own permutation and relies on OCB mode. The Pyjamask permutation only uses a minimal amount of non-linear gates in order to allow for efficient implementations of (higher-order) masked variants [61].

- Qameleon [64] represents another candidate which makes use of a variant of the OCB mode. In particular, Qameleon operates in the PArallelisable NOnce Rotating Authenticated Encryption with Assocaited Data (PANORAMA) mode and instantiates the tweakable block cipher QARMA [65]. QARMA is built as an

Even-Mansour construction featuring three rounds and has a 4-bit S-Box to obtain confusion [66]. Qameleon allows for a high level of parallelization and is specifically optimized for efficient memory encryption.

- As also seen in other NIST LWC submissions, Remus [67] is constructed from the SKINNY cipher introduced in 2016 [68]. Remus shares its basic design with the candidate Romulus, since both rely on the variants of the tweakable block cipher SKINNY. Remus offers two main variants, the nonce-based AEAD cipher Remus-N and the nonce misuse-resistant family Remus-M. Moreover, the designers chose COFB as the basic mode of operation.

- **Romulus** [69] follows a similar design principle as Remus. Generally, SKINNY is used as the internal block cipher which runs in COFB mode in Romulus-N. Yet, Romulus-M adapts the SIV construction instead of the feedback-based mode COFB. Remus and Romulus mainly differ in the way of the instantiation of the tweakable block cipher. In Romulus, the state uses "persistent key material and a changing tweak" [67], while Remus is "utilizing the whole tweakey state of [SKINNY] as a function of the key" [67].

  The designers of Romulus added a third variant of the cipher during the competition, Romulus-T. Romulus-T is developed as a leakage-resilient mode to limit "the exploitability of physical leakages via side-channel attacks, even if these leakages happen during every message encryption and decryption operations" [69]. This mode is designed according to the principles of the AEAD primitive TEDT [70].

- SAEAES [71] is based on the standard AES cipher. Therefore, e.g. protections against side-channel attacks that have been studied for AES can also be easily applied to SAEAES. The AES block cipher is instantiated in an AEAD mode of SAEB, which is short for Small (Simple, Slim, Sponge-based) AEAD from Blockcipher [72]. SAEB was designed to support a minimum state size (4×4 bytes in SAEAES) and uses `XOR` operations only.

- Saturnin [73] specifies a permutation with a $4 \times 4 \times 4$ cube of 4-bit nibbles as its state and a 4-bit bitslice S-Box to realize confusion. The design offers two encryption modes, Saturnin-CTR-Cascade as a general AEAD cipher and Saturnin-Short, a design specifically built for working with messages that are smaller than 128 bits and for scenarios where no AD is needed. Saturnin operates in an encrypt-then-MAC mode, its block and key size is 256 bits.

- Simple [74] offers two main categories of ciphers. One where the state and block size equals 128 bits and a second one, where these parameters are half the size. The length of the key remains at 128 bits for both variants. For tag generation, Simple-128 and Simple-64 define the building block CBCMAC-IV, while Simple128 uses CTR and Simple64 uses CENC mode for encryption. Moreover, the cipher suite provides different primitives, depending on the underlying block cipher. In their submission, the designers state options with PRESENT, GIFT and Speck [75, 44, 36].

- SIV-Rijndael256 [76] consists of the block cipher Rijndael, of which the AES standard has been formed. However, SIV-Rijndael uses Rijndael256, a variant of the block cipher with a block and state size of 256 bits and a 128-bit key length. To provide AEAD functionality, the cipher is paired with SIV, which combines an encryption algorithm – in this case Rijndael256 – with a pseudorandom function [32]. Due to the design of the SIV mode, SIV-Rijndael256 still provides security in case nonces are repeated.

- Another candidate that incorporates the SIV construction is SIV-TEM-PHOTON [77]. Here, SIV gets paired with the tweakable Even-Mansour structure TEM-PHOTON [78, 79]. In TEM-PHOTON, the round functions of the PHOTON permutation [80] are selected as a base for the tweakable Even-Mansour cipher.

- Like other NIST LWC candidates, SKINNY [81] chooses the eponymous cipher as the internal block cipher [68]. The submission provides two instances SKINNY-b-i, where blocksize b is 128 bits for both and the input length i is either 256 or 384 bits. The ciphers are designed according to the TWEAKEY framework and operate in OCB mode [82]. SKINNY represents an already well-researched primitive, which has withstood extensive cryptanalysis since its publication.

- SUNDAE-GIFT [83] specifies an AEAD scheme based on the block cipher GIFT-128 [44]. The block cipher is utilized in the SUNDAE mode of operation. SUNDAE was designed as a Small universal deterministic authenticated encryption for the Internet of Things and presented in 2018 [84]. SUNDAE-GIFT is particularly suitability for short messages, the authors argue that the internal usage of GIFT has advantages in efficiency over e.g. candidates based on PRESENT or SKINNY [83].

- TGIF [85] describes the tweakable block cipher TGIF-TBC. The design of this component is largely inspired by the GIFT cipher and even reuses 4 rounds of GIFTb, which denotes a bitsliced variation of GIFT [44]. TGIF adapts its modes of operation from Romulus and Remus. The variant TGIF-N implements a modified COFB mode, while TGIF-M defines a similar mode to SIV and SCT [31, 32, 55].

- **TinyJAMBU** [86] represents a variant of the JAMBU mode submitted to the CAESAR competition. The block cipher AEAD mode has there been selected as a 3rd round candidate [87]. In TinyJAMBU, the designers show a smaller variant of the JAMBU mode with a reduced message block size of 32 bits. TinyJAMBU defines a keyed permutation featuring a 128-bit Non-Linear Feedback Shift Register (NLFSR). The non-linear part of the permutation is realized with a `NAND` gate. TinyJAMBU supports parallel computation and provides better authentication security in case of nonce-misuse compared to other modes [86].

- TRIFLE [88], as the full name ThReshold Induced Fault resistant Lightweight authenticated Encryption suggests, is designed to provide efficient (and inherent) protection against side-channel and fault attacks. The submission defines a specially-designed MAC-then-encrypt mode that hardens the cipher inherently

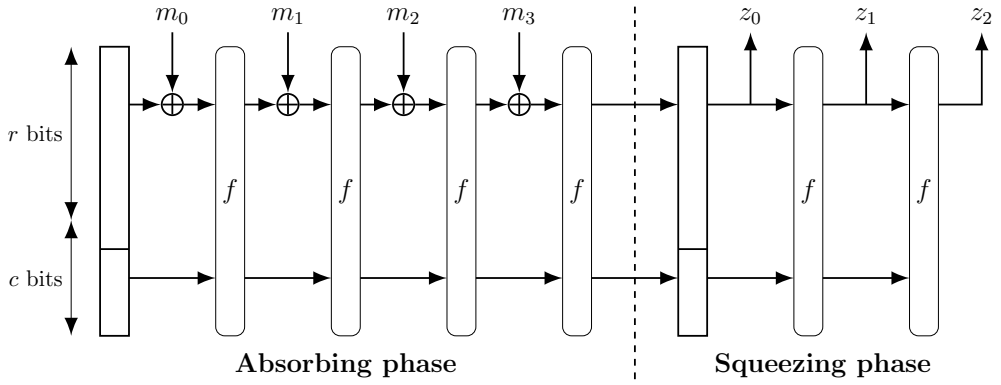


Figure 2.1: General structure of a sponge construction [1]

against well-known fault attacks. Moreover, the authors claim that a protected Threshold Implementation (TI) of TRIFLE can be realized more area-efficient than e.g. for PRESENT or GIFT [44, 75, 88].

- AES-GCM [89] is not a NIST LWC candidate. However, it will be heavily used as a reference to judge the performance of the novel ciphers' implementations. AES-GCM is a well-known standardized AEAD scheme that is applied in lots of use cases, also in the industry. The encryption in AES-GCM is based on CTR mode, while authentication happens in parallel through multiplications in the Galois field $GF(2^{128})$.

2.3.2 Sponge-based Ciphers

A sponge construction describes a cryptographic operation mode, which is based on a permutation that has a fixed length and works on a defined number of bits b [90]. We denote the permutation as a function f. The sponge construction as a whole has an input m of variable length and produces an arbitrary-length output z. The permutation f is typically applied on b several times, similar to a round function. In a sponge construction, the bits b are divided into two groups, the rate r and the capacity c, such that $b = r + c$.

Before the first main phase on the sponge (absorbing), the input string is padded and sliced into blocks of size r. After b is initialized with zeros, the absorbing of the input starts. Input blocks of size r are `XOR`ed into the bits b, then the permutation f is applied. These two steps repeat as long as there are input blocks left. Note that the bits b are also often referred to as the state of the sponge construction in the context of symmetric cryptography or hashing.

In the second part of the sponge (squeezing phase), the construction extracts the first r bits of the current state as an output block, then f is applied. Again, these two operations repeat until the number of desired output blocks have been returned. The last c bits of the state are never altered by the input, nor are they returned as an output. Figure 2.1 illustrates the described steps carried out in a sponge construction.

A popular variation of the standard sponge is the duplex mode. Inspired by full-duplex communication, in a duplex sponge, the input (absorbing) and output (squeezing) phases happen in an alternating fashion. Moreover, the in- and outputs have fixed lengths and any output depends on all previously processed inputs. Duplexed sponge constructions also allow for one-pass authentication and encryption. The SpongeWrap AEAD mode relies on the duplex sponge. This single-pass authenticated encryption construction requires the associated data to be unique for encryptions with the same key. Essentially, the associated data has to behave the same way as a nonce to retain confidentiality. If this property is violated, different plaintexts will be encrypted with the same keystream [91].

An alternative to the standard duplex sponge is the monkey duplex mode. This variant introduces the use of a nonce in the initialization phase. Since the nonce has to be unique, this mode is not applicable in case the uniqueness of a nonce cannot be guaranteed. However, this modification of the regular duplex sponge results in a gain in security [92].

Besides these three sponge designs, the Beetle mode of operation is often integrated in NIST LWC ciphers. Beetle is based on SpongeWrap and implements a sponge construction together with a combined feedback block. The feedback is utilized to introduce a difference between the current ciphertext and the input to the next permutation. In classical sponge constructions, these values are the same. This alternation boosts the security of the mode without sacrificing performance [34].

In the following, we introduce the sponge-based NIST LWC ciphers and their key features. Again, sources with additional details for each cipher are linked and the algorithms are added to the table overview in appendix A to support a quick comparison.

- ACE [93] features a 320-bit state consisting of five 64-bit words. It uses `XOR`, `AND`, shift and shuffle operations together with a reduced version of the Simeck cipher in its core [94]. ACE runs in a sLiSCP sponge mode specifically suited for Simeck-based constructions [95]. ACE is implemented in a bitsliced manner, which makes its implementation resistant against cache-timing attacks.

- **ASCON** [96] has the same state size as ACE and operates in a monkey duplex mode [92]. As mentioned before, in duplex constructions, the absorbing and squeezing phases are not completed subsequently. The processing of an input m_i is directly followed by the squeezing of an output block z_i.

  Ascon has already been part of the CAESAR competition and some of its instances were selected as the primary choice for lightweight authenticated encryption in the final portfolio. The Ascon suite provides three different main variants. Ascon-128 and Ascon-128a are the recommended types for AEAD, while Ascon-128 is set as the primary recommendation. Ascon-80pq offers higher resistance against quantum key-search attacks. The cipher family is designed to be protected against side-channel attacks at a low(er) cost.

- Like Ascon, CiliPadi [97] operates in a monkey duplex mode [92]. It offers four different instances: Mild, Medium, Hot and ExtraHot, which differ in state, tag,

key, block size and number of rounds. CiliPadi applies `XOR` operations and block swaps for diffusion and implements two unkeyed rounds of the LED cipher (with a 4-bit S-Box from PRESENT for confusion [98, 75]). The LED cipher can be replaced with any AES-like block cipher in this design.

- CLX [99] features a sponge duplex mode of operation. CLX defines a permutation $P_{160+x,n}$, where the state size in bits is denoted by $160 + x$ and n specifies the number of rounds. For example, $P_{160,320}$ describes that a 160-bit state is updated in 320 rounds. The state update function in CLX is realized with a NLFSR. The computation of CLX allows for parallel execution of various rounds. The cipher family contains seven AEAD variants, the primary member CLX-128 and three pairs of ciphers with a key size of 128, 192 and 256 bits each. Every pair consists of a version Q, which is optimized for speed and a version H, which ensures protection of the secret key in case of nonce-misuse.

- DryGASCON [100] is based on a mixture of Ascon and the DrySponge mode. DrySponge is another sponge construction, which is similar to the duplex variation, there are however slight differences in the ways its input is merged and the output is returned [91]. DryGASCON is designed such that it allows for protection against most physical attacks already on an algorithmic level. This is achieved through the introduction of a dedicated input which is used for domain separation. The algorithm utilizes a generalized version of the Ascon permutation, GASCON, that supports a bigger state size and reaches up to 256-bit instead of 128-bit security.

- Gimli [101] was designed to fit a broad range of LWC use cases, ranging from hardware targets to server systems that communicate with multiple clients possibly running the same cryptographic algorithm. Gimli operates in a duplex mode and features a 384-bit state. On bit level, the state is organized in a parallelepiped with the dimensions $3\times4\times32$. This corresponds to 12 32-bit words on an implementation level. Gimli uses defined swap methods (big swap and small swap) to produce diffusion and a SP-box consisting of `AND`, `XOR` and rotation operations to reach confusion.

- Again, InGAGE [102] uses the monkey duplex mode of operation. One property that distinguishes InGage from most other sponge-based ciphers is its low rate size, which gets as small as 8 bits, depending on the variant. When $r = 8$, the state holds 232 bits. In its largest configuration, InGAGE has a state size of 512 bits with $r = 64$. As a non-linear layer, InGAGE implements a small 4-to-2-bit S-box and diffusion is achieved through bit shuffling.

- **ISAP** [103] features a sponge-based encrypt-then-MAC construction. The main design focus of the cryptosystem lies on the resistance against side-channel attacks. Isap incorporates a re-keying function that ensures that always fresh secret material is injected when processing new data. This mechanism hardens the primitive against a number of Differential Power Analysis (DPA) attacks. Isap defines

four different instances, ISAP-K-128A, ISAP-A-128A, ISAP-K-128, and ISAP-A-128. The variants that have a K in their name internally employ a 400-bit state processed by the KECCAK permutation, while the other two apply the ASCON permutation on a 320-bit state [41]. KECCAK in a configuration with a larger state is part of the SHA-3 specification [14].

- KNOT [104] is another NIST LWC candidate relying on a monkey duplex construction. The permutations applied in KNOT are based on the bitsliced block cipher RECTANGLE [105]. KNOT offers various instances with different properties. For example, the state size ranges from 256 to 512 bits. In the KNOT permutation, three different steps are conducted subsequently: *AddRoundConstant*, *SubColumn* and *ShiftRow*. These operations are carried out on the state bits b and can be expressed in a bitsliced manner. Therefore, KNOT is well-suited to be implemented in a bitsliced way that is resistant against cache-timing attacks.

- ORANGE [106] and its AEAD variant ORANGE-Zest are built upon a slightly modified sponge construction. ORANGE-Zest uses full state absorption and holds a second 128-bit secret state to allow for the complete absorbing of the first state. ORANGE utilizes the $PHOTON_{256}$ permutation with its 4-bit S-box internally [80].

- Oribatida [107] is operated in a variant of the monkey duplex sponge mode. It incorporates the SimP permutations, which are strongly inspired by instances of the SIMON block cipher [36]. Oribatida defines two cipher versions, Oribatida-256-64 with a permutation size of 256 bits (based on SIMON-128-128) and Oribatida-192-96 with a smaller 192-bit state (based on SIMON-96-96). The second number in the cipher name defines the size of the mask s, which is included to protect the ciphertext additionally and reach the required security level stated by NIST. Oribatida provides robustness in case of the release of unverified plaintexts. If a plaintext is leaked from invalid ciphertexts, Oribatida still retains the integrity of the data.

- **PHOTON-Beetle** [108] applies the sponge variant Beetle [34]. The Beetle mode is instantiated with the PHOTON permutation that is also applied in ORANGE-Zest. The PHOTON permutation, denoted as P_{256} by the designers of PHOTON-Beetle, works on a 256-bit state, the rate of absorption differs between 128 and 32 bits, depending on the variant of PHOTON-Beetle.

- Shamash [109] is operated in a sponge duplex mode with a 320-bit state, expressed in five 64-bit words. It uses its own permutation based on *MixColumns* and *ShiftRows* functions and a specifically designed 5-bit S-box. The authors claim that the low algebraic degree 2 of the S-box, while it has disadvantages in regards of algebraic attacks, makes it easier to implement protected versions of the cipher. The minimum length for tags in Shamash is 8 bytes. The cipher is not parallelizable due to its design.

- SNEIKEN [110] defines the novel BLNK2 mode, which is modelled according to the "BLINKER-style" primitives [111]. These are state-based modes using rates and capacities like standard sponge constructions. SNEIKEN employs a relatively large 512-bit state, the data block size is either 256 bits (SNEIKEN256), 320 bits (SNEIKEN192) or 384 bits (SNEIKEN128). The underlying SNEIK permutation is designed as a NLFSR. Additionally, diffusion is reached through a simple ARX layer that performs additions, rotations and exclusive-or operations on 32-bit words [20].

- **SPARKLE** [112] is inspired by the block cipher SPARX [113] and mainly modifies its block size and key length. The SPARKLE permutation can be used in an AEAD or hash setting. The designers of SPARKLE name the algorithms for the two categories of application SCHWAEMM and ESCH. In SCHWAEMM, the SPARKLE permutation is instantiated within the duplex sponge mode Beetle. However, Beetle is slightly modified by limiting the key length to the size of the capacity c. Internally, SPARKLE implements a four-round ARX-box operating on 32-bit words. SCHWAEMM provides four instances (128-128, 256-128, 192-192, 256-256) with different security levels and state sizes ranging from 256 to 512 bits. Since we do not consider hash functions in this work, we always refer to the AEAD variant SCHWAEMM when mentioning the candidate SPARKLE.

- SPIX [114] incorporates the monkey duplex sponge with a state size of 256 bits. It uses the Simeck-based sLiSCP-light permutation that is also implemented in the ACE cipher [95]. Block swaps and `XOR` operations help in achieving diffusion and the eight-round 64-bit Simeck box is exploited for confusion [94]. SPIX processes blocks of 64 bits and defines only a single instance for AEAD.

- SpoC [115] is operated in a slight variation of the Beetle mode. Like SPIX, it also includes parts of Simeck and the sLiSCP-light permutation at its core. According to the design of Beetle and other sponge constructions, the rate is usually masked by the input bits. Instead, in SpoC the capacity part of the state is masked with the input block. The authors claim that this change in the mode helps to achieve a higher security level while implementing a lower state size. SpoC offers two cipher variants, one with 192 and one with 256 state bits.

- Spook [116] was designed with the two primarily goals of resistance against SCA and energy-efficiency. Therefore, the cipher features a leakage-resilient operation mode that allows for efficient side-channel protection [117]. Furthermore, this mode is coupled with the tweakable block cipher Clyde-128 and the Shadow-512 permutation. These advanced building blocks are inspired by the bitsliced designs of other block ciphers [118]. Clyde-128 and Shadow-512 enable efficient bitsliced and side-channel resistant implementations with a low-energy footprint.

- Subterranean 2.0 [119] origins from its first version published in 1992 [120]. The state of Subterranean is expressed as a one-dimensional array of 257 bits, which makes it hard to implement the cipher efficiently in software. The cipher is operated

in a duplex sponge mode with an injection rate of 33 bits. Due to the state design of Subterranean 2.0, the cipher is suited for low-area and low-energy hardware implementations.

- Sycon [121] features a monkey duplex sponge. It implements the eponymous permutation, which is operating on a 320-bit state. Confusion is achieved by using a 5-bit S-box, the whole permutation consists of two different layers of smaller permutations. The first layer applies simple linear diffusion on the 320-bit state, the second (called FIST) performs five permutation steps on 64 symbols before running the inverse of the first permutation layer on the state. Sycon is designed in a way to be efficient in both software and hardware implementations.
- **XOODYAK** [122] defines a 384-bit state manipulated by the Xoodoo permutation [123], which is inspired by Keccak. The state is organized in three 128-bit planes that are mixed on the basis of 3-bit column chunks. Xoodyak is operated in *Cyclist* mode, a lightweight variation of *Motorist*, which is applied in the CAESAR candidate Keyak [124]. In Xoodyak, nonce-misuse does not lead to leakage of the secret key through cryptanalysis. Also, the authentication part of the cipher does not rely on a nonce at all.
- Yarará [125] is using a duplex sponge construction with a state size of 256 bits with a rate of 64 and a capacity of 192 bits. The state is expressed in the four 64-bit words x_0, x_1, x_2 and x_3. Diffusion is realized by alternating linear diffusion and column mixing, while the confusion part relies on a 4-bit S-box. In Yarará, the plaintext, ciphertext and AD is padded to a length of $r/8 = 8$ bits. The cipher is well-suited for bitsliced implementations.

2.3.3 Stream Ciphers

The NIST LWC team also received some stream-based algorithms as proposals for first round candidates. In contrast to block and sponge-based ciphers, a stream cipher does not feature a block size or rate. Instead, the plaintext is processed symbol by symbol, typically bit by bit. The input is combined with a (pseudo-) random keystream to obtain the ciphertext. Typically, the combination of the input and the keystream is done with a `XOR` operation. As mentioned previously, block ciphers can as well be operated as a stream cipher by choosing a particular mode (e.g. CTR).

In the next section, we provide a brief description of each stream cipher taking part in the NIST LWC project.

- Bleep64 [126] is heavily based on its predecessor BeepBeep introduced in 2002 [127]. Bleep64 shares the basic structure with BeepBeep, while it is improved in performance and provides better avalanche. The key size is also reduced from 223 to 128 bits. Bleep64 implements a 127-bit Linear Feedback Shift Register (LFSR), which acts as Pseudo-Random Number Generator (PRNG). This PRNG is paired with an autokey stream cipher to provide confidentiality and integrity. The cipher introduces non-linearity through ones'-complement addition.

- The state of Fountain [128] is stored in four 64-bit LFSRs. Fountain exploits the 4-bit S-box of GIFT to extract the required number of bits. This 4-bit string is then fed to a Maximum Distance Separable (MDS) matrix which is defined over $GF(2^2)$ [129]. Fountain is oriented towards efficient implementations in hardware and allows for parallel computation.

- **Grain-128AEAD** [130] employs a pre-output generator that consists of a 128-bit NLFSR and a 128-bit LFSR. This component is responsible for producing a pseudo-random bit stream that is needed for encryption and authentication. Both registers together also represent the state of the cipher. The second main component of the algorithm is the authenticator generator, which is made from a shift register and an accumulator. The two building blocks each are 64-bit in size and the shift register contains the current 64 odd bits from the pre-output. Grain-128AEAD represents a slight modification of Grain-128a, which has already been introduced in 2011 [131]. A major difference to Grain-128a is that Grain-128AEAD does not support an encryption-only mode, but always authenticates the input data (according to the NIST LWC requirements).

- HERN [132] forms its state from four LFSRs containing the 64-bit words s_0, s_1, s_2 and s_3. Additionally, the two 1-bit variables a and b are used to hold the in- and output bit, respectively. For the state update function, the LFSRs are connected circularly, such that "one bit of one register affects its previous register" [132]. HERN uses `XOR` together with `AND` operations to introduce non-linearity and allows for 32 function steps to be computed in parallel.

- Quartet [133] organizes the state of the cipher as four 64-bit lanes x_0, x_1, x_2 and x_3. The round function is defined as $r = \tau \circ \lambda \circ \rho \circ \chi$ with λ and ρ providing diffusion through rotations and Ascon-like operations, τ performing constant addition and χ being the only functions introducing non-linearity. In Quartet, the plain- and ciphertext is padded to match a multiple of 8 bit, the cipher can be (partly) parallelized and is designed to be very efficient in hardware while maintaining reasonable performance in software.

- TRIAD-AE, the AEAD variant of TRIAD [134], is built as an encrypt-then-MAC construction with the MAC part relying on a sponge variant. For encryption, TRIAD-AE adopts the TRIVIUM stream cipher [135] and alters it in order to achieve a higher and NIST-compliant security level. The TRIAD state is constructed from three registers with the size of once 80 and twice 88 bits. This results in a state size of 256 bits, which is a 32-bit reduction compared to the TRIVIUM cipher. TRIAD-AE is designed with a focus on low-energy consumption and reasonable area size. Nonce-misuse in TRIAD-AE leads to loss of confidentiality, but no loss of the security of the key.

- WAGE [136] defines a 259-bit permutation derived from the Welch-Gong stream cipher [137]. It was developed to be efficient in a hardware implementation. WAGE is operated in a sponge duplex mode with a rate r of 64-bit. The round function

consists of a LFSR, two Welch-Gong permutations and the application of four 7-bit S-boxes. The LFSR, i.e. the state of the cipher, is expressed in 37 stages, with every stage being an instance of the finite field $\mathbb{F}_{2^7}$. WAGE applies padding to the AD, plain- and ciphertext in case their length is not a multiple of 64 bits.

2.3.4 Others

- CLAE [138] is an AEAD design based on the Learning with Errors (LWE) problem [139]. A certain number of input bytes are encrypted into 2-byte ciphertexts. That these 2-byte chunks appear as randomly sampled – similar to the decision problem in LWE – forms the basic idea of CLAE. The 2-byte ciphertext has some advantages when it comes to integrity and fault attacks. It offers 1-byte redundancy, so its decryption will fail if the ciphertext is altered unintentionally or by a fault attack. Moreover, any modification of the plaintext, the nonce or the AD will be reflected in the whole ciphertext. If the ciphertext is changed, the decryption process spreads the faults to each 2-byte block, where they can be detected. CLAE provides protection against side-channel attacks due to "[t]he bytes in the secret key [being] accessed nondeterministically during encryption" [138].

3 A Framework for Benchmarking NIST LWC Ciphers

The core ideas presented in this chapter have already been published in S. Renner, E. Pozzobon, and J. Mottok. A hardware in the loop benchmark suite to evaluate NIST LWC ciphers on microcontrollers. In International Conference on Information and Communications Security*, pages 495–509. Springer, 2020*

Over the last roughly 20 years, various benchmarking suites for cryptographic algorithms have been developed. Especially, the evaluation projects mentioned in chapter 1 often kickstarted benchmarking efforts and the development of suitable frameworks to assess the participating ciphers. For example, the work of Ankele et. al focuses on software benchmarking of 2nd round CAESAR candidates. They classified the lightweight ciphers and evaluated their speed on two desktop computers processing different message lengths [141].

Older but still very popular frameworks are eBACS and SUPERCOP. eBACS consists of different software components that are made for measuring the performance of asymmetric cryptography, stream ciphers, hash functions and authenticated encryption algorithms. Parts of it origin from the benchmarking efforts carried out during the eSTREAM competition [13]. SUPERCOP supports a significant amount of test cases and platforms for evaluating cryptographic software. The setup features many test devices operated by different engineers in different locations. SUPERCOP is run in powerful desktop environments as well as on higher-power ARM Cortex-A cores. It iterates over lots of compilation options to find the most performant settings for each platform and implementation [142].

Dinu et al. worked on a performance evaluation tool to support the fair assessment of software implementations of block ciphers. Initially, they benchmarked 19 cryptographic algorithms in their FELICS framework [143, 144]. This FELICS framework was later extended to allow for testing AEAD ciphers in the context of the NIST LWC project. It features three different platforms and ranks algorithms accoring to execution time, code size and Random Access Memory (RAM) utilization [145].

Other benchmarking efforts within the NIST LWC project include the research from Weatherley. He focused on speed testing implementations on AVR-architecture. Moreover, he provided variants optimized on assembly level himself and conducted benchmarks on 32-bit and 8-bit platforms. His results are presented in relation to the execution time of the ChaChaPoly AEAD scheme, an alternative to AES-GCM that relies on the stream cipher ChaCha20 and the authentication algorithm Poly1305. ChaChaPoly is standardized in an IETF RFC [146, 147].

Campos et al. base their performance evaluations on a single architecture, RISC-V. They study the optimization potential for C and assembly implementations of six NIST LWC candidates. The cycle count for their software versions is measured in multiple simulation environments and on a SiFive E31 core. The authors show that they can reach significant speed gains when applying certain implementation strategies, such as loop unrolling or bit interleaving [148].

The NIST cryptography team itself applied performance benchmarking of software implementations. Their research includes execution time and code size evaluations on four different architectures. NIST presented speed rankings for various message sizes, both with and without AD. Their experiments have been initially conducted for 2nd round candidates [149].

More examples for cryptographic software benchmarking outside the NIST LWC competition are the research from Hyncica et al. and Tschofenig et al.: While the former work also analyzes symmetric ciphers of embedded platforms, the latter compares the performance of asymmetric algorithms. In particular, the timing differences of the signature and verification process of elliptic curve cryptography are evaluated on Cortex-M microcontrollers [150, 151].

Since the performance of ciphers is important in a software and in a hardware implementation, other researchers also specialize in comparing e.g. the NIST LWC candidates on FPGAs or ASICs. While this is out of scope for our work, their analyses complement the software benchmarks and have to be taken into account when judging the weight of a cipher. The most prominent benchmarking projects for hardware implementations of NIST LWC ciphers are maintained by Kaps et al., Khairallah et al. and Aagaard et al. [152, 153, 154].

Our benchmarking framework differs from the mentioned research in the following aspects. First off, our solution is custom-built and tailored for the algorithms and constraints of the NIST LWC competition. This means both the software and the hardware platforms has been designed and selected to be suitable for evaluating the NIST candidates. Furthermore, the testing process and the tool was built to support a high degree of automation. This allows for fast result gathering and feedback without heavy human interaction. On top of the automated test procedure, we host a publicly accessible website, where the benchmarking results and useful metadata is published for each test. The website blends seamlessly with the idea of updating the comparison data with results from freshly emerged implementations. Since also most parts of the testing process is automated, this allows us to always keep our benchmarking results up-to-date throughout the whole NIST LWC project.

The combination of these key design points and features distinguishes our framework from related work. While most benchmarking projects offer a subset of these properties, none fully matches all of them. In the following section 3.1, we first discuss the requirements for an ideal NIST LWC benchmarking framework in detail and then highlight how our custom-built solution matches these the best in comparison to the most important related work.

3.1 Requirements

When we planned our LWC benchmarking project, we had to decide whether we use and extend an existing framework or if we build our own. In order to make an educated decision, we elaborated requirements that our solution had to meet. While we did not follow a strict software development process, we still researched the landscape of already built frameworks and implemented the step of formulating requirements before the actual design and implementation of the benchmarking tool.

The benchmarking project was already started shortly after the beginning of the NIST LWC competition. With still 56 candidates in competition and multiple implementations per candidate, we knew that having a high degree of automation in the testing procedure and results gathering is critical. Moreover, one of our project goals was to support the selection process by NIST throughout the whole project duration. In order to achieve that, not only a strongly automated testbed is needed, but also a result database has to be maintained and updated regularly.

Another key requirement is a suitable interface to the implementation that can be used for sending input to and receiving output from the algorithms under test. Here, it is helpful that NIST required every submission to include (at least) a reference implementation in C that contains encryption/decryption functions with a certain parameter set. Listing 3.1 shows the C code for the predefined signatures of these functions [24].

```
1  int crypto_aead_encrypt(
2      unsigned char *c,unsigned long long *clen,
3      const unsigned char *m,unsigned long long mlen,
4      const unsigned char *ad,unsigned long long adlen,
5      const unsigned char *nsec,
6      const unsigned char *npub,
7      const unsigned char *k
8  )
9
10 int crypto_aead_decrypt(
11     unsigned char *m,unsigned long long *mlen,
12     unsigned char *nsec,
13     const unsigned char *c,unsigned long long clen,
14     const unsigned char *ad,unsigned long long adlen,
15     const unsigned char *npub,
16     const unsigned char *k
17 )
```

Listing 3.1: Signatures of mandatory C functions for encryption/decryption

The parameters for the `cryto_aead_encrypt` function are the generated ciphertext `*c` with the elements `c[0]`,`c[1]`, ..., `c[*clen-1]` (and `clen` referring to the length of the ciphertext), `m` being the plaintext with the length `mlen` and the elements `m[0]`,`m[1]`, ..., `m[mlen-1]`, `*ad` containing the AD with length `adlen` and `*npub` pointing to the nonce, while `*k` points to the secret key array. `*nsec` is kept for compatibility reasons with SUPERCOP and shall not be used. The same parameter explanations apply to

the function `cryto_aead_decrypt`. This Application Programming Interface (API) is borrowed from SUPERCOP and defines a uniform entry point that can be utilized in a LWC cipher test setup.

Besides fitting software interfaces, also the hardware part of a benchmarking setup has to suit potential LWC use cases. Mainly this means that the devices under test, i.e. the microcontrollers, should be chips that are popular platforms for IoT developments in industry or the maker community. Furthermore, a set of embedded systems should feature different performance categories and architectures to support a holistic testing approach. Also, the benchmarking tool should be designed in a modular fashion, such that adding or replacing devices under test is possible and straight-forward.

It is important as well that a performance testing tool implements the most relevant test cases. This includes not only measuring the right metrics, but also how these metrics are measured and under which circumstances. In the past, most related work in the field of software implementation benchmarking focused on one or more of the following parameters: execution time, binary size, RAM footprint [144, 148, 149]. Especially when measuring timings on the Microcontroller Units (MCUs), it is crucial to use external measurement equipment to not get inaccurate results due to e.g. internal timer delays. Additionally, when quantifying Read Only Memory (ROM) or RAM consumption, one has to take into account the memory utilization due to the testing software itself and exclude this exact amount when judging the utilization of the algorithm under test. The correct application and implementation of these test cases has to be ensured in an accurate benchmarking setup.

Also, some general testing principles need to be respected. Of course, the tests and their corresponding results need to be reproducible and always follow the same test protocol. The benchmark figures and the gathered metadata have to be saved for each test (attempt). For every conducted test, a log file containing details regarding the test procedure shall be available. This can be helpful when tracing errors in the test execution or to verify a correct termination of a test. To support maximum transparency, the results of every benchmark should be made available to the public, accompanied by details on the time of the test execution and the source code of the benchmarked algorithm. In that way, third parties can inspect and follow the benchmarking process and potential execution errors can be detected easier. These transparency requirements can be consolidated under the term *public documentation*.

In table 3.1, we summarize our previously explained requirements and form them to phrases according to the Rupp et al. template [155]. In table 3.2, we show which of these requirements are met (to what degree) by the most popular related work. It can be concluded that none of the available frameworks offered every of our desired features to the full extent. Since we also did not want to study the source of an existing solution in-depth because we believe that this is as time-consuming as a custom build, we decided to develop our own benchmarking framework tailored for the NIST LWC project. Also note that we started the development of our testbed in 2018, so some novel related work might not have been as mature at that time compared to how it is now.

Abbreviation	Requirement
REQ1	The framework shall have a modular design, such that support for additional hardware platforms can be integrated.
REQ2	The framework shall integrate a range of MCUs that feature different architectures and performance levels.
REQ3	The framework shall implement the possibility to publish and update measured results and test meta data.
REQ4	The framework shall support measuring the execution time, code size and RAM utilization of the implementations under test.
REQ5	The framework shall allow for an automated test procedure, no user interaction shall be required from the compilation step until the publication of the results.

Table 3.1: Crucial requirements for an LWC software benchmarking framework

Framework	Requirement	Fulfillment
FELICS-AE	REQ1	partly
	REQ2	yes
	REQ3	no
	REQ4	yes
	REQ5	no
SUPERCOP	REQ1	yes
	REQ2	no
	REQ3	yes
	REQ4	no
	REQ5	no
NIST	REQ1	yes
	REQ2	yes
	REQ3	no
	REQ4	no
	REQ5	no

Table 3.2: Comparison of required features of related benchmarking frameworks

3.2 Software Design and Implementation

The benchmarking framework should be designed to support the fulfillment of the previously engineered requirements. In terms of the software design that mainly concerns the modularity, automation and possibility to store, publish and update results. We opted for a simple design that acts in favor of these desired key properties. To be able to add and remove devices under test from a software perspective, we decided to split the instrumentation software in two categories. A general base software, which handles all common testing tasks, including the collection and storage of the results and a device-specific template firmware, that provides a runtime environment for the test cases. Furthermore, this firmware should be responsible for initializing required peripherals and preparing the MCU for the testing phase. The base software drives the test procedures and is responsible for managing the individual test instances for each device and implementation. This includes scheduling benchmarks and gathering their results in an automated manner.

A third component of the benchmarking framework consists of software that takes care of the publishing and updating of results and additional metadata. While this infrastructure is still part of the framework, it is not involved in the actual test preparation or execution. Instead, this component can be rather seen as a server-side add-on, which shares many properties with standard websites. However, the interface from the testing software to the web-based part needs to be defined in order to meet the requirements regarding result updates and automation.

From an implementation point of view, we followed the ideas elaborated during the design process. The MCU-specific template is written in C/C++, depending on the development kit available for the device. As mentioned previously, the template covers the initialization and startup phase of the individual MCU and provides boilerplate code that is merged with the implementation under test for each test case. The scripts that are managing the benchmarking procedures are the same for each MCU. In general, they implement all steps from loading a cipher implementation to publishing the gained results.

Every benchmarking run starts with an input ZIP file that contains one or more implementations of a cipher family. NIST dictated how a cipher submission shall be structured within the ZIP file [24]. This also concerns the structure of present implementations. Every implementation shall contain an `encrypt.c` file defining the encrypt/decrypt functions described in section 3.1. The C file and other additional source files shall be placed in the path `cipher_family/Implementations/crypto_aead/cipher_variant/implementation_variant/`, with `cipher_family` being the general name of the cipher, `cipher_variant` representing the instance of the cipher and `implementation_variant` describing the kind of implementation (e.g. `armv6m`). Moreover, every cipher variant has to provide a text file with fixed test vectors for different input plaintexts. NIST offers a tool to generate this file automatically such that it can be easily included in the submission archive. Figure 3.1 shows parts of the structure of a ZIP submission file that we take as an input for our testing framework. Since every submission to the NIST LWC competition has to comply with the regulations, we can rely on the folder

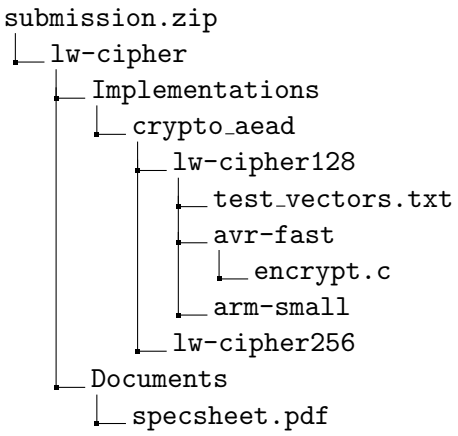


Figure 3.1: Predefined directory tree of a ZIP-archived NIST LWC submission

structure when using the archives as input files for benchmarking. Initially, we extract the archive and check if the submission includes an `encrypt.c` file and corresponding test vectors for each cipher variant. Then we apply some Python scripts to process the source files and instrument the MCUs testing procedures. Note that we are not making any changes to the submitted ZIPs, we only utilize them as input archives. In the next step, the framework fetches the source files for every cipher variant and tries to compile them for each target platform. During that step, the individual C/C++ templates and the implementation sources are merged. We utilize the API introduced in section 3.1 to interface the AEAD algorithms. In its standard configuration, the compiling tool tries to produce a binary for every combination of implementation and target firmware. The compilation procedure is monitored by a logging routine in case one wants to investigate why a compilation fails for a certain MCU.

As a next step, the binaries are flashed onto the devices under test and the benchmark procedures can be started. The flashing process and the communication setup is specific to the MCU and therefore part of the individual template. Nonetheless, all devices share the same interfaces and some test routines, which is why a generic `test_common.py` script provides the base class for the testing functions. We initialize a serial interface on each MCU and use that to communicate between the host system and the MCUs. We also define a simple protocol to signal which kind of data is transferred. This is required e.g. for marking the different categories of data from the test vectors. More details regarding this communication channel will be discussed in section 3.4.

Besides the execution of the test cases, also managing and timing the tests for various implementations on multiple platforms is an important part of the framework. It is especially important to be able to evaluate many implementations subsequently in an automated fashion. Therefore, we implemented a test scheduling script that monitors the execution of the tests. Once an implementation assessment is finished and the MCU under test is available, the script recognizes that and starts the flashing and testing procedure for the next compiled implementation on that platform. This service monitors

all platforms simultaneously in order to execute the test cases as efficient as possible. We also provide a graphical representation of the test/implementation queue via a simple web interface. This interface also supports opening log files and the rescheduling or restarting of single tests. It also can be used to monitor the progress of the current set of benchmarks.

Once a test has terminated, the results for the benchmarked metric are stored in a database, along with acquired metadata. Moreover, the tested implementation is committed and pushed to a public git repository and also the source code for the complete testing framework is available from public sources.[1,2] In addition to that, the measured test results are uploaded to a public website[3] which allows third parties to check and verify them. Furthermore, results for novel implementations can be included in a database and website such that an up-to-date snapshot of results can be provided throughout the duration of the NIST LWC project. By publishing the source code of the benchmarking tool and maintaining a repository of tested software implementations, we offer a high level of transparency. Due to the most parts of the testing routines being automated – from the compilation process to the updating of the result website – we support an efficient and repeatable benchmarking effort. To give rapid feedback regarding the performance of newly developed implementations, we additionally included a submission form on our website. Through that, programmers can send complying ZIP archives that are then automatically handled by our benchmarking framework. With that approach, we enable developers to verify their implementation strategies on different target platforms in an uncomplicated manner.

3.3 Test Setup

The hardware part of the setup consists of a host PC, a Saleae Logic Pro 16 logic analyzer and appropriate development boards containing the selected MCU cores. Every MCU needs to feature a serial Universal Asynchronous Receiver/Transmitter (UART) interface in order to communicate with the host system. In case any device does not offer that by default, external serial modules are applied. Power for the MCUs is provided via USB. We utilize the sigrok library to interface the logic analyzer in the speed measurements. This means any powerful enough logic analyzer that is compatible with this library can replace the Saleae Logic Pro 16. We include this device into our setup because it offers a stable sampling rate of 100 MHz when using five channels in parallel. We connect the logic analyzer to one GPIO pin on each MCU. This GPIO is then toggled once an encryption/decryption is started and again after its termination. Through that signalling, the logic analyzer can measure the time passed precisely and externally.

For flashing the binaries onto the target devices, we either connect via a JTAG interface (when available), or alternatively program the MCU platforms via the serial interface. We provide an overview over the appointed compiler versions, development kits and

[1] https://lab.las3.de/gitlab/lwc/candidates
[2] https://lab.las3.de/gitlab/lwc/compare
[3] https://lwc.las3.de

Software type	Tool	Version
Compiler (Uno)	gcc	5.4.0
Compiler (F1)	gcc	9.2.1
Compiler (ESP)	gcc	5.2.0
Compiler (F7)	gcc	7.3.1
Compiler (R5)	gcc	8.2.0
Framework	PlatformIO	4.3.3
Framework	STM32CubeMX	5.4.0
Interpreter	Python	3.7.3
Logic Analyzer Library	libsigrok	0.5.1
Debugger Software	openocd	0.10.0

Table 3.3: Overview of used software tools

additional libraries used in our framework in table 3.3. The compiler versions correspond to the variants used on the different platforms. Depending on the availability on each MCU, we developed our firmware with either PlatformIO or STM32CubeMX. Hence, we list the incorporated versions for both frameworks. To conclude the explanation of the architectural and implementation-based details, Figure 3.2 visualizes the communication model of the frameworks and its previously described parts.

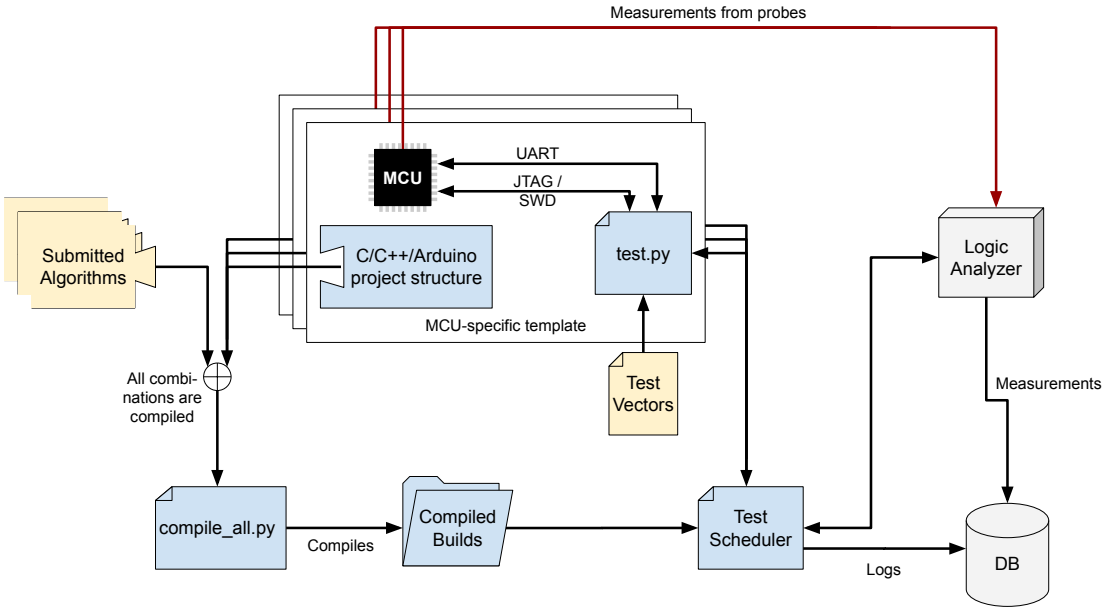


Figure 3.2: Core components and data flow of the test framework

In the following, some configuration details regarding the selected embedded platforms are introduced. At the time of writing this thesis, the framework supports five different

cores with four different architectures. We selected the devices in order to reflect a broad range of target platforms for different use cases. Even if some of the MCUs might seem a bit too powerful for LWC, benchmarking algorithms on them still results in an additional comparison on a different architecture and/or instruction pipeline. Note that due to its modular design, the framework can be simply adapted to support additional or different MCUs. On the hardware side, only a suitable development board, a serial connection and a GPIO pin are required. To support a new board in software, one needs to create a template firmware that initializes the device and acts as a runtime environment for the test cases. The easiest way to achieve that, is to change an existing template such that it fits the new device. We have already extended our initial set of platforms with a RISC-V-based chip. We decided to support the emerging architecture after our initial benchmarks, since we wanted to reflect the popularity of the chipset and provide benchmarking figures for the relatively new instruction set.

Arduino Uno R3. The Arduino Uno R3 [156] is an evaluation board for the ATmega328P 8-bit AVR chip, which is now sold by Mircochip Technology. With its maximum clock frequency of 16 MHz, a flash memory size of 32 KB, 2 KB SRAM and 1 KB EEPROM memory, it forms the lower end in our performance matrix. The device features a 2-stage pipeline and the AVR architecture offers 32 gerneral-purpose registers.

STM32F103C8T6 "bluepill". This lower-end ARM board [157] is known as "bluepill" or "blackpill" in the maker community. It is a cheap device, containing an ARM Cortex-M3 core. The 32-bit chip operates at a frequency of up to 72 MHz and includes 64 KB of flash memory. Its ARMv7 architecture has a 3-stage pipeline and supports the Thumb(-2) instruction set.

STM32 NUCLEO-F746ZG. This NUCELO development kit [158] is driven by a powerful 32-bit ARM Cortex-M7 core clocking at a maximum of 216 MHz. It features 1 MB of flash memory and 320 KB of SRAM, which puts it in the upper half of our performance matrix, together with the following two devices. The M7 chip integrates a pipeline with six stages and includes basic branch prediction.

Espressif ESP32 WROOM. This evaluation board [159] includes the 32-bit Xtensa LX6 MCU. It is running at 240 MHz and can store up to 4 MB of data in its flash memory. Moreover, it also implements 320 KB of SRAM. The LX6 core contains a 5-stage pipeline and supports 82 RISC instructions The ESP32 is a powerful and popular target device for various IoT projects, both in industry and in the private sector.

Sipeed Maixduino RISC-V 64. The Maixduino board [160] represents a high-power RISC-V MCU. It is built around the Kendryte K210 64-bit MCU. This core can run at a maximum frequency of 400 MHz and integrates 8 MB of flash storage. While this platform is probably too powerful for most software LWC use cases, it still can be

utilized as a proxy for the RISC-V architecture, where potential peculiarities of cipher implementations on RISC-V can be revealed.

3.4 Test Cases

Currently, our LWC benchmarking framework features three test cases, where three performance metrics are taken into account. We measure encryption/decryption timings, the binary size and the utilization of dynamic memory (RAM). The first two evaluations are carried out on all five MCU platforms and four different architectures, while the RAM measurement is only taken on one MCU. That is because early experiments during the development process have revealed that the dynamic memory usage is fairly constant across various platforms and even varies comparatively little in between implementations. Therefore, we conduct this test only on our most powerful ARM chip.

3.4.1 Execution Time

In this benchmark, we measure the encryption/decryption speed of the implementations. We use the input/output data from the test vector file supplied with each submission. This approach has multiple advantages. First, we are provided with a well-formed dataset, which has to be part of every algorithm package. Second, this dataset is generated by a NIST tool and features a variety of input sizes and edge cases for plaintexts/ciphertexts. Third, while the test vectors are useful for benchmarking purposes, they can as well be utilized to verify the correctness of the implementation and the communication during the test execution.

```
1 Key   = 000102030405060708090A0B0C0D0E0F
2 Nonce = 000102030405060708090A0B
3 PT    = 0001020304050607
4 AD    = 00010203
5 CT    = 60267634D1D37D06582A9A50A0EBDC62
```

Listing 3.2: Excerpt of a test vector file with example data for an input to the speed benchmark

We measure the time for the encryption/decryption of every test vector available from the text file. It contains 1089 plain- and ciphertext pairs with a key, a nonce and optionally AD. The variety and number of test vectors are defined by NIST. In case the test vector features AD, the benchmark takes the tag generation and verification into account when measuring the timings. Listing 3.2 shows how the test vector file is structured. In the test procedure, we send the test vector data one by one from the host PC to the device under test. When the data is transmitted, we signal the required action (e.g. encryption) to the MCU via a serial control message. Upon this signal, the encryption is started and the time is taken by utilizing the logic analyzer and pin toggling, as described in the previous section. The speed is measured for every individual test vector. The vectors cover different message sizes, from the empty plaintext without AD up to 64 bytes length for both inputs. In parallel to the timing benchmark, we also

check the produced ciphertexts and compare them to the available data in the test vector file. In this way, the correct transmission over the serial line and the right behavior of the tested implementation is verified.

3.4.2 Code Size

To compare the binary sizes of the LWC implementations, we compile a firmware blob for each MCU. We use the recommended compilation flags from NIST and automatically integrate the cryptographic routines into the templates to create a flashable binary. Then, the `du` system command is executed to display the code size. Note that we also compile one binary without any included AEAD algorithm to gain knowledge about the code size of the provided runtime environment. This minimal size can then be subtracted from the total size of an implementation binary to get the raw size of the LWC code.

3.4.3 RAM Utilization

Before we run an algorithm in the RAM test case, we fill the whole dynamic memory with a known pattern. This can be achieved by connecting the evaluation board to the debug interface. In the following, the encryption/decryption routines are run with the integrated algorithm under test. After the processing of the test vector data has finished, the RAM sections are analyzed. We check which of the prefilled memory regions have been overwritten and compare the state of the memory to the one before the algorithm execution. We observe how many consecutive memory segments remain unchanged and calculate the used memory based on that information. Again, we consider the memory consumed by the template alone, in order to compare the RAM utilization of the implementations only.

4 Benchmarking NIST LWC Implementations

The core ideas presented in this chapter have already been published in S. Renner, E. Pozzobon, and J. Mottok. The final round: Benchmarking NIST LWC ciphers on microcontrollers. In International Workshop on Attacks and Defenses for Internet-of-Things at ESORICS*, pages 1–20. Springer, 2022*

In the following chapter, we will present benchmarking results for various LWC implementations obtained with our framework. Before providing numbers for different test cases, we describe how we aimed for a fair performance assessment of the individual candidates and why this is sometimes not entirely possible due to external factors. Moreover, we give some insights regarding the AES-GCM implementation that we compare the LWC candidates to and how we took into account the size of our MCU templates in the memory consumption tests.

AES-GCM represents a state-of-the-art algorithm for authenticated encryption to date. It is a well-studied and standardized cipher that is utilized also in industry. Since the NIST LWC initiative has been started in order to find a more lightweight alternative to AES-GCM, a performance evaluation of the candidates should include a comparison to the de-facto standard. In general, we do not contribute our own implementations of cipher algorithms. We also followed this principle with AES-GCM. The implementation that we include, is stripped from a popular TLS library, Mbed TLS. This C library is part of the IoT operating system Mbed OS and complies with NIST SP800-38 [89]. We did not change the optimization level of the implementation or any code related to the AES-GCM algorithm. However, we integrated some wrapper functions to make it fit the proposed NIST LWC API. In this way, a seamless integration of the cipher could be ensured.

Moreover, we set some implementation flags, such that the AES-GCM variant matches best the design goals of LWC. We add `MBEDTLS_AES_ROM_TABLES` and `MBEDTLS_AES_FEWER_TABLES` to the configuration because these options are typically used in embedded systems with low memory resources. The former flag causes that the AES round constants and the S-Box are placed in the ROM, instead of being initialized in the RAM, the latter configuration reduces the amount of stored tables in the ROM, which is advantageous regarding the binary size. With this setting, we aim to bring the AES-GCM implementation closest to what would be expected from a lightweight cipher.

4.1 How to Support a Fair Evaluation

It is important to pay attention to various potential issues when benchmarking and ranking cryptographic algorithms. This is because the performance of a cipher is not only determined by its design, but also by numerous other factors. For example, the focus of one candidate might be directed towards a specific use case, where one metric (e.g. the binary size) might be very crucial, but another one (e.g. encryption speed) has less or no priority. Other properties that some LWC candidates specialize in are the performance for short messages (e.g. eight bytes) or the (inherent) security against certain side-channel attacks.

While these factors are heavily influenced by design choices, also implementation differences can greatly influence performance. Depending on the programming strategy, an implementation can be tailored to fit a certain test case. A more balanced approach can result in similar performance in a wider range of metrics. This comes down to the fact that many performance goals contradict each other. For example, if an implementation is optimized for speed, that usually means it has to make use of lots of precomputed values, which in the end results in a larger binary size. Furthermore, optimizations cannot only target certain metrics but are often directed towards specific platforms. In the LWC competition, this means some cipher implementations are provided (partly) in assembly language, e.g. for ARM- or AVR-based embedded controllers. Typically, these implementations then perform better than standard C implementations, but on the other hand they can of course only run on dedicated MCUs.

Any optimization usually limits the flexibility of an implementation. No matter if a cipher instance is tailored for a specific platform, metric or use case, while the optimization causes a performance boost in that one category, it makes it less efficient in other disciplines. Depending on the project in which lightweight authenticated encryption is required, either a highly optimized or a more generic implementation might be the best choice.

Note that we present benchmarking results for 3rd round NIST LWC candidates only here. That is because on the one hand, the already eliminated ciphers are no longer relevant for the standardization effort. On the other hand, a variety of different implementations is available for (almost all) finalists since these algorithms have been studied for a couple of years, at least since their introduction into the NIST LWC competition. Moreover, comparing the finalists to e.g. 1st round candidates is not beneficial since often only reference or less optimized implementations are available for already eliminated ciphers. On top of that, the reasons for a cipher exclusion can be manifold. While some 1st round candidates might be able to compete with some finalists regarding performance, they might have security issues which have been exposed during the NIST LWC project. However, we would like to mention that the ciphers Ascon, GIFT-COFB, KNOT, Sparkle, TinyJAMBU and Xoodyak delivered the overall best performance in our 1st and 2nd round benchmarks. Regarding the KNOT cipher, a number of attacks have been published, which can weaken the security of the algorithm [162, 163, 164]. NIST does not reveal in detail why a certain cipher has been selected as a 3rd round candidate. However, they state "ASCON, GIFT-COFB, ISAP, PHOTON-Beetle, Ro-

mulus, SPARKLE, TinyJAMBU, and Xoodyak demonstrated performance advantages over NIST standards in software benchmarks [...]. For hardware applications, the finalists ASCON, Elephant, GIFT-COFB, PHOTON-Beetle, Romulus, TinyJAMBU, and Xoodyak demonstrated performance advantages over NIST standards. ISAP provides promising features for applications requiring side-channel resistance." in their status report on the second round of the NIST LWC standardization process [165].

To maintain the neutrality of our testing efforts, none of the finalist implementations we included in these benchmark experiments has been contributed or altered by us. This means that we do not have any influence on the amount, quality or optimization level of the tested implementations. We only gather every publicly available implementation and save the tested and submitted implementations in our public databases. We do not evaluate the same number of implementations for every cipher. This number depends on the popularity of a candidate and the maintenance of the designer team. There are candidates for which we have access to 100+ implementations and others, where we only tested the initial implementation submission package with less than ten implementations. Obviously, the variety and amount of (optimized) implementations affects the maximum performance a candidate can provide in the different test cases. Figure 4.1 gives an overview of how many implementations we have tested for each candidate and which optimization focus or target architecture these implementations aim for. We can see that there are more than twice as many implementations available for Ascon as there are for the candidate with the second most implementations, Romulus. While we have access to 111 and 50 implementations for these ciphers, respectively, Isap and Sparkle offer between 30 and 40 implementations. Xoodyak, PHOTON-Beetle and TinyJAMBU deliver more than ten but less than 20 variants; Elephant, GIFT-COFB and Grain-128AEAD provide less than ten implementations. Analyzing the programming languages of all implementations, one can derive that almost 50% of the variants have been programmed in C. The most popular MCU target for assembly optimizations is ARM, while Xtensa and AVR are on a similar level but far less prominent. Combining assembly accelerations with C functions for less time-critical operations is the second most common implementation strategy. Table 4.1 further specifies which kind of (optimized) implementations have been available to us for every cipher. The number of available implementations together with their optimization level can have an impact on the measurable performance of a candidate. However, it is the designers' responsibility to make sure the capabilities of a cipher are best represented through their implementations.

In order to support a fair evaluation, we maintain the same benchmarking process for any implementation which is in line with the requirements discussed in section 3.1. Moreover, we include every available implementation of any NIST LWC candidate, complemented by the AES-GCM implementation, in our benchmarking.

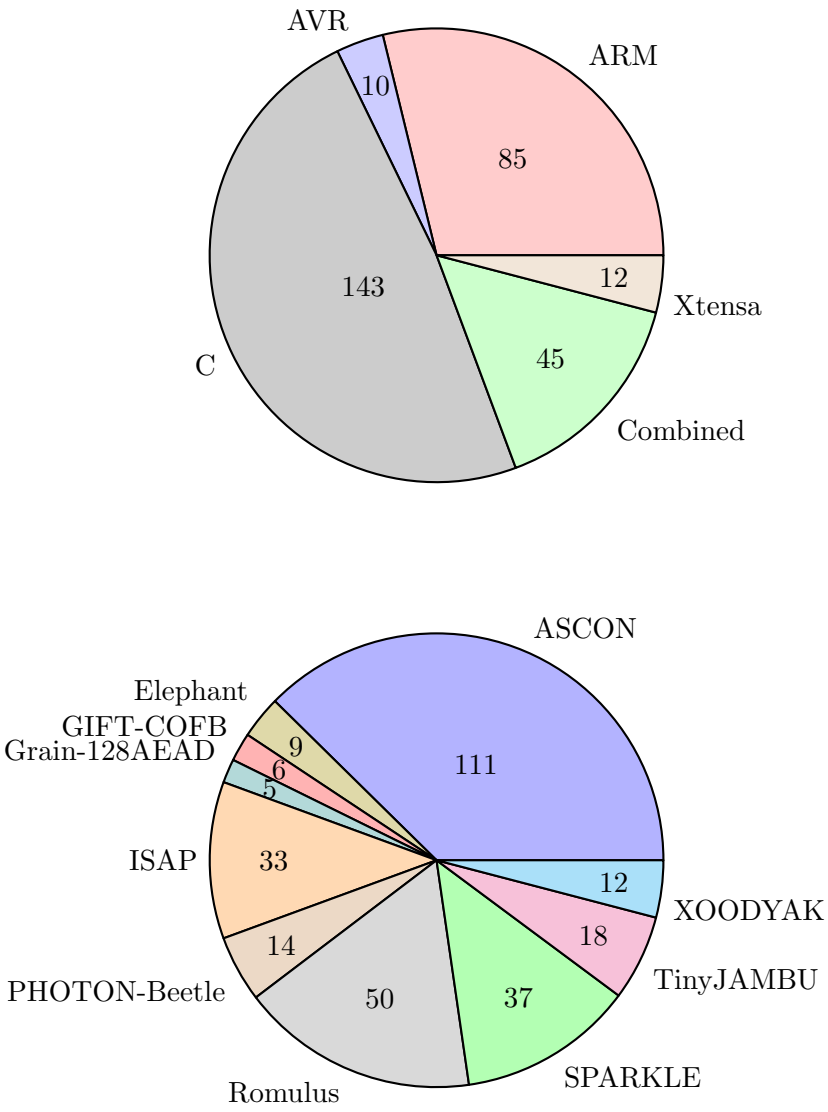


Figure 4.1: Number of implementations grouped by optimization focus vs. number of implementations grouped by candidate

Cipher/Implementation	C	ARM	AVR	Xtensa	Combined
Ascon	yes	yes	no	yes	yes
Elephant	yes	no	no	no	yes
GIFT-COFB	yes	yes	no	no	yes
Grain-128AEAD	yes	no	no	no	yes
Isap	yes	yes	no	no	yes
PHOTON-Beetle	yes	no	no	yes	yes
Romulus	yes	yes	no	no	yes
Sparkle	yes	yes	yes	no	yes
TinyJAMBU	yes	no	no	no	no
Xoodyak	yes	yes	yes	no	yes

Table 4.1: Availability of differently optimized implementations of the LWC finalists (*combined* includes C implementations with building blocks accelerated with assembly)

4.2 Performance of Non-protected Implementations

In the comparisons, always the best result of each candidate is considered. This means that for each test case we sort all implementations of every cipher by the evaluated metric and then include the best competitor into the corresponding plot. Note that this might lead to the inclusion of different implementations of the same cipher for different test cases. With that strategy, we ensure that we consider the most optimal implementation available for any candidate and test case. Also, any implementations included are representations of *primary* cipher variants. We do not consider any other alternative variants because according to NIST these are not required to meet all defined requirements, e.g. in terms of security boundaries.

4.2.1 Speed

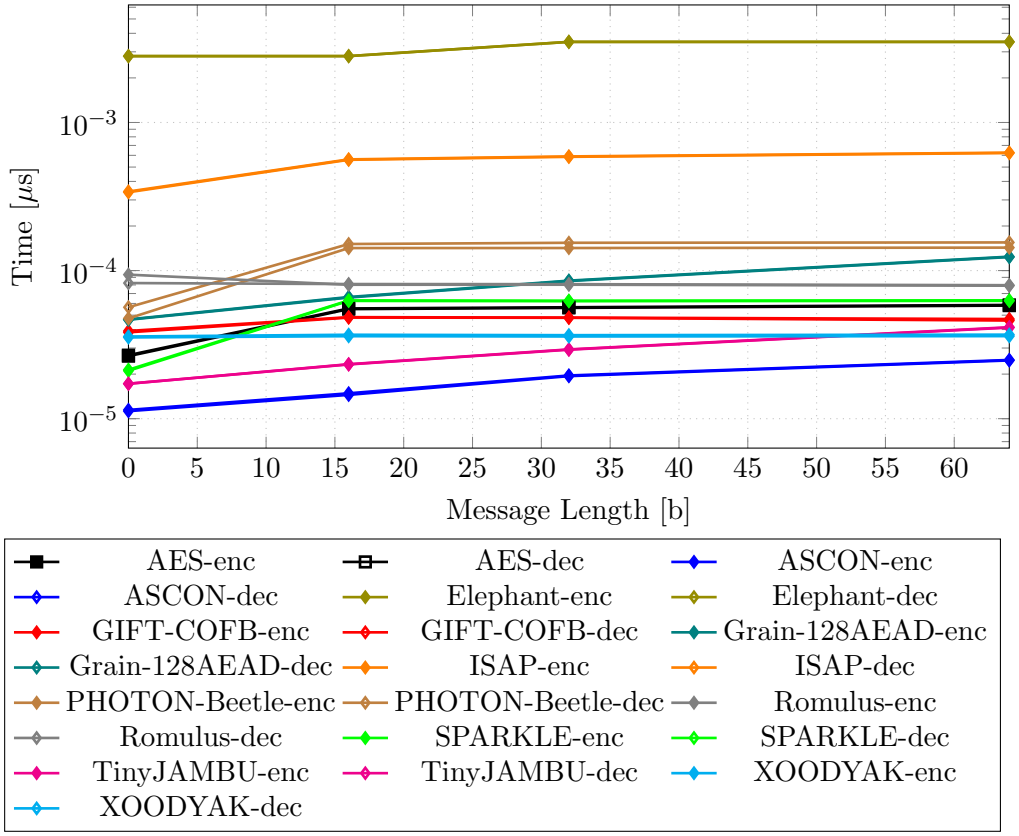


Figure 4.2: Differences in encryption and decryption speeds (on the ESP32, for including authentication, `X-enc` represents the encryption/authentication, `X-dec` represents the decryption/verification with cipher `X`)

For the latency test case results, we show the average encryption time of all 1089 test vectors and timings for distinct sizes and types of input data. The plots feature encryption/authentication only, since we are dealing with symmetric ciphers. While processing the data set, we also measure the time for decryption/verification, however there is no reproducible difference in encryption and decryption timings. This is illustrated in figure 4.2, where we plot both timing measurements for an empty input (0 bytes plaintext, 0 bytes AD) and for plaintext/AD combinations of 8/8, 16/16 and 32/32 bytes of data.

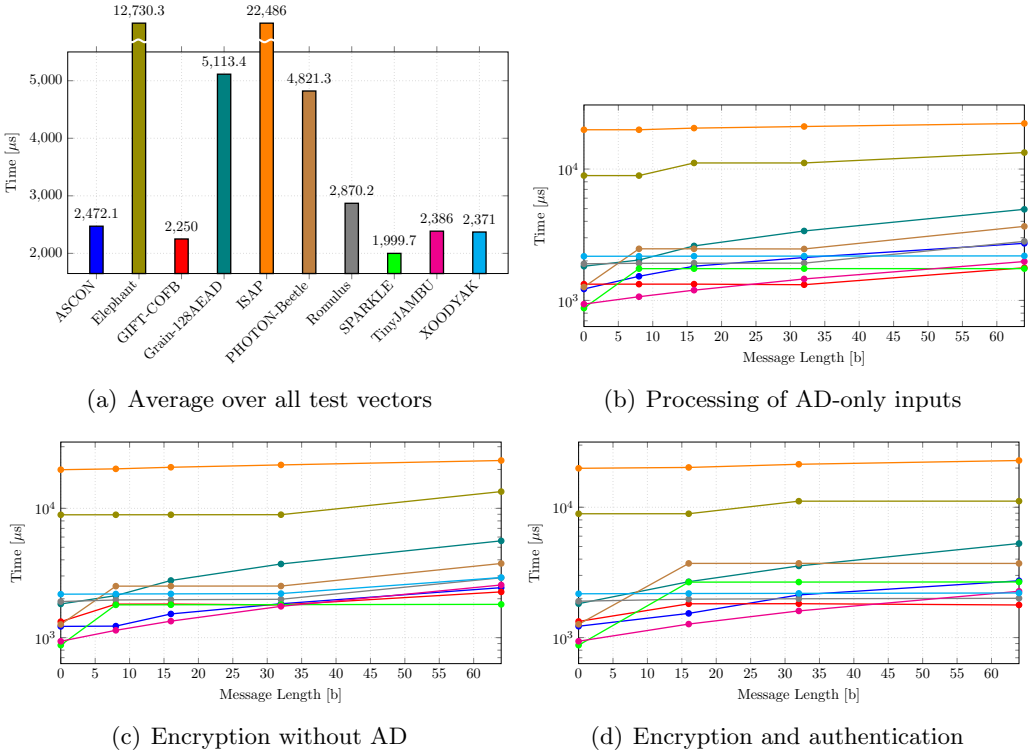
Our speed measurement results are represented by four subplots per test platform (see e.g. figure 4.4). The upper left plot in the group always shows the average encryption timings on the mentioned MCU. Since the test vector input data contains many different message sizes and an average might be unfair for ciphers optimized for specific messages, we include measurements for selected input sizes. This way we can estimate how the speed of a cipher develops with growing plaintexts or altered input lengths in general. In all plots, we give the processing timings in microseconds (μs). The top right plot shows

the speed for processing 8, 16, 32 and 64 bytes of AD without any additional plaintext. The bottom left depiction covers the timings for encryption of the same plaintext sizes, this time without any AD involved. In the lower right plot we show the latency for encryption and authentication of 16, 32 and 64 bytes of data, whereas half of the data is encrypted and half of the data is treated as associated data. Every plot includes the time it takes to process an empty input. So while the (b) and (c) plots magnify the performance of the implementations for one part of the AEAD mode only, in (d) a combined measurement is taken. Figure 4.3 gives the color codes for the graphs of each cipher in the following plots. We do not include the runtime of NoCrypt in our speed comparisons, since its average latency is only roughly 5% the amount the fastest cipher requires. The NoCrypt measurement becomes more relevant in the memory test cases.

Figure 4.3: Legend of cipher color codes for latency plots

In figure 4.4, the speed benchmarking results obtained on the Arduino Uno platform are shown. Note that we do not show a comparison to AES-GCM on this platform, since the binary size of the implementation is too large to fit on the AVR chip. Generally, six ciphers form the top group regarding the average encryption speed: Ascon, GIFT-COFB, Romulus, Sparkle, TinyJAMBU and Xoodyak. Sparkle ranks first here, followed by GIFT-COFB, Xoodyak, TinyJAMBU and Ascon. More than twice as slow as the top candidate are PHOTON-Beetle and Grain. Elephant and Isap are the slowest competitors on this platform. This also applies when looking at the measurements for individual input sizes. However, one has to remember that Isap includes basic protection mechanisms against fault and side-channel attacks by design, which makes it less efficient regarding speed. For Elephant, we only had nine implementations available, none of them being heavily optimized for embedded platforms. It is possible that the lack of highly optimized code amplifies its bad ranking in this test case.

If we analyze the timings for different input sizes, a couple of interesting observations can be made. Although TinyJAMBU and Ascon are slower than Sparkle on average, they beat the winner for small message sizes (8 and 16 bytes) regardless of the combination of AD and plaintext to encrypt. Moreover, we can conclude that the timings for some ciphers remain almost constant for various message lengths (see e.g. Xoodyak), while they gradually increase for others (e.g. for TinyJAMBU). The speed differences of Elephant and Isap in comparison to the rest are still visible in the subplots (b), (c) and (d).



(a) Average over all test vectors

(b) Processing of AD-only inputs

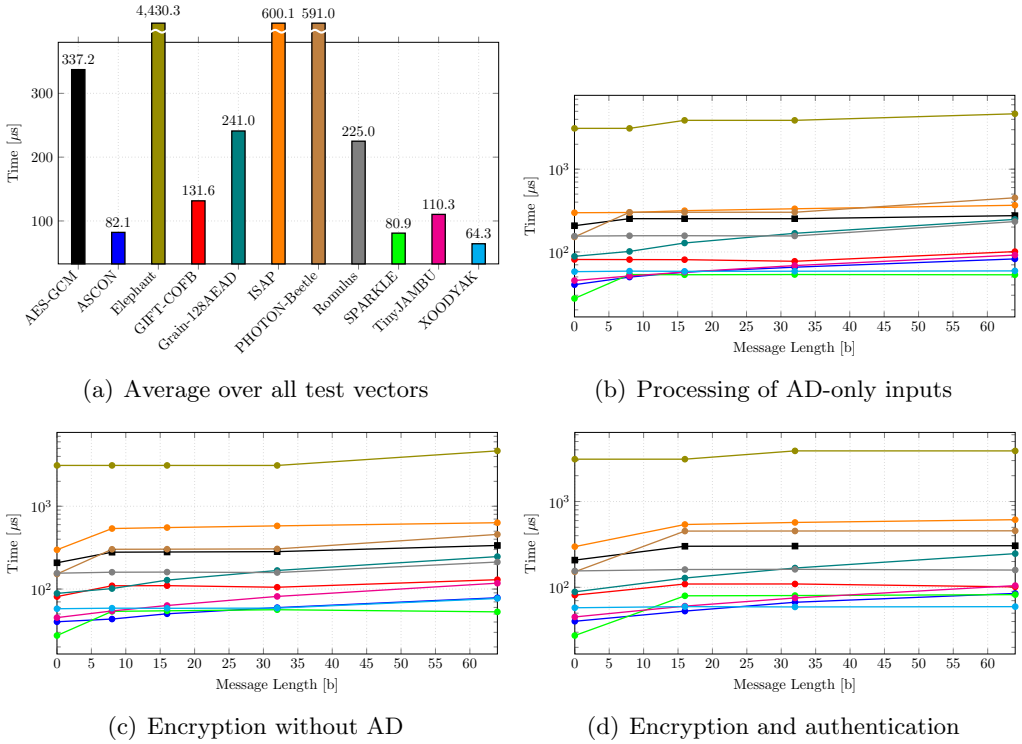
(c) Encryption without AD

(d) Encryption and authentication

Figure 4.4: (Average) encryption timings for different inputs on the Arduino Uno

We can see similar patterns again on the STM32F103 (compare figure 4.5). On the low-powered ARM platform, Ascon, GIFT-COFB, Sparkle, TinyJAMBU and Xoodyak reach the highest average speed. Elephant, PHOTON-Beetle and Isap rank the worst in this test. AES-GCM is slower than most LWC ciphers in this test, reaching the eighth out of eleven spots. The on average fastest candidate is an implementation of Xoodyak, with Sparkle and Ascon being only slightly slower.

The results for different message sizes and modes generally support the results for the average timings. However, again for short messages, the fastest candidate is outperformed by some of its successors (Ascon, TinyJAMBU or Sparkle) regardless of the input type. This is due to the encryption time of Xoodyak being very constant for different input sizes, while it gradually rises (but starts slightly lower) for some of its competitors. The nearly constant performance of Xoodyak for our tested input sizes origins from the design of the cipher. The big 48-byte state, paired with an equally sized permutation and a 44-byte rate, allows the cipher to process larger inputs without any queuing. As long as the plaintext size does not exceed the state size, Xoodyak can directly fit it into the state permutation. This is also reflected by the noticeable increase in latency for the 64-byte plaintext input in subplot (c). In comparison to the LWC candidates, AES-GCM ranks in the middle to middle-end of the field, also in the detailed analysis for various input lengths.



(a) Average over all test vectors

(b) Processing of AD-only inputs

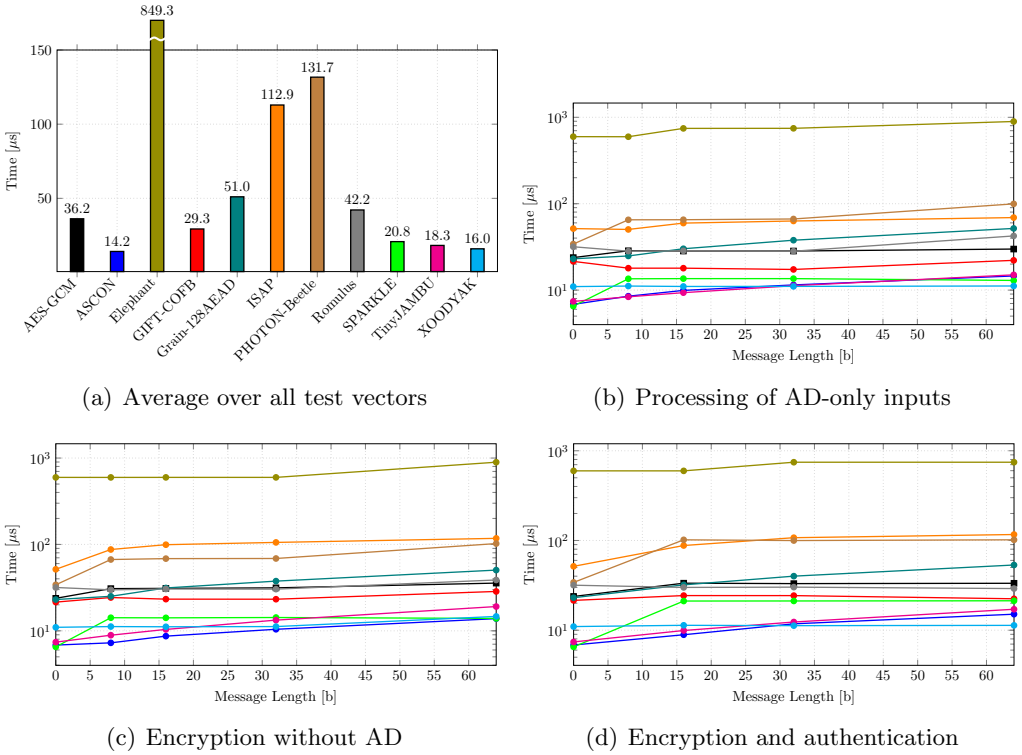
(c) Encryption without AD

(d) Encryption and authentication

Figure 4.5: (Average) encryption timings for different inputs on the STM32F103

In figure 4.6, the measured timings on the STM32F746 are given. First off, it is noticeable that AES-GCM performs better on the higher-powered ARM platform than on the STM32F103. Regarding the average, it is the sixth fastest cipher, meaning it beats five LWC candidates in this benchmark. Besides that, some patterns repeat throughout the platforms. Ascon ranks first here but is tightly followed by Xoodyak, TinyJAMBU, Sparkle and GIFT-COFB. Elephant is the slowest competitor, Romulus and Grain form the middle, Isap and PHOTON-Beetle reach similar but slower speeds than all others but Elephant.

There are no significant anomalies in the assessment of distinct input vectors. Ascon and TinyJAMBU are the most efficient ciphers for shorter messages. Xoodyak and Sparkle deliver nearly constant processing timings across the selected inputs. While this results in a slight disadvantage for small inputs, it leads to an increased performance on average.



(a) Average over all test vectors

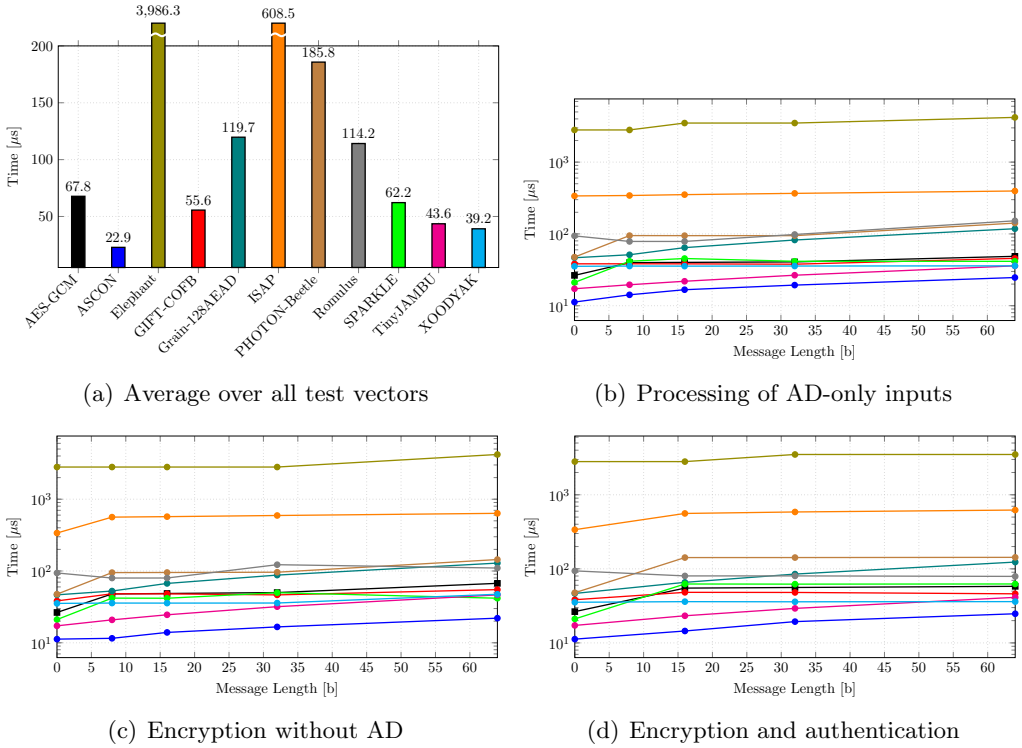
(b) Processing of AD-only inputs

(c) Encryption without AD

(d) Encryption and authentication

Figure 4.6: (Average) encryption timings for different inputs on the STM32F746

The results on the ESP32 (see figure 4.7) are generally comparable to the numbers obtained on the STM32F746. Again, Ascon is leading, with Xoodyak, TinyJAMBU, GIFT-COFB and Sparkle occupying the following spots. AES-GCM ranks sixth like before, but is only marginally slower than Sparkle on average. Isap and Elephant perform the worst, however Isap is still six times as fast as Elephant but more than 26 times slower than Ascon. The leading implementation is the fastest for all message sizes and types on this platform, TinyJAMBU is more efficient than Xoodyak for small inputs. AES-GCM delivers fairly constant timings for all test vectors, which are almost identical to the measurements taken for Sparkle. Once more, the performance of Xoodyak remains at the same level for various inputs, while the encryption/authentication time increases with the message size for Ascon and TinyJAMBU.



(a) Average over all test vectors

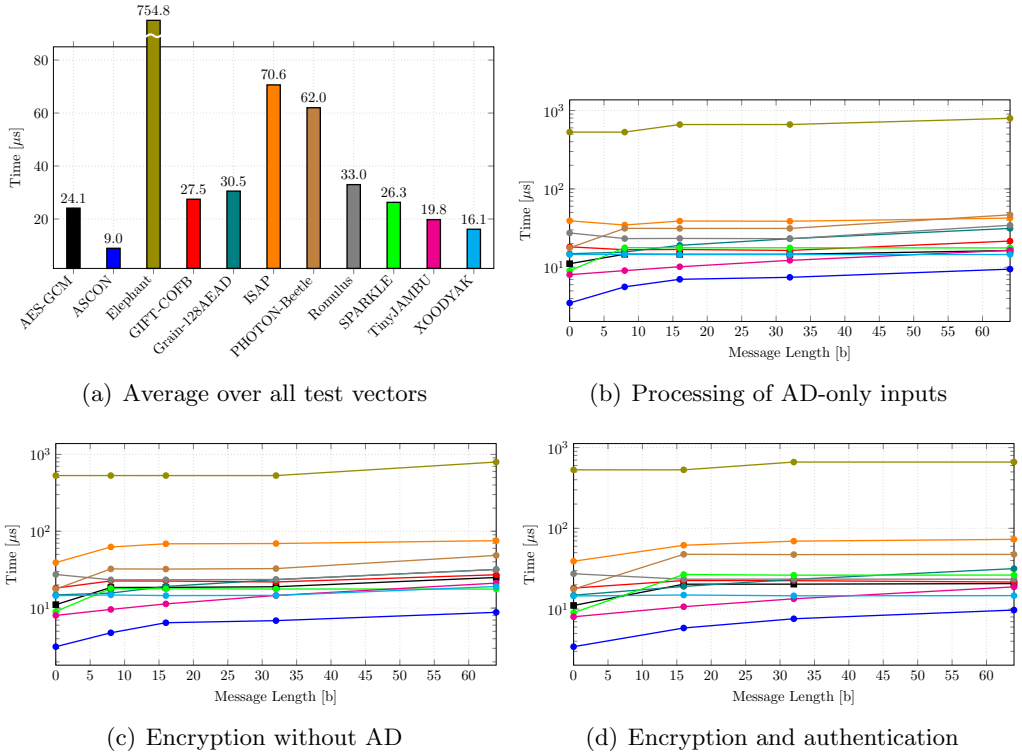
(b) Processing of AD-only inputs

(c) Encryption without AD

(d) Encryption and authentication

Figure 4.7: (Average) encryption timings for different inputs on the ESP32

In figure 4.8, we provide the speed benchmarking figures for the Sipeed Maixduino. On the 64-bit RISC-V platform, Ascon performs encryption and authentication most efficiently. Xoodyak and TinyJAMBU are again placed in the top group. However, on this MCU, AES-GCM beats Sparkle and GIFT-COFB regarding average speed and therefore ranks in the fourth overall spot. Elephant covers the last place, with Isap and PHOTON-Beetle setting as the second and third to last ciphers. In this benchmark, Ascon is the fastest cipher for any highlighted input size/type combination. TinyJAMBU is once again more efficient in processing short messages than Xoodyak. GIFT-COFB and Grain deliver similar speeds on the Maixduino, Romulus is slightly slower than this pair.



(a) Average over all test vectors

(b) Processing of AD-only inputs

(c) Encryption without AD

(d) Encryption and authentication

Figure 4.8: (Average) encryption timings for different inputs on the Maixduino

4.2.2 Binary Size

For the code size measurement, we try to compile each available implementation for every of our testing platforms. Obviously, not every implementation will compile on every MCU due to e.g. use of architecture-specific assembly language or binaries exceeding the memory capabilities of the device. As already mentioned beforehand, this occurs for the AES-GCM implementation from Mbed TLS on the Arduino Uno. To take the code size of the individual runtime environment of the platform into account, we compile our template once without any code for encryption/decryption. The size of this NoCrypt binary can later be compared to the size of the compiled blob of every firmware including an LWC implementation. In that way, we observe the code size overhead introduced by the tested AEAD implementation.

In figure 4.9, we introduce the code size measurements taken on the Arduino Uno. We see Ascon and Xoodyak being in the top half, similar to what we observed in the speed benchmarks. However, both are beaten by an implementation of PHOTON-Beetle and Xoodyak is beaten by Isap. Romulus ranks last, with TinyJAMBU and Grain forming the lower performing end of the rest. It is worth noting that the best implementation of PHOTON-Beetle consists in code written in assembly, which has especially been optimized for code size on AVR platforms.

Comparing the binary sizes on the STM32F103 (see figure 4.10), one can analyze that Ascon represents the most efficient implementation, while TinyJAMBU, GIFT-COFB, Sparkle and Xoodyak form the rest of the top half. Isap leads the second half, in which PHOTON-Beetle is second to last and Elephant ranks last. AES-GCM covers the third to last overall spot regarding code size on the Cortex-M3 MCU.

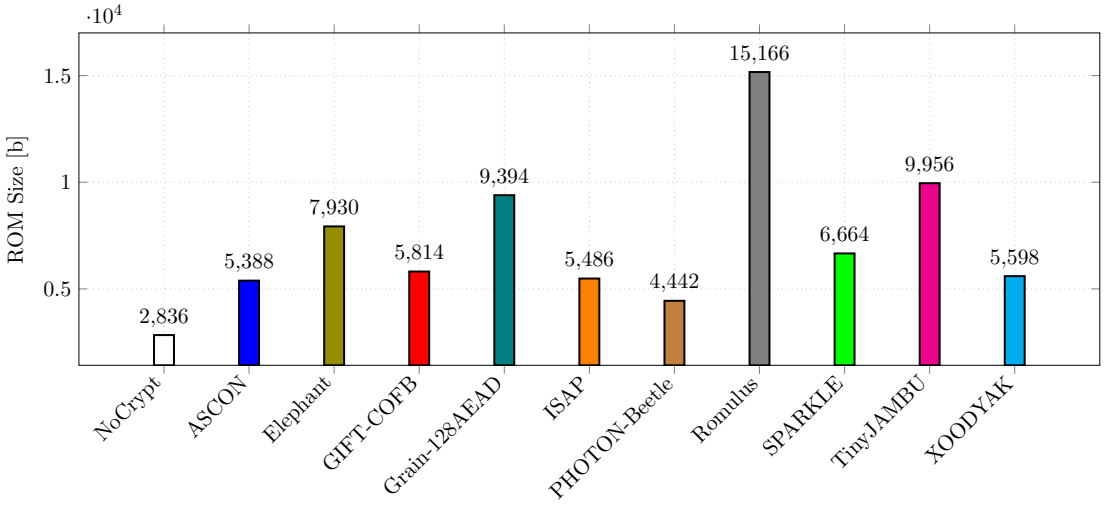


Figure 4.9: Code size measurement results on the Arduino Uno

Very similar rankings can be drawn from the results on the STM32F746. Again, Ascon provides the smallest implementation, closely followed by TinyJAMBU. The three worst candidates in this test are PHOTON-Beetle, Elephant and AES-GCM. On this platform, Isap is part of the leading half, ranking at the fourth spot.

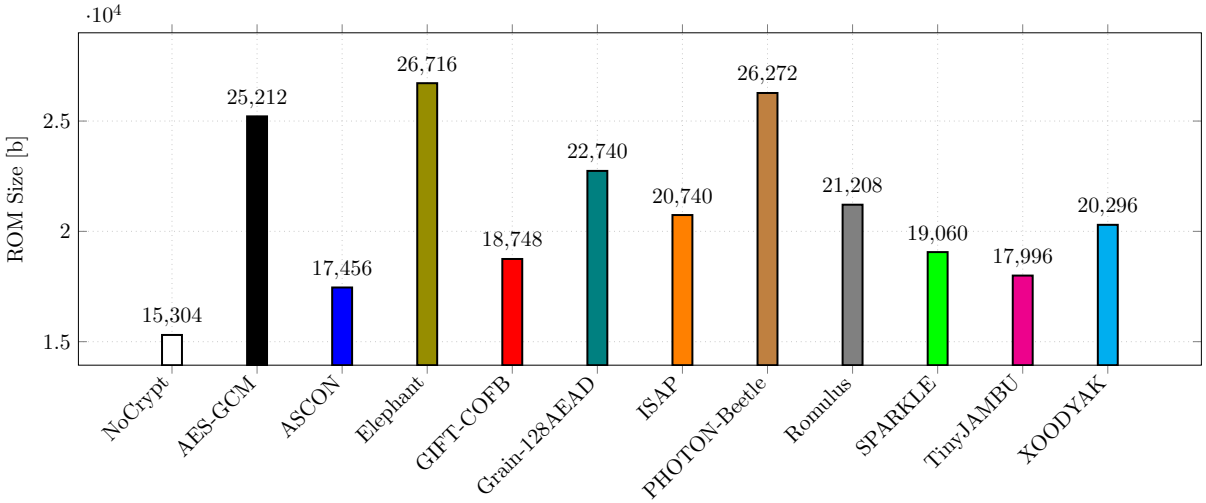


Figure 4.10: Code size measurement results on the STM32F103

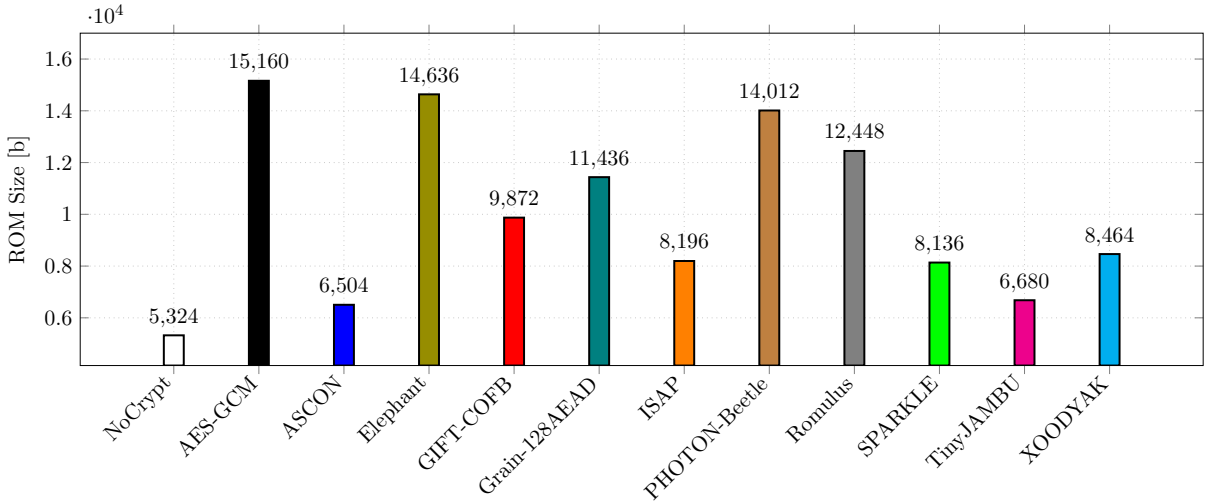


Figure 4.11: Code size measurement results on the STM32F746

On both the ESP32 and the Maixduino (see figures 4.12 and 4.13), Ascon features the most efficient implementation, with TinyJAMBU being very close to the leader. On the RISC-V and the Xtensa architecture, Isap, Sparkle and Xoodyjak are members of the upper ranking half and AES-GCM represents the largest implementation. Also, the rest of the ciphers rank very similarly on these two MCUs, which leads to the two plots being almost identical, qualitatively.

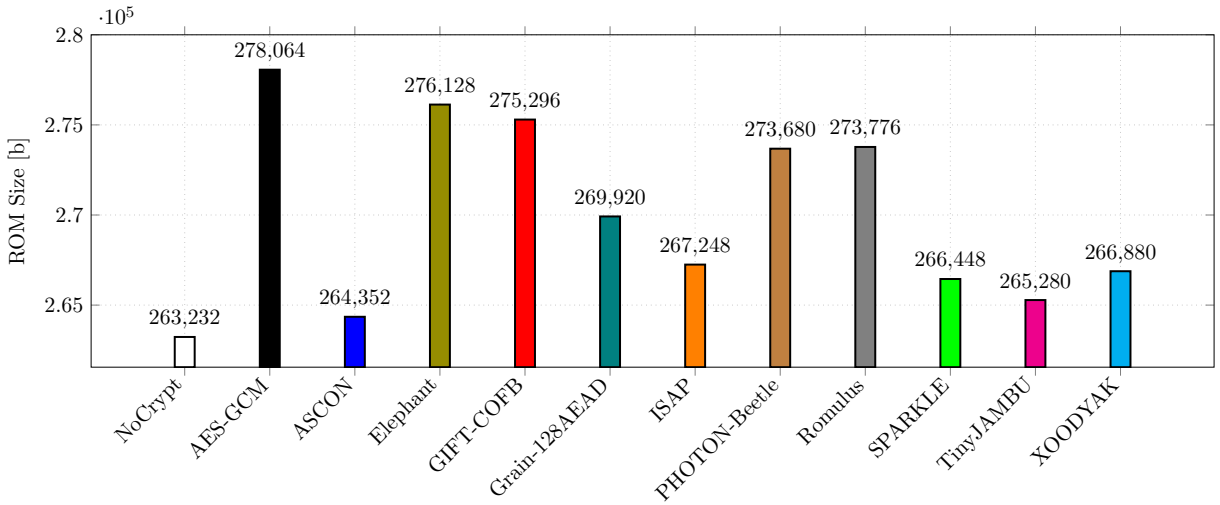


Figure 4.12: Code size measurement results on the ESP32

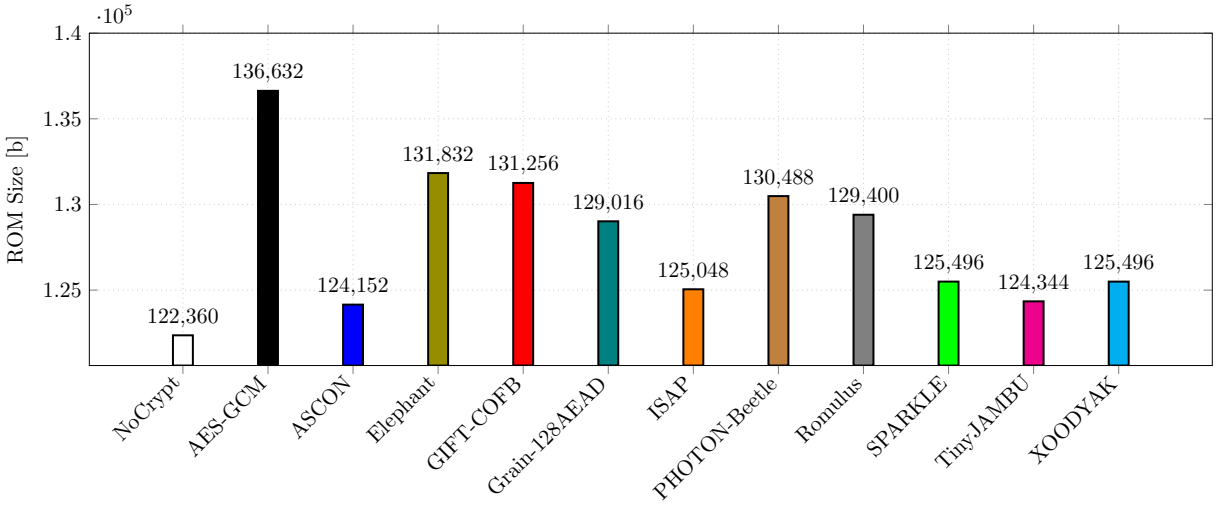


Figure 4.13: Code size measurement results on the Maixduino

4.2.3 RAM Utilization

To measure the RAM utilization of an algorithm correctly on the STM32F746, we check the dynamic memory consumption of every MCU template without any integrated cryptography. This is achieved by utilizing the simple memcopy operation implemented in our NoCrypt routine. Then – as we do for the code size measurement – we test the RAM utilization of every implementation and compare the consumption of the NoCrypt template to the measurements obtained for the LWC candidates.

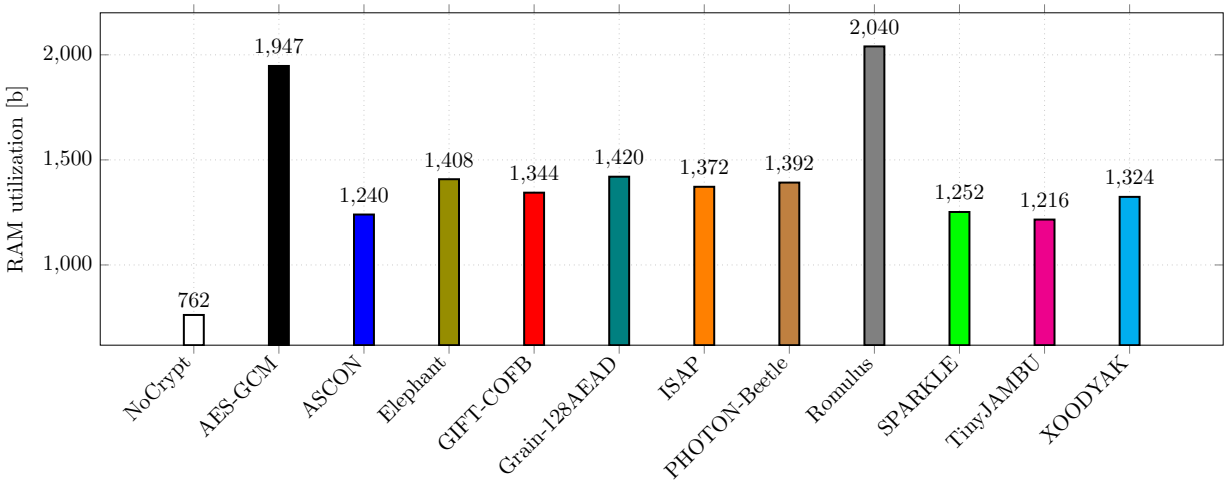


Figure 4.14: RAM utilization measurement results on the STM32F746

We present the results of this test case in figure 4.14. TinyJAMBU consumes the least amount of RAM, while Ascon, Sparkle and Xoodyak are following. Overall, the RAM utilization of the ciphers does not differ as much as the measurements in the speed benchmarks. However, there are two exceptions: AES-GCM and Romulus represent the two worst implementations in this test case and consume 60% and 68% more dynamic memory than TinyJAMBU, respectively. The third to last ranked candidate, Grain, only uses 17% more RAM than the leading algorithm.

Finally, we provide an overall comparison of the average timings and code sizes measured across platforms. In figures 4.15 and 4.16, we show how the performance of the NIST LWC finalists compares to AES-GCM. We normalized the measurements taken on four different platforms (all but the Arduino Uno) regarding the results for our AES implementation on each platform. This means, we set the average encryption/authentication time and code size of AES as our baseline (see the dashed line at $y = 1$). If an LWC implementation performed better than AES on a platform, this is indicated by a scaled value which is smaller than 1. A value $y = 0.5$ would mean that this candidate used half the resources (time or memory) in comparison to AES. In case an implementation achieved worse numbers than AES, this is shown by a value $y > 1$. An identical performance as AES would be reflected by a value $y = 1$. For each cipher, four dots for the four platforms are included in the plot. In figure 4.15, the value is cut off at $y = 2.75$, even if a cipher might perform more than 2.75 times worse than AES. This means, if a dot is placed on this end of the y-axis, the corresponding cipher performs **at least** 2.75 times worse. We do this to allow for a clearer depiction of the more relevant numbers in a lower range.

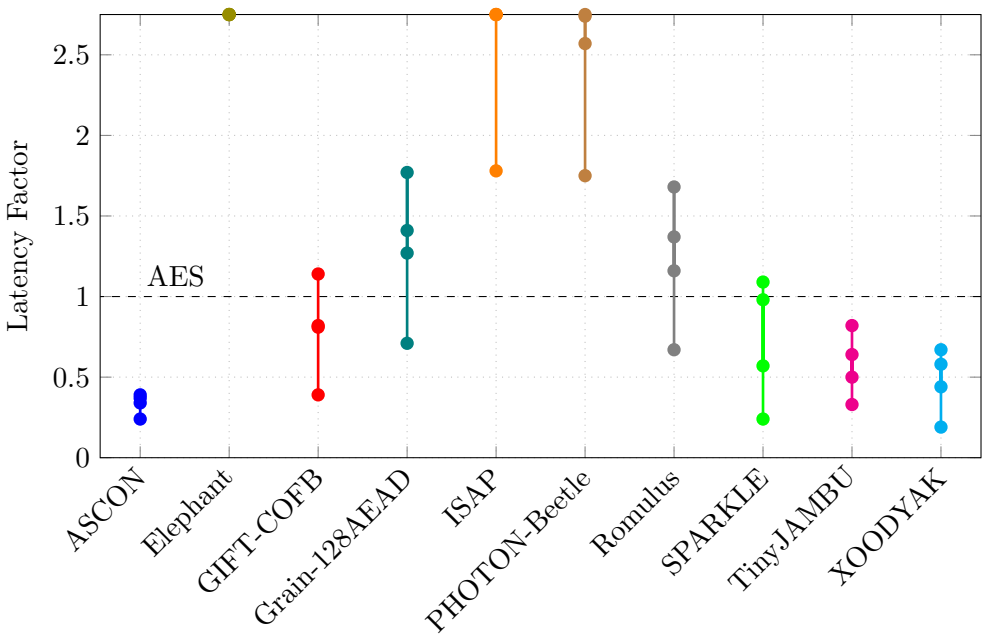


Figure 4.15: Normalized cross-platform latency of NIST LWC ciphers compared to AES-GCM

Obviously, the interpretation of the two concluding figures is identical to the previous per-platform discussion. Here, we just offer another cross-platform display of the results. We can, however, directly observe that Ascon, TinyJAMBU and Xoodyak perform better than AES in any of our test cases. Moreover, we see that GIFT-COFB, Grain-128AEAD, Romulus and Sparkle are sometimes better and sometimes worse than AES-GCM regarding speed. The remaining candidates are always slower than the current industry standard, regardless of the platform (see figure 4.15). Regarding code size, only Elephant and PHOTON-Beetle perform (partly) worse than AES-GCM. All other finalists achieve a lower code size on every tested architecture (compare figure 4.16). Note that we consider the pure code size of each implementation here, with the size of the MCU firmware (NoCrypt) subtracted from every binary.

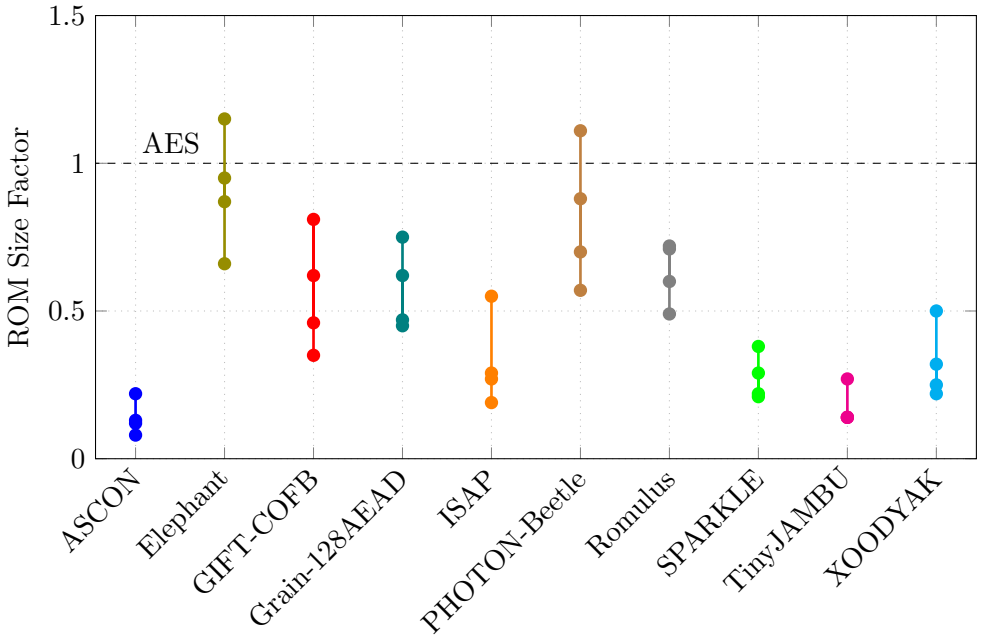


Figure 4.16: Normalized cross-platform code size of NIST LWC ciphers compared to AES-GCM

4.3 Studying Penalty Factors for Protected Variants

A cipher like Isap will always be among the slowest LWC candidates when it is compared to optimized, unprotected implementations from other finalists. This is due to the design properties of the Isap cipher. Its main goal is not to be the fastest candidate. However, it is focused on robustness against various types of side-channel and fault attacks. Due to its leakage-resilient sponge design including a re-keying function, standard side-channel attacks are not applicable to this algorithm [103]. When we discuss side-channel analysis and protection mechanisms in more detail in chapter 5, we will provide in-depth examples for these kinds of attacks.

The inherent protection against implementation attacks comes with a significant decline in performance. Naturally, the generation of new session keys for every encryption/authentication takes time, which leads to the generally lower speed of Isap implementations in comparison to the rest of the remaining LWC finalists. Since NIST stated resistance against side-channel analysis as a secondary selection criterion, the possibility of protecting the algorithms against attacks and the performance of protected implementation variants poses an interesting research field. While any Isap implementation is already secure against basic attacks by design, the other candidates need to provide that security in dedicated implementations. Hardening algorithms by implementation instead of by design is a common approach. There exist different strategies to transform a normal implementation to a protected one. One of the main principles here is to separate secret information (e.g. key bytes) in multiple parts called shares. An attacker is then no longer able to get a hold of the secret, as long as he cannot collect all shares. In general, the security level of an implementation rises with the number of shares. However, also the computing time usually rises when the algorithm has to work with a higher number of shares. Additional performance overhead is introduced by routines for splitting and recombining the sensitive values.

The details of different protection techniques will be explained in a dedicated chapter (see chapter 5). However, now we would like to investigate the overhead that protected implementations introduce in relation to unprotected variants of the same cipher. Moreover, we want to compare the performance of a conventionally hardened implementation of a NIST LWC finalist to the benchmarking results of the hardened-by-design cipher Isap. Unfortunately, protected implementations are rarely available for the relatively new LWC ciphers and their creation and validation represents a difficult task. Since such implementations only exist for few candidates and even less of them are verified to work as intended, we choose one popular candidate – Ascon – for our performance comparison with Isap. As already seen in the benchmark of unprotected variants, we have many Ascon implementations to choose from, including protected ones.

We prepare these protected implementations and analyze their performance using our own benchmarking framework. As a target platform, we choose the STM32F103 MCU that was already part of our hardware portfolio in the previously described benchmarks. We evaluate two protected Ascon implementations, a 2-shares and a 3-shares variant of the main instance, Ascon-128. As a baseline, we use the fastest primary instance implementations of the unprotected Ascon and the Isap cipher. Note that we include these same implementations in both the speed and ROM benchmarks. So, contrary to the performance evaluation of unprotected variants, we do not choose the best implementation per cipher and use case in this set of tests. We do this because we believe it is fair to show the speed and memory footprint for a single (unprotected) implementation of Ascon and Isap since alternative protected implementations for different use cases are not available. Using a single implementation for both test cases results in the implementation not being optimal for one of the test cases. We prioritize the speed test case and select the fastest implementation per cipher for this analysis. Remember that these implementations will not be the best choice for the ROM selection. In a scenario

in which binary size is the highest priority, other (slower) variants should be chosen. In figure 4.17, a legend of the color codes for the following plots is provided.

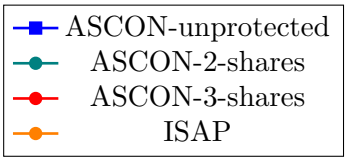
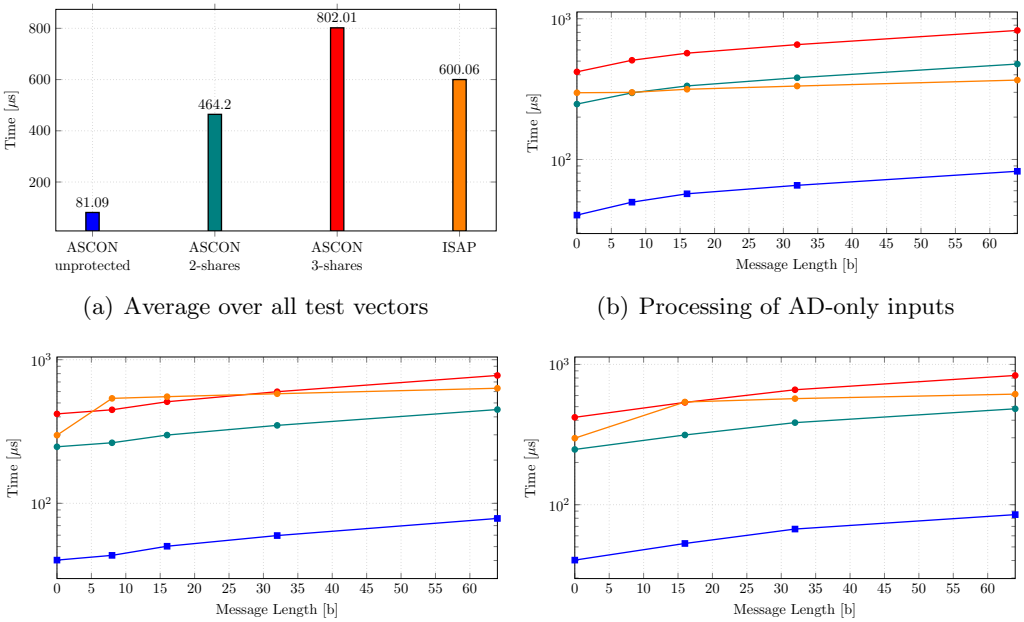


Figure 4.17: Legend of cipher color codes for speed measurements of (masked) implementations



(a) Average over all test vectors

(b) Processing of AD-only inputs

(c) Encryption without AD

(d) Encryption and authentication

Figure 4.18: (Average) encryption timings for (masked) versions of Ascon and Isap on the STM32F103

Figure 4.18 shows the speed measurement results for the mentioned implementations. As in the previous performance tests, we present timings averaged over all test vectors, for authentication only, for encryption only and for specifically both. We observe that, on average, the fastest Isap implementation is slower than a 2-shares version of Ascon but faster than the version with the higher security level (with 3 shares). Moreover, we can derive that Isap is particularly efficient in the authentication step because it outperforms 2-shared Ascon there for larger message sizes. Overall, it is interesting to see that Isap performs better than Ascon implemented with 3 shares, since Isap still provides a higher "by-design" security level than a 3-shared Ascon variation [103].

The performance overhead introduced by the countermeasures for Ascon is directly noticeable in the timing benchmark. The 2-shares implementation causes a penalty

factor of 5.7, while the 3-shares version takes even 9.9× as much time for encryption/authentication on average. These penalty factors remain fairly constant regardless of the input type and show how much more computation is needed in a conventionally protected implementation.

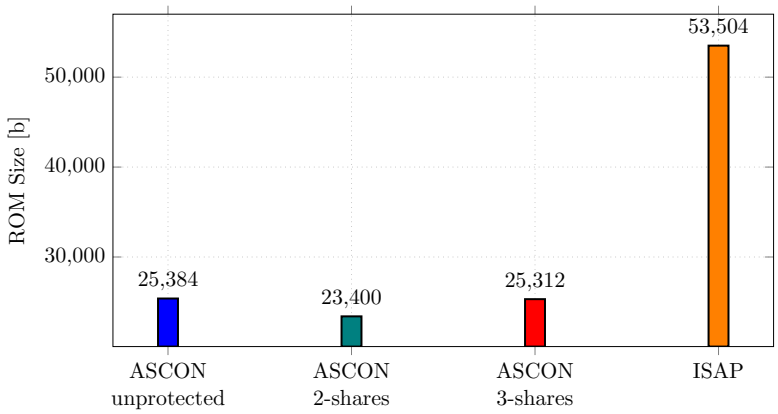


Figure 4.19: Code size measurement results for (masked) versions of Ascon and Isap on the STM32F103

If we compare the results in the ROM test case (see figure 4.19), we recognize that the shared Ascon versions differ only slightly in memory utilization, with the binary including 3 shares requiring roughly 8% more ROM than the 2-shares version. The fastest Isap implementation uses more than twice as much memory than the shared Ascon variants. This is likely caused by an implementation strategy that stores many look-up tables in the binary to optimize for high speeds. Isap can be implemented in a way that it requires much less static memory, then however the speed of the implementation usually declines.

There is virtually no overhead in ROM consumption for the shared implementations of Ascon. Compared to the ROM footprint of its fastest implementation, the 2-share version even consumes 7.8% less ROM than the unprotected one, while the variant with 3 shares practically produces the same binary size as the baseline implementation.

5 Side-Channel Attacks and Countermeasures for LWC

Side-channel attacks pose a relevant security risk in embedded applications, especially if the attacker has physical access to the device. In general, side-channel attacks comprise all attacks that somehow make use of gained information from secondary "channels" of the system under test. Typical side-channels are power consumption, electromagnetic radiation or timing data of the chip. Since these factors are not independent of the processed data on the device, information regarding the running algorithm – including sensitive values – can be derived from these side-channels. In the worst case, an attacker can extract secret key material using side-channel analysis and corresponding statistical methods.

There exist different strategies to protect an implementation of a cryptographic algorithm from side-channel attacks. The main principle here is to make the sensitive values independent of the side-channel, e.g. the power consumption of the device. The two general approaches to achieve this goal are hiding and masking. As the name suggests, hiding aims to hide sensitive information in the power trace. This can be accomplished by either randomizing the power consumption of the embedded device, or by making it (virtually) constant. Different strategies have been presented to realize this, for example the execution of instructions can be randomized in time as long as this does not alter the logic of the algorithm. Randomization in time then leads to a randomized power consumption, which cannot be easily correlated to the processing of key material. The second main protection strategy, masking, is based on dividing sensitive values in multiple parts, such that an attacker only gets hold of a subset of the wanted information, which does not leak the secret (under certain circumstances). Through the sharing of sensitive information, the whole value gets masked and cannot anymore be extracted through basic side-channel analysis [166].

In the following chapter, we will categorize popular types of side-channel attacks. Moreover, we discuss different approaches to hardening cryptographic algorithms on an implementation level. Well-known strategies to evaluate leakage of sensitive values are introduced, before we focus on the efficient protection of implementations of lightweight ARX ciphers, specifically. We will explain, why protecting ARX ciphers is especially challenging and provide an overview over how related work deals with this problem. Then, we will introduce our two novel methods to find efficiently protected implementations for this cipher category and compare the performance of our solutions to previous research. We evaluate the penalty that (our) optimized protection mechanisms introduce in comparison to unprotected ARX implementations. Finally, we analyze how these penalty factors relate to those presented for other types of LWC ciphers in section 4.3.

5.1 Attack Types

Generally, side-channel attacks can be separated in invasive and non-invasive attacks. While the former requires some kind of active tampering of the attacked hardware, for the latter attack only capturing and processing of side-channel information is necessary. In an invasive attack scenario, the device is actively manipulated such that it changes its normal behavior. For example, altering the clock frequency or the power supply to cause hardware faults are typical methods in this attack category. Non-invasive approaches only make use of the side-channel emission that is naturally caused during the normal operation of the hardware. This information is captured and can later be inspected or statistically processed to obtain sensitive values. Both attack types require physical access to the hardware, at least temporarily.

5.1.1 Simple Power Analysis

Simple Power Analysis (SPA) covers scenarios in which it is possible to extract the secret information simply by visual inspection of the power data or another side-channel. This power consumption data of the device over a certain time frame is referred to as a *trace* in side-channel analysis. With SPA, an attacker tries to extract the values of interest only from a couple of traces, or even a single trace. To be able to successfully perform (single-trace) SPA, an attacker has to have knowledge regarding the executed cryptographic implementation on the hardware [167, 168, 169]. Figure 5.1 shows a trace of an AES encryption round. The different phases of the algorithm can be recognized by visual inspection.

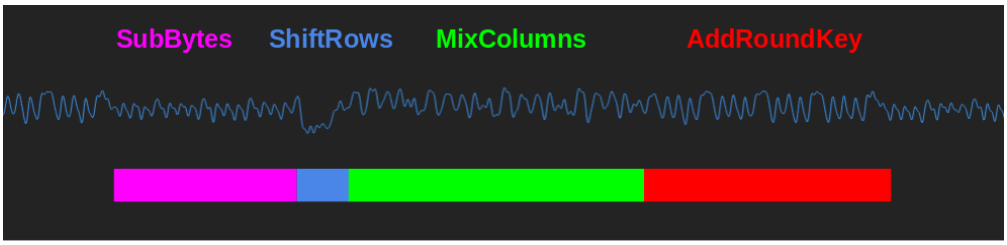


Figure 5.1: Magnified trace of a single AES encryption round

5.1.2 Correlation Power Analysis

Correlation Power Analysis (CPA), in contrast to SPA, requires a high number of side-channel traces to be successful. However, the advantage over SPA is that in a CPA setting, it is usually not necessary for the attacker to have detailed knowledge regarding the hardware or the specific implementation. Often, it is sufficient to know which cryptographic algorithm is executed on the attacked device. For a CPA attack, many (power) traces during encryption/decryption have to be obtained from the hardware. Then, the dependency between the power consumption and the processed data is exploited. By

aligning the power traces and comparing them at defined points in time – where e.g. a round key is calculated – an attacker can spot data-dependent differences.

In more detail, the first step is to identify *points of interest* in the executed algorithm, i.e. one has to find intermediate values within the algorithm execution that are dependent on (a part) of the secret key k. Moreover, the chosen intermediate value has to be a result of a function that takes a known non-constant value v as an input. It is important for the attacker to know the corresponding v for every power consumption trace measured in the following step. When our intermediate value r is a result of $f(k, v)$, the attacker can calculate r for every possible instance of k. With this approach, we obtain a set of values for r of which one of them must have been processed within the device during the power measurement. In the next step, the hypothetical intermediate values have to be mapped to power consumption values. In this step of a CPA attack, a power model has to be applied. The Hamming distance and the Hamming weight model are popular power models for value mapping. Both models exploit the fact that the Hamming weight of a value and the Hamming distance between two subsequent values on a data bus are proportional to the caused power consumption. After the mapping process, every calculated hypothetical power consumption value is compared to the actually measured power consumption at the point of interest. The value hypothesis that correlates best with the power consumption over all recorded traces is most likely the data that has been processed on the actual hardware. For evaluating the correlation between the power model and the actual power consumption, typically the Pearson correlation coefficient is calculated. Once a sufficient correlation peak has been found, the attacker can look up the fitting key hypothesis for the value and thus the secret information is revealed [170, 166].

The Pearson correlation coefficient has been developed a long time ago and was later adopted for use in side-channel evaluations [170]. The Pearson correlation ρ is defined as follows.

$$\rho_{x,y} = \frac{\sum(x - \mu_x)(y - \mu_y)}{\sqrt{\sum(x - \mu_x)^2 \sum(y - \mu_y)^2}} \tag{5.1}$$

The correlation $\rho_{x,y}$ of the variables x and y is calculated using the respective means μ_x and μ_y. In a CPA attack, the compared data sets consist of the measured power traces and the corresponding estimated power drain for different key hypotheses. If we assume N acquired power traces t with M data points, $t_{n,m}$ specifies data point m of trace n. Furthermore, we can denote the estimated power traces H and refer to a modeled key guess data point g for trace n as $h_{n,g}$. Using this notation, we can calculate the Pearson correlation for every key guess g and data point m. The calculation that contains the correct key hypothesis is most likely to have the highest correlation coefficient. If we substitute the generic variables of the formula with the ones in the CPA use case, we can write the equation in the following way.

$$\rho_{g,m} = \frac{\sum(h_{n,g} - \mu_{h_g})(t_{n,m} - \mu_{t_m})}{\sqrt{\sum(h_{n,g} - \mu_{h_g})^2 \sum(t_{n,m} - \mu_{t_m})^2}} \tag{5.2}$$

The Pearson correlation is usually employed in such CPA scenarios. However, also this attack method is still studied and developed further, which is reflected by more recent publications in the area of leakage analysis [171].

Figure 5.2 depicts a CPA attack plot on the first secret key byte using the Pearson correlation. The time span is targeting the first *SubBytes* operation in the AES algorithm. We plot the correlation coefficient for every key byte guess and can derive the correct secret by choosing the key hypothesis with the absolute highest correlation during the sensitive operation (around data point 2470).

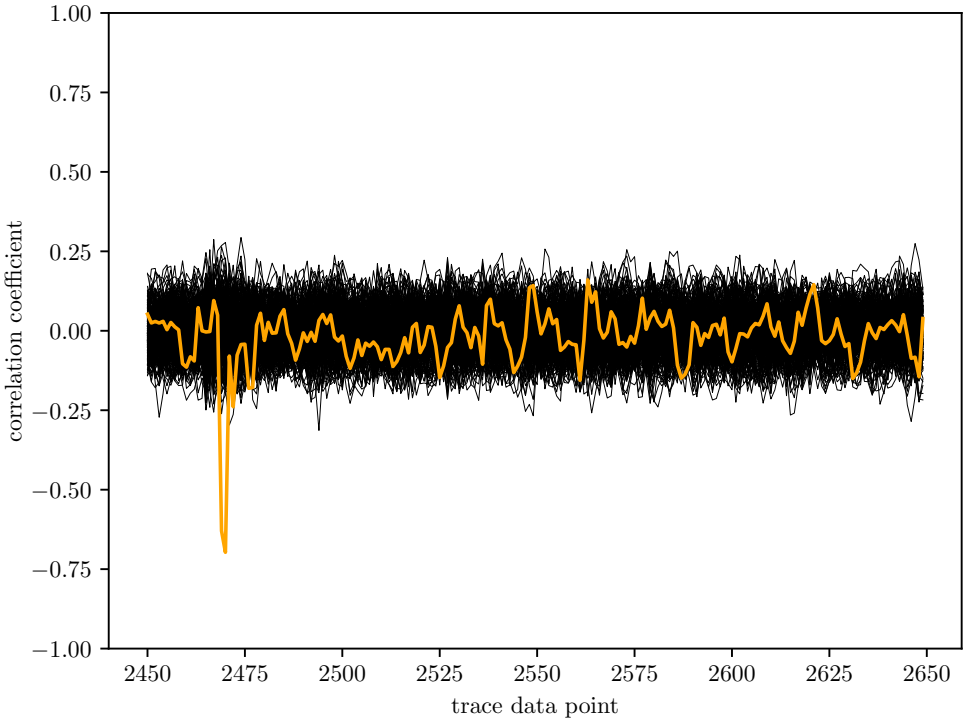


Figure 5.2: Pearson correlation plot for a single AES key byte

5.1.3 Differential Power Analysis

In its basic form, DPA applies the *Difference of Means* approach to extract secret information from power traces. Similarly to CPA, a large number of power traces has to be obtained in order to conduct a meaningful attack. Generally, the acquired power traces have to be separated into two different subsets, of which then the difference of the averages is calculated. In case the two subsets are significantly different, this differential value will be non-zero. If they are statistically non-correlated, the difference of means will be (close to) zero with a rising number of traces. The separation of traces has to

happen according to a *selection function*. This function can imply any logic that groups the traces into two different sets. For example, we can choose the Least Significant Bit (LSB) of the output of the cipher implementation as our selection property. We then class our traces by the value of this LSB being either 0 or 1. When the difference of means is calculated between these two trace classes, we will find non-zero values at data points that are correlated to the output bit. Sometimes, one merges the traces of each group into one master trace to support an easier comparison. In practice, the selection function will contain some operation that holds a known non-constant and a secret constant value, e.g. the key or part of the key. As in CPA, the key byte hypothesis can then be made but in DPA the traces are separated according to the selection function. If the difference of the averages is calculated with the correct key hypothesis, we will see significantly higher correlation than with the traces containing incorrect key guesses. In a DPA attack, typically different points in time are attacked, in which different parts of the secret are used in the algorithm. After all key bytes have been extracted, the whole key can be reconstructed. [172].

Figure 5.3 shows a standard DPA approach using the difference of means on an AES implementation. As a group distinguisher, the Hamming weight of the first S-box output has been used. Subset 0 contains all *low-energy* traces, i.e. traces that incorporate a Hamming weight < 4. All other traces (with a Hamming weight of $>= 4$) have been classified as *high energy* traces. One can clearly observe a high difference of means around data point 2470 for the grouping with the correct key guess (highlighted in orange).

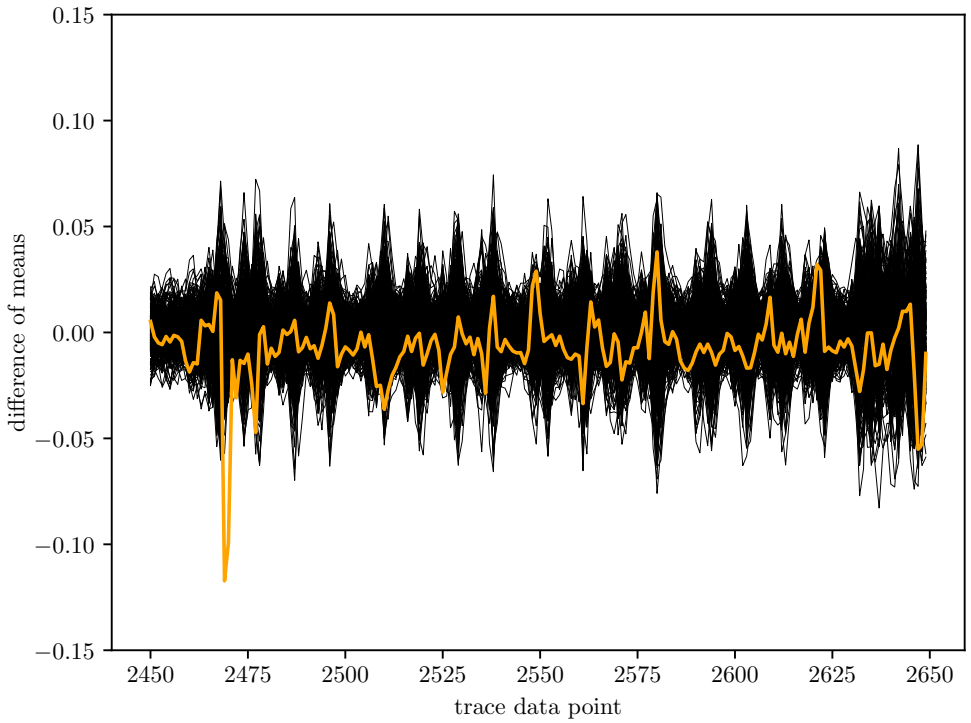


Figure 5.3: DPA difference of means attack plot for a single AES key byte

If we compare this analysis to the CPA attack seen in figure 5.2, we can conclude that both CPA and DPA are successful for the standard AES use case. We compare the efficiency of a CPA and a DPA approach in Figure 5.4. We use the difference of means method for DPA and the Pearson correlation coefficient for the CPA attack. From the side-to-side plots, one can derive how many traces are needed to attack a single key byte in an unprotected AES implementation. Our plot shows the correlation and difference of means for all key bytes, while the orange line represents the correct key guess. The number of traces grows along the x axis. We can see that – in this example – the required confidence for the correct key byte is reached from 25 traces upwards in the CPA scenario. In the DPA setting, we require a bit more data ($\tilde{4}0$ traces) to confidently determine the part of the key.

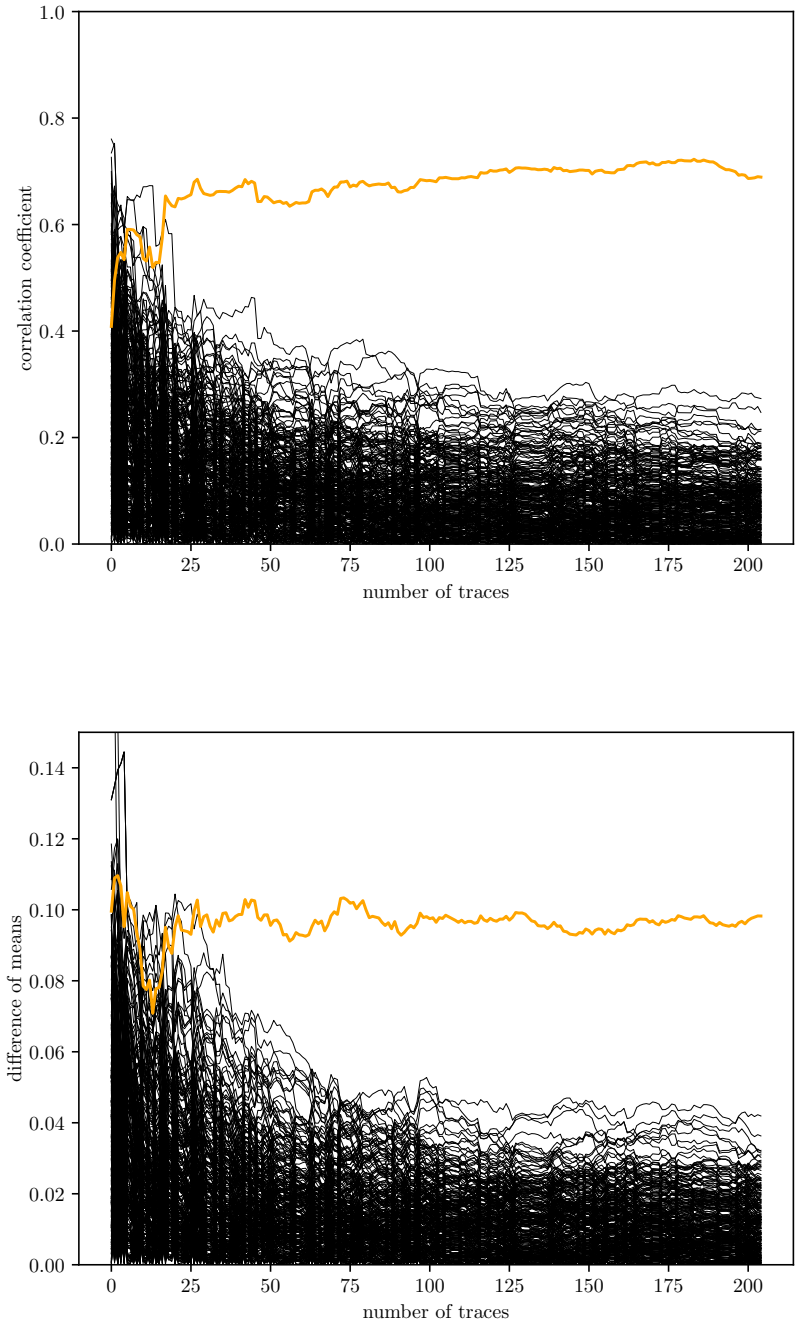


Figure 5.4: Absolute correlation (top) and difference of means (bottom) for AES key byte hypotheses in relation to the number of power traces

5.1.4 Mutual Information Analysis

Mutual Information Analysis (MIA) represents a generic side-channel distinguisher that measures the dependency of two random variables [173]. The basic analysis procedure follows the principles of DPA and CPA. However, when comparing the key hypothesis, the entropy H defined by Claude Shannon is used instead of e.g. the Pearson correlation coefficient [174]. Another difference between MIA and previous model-based side-channel analyses is that MIA can capture non-linear similarities, while correlation can only find two-dimensional dependencies. Moreover, univariate approaches need to apply pre-processing transformations to tackle a multivariate problem, while MIA allows for direct multivariate inspection by default [175]. This can make the MIA model more suitable for higher-order attacks on protected implementations, in which multiple points of interest are analyzed simultaneously.

5.1.5 Template Attacks

Template attacks comprise the most powerful side-channel attacks. For this type of analysis, an attacker has to have full control over the hardware or possess an identical device. As the name suggests, then a *template* of the device under attack is built. This means its behavior is studied in detail and various effects on the power consumption caused by different inputs are observed. Depending on which way and how intensive this profiling phase is carried out, an attacker can extract device-specific dependencies between inputs and the acquired traces. This can even lead to a mapping of instruction series and peculiarities in the power consumption. After a sufficiently exact template is set up, an attacker can use this knowledge in e.g. an attack based on the profiled power model. While this approach can make an attack more efficient and exact, it requires the most extensive physical access to the hardware that processes the secret information of interest [176].

5.2 Countermeasures

As mentioned in the introduction to this chapter, mainly two strategies are used when it comes to protecting implementations from side-channel attacks – hiding and masking. Each of the two approaches can be implemented in different ways, however, most of the recently published protection mechanisms can be assigned to either hiding or masking. In the following two sections, we will introduce the general idea behind these countermeasures and provide some details regarding their most popular variations.

5.2.1 Hiding

Hiding tries to conceal the dependency between the power consumption of the device and the processed intermediate values. This is usually approached by either randomizing the power consumption or equalizing it. Even if it is not possible to make the current drain completely independent of the carried out operations, reaching a high degree of

randomness/equality can be enough to protect the secret information from basic DPA. A (close to) random power consumption can be achieved in two ways. One possibility is to randomize the algorithm itself in time. This means that the hiding countermeasure consists in reordering operations within the implementation and changes the order during the execution. The challenge here is to reach a high-enough level of mixed-up operations in order to randomize the power consumption, while still maintaining the logical correctness of the algorithm [177]. A simpler approach to alter the execution timings and the resulting consumption is to insert additional instructions within the algorithm implementation. Here, no reordering of the actual instructions is necessary and still the power trace can be shifted in time.

Another strategy to randomize the measurement aims to change the actual power consumption of the operations. This has the advantage that no changes have to be made to the cipher implementation. However, equalizing or randomizing the power consumption of actual instructions is a challenging task. Typically, the signal-to-noise ratio is lowered in these scenarios, which effectively also lowers the leakage of the secret information (see section 5.3.1). To reduce the signal-to-noise ratio, either the signal can be reduced or the noise can be increased. The former aims to equalize the power consumption by introducing e.g. filters in the layout, the latter tries to shadow the sensitive operations through parallelism in the data path or particular components that inject additional noise. Both of these measures should be considered already in the design of a cryptographic device in order to be implemented properly [178, 166].

5.2.2 Masking

Masking refers to destroying the relation between known hypothetical values and processed data through splitting the value in multiple parts. These parts are usually called shares. The idea behind masking is that an attacker cannot construct the secret information from its parts, unless he has access to all shares. In software implementations, a masked representation of an algorithm is achieved by dividing the sensitive values in multiple shared variables. It is important that the shared representation is correct, i.e. the whole value can be reconstructed from its shares to build the appropriate output of the algorithm. The computation time and the protection level rise with the number of shares d. An implementation incorporating d shares protects (at most) against attacks of order $d-1$. In practice, this means that an attacker probing $d-1$ wires on a device running a d-share implementation cannot extract sensitive information caused by distance-based side-channel leakage [179].

5.2.2.1 Boolean Masking

We distinguish two main strategies of masking by their composition function, meaning the operation that combines the shares back into the original value. When using Boolean masking, the composition operation is an exclusive-or, so the secret value v, masked with three shares v_1, v_2 and v_3, is determined by $v_1 \oplus v_2 \oplus v_3$. The equation $f(a*b) = f(a)*f(b)$ is true, given that the operation $*$ is a linear operation. If this linear operation is e.g. a

XOR, $f(a \oplus b) = f(a) \oplus f(b)$ applies. This means, a Boolean masking scheme is linear, as long as the original operation is also linear. Therefore, Boolean masking is particularly efficient for linear operations [166].

5.2.2.2 Arithmetic Masking

In arithmetic masking, the shares are combined by a non-linear operation. Here, either modular addition or multiplication in a residue class ring is used. Our composition equations for value v are then defined by $v_1 + v_2 + v_3$ or $v_1 \times v_2 \times v_3$, respectively. Arithmetic masking is linear for non-linear operations. It is more suitable for non-linear operations, similarly to Boolean masking being more efficient for linear operations.

Many cryptographic algorithms mix linear and non-linear operations, which means changing Boolean masks into arithmetic masks (and vice versa) is required during runtime. Various publications provide suggestions on how to securely transition in between Boolean and arithmetic masking. However, the switching remains a resource-intensive matter [180, 181].

5.2.2.3 Threshold Implementations

TIs haven been introduced as an alternative to Boolean masking. This concept has first been developed for hardware implementations only and has the advantage that it remains secure even in case of glitching attacks, contrary to other approaches [182, 183]. TIs also work on shares of sensitive values, which are reassembled to the whole value by an exclusive-or operation. When TIs have been first introduced, the designers required implementations to carry three shares to reach 1st-order security, one more share than conventional methods. It has later been shown that a variation of the original TI-definition can also be implemented with two shares only while still maintaining 1st-order security [184]. Lately, this relatively new TI-scheme has also been adapted for use in software implementations. At a basic level, protected implementations have to fulfill three requirements in order to identify as a TI [182].

Correctness The exclusive-or combination of the input and output share has to resemble in the correct (unshared) input or output value. This property is required to make sure that the algorithm itself is implemented correctly and delivers the expected output to the corresponding inputs. Correctness is obligatory only for input and outputs and not necessarily for intermediate values during the different stages of the algorithm.

Non-completeness An implementation that operates on d shares shall only combine at most $d - 1$ shares in a computation step in order to be d-order secure. This can require a two-share algorithm to expand the number of output shares temporarily, in order to not have to combine the only two shares at once. After the expansion, the shares can be collapsed back, again. This expansion and collapsing layer allows for 1st-order secure implementation with only two shares in a software setting.

Uniformity All in- and outputs of the TI shall be uniformly shared. In order to prove that, one can check if the input vector represents a uniform sharing for all possible inputs. It can be shown that in case of a uniform input sharing, the output sharing is uniform as well. The uniform distribution of the shares is not required for intermediate states, but for the in- and outputs only.

With these requirements, one can build 1st-order secure implementations with (at least) two shares per input and output.

5.2.2.4 Domain-Oriented Masking

Another novel masking scheme that relies on secret sharing is Domain-Oriented Masking (DOM). DOM provides the same security level as TI, but has been shown to be more resource-efficient for hardware implementations of AES. The main idea of DOM is to organize shares in domains and not at a function level. Per share, DOM creates one domain. The basic approach is to separate the shares of one domain from the shares of other domains. Through the domain separation, no recombination of all shares of a value can occur, which leads to $d-1$-order security for a d-share implementation [185].

5.3 Leakage Detection

The aforementioned side-channel attacks all share one step in which the acquired traces are processed with the goal to discover leakage of secret information. This processing of the power consumption data incorporates some kind of statistical analysis in any type of attack. Depending on the strategy and the gathered traces, different statistical models can be applied to reveal correlations between the input data and the measured power consumption. The selected method for detecting leakage is also relying on the preprocessing of the traces and the capturing methodology. In most cases, points of interest are determined at certain time slots and the traces are separated according to the value of a relevant input word (see section 5.1.3). In dependency to the type of partitioning and other properties of the traces, an appropriate statistical analysis step is applied to reveal the leakage.

5.3.1 Signal-to-Noise Ratio

The Signal-to-Noise Ratio (SNR) is a data property that is often used in the digital signal processing domain outside the context of side-channel analysis. The SNR can be defined as seen in equation 5.3, where Var is short for the *variance*.

$$SNR = \frac{Var(Signal)}{Var(Noise)} \tag{5.3}$$

For the SNR in side-channel attacks, the *signal* part of the equation only concerns the power consumption that is actually exploitable by the attacks. Since the non-exploitable signal (i.e. power consumption) does not carry any desired information, it should not be included in the SNR calculation. The *noise* however composes all types of noise that

the traces contain. This is because, in contrast to the signal, the noise is relevant to the attacker regardless of its origin, so none of it can be disregarded. To be able to calculate the SNR, an attacker has to have control over the inputs of the algorithm. By carefully selecting the plaintexts and the point(s) of interest, the desired trace set can be built and used as the source data for an attack.

5.3.2 Welch's t-test

Welch's t-test has been introduced as a statistical comparison tool in 1947 [186]. Since then, it has been adopted for and applied in different research disciplines. In general, the t-test compares the mean of two datasets. If the mean of the inputs is equal, they are considered to be not different. Goodwill et al. applied the t-test to side-channel evaluations, and since then it has been a popular tool for basic leakage inspections [187]. The test is defined as depicted in equation 5.4.

$$t = \frac{\mu_A - \mu_B}{\sqrt{\frac{\sigma_A^2}{N_A} + \frac{\sigma_B^2}{N_B}}} \tag{5.4}$$

A and B represent the two compared groups with their standard deviations σ_X and their means μ_X. N_A and N_B are the sizes of the data sets (i.e. the number of traces in side-channel evaluations). To use Welch's t-test to determine side-channel leakage, a threshold for $|t| = 4.5$ is set. This allows to make a statement regarding the difference of the traces with a confidence of > 0.99999. If $|t| < 4.5$, the two datasets are regarded as statistically not different, meaning no side-channel leakage can be detected by the t-test. The t value itself can only be used to measure statistical difference of traces, qualitatively. The actual value does not allow for an evaluation of the amount of leakage, e.g. the result $|t| = 2.3$ does not imply worse side-channel protection than a result $|t| = 1.2$.

We distinguish fixed vs. fixed (specific) and fixed vs. random (non-specific) t-tests. In a non-specific scenario, trace set A is collected with the same plaintext in every measurement and trace set B contains traces acquired while incorporating random plaintexts with a uniform distribution. The specific t-test asks for two datasets, both instrumented with a fixed value. Welch's t-test remains a standard analysis for leakage detection since its introduction. However, it has been shown that specific properties of the input data can lead to either non-detectable leakage or the detection of leakage that is not present in practice. Hence, the methodology and application of the t-test is still discussed in recent literature [188].

5.3.3 χ^2-test

The χ^2-test is used to analyze if unpaired observations on two variables are independent of each other. The null hypothesis is that the measured observations are indeed independent. The evaluation uses a contingency table to show the frequency distribution of the variables. No statistical methods such as the investigation of the means are applied, compared to the t-test. The distribution is spread according to χ^2, hence the name of the test. We now illustrate the concept of using a contingency table with a simplified

Fi,j	$j=0$	$j=1$	$j=2$	$j=3$	total
$i=0$	17	47	32	14	110
$i=1$	16	38	28	1	83
total	33	85	60	15	193

Table 5.1: Contingency table according to the histograms in figure 5.5

example derived from the original paper [189]. In this scenario, we use hypothetical data distributions because our only goal is to present the χ^2−-test approach.

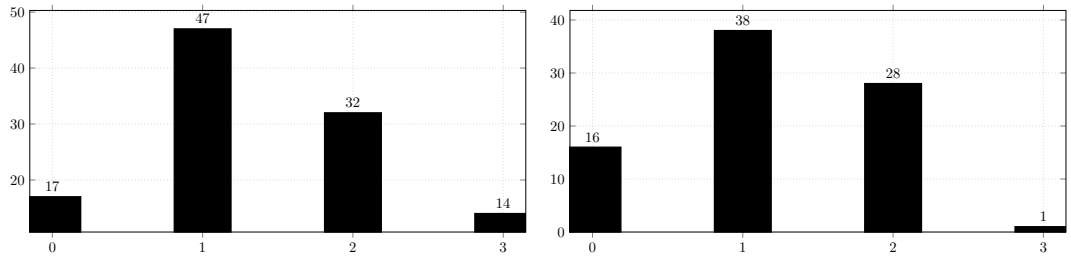


Figure 5.5: Histograms for example data sets

We assume two data sets with 110 and 83 elements each. The frequency of observations is given by the histograms in figure 5.5. We build the contingency table 5.1 with the data from the histogram and calculate the degrees of freedom v in the following. v is defined as seen in equation 5.5, with r being the number of table rows and c being the number of columns of the contingency table. If we fill in the values from our example, we get $v = 3$.

$$v = (c-1) \cdot (r-1) \tag{5.5}$$

In the next step, we need to calculate the expected frequency for every table cell. We denote the expected frequency for a cell (i,j) as $E_{i,j}$. In general, the expected frequency is defined according to equation 5.6, with N specifying the sum of all cells.

$$E_{i,j} = \frac{(\sum_{k=0}^{c-1} F_{i,k}) \cdot (\sum_{k=0}^{r-1}) F_{k,j}}{N} \tag{5.6}$$

If we again fill in the frequencies from table 5.1, we can derive the following table 5.2 with the expected frequencies for every cell. The χ^2 statistic x can then be computed by inserting the values from both tables into equation 5.7. We compute a χ^2 statistic again for every cell. In the end, we sum up the cell values to obtain the overall χ^2 statistic as follows:

$$x = 7.47 \times 10^{-3} + 0.11 + 5.59 \times 10^{-3} + 3.47 + 8.76 \times 10^{-3} + 0.13 + 6.21 \times 10^{-3} + 4.61$$

Ei,j	$j=0$	$j=1$	$j=2$	$j=3$
$i=0$	17.36	44.74	31.58	8.55
$i=1$	15.63	40.26	28.42	6.45

Table 5.2: Calculated frequencies for every cell

$$x = \sum_{i=0}^{r-1} \sum_{j=0}^{c-1} \frac{(F_{i,j} - E_{i,j})^2}{E_{i,j}} \tag{5.7}$$

Finally, we use the χ^2 probability density function f (see equation 5.8, with Γ specifying the gamma function) to calculate the probability to accept the null hypothesis. For our example values, we get $p \approx 0.0139$, which means the occurrences of the observations in our example data sets are dependent on each other.

$$\int_x^\infty f(x,v)dx, f(x,v) = \begin{cases} \frac{x^{\frac{v}{2}-1}e^{-\frac{\pi}{2}}}{2^{\frac{v}{2}}\Gamma(\frac{v}{2})} & x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

For side-channel leakage evaluation, the χ^2-test evaluates the dependence of trace classes and measured leakages. In case the null hypothesis holds true and this statement can be made with appropriate confidence, it means – according to the χ^2-test – the leakages are not informative, i.e. not useful for an attack. If the null hypothesis can be confidently rejected, informative leakage is present in the data [189].

While with the t-test evaluates the presence of leakage in a specific statistical order, the χ^2-test targets the whole distribution. This has the advantage that it can be used to analyze the resistance of an implementation against distribution-based attacks. The t-test is suitable to check if a masked implementation reaches the desired protection level. However, the χ^2-test can be of use for leakage evaluations regardless of a particular statistical order [189].

5.4 Efficient Protection of ARX Ciphers

The core ideas presented in this section have already been published in S. Renner, E. Pozzobon, and J. Mottok. Evolving a Boolean masked adder using neuroevolution. In International Workshop on Attacks and Defenses for Internet-of-Things at ESORICS*, pages 21–40. Springer, 2022 and E. Pozzobon, S. Renner, J. Mottok, and V. Matoušek. An optimized bitsliced masked adder for ARM Thumb-2 controllers. In* International Conference on Applied Electronics*, pages 1–4. IEEE, 2022*

Building symmetric ciphers from only modular additions, rotations and `XOR` operations is a popular design pattern, especially for lightweight algorithms. Due to the simplicity of the core operations, a cipher only using these three elements can reach a high encryption/decryption speed. These so-called ARX ciphers have therefore been

proposed as alternatives to AES in resource-constraint environments. The most well-known algorithms from that category are ChaCha20, Speck and SPARKLE, which is 3rd round candidate in the NIST LWC competition. Chacha20 can be seen as a successor variant of the Salsa20 cipher, which has been selected for the winning portfolio of the eSTREAM project [13]. Moreover, ChaCha20 has been shown to sometimes deliver better performance than AES, specifically on devices that do not support AES hardware acceleration [192]. The SPARKLE ARX mode also offers highly competitive encryption/authentication speeds among the LWC candidates and AES, as our benchmarking results presented in chapter 4 suggest.

While ARX structures deliver high processing speeds and are simple to implement, masking them against the mentioned side-channel attacks is challenging. Protecting for example against 1st-order DPA attacks comes with a big performance penalty, even if latest optimization strategies are incorporated [193, 194, 195]. This overhead can be identified as particularly high, if we compare the performance drop due to masking to e.g. AES, where the penalty is much lower [196, 197].

In the following sections, we will introduce our research in the field of optimizing the protection of ARX ciphers. First, we elaborate on why masking these structures comes with such a high penalty. Then, conventional strategies to protect ARX ciphers are mentioned and related work is discussed. Moreover, we present two novel methodologies for finding efficiently masked ARX representations. We show under which circumstances our solutions perform better than related work by giving detailed benchmarking figures. Therefore, we implement ARX LWC algorithms using different masking strategies. Finally, we discuss real-world leakages of protected implementations on hardware suited for LWC.

5.4.1 Complexity of Masking ARX Structures

We have already introduced the main principles and strategies for masking an algorithm. When it comes to protecting ARX ciphers, the differences between Boolean and arithmetic masking are especially relevant. Since the ARX algorithm design utilizes Boolean (exclusive-or, rotation) and arithmetic operations (the addition), in general both masking strategies can be used. When this is the case, switching in between masking types within the implementation is required. Goubin et al. and Coron et al. published research regarding efficient algorithms for changing between arithmetic and Boolean masks. While a masked ARX implementation can be realized with these switching algorithms, their use requires many additional operations and therefore significantly lowers the performance of the implementation [181, 180].

5.4.2 Conventional Masking of Addition

The deciding performance factor for a protected ARX implementation is the masking of the addition part. In the early 2000s, research suggested transitioning to and from arithmetic masking prior and after the addition, respectively. Here, the conversion from arithmetic to Boolean masking is especially resource-intensive [181]. The work

from Goubin et al. has a complexity of $O(k)$, with k representing the bit width of the addition [180]. Coron et al. published an alternative switching algorithm in 2015 which reduces this complexity to $O(log(k))$ [198].

In more recent work, researchers abandoned the idea of transitioning in between masking types within the implementations. Instead, they seek ways to use Boolean masking only, in order to save the processing time for the mask conversion. This however implies that the algorithm has to be representable with Boolean operations only. Regarding ARX ciphers, the crucial part is again the addition. While using Boolean masks for exclusive-or and rotation operations is standard, providing a Boolean masked variant of the addition is challenging. One strategy to achieve this, is to utilize Parallel Prefix Adders (PPAs). These are adder structures that add binary words of a specific width and use prefix operations to be more efficient. Typically, PPAs such as the Carry-Save Adder (CSA) or the Kogge-Stone Adder (KSA) are implemented in larger hardware circuits. Due to their gate-based design, they provide a Boolean representation of the addition and therefore Boolean masking is applicable. This is true also for software implementations, as long as their gate structure is transferable to assembly instructions.

Biryukov et al. followed a PPA approach to be able to directly mask the Boolean operations. They introduce a tree-based exhaustive search algorithm to find an optimally masked representation of the gates in a KSA. The masked gates are then plugged into the KSA, which leads to a masked addition that can be included e.g. in ARX implementations. The authors validate the protection level with a t-test. The secure gate KSA delivers significantly better performance than the earlier conversion-based approaches [193]. Similar work from partly the same research team has also been published in 2017. In this second project, Dinu et al. adopt secure gate representations and complement them to build a masked addition based on a CSA [195]. Other research targets the masked addition in a hardware setting. Schneider et al. employ the ideas of TIs (see section 5.2.2.3) to form a 3-share protected implementation. They present speed gains for hardware implementations, however, they do not optimize for software applications [199].

Jungk et al. pick up the TI approach and combine it with ideas from Biryukov and Dinu (et al.). The research from Jungk et al. adopts a TI-like methodology for software implementations. In this work, the authors dissect again a KSA and then mask its parts in a TI scheme. However, they protect not only single gates at a time, but form *gadgets* combined from subsequent operations. Similar to other research, after these parts are properly masked, they are used to reassemble the now protected KSA in a TI representation. Through this strategy, Jungk et al. reach improved performance compared to related work. They validate their results in a ChaCha20 implementation benchmark and report significant speed gains.

In our optimization research, we do not choose a PPA as our target. Moreover, we do not apply switching between the two different masking types. We choose the masked full adder as our subject of study and define the inputs a, b, c_{in} and the outputs s and c_{out}. We believe there is more room left for optimization than in the heavily researched PPAs. Furthermore, an efficiently masked full adder can later be used in a bitsliced representation of an ARX cipher, i.e. the outcome is applicable to protected software

implementations. We are researching two different paths regarding efficient masking of the full adder. First, we follow an automated learning approach based on genetic algorithms and neural networks to search for suitable designs. Here, we emphasize a more unconventional and open strategy to solve this masking problem. Second, we apply a more guided path, in which we exploit more standard exhaustive search methods, similar to the work from the team of Biryukov. However, we modify their strategy based on the developments we have observed from the neuroevolution approach.

We choose these different methods to study if an instrumented machine learning algorithm is applicable to such masking optimization problems. Moreover, we would like to compare the neuroevolutional technique to more conventional search approaches. In the following, we introduce some basics regarding neuroevolution and genetic algorithms and show how we fit these tools to our problem setting. Subsequently, we present our guided exhaustive search methodology for finding our most efficient masked adders. In later sections, we will provide software implementations for lightweight ARX ciphers and show benchmarking results for our solutions and related work. Finally, we will touch on our leakage evaluation and discuss power side-channel leakages on actual MCUs.

5.4.3 Finding Masked Adders Using Neuroevolution

Neuroevolution is a discipline within the field of artificial intelligence, its name signifies the evolution of neural networks. Usually, genetic algorithms are utilized to generate and evolve neural networks with some pre-defined properties [200, 201]. Through the application of genetic algorithms, processes known from biology and nature are transferred into computer science. Neuroevolution techniques are often applied to reinforcement learning problems. As a benchmarking experiment, new variants of neuroevolution algorithms are tested within video game environments, e.g. to solve navigation problems [202, 203].

Neuroevolution algorithms can be separated by the way they alter networks during their execution. Topology and Weight Evolving Artificial Neural Networks (TWEANNs) change the nodes/connections of a network and the associated weights, while conventional algorithms modify the weights only and leave the topology untouched. The incorporated genetic algorithm, however, always follows the same steps: initialization, selection, crossover and mutation. Initially, a startup population p_0 of n networks is created in line with the settings in the configuration. Each member of this population is then evaluated according to the *fitness function*. This fitness function is defined such that it can rate how good each network fulfills the requirements of the target network. Each member of p_0 is labeled with a rating from the fitness function. The population is sorted by the fitness value and a certain part of the highest-rated networks proceeds to the crossover step. Here, a mating process of the population is simulated, which is inspired by biology. The best-fit networks are allowed to reproduce and their *children* form the next generation of networks p_1. Similar to what happens in nature, some population members also experience mutations, which further diversifies the group of networks. Once a new generation is formed, the next algorithm iteration is started. This process is carried out until either a certain number of populations has been produced or a sufficiently fit solution network has been found. Technically, this exit condition is realized

through a condition that terminates the algorithm in case a network surpasses a certain fitness threshold in the evaluation phase.

5.4.3.1 The NEAT Algorithm

A well-known neuroevolution method is called NeuroEvolution of Augmenting Topologies (NEAT). It was introduced by Stanley et al. in 2002 and has been regularly adopted in various research works over the years [204]. NEAT is ranked among the best-performing techniques in multiple benchmarking experiments and different works have shown that it beats other neuroevolution strategies in performance tests [205, 206, 207]. Due to its good reputation for solving difficult non-continuous problems and the availability of a well-documented open-source implementation, we selected NEAT as our base algorithm for evolving masked full adders.

Like other neuroevolution techniques, NEAT implements a standard genetic algorithm. However, NEAT has some properties which differentiate it from other neuroevolution methods. The members of a population are referred to as *genomes*. A genome is an encoding of a neural network. Every network is built from nodes that are linked by connections. If a node has connections going to it and originating from it, it is called a *hidden* node. *Input* nodes are nodes that only have outgoing connections, an *output* node is an endpoint of a network which only has inbound connections. Every connection is associated with a *weight*. This weight is subject to change during the mutation phase. Since NEAT operates on TWEANNs, connections can be rerouted or deleted, new connections can be inserted and also nodes can be deleted or inserted. How much change happens in one iteration is dependent on the configuration of the mutation rates prior to runtime.

Another peculiarity of NEAT is that the algorithm is keeping track of the history of each gene. A gene represents either a connection or a node. By saving an *innovation number* for every newly appearing gene, the origin of each structure can be tracked. That allows the algorithm to evaluate the similarity of single structures and whole genomes when comparing them in the reproduction process. Since the historical markers reveal how much alike genomes are in functionality, this information can be used to calculate how distant one genome is to others. The genomic distances are taken into account in another peculiarity of NEAT, which is speciation. By default, NEAT is separating genomes into a number of species, based on the aforementioned distance. Members of a species are then only competing with members within their own species, which helps in protecting innovative genomes that do not fit the problem setup (yet) when arising. The designers of NEAT motivate the use of speciation with a similar behavior in nature, where different structures also compete with each other solely within their own species.

The NEAT algorithm typically works on continuous values within neural networks. This means e.g. the weights of the connections are floats and the networks are built from structures that cannot be directly translated into software implementations. This is due to the fact that suitable problems for neuroevolutionary computing are very result-oriented. In our use case, we are however not only very interested in *what* the networks return, but also *how* they produce the output. Our goal is to alter the NEAT algo-

rithm such that it can work on specific TWEANNs that represent networks of gates or instructions. This way we can still make use of the mentioned nature-inspired features of NEAT and also receive an implementable network for our masked implementations. Typically, NEAT includes weights, biases, responses, activation and aggregation functions that all are defined for floating-point values. The output of a single node gene is then determined using these values (see equation 5.9). However, continuous activation functions, or correctional bias are only relevant for conventional neural networks.

$$output = activation(bias + (response * aggregation(inputs))) \tag{5.9}$$

In our use case, we want to emulate networks of gates/instructions and therefore, output-oriented operations based on floating-point values are not applicable to our problem. This is why the first step in customizing the NEAT algorithm was to remove all float-based calculations or to replace them with fitting discrete operations. As stated before, we wanted NEAT to work on node networks of instructions – or more general – logical expressions that are interconnected through connection genes. We limited the connection weights to 0 and 1. A weight of 0 results in the corresponding input being disabled, while a 1 allows the input signal to reach the node. The possible node aggregations have been restricted to a set of well-defined logical expressions, allowing us to evolve genomes that are essentially gate netlists. After we remove the unnecessary float parts from the output equation 5.9, we can simplify it as seen in equation 5.10.

$$output = aggregation(inputs) \tag{5.10}$$

After that modification, the output of a node gene is only determined by the aggregation function, which is carried out on the corresponding inputs. We implemented custom aggregation functions to realize our gate networks. If a node, for example, holds a `XOR` aggregation and features two 1-bit inputs, it literally represents a `XOR` gate in the form of a node gene. Our defined aggregation functions are transferable to ARM assembly instructions, either directly or after a transformation. This way, we make sure that the evolved networks can be incorporated into software implementations. We define aggregation functions for `XOR`, `OR`, `AND`, `NOR`, `NAND` and `NOT`. This allows us to exploit the neuroevolution techniques from NEAT to search for logic gate networks in general. Note that the underlying NEAT algorithm and its genetic nature remained unchanged in this step, since only output functions for the nodes and the floating point-based properties were altered to fit our Boolean use case.

For our masked adder use case, we define a genome g as an encoding of a Boolean NEAT network N. N consists of a series of node genes ng_x and connection genes cg_x. Node genes are defined by their node type, nt (which can be *input*, *hidden* or *output*) and a unique ID, nid. A connection gene has three properties, the two nodes it is connected to, nid_1 and nid_2, and a weight, cw, which can be either 0 for disabled or 1 for enabled. The NEAT algorithm operates on a number of encoded Boolean networks. This set of genomes is referred to as a population p. Each genome of the current population is a member m. All of our individual in- and outputs of a network are exactly 1-bit wide.

5.4.3.2 The Fitness Evaluation

To evaluate our modifications, we needed to test the NEAT implementation within our problem setting. Therefore, the adder problem first has to be defined in the NEAT configuration and setup. Our goal is to evolve a masked full adder with two shares per in- and output value. This is suitable for an ARX implementation that shall be protected against 1st-order attacks. With that same security level as the most recent related work from Jungk et al., we also make sure to have a solid base for a fair comparison [194]. Contrary to related work, we do not focus on the problem of masking individual parts of a PPA. We define a 1st-order masked full adder as our design goal. We search for a masked representation of the complete full adder, instead of trying to secure certain structures of a 32-bit adder. In this way, we can evolve an adder model as a whole. However, working with a full adder comes with the downside that its masked representation will only be useful in bitsliced implementations of e.g. ARX ciphers. Anyway, the rather small number of 64 possible input vectors allows us to check for full correctness of evolved genomes. This would obviously be computationally impossible in a reasonable time frame for e.g. a KSA with 2^{32} inputs.

We specify the inputs of our adder as a_0, a_1, b_0, b_1, c_{in0} and c_{in1} and the outputs as s_0, s_1, c_{out0} and c_{out1}, while v_0 and v_1 represent the shares of the variable v. In the configuration of the NEAT implementation, we include these variables and the truth table of the full adder. This way, the NEAT algorithm is capable of checking the correctness of the evolved networks during runtime. The correctness check is done by feeding each adder candidate all 2^6 possible inputs. Since the algorithm knows which output values should be produced for each input vector, it can judge the correctness (fitness) of the network for the defined problem.

In the standard setting, the neat-python library implements a scalar fitness value. This value is calculated for every genome of the population in every iteration. More specifically, the fitness is the output of the user-defined fitness function, which is applied n times per algorithm iteration (for a population of n members). This evaluation happens prior to the selection phase. Typically, the networks are then sorted according to their fitness values and only the top b percent of genomes in that sorted list are allowed to proceed to the next algorithm step. For the fitness value, the developer also specifies a termination threshold. The fitness function is defined such that this threshold can only be reached by a perfect network. In case a perfect solution occurs during the algorithm execution, the threshold is met and the algorithm terminates and presents the solution network [208].

Our fitness threshold is set to 0. In the fitness evaluation, every network is stimulated with all possible inputs. We then observe the output values and compare them to the full adder truth table. Since the in- and outputs are available in a shared representation, we check if $c_{out0} \oplus c_{out1}$ equals c_{out} and $s_0 \oplus s_1$ equals s for the given input vector. We allow each network to have a starting fitness value of 0. However, for every wrong output bit, 1 is subtracted from the initial fitness. With 64 input stimuli and 2 output bits, the minimal adder fitness of a network is -128. Only if a network delivers the correct 2-bit output for every input vector, no degradation of the fitness takes place. This is the

only case in which a network manages to maintain the fitness goal of 0 – which means it represents a perfect shared full adder. With this fitness evaluation and the Boolean aggregation functions, we can evolve a shared full adder. However, this strategy does not yet include any kind of leakage analysis, which means it is very likely that any evolved solution is prone to standard DPA or CPA attacks as presented in sections 5.1.2 and 5.1.3.

To be able to evaluate the distance-based leakage of our networks, we introduce a second fitness value. This leakage fitness is also set to a threshold goal of 0. The leakage fitness function is based on the method published by Gross et al. in 2019 [209]. Note that we refer to a *shared* input, when mentioning an input vector consisting of the six input shares and a *secret* input, when considering the 3-bit input vector, obtained from the `XOR` operations on the 6-bit input. We build a leakage table with a single row per possible shared input. These shared inputs are then grouped by secret inputs, i.e. the input vectors $a_0 = a_1 = b_0 = b_1 = c_{in0} = c_{in1} = 1$ and $a_0 = a_1 = b_0 = b_1 = c_{in0} = c_{in1} = 0$ are put into the same secret input group since for both the secret inputs are the same ($a = b = c_{in} = 0$). Then, the output value at every node of the network is calculated and saved. We compare the Hamming weight of these outputs for every secret input group. In case the Hamming weight is equal among all groups, we can state that there is no distance-based leakage originating from that node output. This can be concluded because an equal Hamming weight implies the statistical independence between the secret inputs and the power consumption at that intermediate node. As an example, assume an adder network with 16 nodes/gates (t_0 to t_{15}). To check the leakage at the first node (t_0), we stimulate this node with all possible shared input vectors. As mentioned before, we group all shared inputs that translate to the same secret input. Lastly, we measure the Hamming weight of the output truth table (of t_0) and compare if weights are equally distributed, regardless of the secret input.

We apply the secret input grouping and Hamming weight check for every intermediate node in a genome. The leakage fitness is initialized with the value 0. In our full adder use case, we have 2^3 possible (unshared) secret inputs, i.e. we also have 8 Hamming weight groups. This means 8 Hamming weights are compared at every intermediate point of the network. In case we find one unequal Hamming weight for any secret input group, we subtract the value 1 from the leakage fitness value. This subtraction happens for every detected unequal Hamming weight throughout every node in the whole network. With this strategy, we can reach a minimal leakage fitness of $-(8 * n)$ in a network with n intermediate nodes. As with our adder correctness fitness, the perfect leakage fitness is 0, this is why the threshold is also set to that value. Since the fitness value is set with the target value at startup, a perfect network has to maintain that setting. Because the fitness value is only lowered, if we detect different Hamming weights (i.e. leakage) for the secret input groups, the leakage fitness of 0 can be reached when the Hamming weight is equal at every intermediate node. If this goal is met, the Hamming weights of the intermediate values are independent of the secret inputs in the whole network.

In table 5.3, we show an example of a Hamming weight distribution table for a full adder network. This table is created for each network during the leakage evaluation. In this example, it contains the same Hamming weights at every intermediate node/gate

Secret Inputs	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
0, 0, 0	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
0, 0, 1	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
0, 1, 0	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
0, 1, 1	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
1, 0, 0	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
1, 0, 1	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
1, 1, 0	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
1, 1, 1	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4

Table 5.3: Each cell represents the sum of the Hamming weights of the output of one gate (column) grouped by the same secret input (row)

Shared Inputs						Secret Inputs			Output
a_0	a_1	b_0	b_1	c_0	c_1	a	b	c	$a_0 \oplus b_0$
0	0	0	0	0	0				0
0	0	0	0	1	1				0
0	0	1	1	0	0				1
1	1	0	0	0	0	0	0	0	1
0	0	1	1	1	1				1
1	1	0	0	1	1				1
1	1	1	1	0	0				0
1	1	1	1	1	1				0
Truth Table Hamming Weight									4

Table 5.4: Snippet of the truth table input grouping for a single `XOR` node

of the network, meaning this network does not leak secret information according to our definition. Due to the completely even Hamming weight distribution, this network would reach a perfect leakage fitness of 0. Note that each column t_n of the table specifies the Hamming weights per secret input at the intermediate node n.

To further clarify the process of grouping the inputs and measuring the Hamming weights, we depict the assignment of shared inputs to a secret input group in table 5.4. Here, we assume a node carrying out a `XOR` operation, directly on the two inputs a_0 and b_0. We list all corresponding shared inputs to the secret input vector (0,0,0) together with the truth table for the node output and the Hamming weight. This grouping and evaluation is done for all 64 and 8 shared or secret input vectors, respectively. In order to achieve 1st-order side-channel resistance, the measured Hamming weight has to be equal for every secret input at a particular node output (i.e. column in table 5.3).

To conclude this section, we briefly summarize the properties of our two fitness functions in table 5.5. The *Value Interval* represents the possible range of values, one can expect for each fitness part. The upper bound (0 for both) indicates the fitness goal is reached by the corresponding network. With the definitions of the two thresholds and

Function Name	Value Name	Goal	Value Interval
Adder Fitness	add fit	model a correct adder for all inputs	$[-128, 0]$
Leakage Fitness	leak fit	no distance-based 1st-order leakage	$[-(8 * n), 0]$ with n being the number of intermediate nodes in the network.

Table 5.5: Overview of defined fitness functions for the masked adder problem

their corresponding functions, we modeled the requirements for our target genome. For a network to be eligible for the ARX adder use case, it has to be correct (indicated by an adder fitness of 0) and free from Hamming weight-based leakage (indicated by a leakage fitness of 0). Fulfilling only one of the goals completely is not sufficient, because the resulting network would either leak secret information or not provide adder functionality in every input case. Naturally, during the evolution of the networks with NEAT, non-perfect networks will be the majority, especially in the first instantiations of the genetic algorithm. Therefore, it is important that the most promising members of the population are identified and allowed to reproduce in the upcoming iteration.

5.4.3.3 The Selection Strategy

While the two fitness values are necessary for our use case, the use of both leads to a much harder problem. Having two fitness targets transforms the problem to a Multi-Objective Optimization (MOO) task. This means it is harder to judge which network should be allowed to proceed during the selection phase. In a single-objective setup, we can simply sort the genomes by their fitness values and then cut off a certain percentage of the lowest performing networks. However, if we encounter more than one goal, the question is how the members should be sorted before dismissing a part of the population. We experimented with different reproduction approaches to find the most efficient MOO pattern for our problem.

One idea in the field of MOO is to calculate a weighted sum of the n objectives and use that sum as a reproduction indicator in the genetic algorithm. In such a setup, we would sum the adder fitness and the leakage fitness together to obtain one scalar value, which then serves as a combined fitness that indicates the overall fitness of a genome [210]. The problem with this approach is that the importance of each objective needs to be defined *a priori*, meaning a weight for each fitness is to be set to reflect how strong an objective should contribute to the summed fitness. Before the sum is created, each objective value is multiplied by an individual weight. For our masked full adder setup, the weighted sum would be calculated as seen in equation 5.11, with w_1 and w_2 representing the weights

for the two fitnesses. If we considered each objective to be equally important, we would need to set $w_1 = w_2$.

$$fitness = addfit * w_1 + leakfit * w_2 \tag{5.11}$$

An issue with this approach lays in the process of finding the right weights for the different objectives. If we prioritize the adder fitness, the algorithm might converge to a structure fulfilling the adder correctness, but this solution might then have a low leakage fitness. This is especially difficult to handle when the algorithm finds a fully correct but leaking adder network. When the adder fitness is of higher priority, the reproduction logic would not allow a less leaking structure to proceed to the next iteration, if the leakage reduction results in less adder fitness. A similar effect can be observed when the leakage fitness is heavily weighted. The genetic algorithm might then find non-leaking logical networks which are however not representing an addition for all possible inputs. If we reach such a situation due to non-optimal weighting or other factors, we observe that the evolution of the network stagnates at a specific fitness vector, meaning the top fitness vector of the whole population does not further improve over several iterations. Since this stagnation is a common issue in MOO problems, where different objectives compete with each other, research suggests different algorithm tweaks to support converging towards a solution in such scenarios [211][212][213].

Another reproduction method proposed for MOO is the Nondominated Sorting Genentic Algorithm II (NSGA-II) [2]. NSGA-II is based on the idea of extracting the most suitable genomes of the parent and the current population with the help of a special sorting algorithm, taking into account the dominance of a network and its distance to other members. In detail, first the two populations are regarded as one population of the size $2 * popsize$. Through using Pareto-optimal search, NSGA-II finds various nondominated candidates clustered in fronts. A population member is defined as nondominated and part of the Pareto-front if there exists no other member in the current population who is equally fit in all objectives and fitter in at least one objective. If we apply this relation to our adder problem and define the combined population as M, a member of M as m and the nondominated genome as n, a member n is part of a Pareto-front if it fulfills equation 5.12. [1]

$$\begin{aligned} \nexists m \in M((addfit(m) \geq addfit(n) \wedge leakfit(m) \geq leakfit(n)) \wedge \\ (addfit(m) > addfit(n) \vee leakfit(m) > leakfit(n))) \end{aligned} \tag{5.12}$$

NSGA-II is clustering the population members in multiple fronts if the first front is not already exceeding the desired population size for the next evolution cycle. This is achieved by removing the candidates of previously identified Pareto-fronts from the search space for subsequent Pareto-fronts. After the nondominated sorting, the fitter

[1]

Symbol	Meaning
$\nexists$	there is no
$\in$	is member of
$\wedge$	logical and
$\vee$	logical or

upper half of the combined population is transferred to the next iteration. If the end of the least fit transferred front is not directly at the population size barrier, crowded distance sorting is applied to that front. This second sorting step is calculating how similar solutions within one front are by comparing their fitness vectors to each other. If there exist multiple genomes with similar fitness values, they are identified as a crowd. Since NSGA-II is regarding little differences in fitness as little overall difference, only the members of the front that are most distant from each other are kept in the next iteration, if the population size needs to be reduced. This results in all but one member of a crowd being potentially eliminated. Figure 5.6 provides a graphical representation of the main selection principles of NSGA-II. In this figure, we denote the two combined populations as P_t and Q_t. The transition from the initial double population to the clustered bar in the middle illustrates the nondominated sorting. Each front F_n represents a Pareto front. In the transition from the clustered middle to the rightmost bar, the crowded-distance sorting is applied. In this step, a part of the front F_3 is rejected. This is necessary to reduce the size of the resulting population P_{t+1}. This population serves as an input to the next NSGA-II iteration.

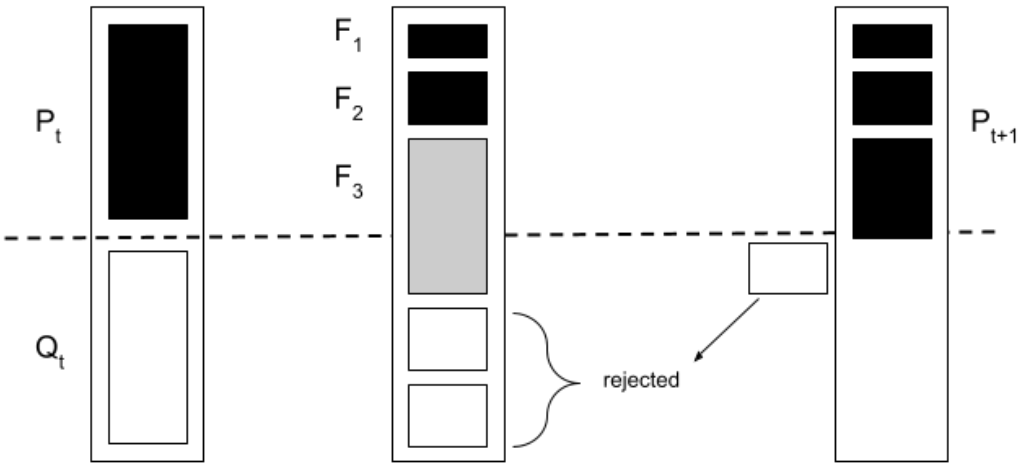


Figure 5.6: Non-dominated (first step) and crowded distance sorting (second step) of the NSGA-II selection [2]

Another rather unconventional selection method is called novelty search. What distinguishes this approach from the ones mentioned before is that it steps away from the idea of basing reproduction (only) on the fitness vector of a genome. Instead, the researchers propose to measure the *novelty* of a candidate and judge it upon this value [214]. This way, the most innovative networks will be advanced into the next iteration, while (some of) the most fit genomes will be eliminated. This strategy rewards networks that deviate from the mass. How the novelty is defined, depends on the specific problem. Sometimes it might correlate with the fitness of a candidate and for other structures it might be completely independent. Obviously, the goal is to define novelty such that it is different

from the fitness evaluation. If not, novelty search will yield similar results as conventional fitness-based approaches.

Risi et al. illustrate NEAT paired with novelty search in a game-based setup. In their example, an agent has to navigate through a maze with the goal to reach a certain target. Once the agent touches a wall, it crashes (fails) and it is again seeded at the starting point. Here, a novelty score can be derived from the taken path of every agent and every iteration. For example, if a candidate tries to take a novel path but crashes immediately, its novelty score would still be very high, resulting in a selection for the next round. In a fitness-based setting, only the early crash would be considered, regardless of how it happened. Then, the selection routine would likely dismiss such an agent, since an early crash and great distance from the target results in a low fitness. In figure 5.7, we depict a maze agent example, similar to the one from the Risi et al. experiment [214]. The pink rectangle marks the starting point of each agent. The goal is to reach the end point (blue rectangle) without touching a wall of the maze. Conventional selection methods based on fitness would rank the agents (circles) according to their proximity to the target. Hence, in this example, the gray agents would be preferred and allowed to reproduce in the next iteration. Novelty search however rewards novel approaches that deviate from the paths that the majority of genomes chooses. In this example, the yellow agents would reach a higher novelty score because their behavior is regarded as innovative compared to the average agent. Note that when novelty search is utilized, the yellow genomes would be selected for the next iteration, despite they finish farther away from the target than the gray ones. The designers of novelty search argue that this selection strategy can be more efficient in non-trivial fitness landscapes that lead to stagnation with conventional approaches. Especially when small alternations of genomes result in big jumps in fitness, novelty search can be a promising alternative to fitness-based evolution techniques.

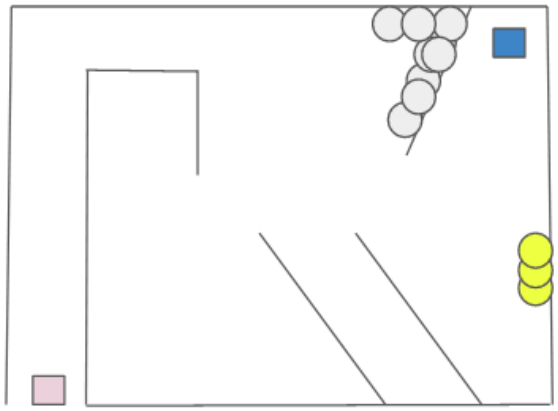


Figure 5.7: Target-seeking agents navigating through a maze structure

While the novelty approach is easy to fit onto the maze problem, defining novelty for our use case is more demanding. We are operating on Boolean networks and we do not have a fixed target or a comparable maze-like structure. Moreover, we cannot completely abandon the fitness function(s), because we need them to ensure that our

resulting network is a correctly formed adder. Without evaluating the adder and leakage fitness, we have no way of judging the genomes at all. The only information we can use in our evaluations, is the Hamming weight distribution at the intermediate states and the output vector produced for each input. However, we can still use this information to manufacture a partly novelty-based selection. For example, we can save an output tuple (c_{out}, s) per input stimulation for every network. For every network, we can store a vector of 64 of these output tuples $((c_{out}, s)_0,..,(c_{out}, s)_{63})$, one for every input stimulus. As a 65th value, we can add the leakage fitness associated with the evaluated network. Then we can count the occurrences of every output vector/leakage combination and reward the networks with those combinations that have rarely been seen within the current population or even across iterations. With that strategy, we can manage to mount a hybrid novelty-fitness technique on our masked adder problem. With the leakage measurement, we still include a part of our fitness function, while the output matrix is exploited to integrate the novelty portion into the selection phase.

5.4.4 Optimizing Masked Addition with Guided Exhaustive Search

Related work shows that optimal 1st-order Boolean masks for single operations can be found by exhaustive search [193, 215]. The execution time however depends heavily on the input size, the target platform and the included instructions. The maximum search space can be estimated by calculating the number of columns inside the produced truth table. This column size is defined by $2^{(2^{(n \cdot m)})}$, with n being the number of inputs and m representing the number of shares. Every expression combination based on the inputs will create one of these columns, so the search for a valid masking can be translated to a search of n columns that deliver the desired unmasked output.

If we for example consider a 1st-order masked `AND` operation, a basic exhaustive search algorithm provides an optimal six-instruction-long implementation after 11539239 iterations. For this problem, the set of possible input combinations has a size of 2^{16} for this two-input `AND` gate with two shares. However, if we try to solve our masked full adder problem, we are faced with three inputs and two shares, leading to a total number of 2^{64} combinations. This problem space can no longer be evaluated in an acceptable time frame on a standard personal computer, meaning no solution can be delivered after several days of processing.

Still, we will present a modified exhaustive search approach for our full adder problem as an alternative to the neuroevolutional technique. This technique is more similar to the related work that we just discussed and can be applied to our problem after some modifications. Although we change the searching technique, we do not shift our optimization goal. Instead of focusing on parts of a PPA, we search for a 1st-order masked full adder. This allows us to evaluate two things: On the one hand, we can analyze if the full adder target yields better or worse results compared to the PPA-based related research and on the other hand, we can investigate if a neuroevolutional approach can outperform conventional methods in this masking use case.

While the following exhaustive search algorithm is fundamentally different from the genetic operations happening within NEAT, we still adopt some properties from our

machine-leaning setup. First, we also include the leakage detection mechanism, in which the uniformity of the Hamming weight distribution is checked with the help of a table, grouped by secret input (see table 5.3) [209]. We are able to conduct this leakage evaluation since we calculate and store a truth table column of the outputs of every node during runtime. Despite this additional feature, our first representation of the exhaustive search algorithm is a modified version of the one from Biryukov et al., of course setup to fit our full adder problem [193]. This unguided version produces a large truth table, where all possible input combinations meeting the leakage requirements are saved in columns. Every node of the network is represented by a tuple (n, f, x, y), with n being a 64-bit integer value that encodes the 1-bit node output corresponding to the set of input values. f specifies the logical operation (e.g. `AND`) that is carried out on the input columns. The values x and y are indices of the two operand input columns stored in the truth table. The output n of the current node is defined by $f(t_x, t_y)$, with t_x and t_y representing the columns at positions x and y in the truth table t. This data structure allows us to derive the performed operations that led to the column values in the truth table.

When the search algorithm is initiated, we fill the truth table with six columns, including the identity function of the shared adder inputs. In every iteration, more columns are added to the truth table. We combine any available columns with every available operation. We only exclude those that do not fulfill our leakage requirement regarding the uniform Hamming weight distribution. If we find a column that is logically identical to one we have already produced, this new column is disregarded. That is because due to the growing nature of the algorithm, the older combination has to have reached the same result with less cost (i.e. instructions). In case `XOR`ing two columns results in one of the desired outputs s or c_{out}, that search output is marked as reached. When all search outputs have been found, the algorithm terminates.

5.4.5 Results

In this section, we will present the resulting adder networks found with our two different approaches, neuroevolution with a genetic algorithm and guided exhaustive search. Note that in both searching strategies, we restricted the gate/node count to 21, since using the most recent techniques from related work, one can build a masked full adder with 22 instructions. While this makes our problem harder, we can guarantee that if we find a result, it will represent a more efficient full adder than one that can be created with the current state-of-the-art.

In the NEAT setup, the most crucial part of the algorithm was the selection strategy. We implemented all different MOO methods mentioned in section 5.4.3.3. We could not reach desirable results with standard fitness-based techniques like weighted sum or the Pareto selection within NSGA-II. The two fitness goals, (adder) correctness and 1st-order protection were working too much against each other, so we either terminated with a non-leakage free adder or a protected structure that was not correct in the sense of the full adder definition.

Since in our use case a small mutation of network can result in a large (logical) change, we finally moved our selection towards novelty search. Consider a genome with an `AND` aggregation at an intermediate node. If during the mating phase this aggregation function is changed e.g. to an `OR` gate, this can lead to a completely different output and therefore largely affects the correctness, i.e. adder fitness. This is also true for a mutation of a connection, which could e.g. be represented by a rerouting of an input line. We modelled a variant of novelty search which fits our Boolean problem, in order to respect this non-continuous behavior of the system. We keep the leakage fitness from the other selection techniques and replace the adder fitness with a list of output vectors produced per input stimulation. We save the combination of outputs and the leakage fitness in a tuple. During each algorithm iteration, the number of occurrences of these tuples is then saved. We keep these measurements stored across iterations and can hence evaluate how often a certain output/leakage tuple has already been created since the start of the algorithm execution. We rate the novelty of a network upon how often its tuple has been counted in the past. The lower the occurrence count, the higher the novelty score. The population of networks is sorted by this novelty value and the top n genomes with the highest novelty rating are allowed to proceed into the next phase. If networks have an equal novelty rating, they are then conventionally sorted by their adder fitness, leakage fitness and lastly by their size (networks with fewer nodes are preferred). This way we prefer correct structures to leakage-free networks, but only if they have the same novelty score.

With this novelty search selection, we recognize better performance compared to the previously implemented MOO evaluations. This modification allows us to evolve genomes with an (adder, leakage) fitness vector of (0, -4), i.e. the algorithm is capable of finding full adders, but they still leak secret information. The leakage score is however also close to the optimal, considered that a leakage fitness of -4 means that only four uneven Hamming weights in the whole network lead to leakage. Often, this leakage originates from a single intermediate node.

Because the fitness of our best solutions was already very close to optimal, we introduced a second NEAT run with the goal to apply error correction to the previously evolved network. Our best solutions showed leakage only at one node, that was as well only contributing to a single output share (c_{out0} or c_{out1}). This means the basic novelty search algorithm found an applicable solution for 3 out of 4 outputs. With that knowledge, we set up a second NEAT stage with the goal of finding a network that produces the single output without causing leakage. Since the network already reaches full correctness with an adder fitness of 0, we know which exact outputs have to be calculated by the network for our missing share. We initiate a second NEAT run with the goal to find a logically equivalent network part for the single share that does not introduce leakage. In this network, we can make use of all six inputs, but have to find only one leakage-free output, instead of four. This simpler problem in the second-stage can be fully solved by our modified NEAT algorithm, again while using novelty search. When we find a leakage-free logical clone for the share in question, we can replace this path of the old network with the new representation. With this approach, we eliminate the

remaining leakage and maintain the correctness of the previous network – resulting in a masked adder network with the fitness vector (0, 0).

Figures 5.8 and 5.9 depict the result networks from our first NEAT run and the second error-correcting stage. Both gate nets are valid adders, the one in figure 5.8, however, does not lead to a uniform Hamming weight distribution at the intermediate node 10. Since node 10 is only contributing to output share c_{out1}, we can instantiate a second NEAT run to find a leakage-free logical twin of the path to output c_{out1}. Figure 5.9 shows the adder network with the patched part for this share. This circuit consisting of 14 gates is a correct full adder with no 1st-order leakage according to our evaluation explained in section 5.4.3.2.

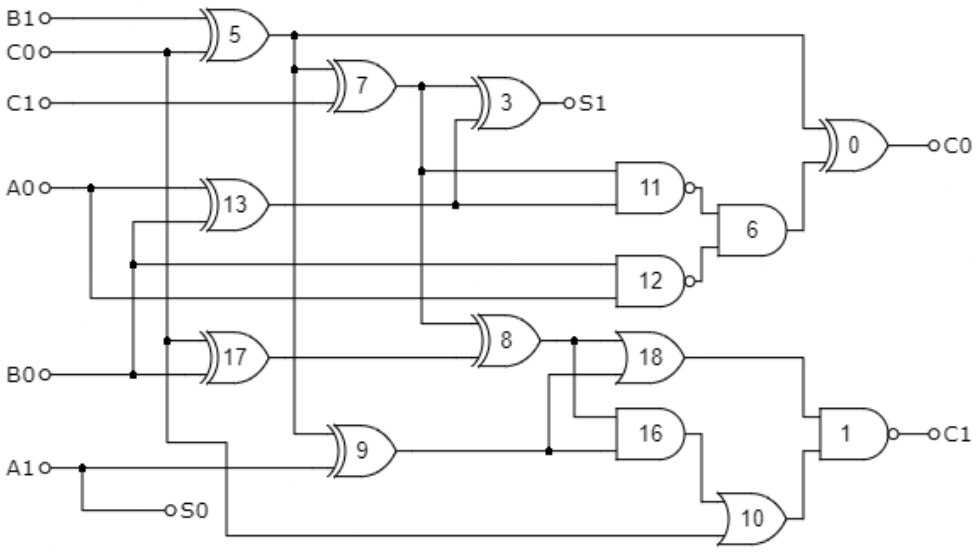


Figure 5.8: Shared full adder with distance-based leakage at node 10

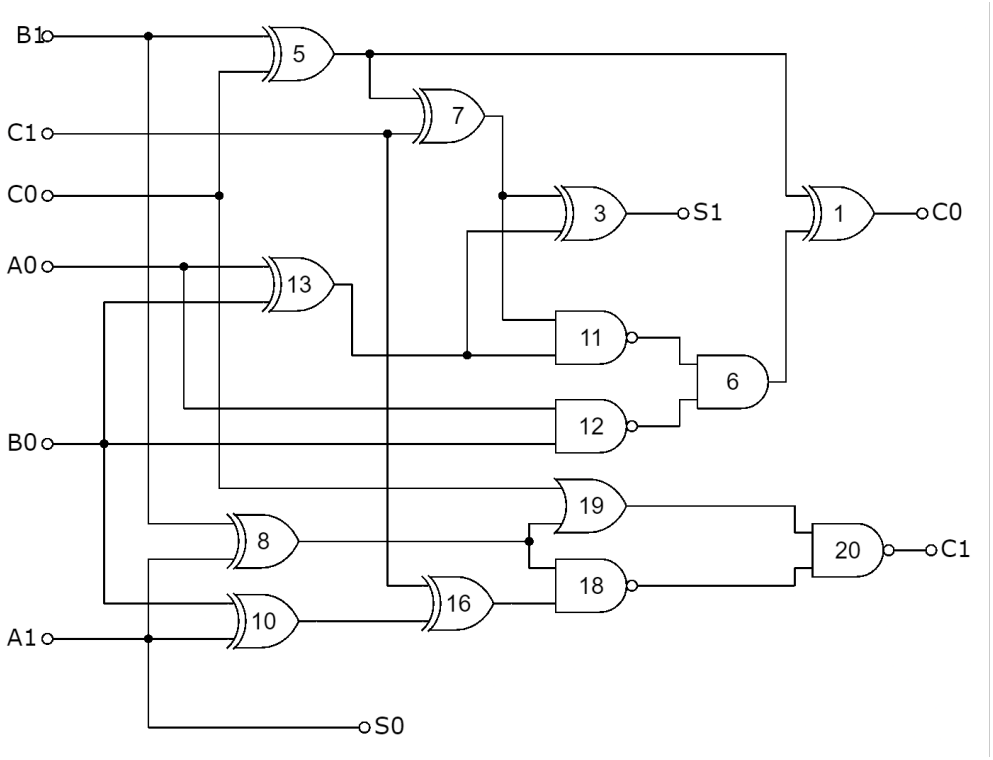


Figure 5.9: First-order leakage-free shared full adder network

Our configuration file for the NEAT implementation can be found in appendix B. We limit mutations to at most one per network/iteration in order to avoid too significant changes at a time. A mutation can be either deleting a connection, adding a connection, deleting a node, adding a node or altering the logical function of a node. We utilize a custom method to specify how nodes should be initially connected when the networks are spawned. This method is mentioned in the configuration file (*neat_double*) and makes sure that first each node is connected to exactly two inputs. While this can change during evolution, this way we can preset the networks towards two-input gate circuits.

We can slightly modify our result network such that it matches the relaxed TI requirements set by Jungk et al. in their side-channel research [194]. This way, we manage to form a solution that has very similar properties as related work and create a fair baseline for a (performance) comparison. We need to substitute the `NAND` aggregation of gate 20 with a `XOR`. That is possible because this gate never receives two zeros as an input and regarding the rest of the inputs a `XOR` and a `NAND` gate are logically identical. The reason why an input vector of (0, 0) is never occurring at this gate, is that its two inputs are outputs of a `NAND` and `OR` aggregations that share a single input. With the gates 1 and 20 now being a `XOR` operation, we can more easily observe that these gates are collapsing 4 non-uniform output shares into 2 uniformly distributed output shares in the adder network. This collapsing layer is required for software TIs as introduced by

Jungk et al. and because of this gate modification we can state that our network is also a representation of such a software TI.

With the ideas of the software TI in mind, we now look into which results we can achieve with our non-neuroevolution-based algorithm. In the form described in section 5.4.4, the exhaustive search algorithm still takes a large number of iterations and exceeds the available dynamic memory on a modern personal computer after hours of runtime. To reduce the search space, i.e. the number of necessary iterations, we use the fact that the adder output s can be obtained with `XOR` gates only. We only need to be careful to not combine two shares of the same variable to not introduce leakage into the calculation of s_0 and s_1. We divide the inputs in two groups, one group containing the first share x_0 of each input x and one group including all second shares x_1 of the inputs. Then, all possible `XOR` input combinations across the two groups are calculated. That results in 114 layers, of which each can represent a column in our truth table. We call this set of `XOR`-networks the *linear expansion layer*. This layer contains multiple valid gate nets to calculate the output s. However, we will determine which one of these we use after we have knowledge of the whole network. Later, we choose the input combination that allows us to reuse the most gates for other operations. This way, we can minimize the number of required gates.

In the search for the next adder layer, we include the `XOR`-columns as a baseline. We perform a single iteration, in which we expand the set of allowed bitwise instructions. For example, for an ARM Thumb-2 target, these operations are `EOR`, `AND`, `ORR`, `BIC` and `ORN`. Since in this iteration also non-linear input combinations can occur, we call the resulting columns the *non-linear layer*. After this step is performed, we can already produce a non-uniform output for c_{out}, which is split into four shares.

We recall the definition of a software TI from Jungk et al. and run another algorithm iteration to find expressions that collapse this non-uniform four-share representation into a uniform two-share output. In this phase, as in the linear expansion step, we only allow linear `XOR` combinations. The columns of this *share collapsing layer* contain a solution for a uniform two-share output c_{out}. Starting from the solution for c_{out}, we can now inspect the truth table backwards and construct the 1st-order secure full adder network with the information stored in the table tuples mentioned in section 5.4.4. The described searching procedure is presented in algorithm 1. Note that the *availableInputs* structure contains the truth table columns in the format (n, f, x, y), as it was introduced in the previous section. These tuples need to be preset with $(i, none, none, none)$ for every input share i. The algorithm represents the optimized 3-layered search technique that was just explained.

The most efficient adder found with the depicted search algorithm requires 12 instructions, of which 11 are used to compute c_{out}. This adder is defined in algorithm 2, the gate network is shown in figure 5.10. The unnumbered `NOT` gate does not translate to an additional instruction, since ARM offers the `BIC` instruction that performs an `AND` and a `NOT` operation in one cycle.

Algorithm 1 Layered search algorithm for masked full adder

Require:
$a = a_0 \oplus a_1; b = b_0 \oplus b_1; c_{in} = c_0 \oplus c_1$
$operations = EOR, AND, ORR, BIC, ORN$
Ensure:
$s = s_0 \oplus s_1; c_{out} = c_0 \oplus c_1$

1: $availableInputs \leftarrow (a_0, a_1, b_0, b_1, c_{in0}, c_{in1})$ ▷ init array of inputs
2: $maxNetworkSize = 21$ ▷ upper bound, less gates than related work
3: $networkSize = 0$

4: FindNextLayer(EOR, availableInputs) ▷ get linear linear

5: **while** $networkSize < maxNetworkSize$ and c_{out} not found **do**
6: findNextLayer(operations, availableInputs) ▷ build non-linear part
7: **end while**
8: **if** c_{out} was found **then**
9: findNextLayer(EOR, availableInputs) ▷ find share collapsing
10: **else**
11: unsuccessful termination
12: **end if**

13: **function** findNextLayer(instructions, inputs)
14: **for** inputPair in inputs **do**
15: $in_0, in_1 \leftarrow inputPair$
16: **for** instruction in instructions **do**
17: $newNode \leftarrow (output, instruction, in_0, in_1)$
18: **if** $newNode$ does not leak and $newNode$ is no logical duplicate **then**
19: $availableInputs = availableInputs + newNode$
20: **end if**
21: **end for**
22: **end for**
23: $networkSize = networkSize + 1$
24: **end function**

Algorithm 2 2-shares masked full adder

Require: $b = a_0 \oplus a_1; b = b_0 \oplus b_1; c_{in} = c_0 \oplus c_1$
Ensure: $s = s_0 \oplus s_1; c_{out} = c_0 \oplus c_1$
1: $t_1 \leftarrow a_1 \oplus c_1$
2: $t_2 \leftarrow c_0 \oplus t_1$
3: $t_3 \leftarrow a_1 \oplus b_1$
4: $t_4 \leftarrow a_0 \oplus b_1$
5: $t_5 \leftarrow a_1 \oplus b_0$
6: $t_6 \leftarrow t_4 \oplus t_2$
7: $t_7 \leftarrow t_5 \wedge t_2$
8: $t_8 \leftarrow t_4 \wedge \neg t_2$
9: $t_9 \leftarrow t_3 \vee a_0$
10: $t_{10} \leftarrow a_0 \wedge b_0$
11: $t_{11} \leftarrow t_9 \oplus t_{10}$
12: $t_{12} \leftarrow t_8 \oplus t_7$
13: $s_0 = b_0; s_1 = t_6$
14: $c_0 = t_{11}; c_1 = t_{12}$

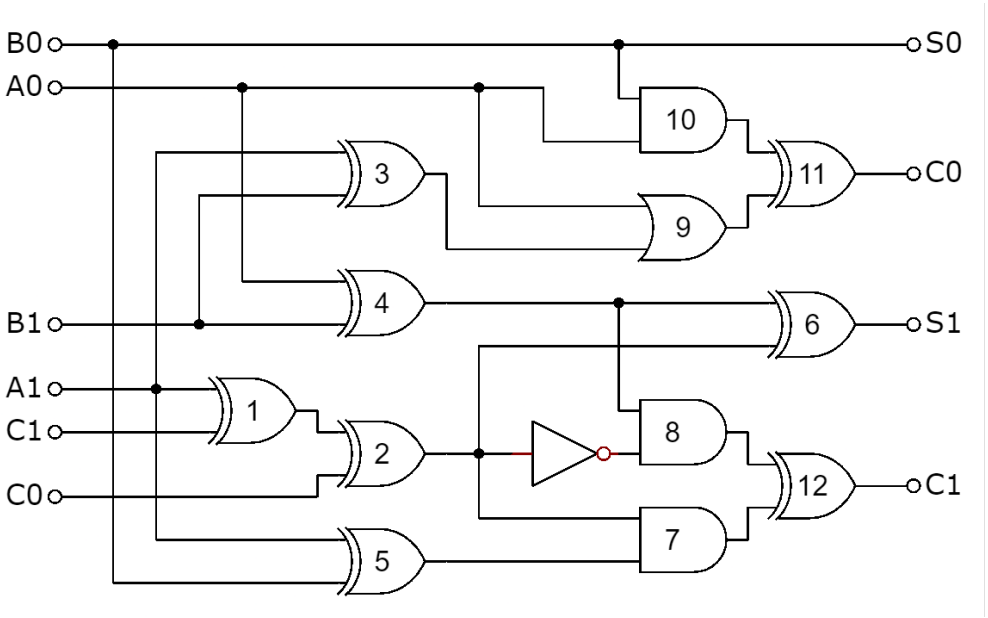


Figure 5.10: First-order leakage-free shared full adder network implemented using 12 ARM Thumb-2 instructions

5.4.6 Application to Software Encryption

The primary goal of this research is to optimize the protected addition, which should eventually lead to better performing masked software implementations of lightweight ARX ciphers. In order to use and evaluate our developed adders, we have to translate the gate networks in assembly implementations. Since we target an ARM implementation, we have to substitute logical expressions that are not available on this architecture with appropriate instructions. We rearrange the structure of the neuroevolved network such that we can use the `BIC` (`AND NOT`) and the `ORR` (`OR NOT`) instruction instead of a `NAND` and a `NOR` gate, respectively. Moreover, we apply DeMorgan's laws to achieve the most efficient ARM assembly representation from the masked gate network. Obviously, we could have limited our search algorithm(s) to exact ARM instructions only to be able to directly use the resulting instruction network in an assembly implementation. However, with the more broad logic gate approach, we get a more generic solution, which could be adapted for multiple instruction sets.

We depict the ARM Thumb-2 assembly implementations of the neuroevolved and the exhaustive search-based masked full adder in Listings 5.1 and 5.2.

```
1  // r2, r3 are shares of A
2  // r4, r5 are shares of B
3  // r0, r1 are shares of C
4      eor    r6, r2, r4  // gate 13
5      and    r2, r4      // gate 12
6      eor    r4, r3      // gate 10
7      eor    r7, r3, r5  // gate 8
8      eor    r4, r1      // gate 16
9      eor    r5, r0      // gate 5
10     orr    r0, r7      // gate 19
11     eor    r8, r5, r1  // gate 7
12     and    r4, r7      // gate 18
13     and    r1, r6, r8  // gate 11
14     bic    r0, r4      // gate 20, output C1
15     orr    r1, r2      // gate 6
16     eor    r6, r8      // gate 3, output S1
17     eor    r1, r5      // gate 1, output C0
18 // r0, r1 are the output carry shares
19 // r3, r6 are the output sum shares
```

Listing 5.1: ARM assembly implementation of our full adder evolved with NEAT and novelty search. Gate numbers in comments reference Figure 5.9.

```
1  // r2, r3 are shares of A
2  // r4, r5 are shares of B
3  // r0, r1 are shares of C
4
5  // linear expansion layer
6      eor    r1, r3    // gate 1
7      eor    r0, r1    // gate 2
```

```
 8       eor     r1, r5, r3 // gate 3
 9       eor     r5, r2     // gate 4
10       eor     r3, r4     // gate 5
11       eor     r6, r5, r0 // gate 6, output S1
12   // non-linear layer
13       and     r3, r0     // gate 7
14       bic     r0, r5, r0 // gate 8
15       oor     r1, r2     // gate 9
16       and     r2, r4     // gate 10
17   // share collapse layer
18       eor     r1, r2     // gate 11, output C0
19       eor     r0, r3     // gate 12, output C1
20
21   // r0, r1 are output carry shares
22   // r4, r6 is the new output sum share
```

Listing 5.2: ARM assembly implementation of our full adder developed with guided exhaustive search. Gate numbers in comments reference Figure 5.10.

5.4.6.1 Shared Bitsliced Adder Implementation

The most popular ARX algorithms use a bit width of 32 in their addition operations. This means our shared full adder has to be applied 32 times in a bitsliced implementation. While bitsliced implementations are useful to avoid timing attacks on AES, they are not common for ARX ciphers [216]. This is due to the property that ARX designs can usually be implemented in constant-time *per se*, and are therefore inherently protected against timing attacks. So in general, our bitsliced approach will have a performance disadvantage compared to non-bitsliced related work, because the slicing before and the combination of the bits after the addition naturally introduces an overhead.

To produce a n-bit adder, we have to cycle through n iterations of the shared full adder. In our implementation, we assume that the shares for a and b can be read from memory and that output s can be stored into the registers of a, effectively overwriting the previously stored shares. In the first sliced iteration, the input carry bit c_{in} will always be 0, which requires its two shares (c_{out0} and c_{out1}) to be equal. However, setting both input shares of c_{in} to 0 violates the uniformity property of the inputs. Therefore, the two shares need to be initialized with the same random value. This amount of required randomness is expected and equal to related work [194]. The bitsliced adder is implemented such that the output (c_{out0}, c_{out1}) of iteration i becomes the input (c_{in0}, c_{in1}) of iteration $i+1$. Note that the registers of the input and output carry shares are the same in the assembly implementations, meaning no extra `mov` instruction is required to move the outputs into the registers of the next inputs.

5.4.6.2 Benchmarking

Similar as in our LWC MCU benchmarking, we analyze the performance of our two adder solutions in a full ARX implementation. Therefore, we provide 1st-order masked implementations of two popular lightweight ARX designs: ChaCha20 and CRAX. CRAX

represents a novel block cipher based on the *Alzette* ARX-box that is also used in the NIST LWC finalist algorithm Sparkle [217]. ChaCha20 is a well-researched symmetric cipher that is often referred to as an alternative to AES and has been standardized for use in the TLS protocol in RFC 8439 [218].

Note that all of the evaluated implementations in this section, ours and the ones from related work, are only protected against distance-based 1st-order leakage. Other leakage sources, that can be relevant in practice, are not taken into account in these implementations. We followed this strategy to be able to provide implementations that are easily comparable to the latest related work (which does not protect against non-distance-based leakage as well).

The presented speed benchmarks (see figure 5.12) have been carried out on a target device for LWC, the STM32F103 MCU including an ARM Cortex-M3 chip. We measure how many cycles the device needs to encrypt a payload of 512 bytes with different implementations of the ARX cipher(s). We provide performance benchmarks for the following ARX variants:

- ChaCha20 (unprotected, with our NEAT adder, with our exhaustive search adder, with the masking from Jungk et al. [194])

- CRAX (unprotected, with our NEAT adder, with our exhaustive search adder, with the masking from Jungk et al. [194])

For our bitsliced implementations, we show the cycle count including and excluding the overhead caused by bitslicing. Note that these implementations will perform worse in case the input size is not a multiple of 32 bits. This additional penalty is caused by the inherent design principle of bistliced implementations. However, the performance of bitsliced implementations rises with a greater plaintext size. This leads to bitsliced variants outperforming conventional implementations even for non-optimal input lengths, when the input size is large enough ($> 1KB$).

We illustrate this behavior in figure 5.11. One can observe that our best bitsliced adder implementation (plotted in orange) performs worse than related work for certain small input sizes that do not allow for great parallelization of the slices. However, the advantage of the more efficient masked adder operation grows with a growing input size, leading to higher speeds of our adder implementations, which eventually outperform related work (plotted in blue) for larger message sizes.

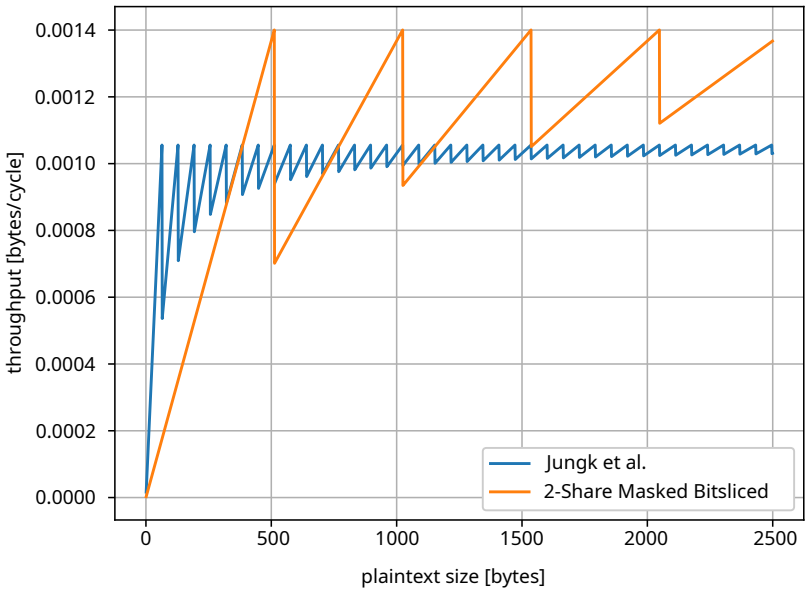
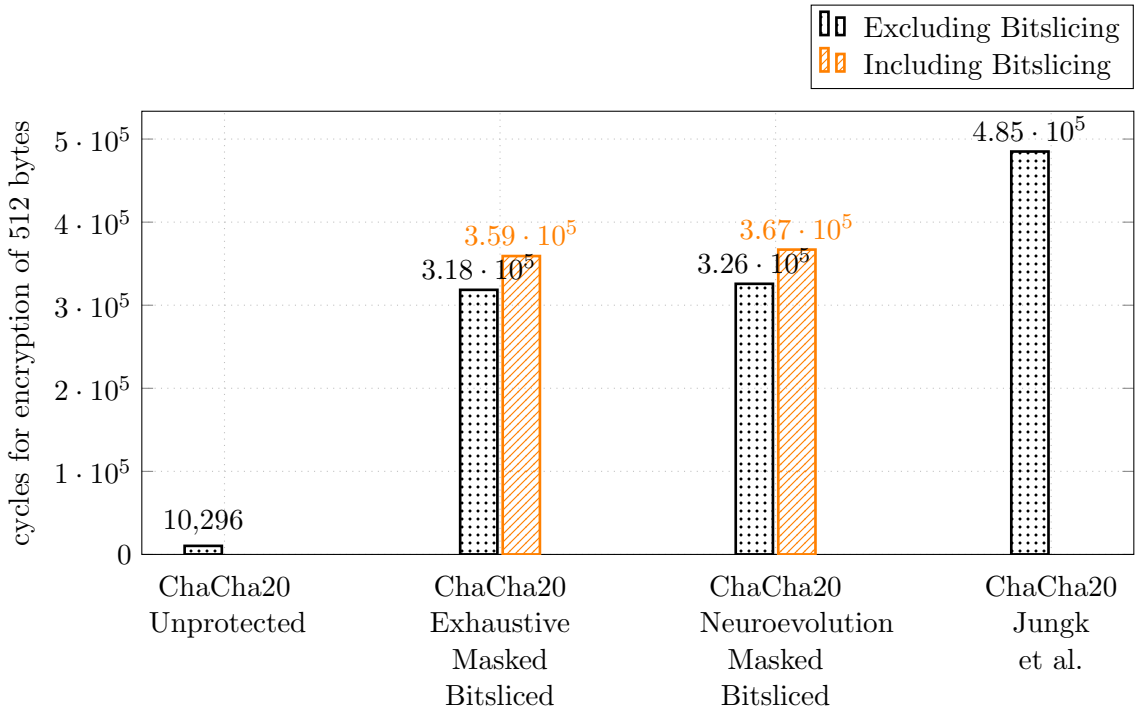
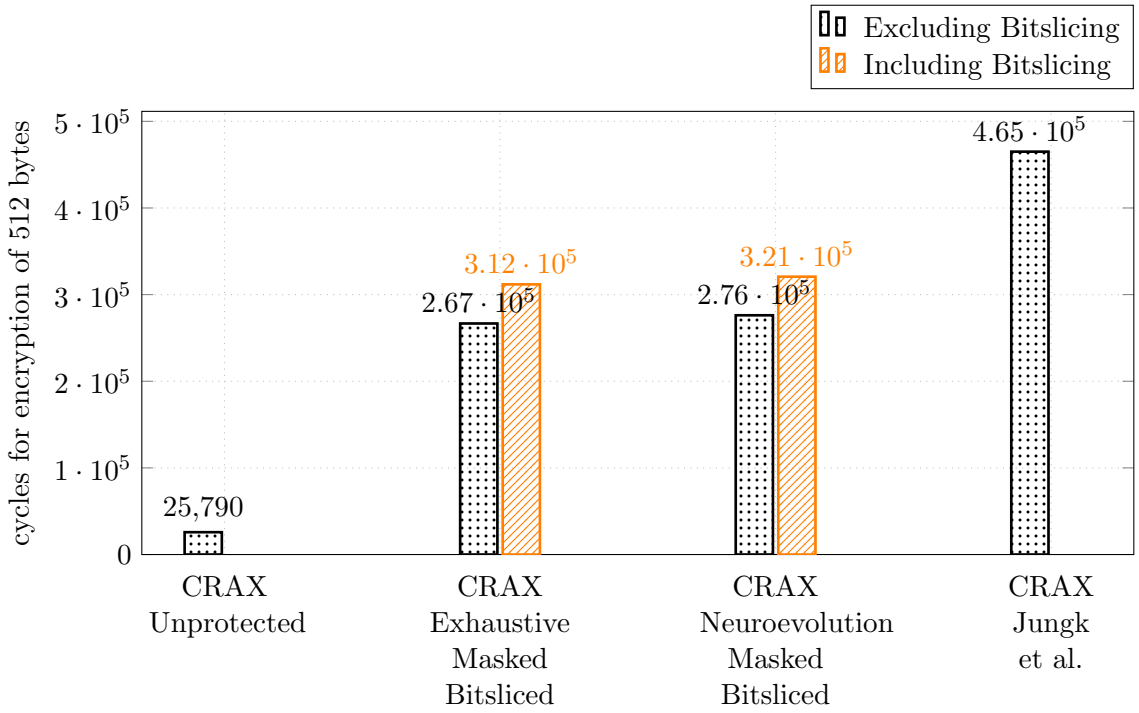


Figure 5.11: Throughput comparison of protected ChaCha20 implementations

If we compare our two approaches to each other, we can see the same tendencies for both ARX ciphers. The implementation with the exhaustive search adder runs faster than the one with the neurevolved solution. This is expected, since the former contains a masked adder featuring 12 instructions and the latter contains one requiring 14 instructions. The performance difference is however not as big as one might imagine. The reason for this result is that the adder only represents a small amount of the whole ARX cipher implementation. The other operations within the cipher design and the bitslicing routines also contribute largely to the cycle count, which results in a 2-instruction-difference per (bitsliced) addition being noticeable but not crucial. All in all, the version including the exhaustive search adder requires 2.8% percent less cycles than our own competitor in the CRAX use case, and 2.2% percent less cycles in the ChaCha20 benchmark.



(a) Cycle counts for ChaCha20



(b) Cycle counts for CRAX

Figure 5.12: Speed benchmark results for different ARX implementations

Implementation	Code	Memory
Unprotected	3174	228
Best Boolean Masked Bitsliced	1024	2260
Jungk et al. Masked	1212	316

Table 5.6: Code size and memory requirements in bytes for a 512-byte encryption with different ChaCha20 implementations

If we take a look a how much penalty the protection of the implementations causes in the 512 byte use case, we recognize a penalty factor of 12.1 for our best masked CRAX implementation and a penalty factor of 34.9 with our best ChaCha20 implementation. We can however reach a speed up of 33% and 26% with our protected implementations compared to the related work from Jungk et al., respectively. Still, the masking penalty is significantly higher in comparison to the one measured for a comparable 2-share implementation of Ascon. In the ChaCha20 example, the 1st-order protection is 6.1× as costly as for Ascon. The masking of CRAX slows down the implementation roughly twice as much as it is the case for the NIST LWC candidate.

We finally compare the memory footprints of different ChaCha20 implementations in table 5.6. While the code size of our bitsliced approach is lower than in related work, our implementation requires much more dynamic memory during execution. This is a general downside of bitsliced designs. Since bitslicing heavily exploits parallelism, many values have to be held in memory. In our ChaCha implementation, we e.g. process eight blocks in parallel on the 32-bit ARM core.

5.4.7 Leakage Evaluation

We conduct a simulated leakage assessment of our implementations to validate our protection techniques. Similar to related work, we run a fixed vs. random t-test (see section 5.3.2) in a simulation environment [193]. We choose MAPS as an evaluation tool, because it has been developed for the ARM Cortex-M3 MCU and has been used in related scenarios in the past [3]. Then, we generate two sets of traces, one with a constant and one with random plaintexts, while the key remains the same for all captures. Figure 5.13 shows the result of the fixed vs. random t-test. MAPS cannot detect any distance-based leakage in our ChaCha20 implementation.

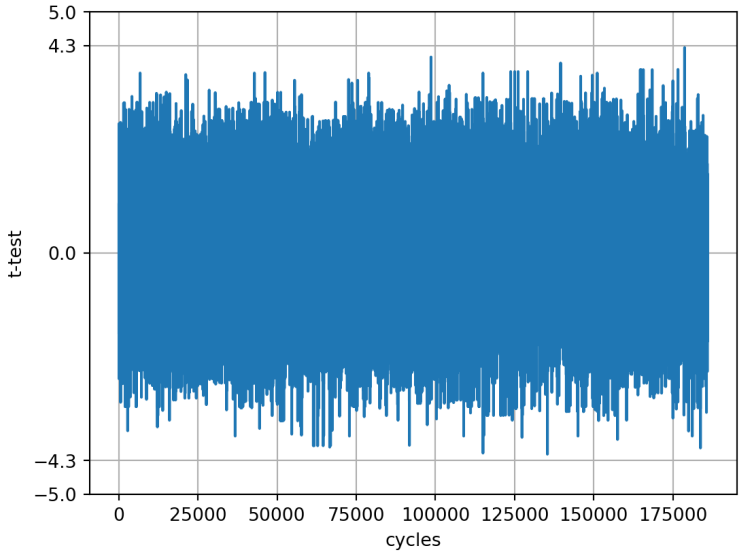


Figure 5.13: Plot of the t-test performed across 1000000 power traces of the ChaCha20 encryption simulated in MAPS, with pipeline leakage simulation disabled [3]

5.4.8 Side-Channel Leakage in Practice

Even if the leakage simulation with MAPS shows no distance-based leakage in our implementation, it is still possible that sensitive values are exposed when the algorithm is executed on real hardware. This is because there are more leakage sources than Hamming weight/distance leakage on an embedded chip. These hardware related leakages are caused by e.g. leftover values in the pipeline or memory registers like the Memory Data Register (MDR) and the Memory Address Register (MAR).

More specifically, when evaluating the leakage of a software encryption algorithm on real hardware, the following additional leakage sources need to be taken into consideration:

- Pipeline leakage
- Memory registers leakage (MAR, read MDR, write MDR)
- Leakage caused by branch prediction and speculative execution

To evaluate our implementation against all these leakage sources, we set up another t-test with real-world data, captured on actual hardware. We execute our ChaCha20 implementation on the STM32F103 MCU. Power traces are acquired using a Picoscope 6000E sampling at 2.5 GHz using a 10:1 300MHz oscilloscope probe. We use a shunt resistor with 22Ω resistance between the power supply and the MCU. Moreover, all bypass capacitors were removed from the MCU and placed on the power supply side of the shunt to ensure maximal bandwidth of the acquired traces.

Figure 5.14 shows the result of the t-test for our ChaCha20 implementation that showed no distance-based leakage on MAPS (see figure 5.13). Clearly, this software still leaks sensitive information through other sources. Note that while we only show this hardware t-test for our standard masked implementation for comparison reasons, the implementations from related work also show similar leakage patterns. This leakage occurs since none of these implementations is additionally guarded against advanced micro-architectural leakage.

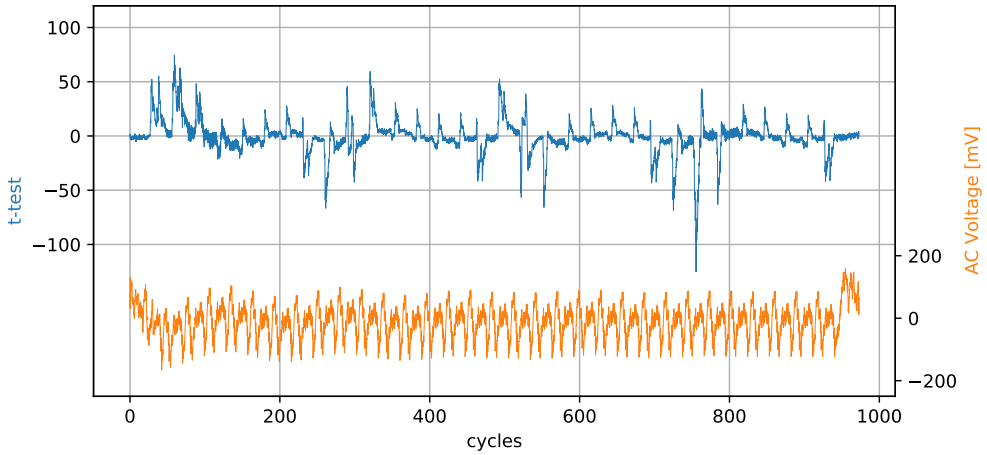


Figure 5.14: t-test result plot of our standard masked ChaCha20 implementation, featuring power traces acquired on the STM32F103

The main source of pipeline leakages is the reuse of the A and B registers, which are used as inputs for the ALU. For removing these leakages, the Thumb instruction `oor.n r0,r0` is inserted between instructions to overwrite the ALU registers. We found that in some cases an additional `nop.n` instruction is required to further separate stages of the pipeline. Obviously, the addition of these instructions introduces a large penalty in both the speed and size of the algorithm.

Since inputs and output of the adder are loaded and stored from and to the memory of the MCU, the MDRs can also be a source of leakage. In Cortex-M3 processors, load and store operations use two different MDRs, so the leakage of each must be considered independently. For removing the leakage from the read MDR used in the load operation, it is sufficient to reorder the load of the inputs in such a way that the two shares of the same secret input are not read consecutively, so the input shares are read in the order a_0, b_0, a_1, b_1. Overwriting the write MDR can be done by simply writing a constant value in an unused location in the stack.

Finally, we found that leakage happened when the destination address of the branch was a multiple of 4 and/or when the address of the branch instruction was not a multiple of 4. Our hypothesis for this is that the limited speculative execution of the Cortex-M3 would attempt loading registers in early stages of the pipeline when branches are incorrectly predicted, causing leakage.

After implementing countermeasures for all the described sources of leakage, the number of cycles per iteration of the proposed bitsliced masked implementation almost doubles. However, the implementation of these countermeasures could likely be optimized more, leading to less additional penalty. Unfortunately, the latest related work does not provide masked implementations that are also guarded against e.g. pipeline-based leakage, i.e. we cannot conduct another performance comparison including the better protected variant.

Figure 5.15 shows the t-test plot featuring the traces acquired from the STM32F103 running our ChaCha20 implementation, including the countermeasures described above. As we can see, the introduced countermeasures eliminate any kind of leakage, which results in a true 1st-order secure implementation of the ARX cipher ChaCha20.

A complete ARM Thumb-2 assembly implementation of one of our shared bitsliced 32-bit adders with countermeasures against memory and pipeline-based leakage can be found in appendix C.

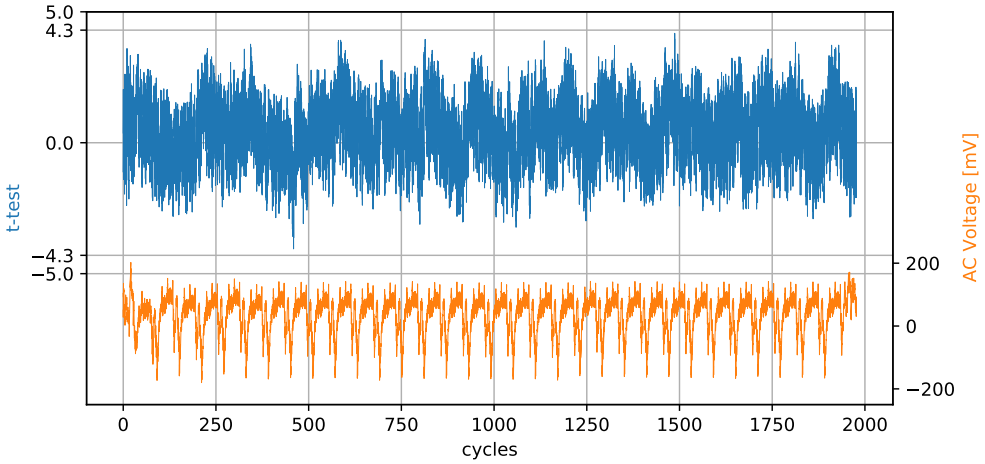


Figure 5.15: t-test result plot of our masked ChaCha20 implementation (incl. additional countermeasures), featuring power traces acquired on the STM32F103

The evaluation in a real-world scenario reveals that *leakage-free* can mean two different things in theory and practice. While dependency-based leakage assessments – as used by related work and our own search algorithms – help in detecting distance-based leakage, additional countermeasures have to be taken into account to ensure no leakage is induced by other hardware-related effects. Recognizing and eliminating these additional leakage sources without conducting specific experiments on the target hardware poses a difficult challenge in the field of side-channel research.

6 Use Case: Lightweight Cryptography in Battery-powered Environments

As discussed in previous chapters, typically three metrics are evaluated when benchmarking software implementations of (cryptographic) algorithms: speed, binary size and RAM utilization. While computing power and memory capabilities are usually very limited on resource-constraint platforms, in certain use cases also the energy consumption of a device can have a high priority. This is especially true for environments in which no permanent electricity can be supplied and where the target systems have to be available for many consecutive years.

Often, the appointed embedded systems have to be powered by a battery that guarantees a certain minimum lifetime before maintenance or replacement is required. In these scenarios, the device can be placed in remote locations that are not always easily accessible after the installation process. Examples for such setups are various kinds of sensors, e.g. temperature or humidity sensors in the nature, as well as water meters or other monitoring devices in industrial facilities. Depending on the use case, these devices have to have a (wireless) interface to be able to transmit or receive data concerning operation modes or measurement results. To enable that, usually well-established wireless protocols are adapted. These again might require the transmission to meet certain security goals, especially if the information is sent over the air.

6.1 Cryptography in Energy-constraint Infrastructures

When the communication demands protection due to the used protocol or the requirement of the scenario, cryptography is needed within the protocol stack. As of now, typically a mode of AES is standardized for many communication protocols. However, if we focus on battery-powered and low-performing embedded systems with long lifespans, LWC ciphers could present an alternative to the current standard for symmetric encryption. When the energy consumption of a system is highly prioritized, rather than its processing speed or low memory footprint, the design and implementation of its firmware still plays an important rule. Even if obviously the chosen hardware platform and its realization also has a great impact on power consumption, the use of energy-saving operation modes and efficient software implementations also contributes to the power consumption. When protected data transfer or storage is required, the applied cryptography causes a part of this power drain. Therefore, such secured energy-constraint environments present potential future real-world targets for the winner of the LWC competition. The obvious question is, if exchanging ciphers and implementations of cryptographic algorithms has enough impact on the whole system that a change results in a reproducible difference in

power consumption. This is especially interesting in these scenarios, since a lower power consumption could yield in a longer lifetime of the battery and therefore the complete system.

6.2 Energy Consumption of NIST LWC Ciphers in an Industrial Use Case

In the following section, we will introduce a test setup to evaluate the contribution of the LWC candidates to the overall power consumption of an embedded system. In order to conduct a meaningful comparison, we select hardware components applied in industrial products and emulate the firmware and other system parts of a suitable use case for lightweight ciphers. We will first describe the general environment of the chosen use case, before we give some details regarding the applied protocol and its cryptography building blocks. Then, our test setup and measurement routines are shown, along with the utilized hardware. Lastly, we will explain the design and implementation of the exact use case and conclude the section with the presentation of the results.

6.2.1 Smart Meters in Water Supply

Typically, water or electric meters are smart IoT devices that are connected to some kind of larger network. In this domain, this is the case to allow trusted parties to access consumption data or even manage the systems from a remote location. While the latter requires advanced hardware and software to be able to provide a reachable administration interface, the former read-only functionality can be realized with a much simpler embedded platform.

Nowadays, modern water meters incorporate a radio interface that can send out data as defined in the firmware of the system. In water metering for private households, this is used to periodically transmit consumption data wirelessly. The transmitted data can be captured by authorized supply companies that possess a certain receiver, which can for example be placed in a maintenance car passing a nearby road. Sometimes even garbage collection vehicles are equipped with read-out equipment and can therefore periodically collect consumption data when being on duty [219]. Similar communication models are also used for other smart devices in private homes, such as smoke detectors or air conditioning sensors.

However, water meters or sensors are also part of industrial systems. For example, sensors can monitor flow control in pipes to detect leakages or measure the quality of transported water or other liquids. For such use cases, these devices can be incorporated in critical infrastructures and have to be installed in hardly accessible locations. Similar to regular water meters, electricity to operate the system can then often not be provided, meaning a battery-powered setup is required. Moreover, the communication cannot be established over a wired connection, so a wireless data transmission needs to be introduced.

6.2.2 The Wireless Meter-Bus Protocol

For scenarios, in which low-power, wireless and secure data transmission is required for metering, the Wireless Meter-Bus (wM-Bus) protocol was developed as an addition to the standard (wired) Meter-Bus (M-Bus) specification first introduced in the 1990s. The wired and wireless variants of the protocol have been standardized in the European norm DIN EN 13757. The wM-Bus communication is detailed in part 4 of this specification [220]. The standard has been largely adapted by the Open Metering System Group in their open specification for metering communication [221].

The wM-Bus protocol is used for exchanging data with meters, sensors or other embedded IoT devices in the private and industrial domain. The corresponding standard parts define the physical and link layers, as well as the application protocols and the remote readout process for wired and wireless scenarios. In general, the radio modules operate on the license-free frequency bands 868 MHz, 433 MHz and 169 MHz in wM-Bus communication. The data rate varies between 300 bit/s and 16,5 kbit/s. Both the exact frequency and data rate are dependent on the mode of operation. Six groups of modes are specified in the DIN EN 13757-4.

Stationary mode (S) describes the communication between a meter and a stationary or mobile system. There exist unidirectional (S1, S1-m) and bidirectional (S2) variants of this mode. S1 is optimized for battery-powered meters, while S1-m is particularly suitable for mobile receivers. S2 is a more generic mode, allowing for sending and receiving data. Frequent transmit mode (T) also offers a uni- and bidirectional communication setting (T1 and T2). T1 is designed for sending single meter values periodically in order to support drive-by readout. In mode T2, the meter transmits a short message frequently but waits for a response. If a response is received, a bidirectional channel is established and more data can be exchanged. In frequent receive mode (R2), a meter waits for a wake-up message from a communication party. If such a message is received, a bidirectional communication is setup. This mode enables readout devices to extract metering data from multiple stations in parallel. Compact mode (C) offers similar features as frequent transmit mode, however, it supports a more energy-efficient data exchange, i.e. more data can be transmitted with consuming the equal amount of energy. Both corresponding sub-modes (C1 and C2) are well-suited for drive-by readout. There exist two more less used modes, which operate on lower frequencies (433 MHz and 169 MHz) in order to allow for longer range communications. Frequent receive and transmit mode (F) defines a sub-mode (F2-m) in which the meter again waits for a wake-up message from a transceiver, similar to mode R2. The narrowband mode (N) is also used for meter reading and is specifically optimized for narrowband communication on 169 MHz.

Regardless of the chosen mode, the wM-Bus protocol operates in a master-slave principle. Typically, a readout or management device acts as the master, while the meters or sensors represent a group of slaves. The master hardware can also function as a gateway that accumulates and processes data from multiple slaves, before it transfers it to further systems. Every transmitted frame is based on a structure consisting of the following six different layers.

Physical Layer (PHY) In this layer, the physical specifications of the bus are defined. This includes electrical regulations, the appointed cables for wiring, data rates, transmission intervals and bandwidths, as well as the frequency ranges for wireless modes. Moreover, minimum time delays between transmissions and access windows for bidirectional communication are covered by the physical layer. The Open Metering Group specification also defines a length interval for the preamble, which initiates the wM-Bus frame transmission [221].

Data Link Layer (DLL) The data link layer is preceded by a synchronization pattern and the preamble and represents the first layer that contains variable data, depending on the purpose and rest of the frame. The data link layer consists at least of one 10-byte block. This first block starts with a length field that defines the length of the following frame. Another byte is taken by the control field (C-field), which describes the type of the frame. Examples of frame types are `SDR-NR`, which stands for Send/No Reply and is used for periodically sent consumption data from the meter, or `SDR-UD` (Send User Data), which is a standard send command transmitted from the master to a slave. The C-field is superseded by a 2-byte manufacturer ID and 4-byte serial number. The last two bytes of the DLL block indicate the version of the device and its type (e.g. water meter). The optional second block can consist of several kinds of non-application data. Any additional block contains a control information field (CI-field) that specifies the type of the following data. Except the last block, every subsequent block has a size of 16 bytes. Every data block (including the first 10-byte block) incorporates a 2-byte Cyclic Redundancy Check (CRC) trailer for error detection.

Extended Link Layer (ELL) The extended link layer is an additional frame part used in wireless communication. DIN EN 13757-4 specifies five different variants of this layer, while the open metering specification only lists two different types, a short and a long version of the ELL. Both versions start with a CI-field that indicates the kind of ELL. In the short variant, only a communication control field (CC-field) and an access counter are included. The long ELL additionally contains a receiver address consisting of a manufacturer ID, a device ID, a version and a device type. The CC-field specifies constraints regarding the communication, e.g. if messages will be repeated or if the data channel shall be bidirectional. The access counter is a single number that is increased with every transmitted frame and therefore provides information on how many messages have already been sent from the device.

Authentication and Fragmentation Layer (AFL) This layer is defined in part 7 of DIN EN 13757. It starts with a CI-field indicating the beginning of an AFL. This is followed by a length specifier for remaining bytes in this layer, as well as a fragmentation control field (AFL.FCL). The FCL-field contains information on how many fragments are part of the AFL and if these contain a frame counter or a MAC for message authentication. The subsequent message control field (AFL.MCL) indicates the authentication type,

Security Mode	Properties
0	no encryption
2	Data Encryption Standard (DES) with Initialization Vector (IV)=0
3	DES with IV≠0
5	AES-CBC-128 with IV≠0
7	AES-CBC-128 with IV=0
8	AES-CTR-128 with CMAC
9	AES-GCM-128
10	AES-CCM-128

Table 6.1: Overview over defined security modes according to DIN EN 13757-7 (condensed) [4]

e.g. AES-CMAC-128 with 16 bytes of tag length. Furthermore, the AFL consists of the calculated MAC over associated data (if applicable) and a message counter [4].

Transport Layer (TPL) The first byte of the transport layer is a CI-field that declares the header type of the following data. The metering standards define a short and a long header, it is also possible that a transport layer block is sent without any header. Header-less messages are only used for unencrypted M-Bus messages, a short TPL header is only used in wireless communication, the long header is applied in both protocols. The main difference between the short and long header lays in the additional manufacturer information. The long header contains a manufacturer ID and the device type, in addition to the status byte, an access number and the configuration field, which are present in both headers. The status byte stores information on the health of the device and is used for reporting any arising error states. The configuration field specifies the security mode for the sent message. DIN EN 13757-7 describes various security modes, some of them also support AEAD. Table 6.1 gives an overview over the possible security mode configurations in the TPL.

The following data is dependent on the selected security mode. For security mode 9 (AES-GCM) length fields for the encrypted and unencrypted payload are superseded by a message counter, the payload from the application layer and the authentication tag.

Application Layer (APL) The application layer contains the (encrypted) payload of the wM-Bus message. In a meter readout use case, this payload is structured in triplets of a Data Information Field (DIF), a Value Information Field (VIF) and a value. Each triplet is called a Data Record (DR). The DIF specifies the type of data (e.g. temperature), the VIF contains further data information (e.g. the unit, °C) and the third field holds the corresponding value (e.g. 58). The APL can have multiple DRs that are interrupted by CRC bytes after every 16 bytes of application data. When the payload is encrypted, the data will be padded to match the block size of AES.

6.2.3 Use Case

We wanted to build a close-to-industry use case for testing the LWC candidates in a real-world environment. In general, we focus on the scenario of a battery-powered water meter, which periodically sends consumption data over a wireless interface. We studied product specifications and communication descriptions to investigate how the wM-Bus messages from such devices are typically crafted and how (often) they are transferred. We model our use case after a popular water meter device sold by a well-known German meter company.

In particular, we define the following communication and evaluation properties for our analysis:

- A wM-Bus message is periodically sent every 8 seconds.
- The transferred data is static and its content is modelled according to the communication description of a well-established water meter.
- The payload of the application layer is encrypted with AES-GCM (security mode 9) to provide a high security level suitable for critical infrastructures.
- The default AEAD algorithm is exchanged with implementations of the NIST LWC candidates to evaluate their impact on the energy consumption of the whole system.
- The MCU under test and its peripherals are put in standby mode when no data is transmitted to achieve a low power consumption.
- The measurement device senses the current drain of one operation cycle (wake up, transmit, standby) at a time.
- The cycle measurement is repeated 50 times in order to be able to calculate a meaningful average consumption.

To create a static and realistic wM-Bus message, we build the data up layer by layer according to DIN EN 13757 and copy the DRs from the specification of an actual water meter. The detailed structure of the plaintext message is depicted in table 6.2. In our example evaluation, we target authenticated encryption of the payload without any additional MAC calculation for associated data. CRC bytes are not included since the integrity checksums are directly calculated in hardware. Every byte of the APL is encrypted prior to the transmission and an authentication tag is appended at the end of the message to support authenticated encryption. The DRs contain information regarding the following values: telegram count (number of messages sent by the meter), total consumed water volume, four due dates for inspections and replacement, flow rate, remaining battery lifetime, temperature and two logging data sets. Depending on the type of data, the value size varies between one and four bytes.

Data content	Length	Layer
Data length	1 byte	DLL
Message type	1 byte	DLL
Manufacturer ID	2 bytes	DLL
Serial number	4 bytes	DLL
Version	1 byte	DLL
Device type	1 byte	DLL
Layer type	1 byte	ELL
Communication type	1 byte	ELL
Access counter	1 byte	ELL
Layer specification	1 byte	AFL
Length of AFL	1 byte	AFL
Fragmentation info	2 bytes	AFL
Message configuration (incl. authentication type)	1 byte	AFL
Message counter	4 bytes	AFL
Layer header type	1 byte	TPL
Access counter	1 byte	TPL
Status	1 byte	TPL
Security configuration (mode 9 for AES-GCM)	2 bytes	TPL
Length of encrypted data	1 byte	TPL
Length of unencrypted data	1 byte	TPL
Message counter	4 bytes	TPL
Payload, containing 11 DIF/VIF/Value triplets	64 bytes	APL

Table 6.2: Structure of used wM-Bus message in the energy evaluation

6.2.4 Hardware Setup

As a basic MCU platform, we use an STM32L053R8 board featuring an ARM Cortex M0+ core, 64 KB of flash memory and 8 KB of RAM. The 32-bit chip is running at a frequency of 32 MHz. It supports different low-power modes and is used for energy-sensitive applications, e.g. in Long Range (LoRa) sensor nodes.[1] We removed/desoldered all unnecessary hardware parts (e.g. resistors and capacitors) in order to minimize the power drain of the system. To be able to wirelessly send out M-Bus messages, we added a Semtech SX1262[2] core to the setup, which is connected via a Serial Peripheral Interface (SPI) to the main MCU. The SX1262 is a sub-1-GHz transceiver that was developed for long-range data exchange with an efficient energy footprint. The co-processor is featuring checksum (CRC) calculation in hardware and is capable of modulating the transmitted signal as required for wM-Bus. It can be configured to send on the 434 MHz frequency band, the definition of the protocol and the creation of suitable messages has to be handled by the developer in software. This allows the user to apply the module in different use cases, e.g. for wM-Bus and LoRa communication.

To accurately measure the energy consumption of the setup, we used a Nordic Power Profiler Kit II[3] to power the test platform. This device was built to accurately measure the power drain of a target system and can as well be used as a power source in parallel. It allows measuring currents as low as 200 nA and has a maximum sampling rate of 100 ks/s. It also provides an API which lets the developer control the measurement process via a scripting language, automatically.

6.2.5 Test Framework

The test setup is built similarly to the speed test case from the main benchmarking framework. This is because the requirements for both test cases are strongly overlapping. Again, any available 3rd round implementation shall be tested and the programming of the flash, the test execution and the result collection shall happen automatically. In the energy measurement test case, this is even more critical since a test for a single implementation runs through 50 test cycles, each lasting 8 seconds, according to the definition in subsection 6.2.3. This means, testing one binary takes $50 * 8s = 400s = 6min40s$ without the compilation/flashing process taken into account. Considering 100+ implementations, the whole energy benchmark takes more than 11 hours. Therefore, building a reliable and highly-automated test procedure is of high importance.

For the energy test case, a template for the STM32L053 had to be developed. As in the main benchmarking framework, this acts as a runtime environment for the test case itself. It implements the initialization of the MCU and the radio module, the interface to the cryptographic operations and routines to send the data and put the devices into standby mode when necessary. The implementation of the low-power modes is essential in this use case in order to bring the test case as close as possible to industry practices. The

[1] https://www.dragino.com/products/lora-lorawan-end-node/item/128-lsn50.html
[2] https://www.semtech.com/products/wireless-rf/lora-core/sx1262
[3] https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2

firmware part of the framework is accompanied by dedicated testing scripts that start and control the individual tests and collect the results upon their termination. These scripts implement the instrumentation of the power profiling device, the compilation of the binaries and the programming of the flash memory, which is realized through a serial interface. The current drain is measured constantly through the 8 second cycle, including the data preparation, encryption, transmission and standby phase. The energy consumption is then calculated using the measured current over time. Figure 6.1 pictures the design of and the data flow within the testing procedure. The complete process – from the compilation until the result collection – happens in an automated manner.

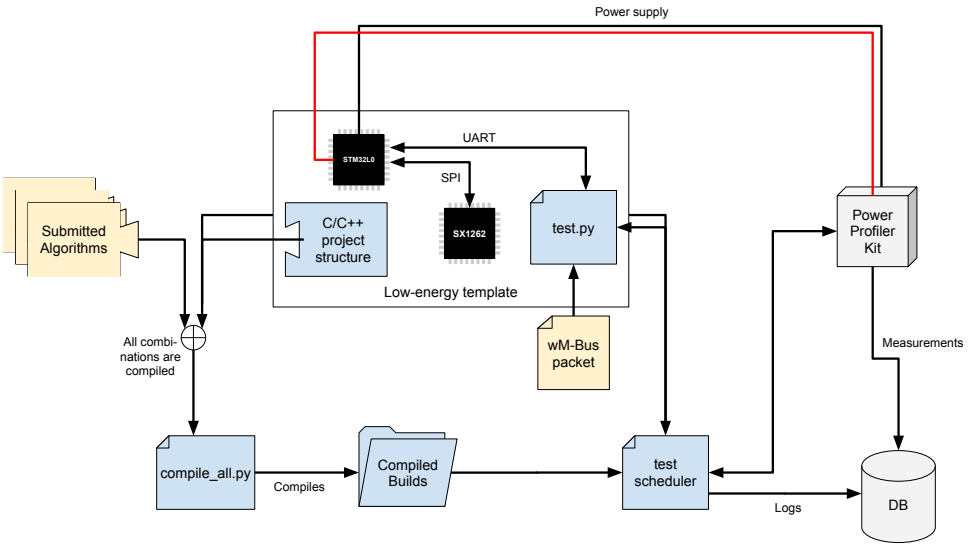


Figure 6.1: Design and data flow of the energy measurement setup

6.2.6 Results

We collect energy consumption data for every implementation available for the 3rd round candidates. The STM32L0 is flashed with every binary one after the other. Once the flashing process is finished, the testing routine is started. This includes the wake-up of the MCU and the radio module, the preparation of the wM-Bus message, its encryption/authentication with the implementation under test and the sending of the (partly) encrypted data. After the transmission has finished, the devices enter the standby mode until the cycle timer is triggered and the next sending procedure is started. A wM-Bus message is sent every 8 seconds, which mimics the behavior of an actual water meter. Per implementation, we execute the described cycle 50 times. The current drain per cycle is measured and from there, the consumed energy in Joule is calculated by using the current, the device voltage and the elapsed time.

To be able to make a statement regarding the energy consumption overhead caused by the cryptography, we also measure the current drain for a firmware that contains no

cipher implementation at all. Similar like in our binary size benchmarks, we introduce a *NoCrypt* "algorithm" that only copies the plaintext in the ciphertext array when the encryption interface is called. The energy consumption of the NoCrypt firmware serves as a minimal baseline and can be used to judge the additional power drained by encryption with different ciphers. Besides the measurement for NoCrypt, we include an energy benchmark for AES-GCM, for which we integrate the same implementation as in our speed and memory test cases. The bar plot in figure 6.3 shows the results for the best ranking implementation for each finalist. With this depiction, we follow the result presentation from the other test cases and only provide the best result in this test per candidate.

We can observe that the average energy consumption does not vary a lot in between most candidates. Besides the firmware compiled with Elephant draining more than three times as much power as the one with the best implementation, the finalists' results are close together. Xoodyak provides the most energy-efficient implementation, but is tightly followed by TinyJAMBU, Sparkle, GIFT-COFB and Ascon. AES-GCM ranks sixth, while Isap reaches the second to last spot. The minimal energy overhead per cycle compared to NoCrypt is 14 μJ. Overall, the differences in energy are minute, when comparing them in the big picture. The Xoodyak firmware consumes $\frac{445\mu J * 7.5 * 60 * 24 * 365}{10^6} = 1754.19J$ per year. If we assume a water meter with a battery containing 30.000 J, this battery would (theoretically) last for $\frac{30.000J}{1754.19} = 17.10$ years, assuming consistent discharging without considering any side effects. The same calculation for AES-GCM yields a yearly consumption of $1793.61J$ and a lifetime of 16.73 years. This means – in this simplified scenario – the device could run ca. 4.5 months more with the LWC algorithm before the battery is entirely discharged.

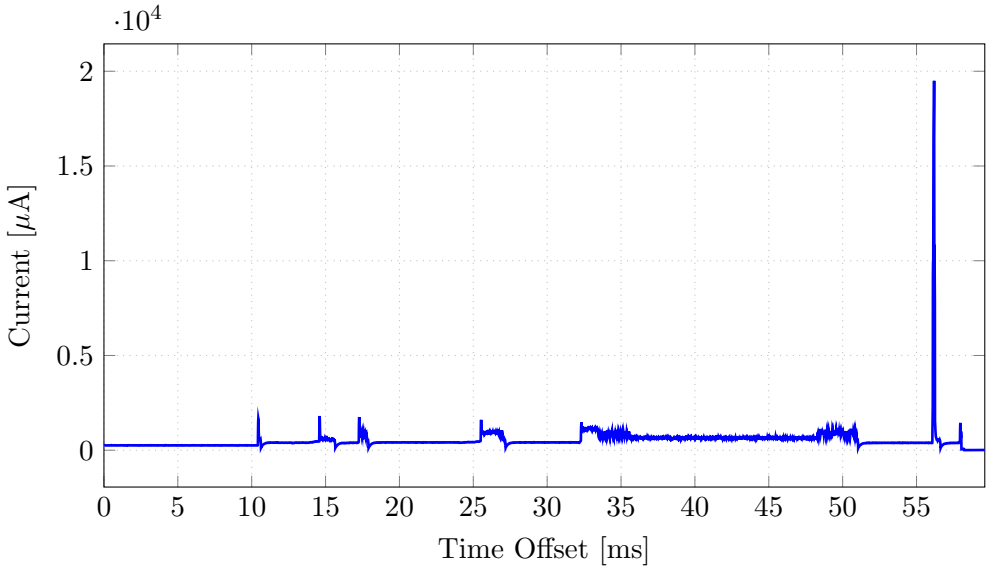


Figure 6.2: Sampled current drain from startup of the device until going back to standby. The included AEAD cipher is AES-GCM.

This comparably little impact of the cipher can be explained when taking a detailed look at the power drain of the embedded system in the smart metering use case. While the chosen cipher implementation definitely has an impact on the power consumption, this impact is shadowed by the high currents drained by the transmission of the message. In figure 6.2, we can recognize the device wake-up and configuration phase, together with the data preparation and encryption. These operations explain the little peaks ($<< 2mA$) between 10 and 52 ms after starting up. The single big peak ($> 15mA$) shows the current consumption during the message transmission. Even if this high amount of current is only needed for a short amount of time, it obviously highly influences the average power drained over one cycle. After the termination of the sending procedure, the current drops to a minimum of $5\mu A$ due to the hardware entering ultra-low-power modes. While comparing the power consumption only during the encryption process of the implementations would probably lead to easier measurable performance differences, our approach highlights the impact of selecting different LWC cipher implementations for a real-world industrial use case.

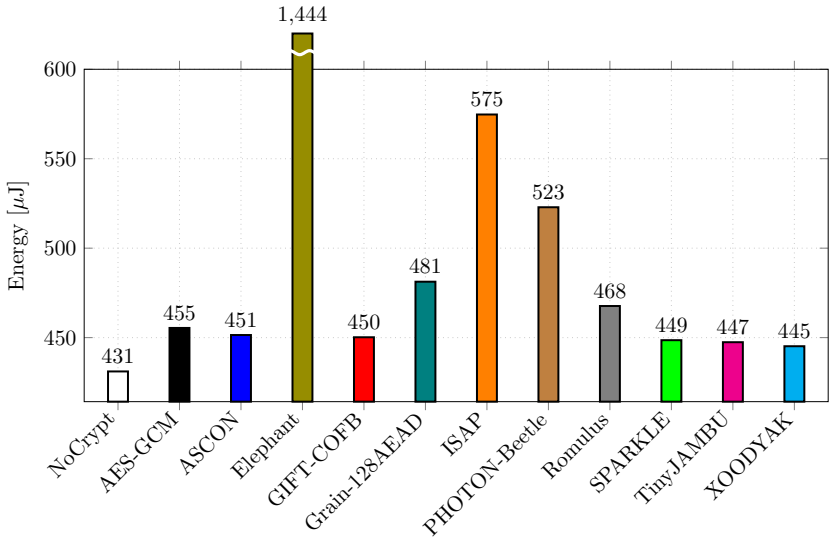


Figure 6.3: Average energy consumption of the best implementation per finalist. The measurement reflects the energy consumed for one eight-second-long cycle including wake up, encryption, transmission and standby phases.

Naturally, the speed of an implementation correlates with its energy consumption [222, 223, 224]. Whenever significantly fewer instructions need to be executed for an algorithm, its energy footprint will usually be smaller. To evaluate if there are any especially energy-efficient implementations that limit the prevalence of this correlation, we qualitatively compare the speed of the candidates and their ranking in the industrial use case benchmark. In that way, we can also estimate if results from benchmarks in synthetic test environments are comparable to those obtained in close-to-industry settings. In our speed benchmarks, we do not support a platform with the same Cortex-M0+

MCU as found on the STM32L0. The closest low-power device in our framework is the STM32F103, it however features an ARM Cortex-M3 MCU that implements a different pipeline than the Cortex-M0+. Since these architectural differences can highly impact the performance of an implementation, we resort to the speed benchmark ranking provided by NIST itself. Their testbed includes an Arduino MKR Zero MCU that makes use of an ARM Cortex M0+ chip with similar characteristics as the one featured on the STM32L0 evaluation board. We can extract the following cipher ranking for the encryption/authentication of 64 bytes (without AD) from the NIST results: 1)TinyJAMBU 2)Sparkle 3)Xoodyak 4)GIFT-COFB 5)Ascon 6)AES-GCM 7)Romulus 8)Grain-128AEAD 9)PHOTON-Beetle 10)Isap 11)Elephant. In the NIST speed comparison for the appropriate input size, the same five candidates rank in the first half, although in a slightly different order as in our energy benchmark. AES-GCM occupies the same spot in both result sets, as well as Romulus, Grain-128AEAD, PHOTON-Beetle, ISAP and Elephant. While this is of course only a qualitative comparison, it shows that no cipher is especially energy-efficient while being comparatively slow or vice versa. This generally confirms a strong correlation between speed and energy consumption of an algorithm.

7 Conclusion and Outlook

In this thesis, we contributed to the analysis of lightweight symmetric ciphers within the NIST LWC project. We designed and implemented an open-source and transparent evaluation platform for software implementations of LWC ciphers (CB1). Moreover, we discussed and modelled our requirements and constraints to support a fair comparison of the algorithms (CB2).

We evaluated the performance of NIST LWC ciphers under various aspects. The first benchmarking study analyzed the speed and memory footprints of software implementations on different MCUs. We have observed that the optimization level of an implementation and the tailoring for certain use cases can significantly influence the performance. However, still the same group of ciphers almost always ended up in top ranking spots across platforms and test cases. Moreover, we can conclude that AES-GCM is regularly outperformed by some NIST LWC candidates, while it also regularly beats some of the finalists (CB3).

Furthermore, our side-channel analyses have shown that efficient protection against implementation attacks poses a tough challenge. Standard leakage detection methods are helpful in finding distance-based leakage, however, this is insufficient in case an actually leakage-free implementation is required for an embedded system. For this use case, one has to conduct further leakage analysis on the target device to recognize and be able to eliminate other leakage sources. The protection of ARX ciphers continues to be a costly challenge. We have found that even with optimized protection techniques, the penalty for masking ARX structures – in particular the addition – is still very high. In comparison to masking e.g. sponge-based ciphers, this performance drop remains significantly higher, which represents a disadvantage of ARX ciphers (CB4). Moreover, our search for efficiently masked adders revealed that such Boolean optimization problems can be solved using techniques from the field of neural networks and genetic algorithms. However, we have found that modified conventional searching methods still yield better results, at least for our problem setup.

Lastly, this work provided benchmarking figures for LWC cipher implementations in an industrial use case. In our wM-Bus experiment, we noticed that, while the LWC finalists partly deliver better performance than AES-GCM, the difference is tiny and not significant in this use case (CB5). Additionally, this experiment confirmed that there is a high correlation between the speed of a cipher implementation and its energy consumption. For all finalists, we observed that they (generally) reflect their measured speed also in the energy benchmark. Even if the LWC candidates were not crucially more efficient than AES-GCM in our use case, their use might make sense for other scenarios. In a use case in which the speed, memory or energy footprint of the cryptographic

algorithm has a high(er) influence on the performance of the whole system, employing a suitable LWC cipher instead of AES-GCM might result in an advantageous difference.

Regarding the NIST LWC competition, it will be interesting to see which cipher NIST will select as the winner in the near future. Furthermore, the following standardization process and the possible adoption of the novel algorithm in other regulations will be challenging. It will be especially exciting to observe when and how the new cryptosystem will be applied in industrial projects. It will definitely take time and more performance and security testing to build trust on the new cipher. Nevertheless, there will likely occur practical use cases in which the LWC cipher is significantly more suitable than a variant of AES, which could then lead to a spread of the lightweight algorithm. Side-channel protection might only be required in a small subset of those use cases. However, in hardening scenarios, it is especially important to conduct further research. As we have shown in our studies, it is not enough to apply simulated leakage analysis only. For industrial use, it is required to take measurements on the actual target hardware. The area of conflict between high-performance implementations and actual protection against side-channel attacks still represents an interesting field of research for the future. Here, a further analysis of the application of neuroevolutional techniques could be a rewarding path to follow. All in all, the appropriate evaluation, optimization and selection of ciphers, regarding their performance and security level, will remain a relevant topic in both academia and industry – especially when it comes to building secure connected systems.

Bibliography

[1] J. Jean. TikZ for cryptographers, 2016. https://www.iacr.org/authors/tikz/ (visited on March 25, 2023).

[2] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182–197, 2002. doi:10.1109/4235.996017.

[3] Y. L. Corre, J. Großschädl, and D. Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM Cortex-M3 processors. In *Constructive Side-Channel Analysis and Secure Design*, pages 82–98. Springer, 2018.

[4] German Institute for Standardization. DIN EN 13757-7:2018 - communication systems for meters - part 7: transport and security services, 2018.

[5] A. Kerckhoff. La cryptographie militaire. *J. des Sci. Militaires*, 9:161–191, 1883.

[6] E. Barker and J. Kelsey. NIST special publication 800-90A: recommendation for random number generation using deterministic random bit generators. 2012.

[7] D. R. L. Brown. Conjectured security of the ANSI-NIST elliptic curve RNG. Cryptology ePrint Archive, 2006. https://eprint.iacr.org/2006/117 (visited on March 25, 2023).

[8] B. Schoenmakers and A. Sidorenko. Cryptanalysis of the dual elliptic curve pseudorandom generator. Cryptology ePrint Archive, 2006. https://eprint.iacr.org/2006/190 (visited on March 25, 2023).

[9] B. Schneier. Did NSA put a secret backdoor in new encryption standard, 2007. https://www.wired.com/2007/11/securitymatters-1115/ (visited on March 25, 2023).

[10] N. Perlroth. Government announces steps to restore confidence on encryption standards, 2013. https://archive.nytimes.com/bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards/ (visited on March 25, 2023).

[11] J. Huergo. NIST removes cryptography algorithm from random number generator recommendations, 2014. https://www.nist.gov/news-events/news/2014/04/

nist-removes-cryptography-algorithm-random-number-generator-recommendations (visited on March 25, 2023).

[12] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, J. F. Dray Jr, et al. Advanced Encryption Standard (AES). 2001.

[13] M. Robshaw. The eSTREAM project. In *New Stream Cipher Designs*, pages 1–6. Springer, 2008.

[14] M. J. Dworkin et al. SHA-3 standard: Permutation-based hash and extendable-output functions. 2015.

[15] L. Chen, D. Moody, and Y.-K. Liu. Post-quantum cryptography, 2018. https://csrc.nist.gov/Projects/post-quantum-cryptography (visited on March 25, 2023).

[16] D. J. Bernstein. CAESAR: competition for authenticated encryption: Security, applicability, and robustness, 2014. https://competitions.cr.yp.to/caesar.html (visited on March 25, 2023).

[17] L. Bassham, D. Chang, D. Hong, J. Kang, J. Kelsey, K. McKay, M. Sönmez Turan, and N. Waller. Lightweight cryptography project, 2018. https://csrc.nist.gov/Projects/Lightweight-Cryptography (visited on March 25, 2023).

[18] P. Newman. The Internet of Things report, 2020. https://www.businessinsider.com/internet-of-things-report (visited on March 25, 2023).

[19] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al. Understanding the Mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, 2017.

[20] D. Khovratovich and I. Nikolić. Rotational cryptanalysis of ARX. In *International Workshop on Fast Software Encryption*, pages 333–346. Springer, 2010.

[21] NIST. Lightweight cryptography mailing list, 2018. lwc-forum@list.nist.gov.

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

[23] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 531–545. Springer, 2000.

[24] NIST. Submission requirements and evaluation criteria for the lightweight cryptography standardization process, 2018. https://csrc.nist.

gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf (visited on March 25, 2023).

[25] M. Liskov, R. L. Rivest, and D. Wagner. Tweakable block ciphers. In *Annual International Cryptology Conference*, pages 31–46. Springer, 2002.

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

[27] H. Feistel. Cryptography and computer privacy. *Scientific american*, 228(5):15–23, 1973.

[28] M. J. Dworkin. NIST special publication 800-38A: Recommendation for block cipher modes of operation: Methods and techniques, 2001.

[29] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transactions on Information and System Security*, 6(3):365–403, 2003.

[30] A. Inoue, T. Iwata, K. Minematsu, and B. Poettering. Cryptanalysis of OCB2: attacks on authenticity and confidentiality. *Journal of Cryptology*, 33(4):1871–1913, 2020.

[31] A. Chakraborti, T. Iwata, K. Minematsu, and M. Nandi. Blockcipher-based authenticated encryption: how small can we go? *Journal of Cryptology*, 33(3):703–741, 2020.

[32] P. Rogaway and T. Shrimpton. A provable-security treatment of the key-wrap problem. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 373–390. Springer, 2006.

[33] S. Gueron, A. Jha, and M. Nandi. COMET: Counter mode encryption with authentication tag. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[34] A. Chakraborti, N. Datta, M. Nandi, and K. Yasuda. Beetle family of lightweight and secure authenticated encryption ciphers. In *Transactions on Cryptographic Hardware and Embedded Systems*, page 218–241. IACR, 2018.

[35] B. Koo, D. Roh, H. Kim, Y. Jung, D.-G. Lee, and D. Kwon. CHAM: A family of lightweight block ciphers for resource-constrained devices. In *International Conference on Information Security and Cryptology*, pages 3–25. Springer, 2017.

[36] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[37] T. Beyne, Y. Long Chen, C. Dobraunig, and B. Mennink. Elephant v1. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[38] R. Granger, P. Jovanovic, B. Mennink, and S. Neves. Improved masking for tweakable blockciphers with applications to authenticated encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 263–293. Springer, 2016.

[39] M. N. Wegman and J. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981. URL: `https://www.sciencedirect.com/science/article/pii/0022000081900337`, `doi:10.1016/0022-0000(81)90033-7`.

[40] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede. SPONGENT: A lightweight hash function. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 312–325. Springer, 2011.

[41] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 313–314. Springer, 2013.

[42] A. Chakraborti, N. Datta, A. Jha, C. Mancillas-López, M. Nandi, and Y. Sasaki. ESTATE: A lightweight and low energy authenticated encryption mode. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[43] J. Black and P. Rogaway. CBC MACs for arbitrary-length messages: The three-key constructions. In *Annual International Cryptology Conference*, pages 197–215. Springer, 2000.

[44] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo. GIFT: a small present. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 321–345. Springer, 2017.

[45] E. M. do Nascimento and J. A. M. Xexéo. FlexAEAD-a lightweight cipher with integrated authentication. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[46] E. M. do Nascimento and J. A. M. Xexéo. A flexible authenticated lightweight cipher using Even-Mansour construction. In *IEEE International Conference on Communications*, pages 1–6. IEEE, 2017.

[47] E. Andreeva, V. Lallemand, A. Purnal, R. Reyhanitabar, A. Roy, and D. Vizár. ForkAE v1. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[48] E. Andreeva, R. Reyhanitabar, K. Varici, and D. Vizár. Forking a blockcipher for authenticated encryption of very short messages. Cryptology ePrint Archive, 2018. `https://eprint.iacr.org/2018/916` (visited on March 25, 2023).

[49] S. Banik, A. Chakraborti, A. Inoue, T. Iwata, K. Minematsu, M. Nandi, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo. GIFT-COFB v1.0. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[50] A. Adomnicai, Z. Najm, and T. Peyrin. Fixslicing: a new GIFT representation: fast constant-time implementations of GIFT and GIFT-COFB on ARM Cortex-M. In *Transactions on Cryptographic Hardware and Embedded Systems*, pages 402–427. IACR, 2020.

[51] A. Chakraborti, N. Datta, A. Jha, and M. Nandi. HyENA. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[52] H. Sui, W. Wu, L. Zhang, and D. Zhang. LAEM: Lightweight authentication encryption mode. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[53] A. Adomnicai, T. P. Berger, C. Clavier, J. Francq, P. Huynh, V. Lallemand, K. Le Gouguec, M. Minier, L. Reynaud, and G. Thomas. Lilliput-AE: a new lightweight tweakable block cipher for authenticated encryption with associated data. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[54] T. P. Berger, J. Francq, M. Minier, and G. Thomas. Extended generalized feistel networks using matrix representation to propose a new lightweight block cipher: Lilliput. *IEEE Transactions on Computers*, 65(7):2074–2089, 2015.

[55] W. Stallings. The offset codebook (OCB) block cipher mode of operation for authenticated encryption. *Cryptologia*, 42(2):135–145, 2018.

[56] T. P. Berger, M. Minier, and G. Thomas. Extended generalized feistel networks using matrix representation. In *International Conference on Selected Areas in Cryptography*, pages 289–305. Springer, 2013.

[57] C. E. Mehner. Limdolen. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[58] J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 384–397. Springer, 2002.

[59] A. Chakraborti, N. Datta, A. Jha, C. M. López, M. Nandi, and Y. Sasaki. LOTUS-AEAD and LOCUS-AEAD. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[60] K. Minematsu. AES-OTR v3. *Submission to the CAESAR competition*, 2016.

[61] T. Krovetz and P. Rogaway. OCB (v1. 1). *Submission to the CAESAR Competition*, 2016.

[62] B. Chakraborty and M. Nandi. mixFeed. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[63] D. Goudarzi, J. Jean, S. Kölbl, T. Peyrin, M. Rivain, Y. Sasaki, and S. M. Sim. Pyjamask v1.0. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[64] R. Avanzi, S. Banik, A. Bogdanov, O. Dunkelman, S. Huang, and F. Regazzoni. Qameleon v. 1.0. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[65] P. Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 16–31. Springer, 2004.

[66] R. Avanzi. The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric Even-Mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. In *Transactions on Symmetric Cryptology*, pages 4–44. IACR, 2017.

[67] T. Iwata, M. Khairallah, K. Minematsu, and T. Peyrin. Remus v1.0. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[68] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *Annual International Cryptology Conference*, pages 123–153. Springer, 2016.

[69] T. Iwata, M. Khairallah, K. Minematsu, and T. Peyrin. Romulus v1.0. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[70] F. Berti, C. Guo, O. Pereira, T. Peters, and F.-X. Standaert. TEDT, a leakage-resist AEAD mode for high physical security applications. In *Transactions on Cryptographic Hardware and Embedded Systems*, pages 256–320. IACR, 2020.

[71] Y. Naito, Y. Sakai, and K. Sakiyama. SAEAES. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[72] Y. Naito, M. Matsui, T. Sugawara, and D. Suzuki. SAEB: A lightweight blockcipher-based AEAD mode of operation. Cryptology ePrint Archive, 2019. `https://eprint.iacr.org/2019/700.pdf` (visited on March 25, 2023).

[73] A. Canteaut, S. Duval, G. Leurent, M. Naya-Plasencia, L. Perrin, T. Pornin, and A. Schrottenloher. Saturnin: a suite of lightweight symmetric algorithms for post-quantum security. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[74] S. Gueron and Y. Lindell. Simple: a simple AEAD scheme. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[75] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 450–466. Springer, 2007.

[76] Z. Bao, J. Guo, T. Iwata, and L. Song. SIV-Rijndael256 authenticated encryption and hash family. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[77] Z. Bao, J. Guo, T. Iwata, and L. Song. SIV-TEM-PHOTON authenticated encryption and hash family. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[78] S. Even and Y. Mansour. A construction of a cipher from a single pseudorandom permutation. *Journal of Cryptology*, 10(3):151–161, 1997.

[79] B. Cogliati, R. Lampe, and Y. Seurin. Tweaking Even-Mansour ciphers. In *Annual Cryptology Conference*, pages 189–208. Springer, 2015.

[80] J. Guo, T. Peyrin, and A. Poschmann. The PHOTON family of lightweight hash functions. In *Annual cryptology conference*, pages 222–239. Springer, 2011.

[81] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim. SKINNY-AEAD and SKINNY-HASH. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[82] J. Jean, I. Nikolić, and T. Peyrin. Tweaks and keys for block ciphers: the tweakey framework. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 274–288. Springer, 2014.

[83] S. Banik, A. Bogdanov, T. Peyrin, Y. Sasaki, S. M. Sim, E. Tischhauser, and Y. Todo. SUNDAE-GIFT. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2020.

[84] S. Banik, A. Bogdanov, A. Luykx, and E. Tischhauser. Sundae: Small universal deterministic authenticated encryption for the internet of things. In *Transactions on Symmetric Cryptology*, pages 1–35. IACR, 2018.

[85] T. Iwata, M. Khairallah, K. Minematsu, T. Peyrin, Y. Sasaki, S. M. Sim, and L. Sun. Thank goodness it's friday (TGIF). *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[86] H. Wu and T. Huang. TinyJAMBU: A family of lightweight authenticated encryption algorithms. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[87] H. Wu and T. Huang. JAMBU lightweight authenticated encryption mode and AES-JAMBU. *CAESAR competition proposal*, 2014.

[88] N. Datta, A. Ghoshal, D. Mukhopadhyay, S. Patranabis, S. Picek, and R. Sadhukhan. TRIFLE. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[89] M. J. Dworkin. NIST special publication 800-38D: Recommendation for block cipher modes of operation: Galois/Counter mode (GCM) and GMAC, 2007.

[90] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007. Citeseer, 2007.

[91] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *International Workshop on Selected Areas in Cryptography*, pages 320–337. Springer, 2011.

[92] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Permutation-based encryption, authentication and authenticated encryption. *Directions in Authenticated Ciphers*, pages 159–170, 2012.

[93] M. Aagaard, R. AlTawy, G. Gong, K. Mandal, and R. Rohit. ACE: An authenticated encryption and hash algorithm. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[94] G. Yang, B. Zhu, V. Suder, M. D. Aagaard, and G. Gong. The Simeck family of lightweight block ciphers. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 307–329. Springer, 2015.

[95] R. AlTawy, R. Rohit, M. He, K. Mandal, G. Yang, and G. Gong. sLiSCP: Simeck-based permutations for lightweight sponge cryptographic primitives. In *International Conference on Selected Areas in Cryptography*, pages 129–150. Springer, 2017.

[96] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. Ascon v1.2. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[97] M. R. Z'aba, N. Jamil, M. S. Rohmad, H. A. Rani, and S. Shamsuddin. The CiliPadi family of lightweight authenticated encryption. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[98] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw. The LED block cipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 326–341. Springer, 2011.

[99] H. Wu and T. Huang. CLX. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[100] S. Riou. DryGASCON. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[101] D. J. Bernstein, S. Kölbl, S. Lucks, P. M. C. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo, and B. Viguier. Gimli: a cross-platform permutation. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[102] D. Otte. GAGE and InGAGE v1. 03. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[103] C. Dobraunig, M. Eichlseder, S. Mangard, F. Mendel, B. Mennink, R. Primas, and T. Unterluggauer. ISAP v2.0. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[104] T. Ding, B. Yang, Z. Bao, Z. Xiang, F. Ji, and X. Zhao. KNOT. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[105] W. Zhang, Z. Bao, D. Lin, V. Rijmen, B. Yang, and I. Verbauwhede. RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms. *Science China Information Sciences*, 58(12):1–15, 2015.

[106] B. Chakraborty and M. Nandi. ORANGE. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[107] A. Bhattacharjee, E. List, C. M. López, and M. Nandi. The Oribatida family of lightweight authenticated encryption schemes. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[108] Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, and K. Yasuda. PHOTON-Beetle: Authenticated encryption and hash family. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[109] D. Penazzi and M. Montes. Shamash (and shamashash). *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[110] M.-J. O. Saarinen. SNEIKEN and SNEIKHA. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[111] M.-J. O. Saarinen. Beyond modes: Building a secure record protocol from a cryptographic sponge permutation. In *Cryptographers' Track at the RSA Conference*, pages 270–285. Springer, 2014.

[112] C. Beierle, A. Biryukov, L. C. dos Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, Q. Wang, and A. Biryukov. Schwaemm and Esch: lightweight authenticated encryption and hashing using the Sparkle permutation family. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[113] D. Dinu, L. Perrin, A. Udovenko, V. Velichkov, J. Großschädl, and A. Biryukov. Design strategies for ARX with provable bounds: Sparx and LAX. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 484–513. Springer, 2016.

[114] R. AlTawy, G. Gong, M. He, K. Mandal, and R. Rohit. Spix: An authenticated cipher submission to the NIST LWC competition. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[115] R. AlTawy, G. Gong, M. He, A. Jha, K. Mandal, M. Nandi, and R. Rohit. SpoC: An authenticated cipher. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[116] D. Bellizia, F. Berti, O. Bronchain, G. Cassiers, S. Duval, C. Guo, G. Leander, G. Leurent, I. Levi, C. Momin, O. Pereira, T. Peters, F.-X. Standaert, and F. Wiemer. Spook: Sponge-based leakage-resilient authenticated encryption with a masked tweakable block cipher. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[117] C. Guo, O. Pereira, T. Peters, and F.-X. Standaert. Towards lightweight side-channel security and the leakage-resilience of the duplex sponge. Cryptology ePrint Archive, 2019. `https://eprint.iacr.org/2019/193` (visited on March 25, 2023).

[118] V. Grosso, G. Leurent, F.-X. Standaert, and K. Varıcı. LS-designs: Bitslice encryption for efficient masked software implementations. In *International Workshop on Fast Software Encryption*, pages 18–37. Springer, 2014.

[119] J. Daemen, P. M. C. Massolino, A. Mehrdad, and Y. Rotella. The Subterranean 2.0 cipher suite. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2020.

[120] L. Claesen, J. Daemen, M. Genoe, and G. Peeters. Subterranean: A 600 Mbit/sec cryptographic VLSI chip. In *Proceedings of 1993 IEEE International Conference on Computer Design*, pages 610–613. IEEE, 1993.

[121] S. Sarkar, K. Mandal, and D. Saha. Sycon v1.0. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[122] J. Daemen, S. Hoffert, M. Peeters, G. V. Assche, and R. V. Keer. Xoodyak, a lightweight cryptographic scheme. *A Submission to NIST Lightweight Cryptography Standardization Process*, 2019.

[123] J. Daemen, B. Mennink, and G. V. Assche. Full-state keyed duplex with built-in multi-user support. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 606–637. Springer, 2017.

[124] G. Bertoni, J. Daemen, M. Peeters, G. Assche, and R. Keer. Keyak v2. *Submission to the CAESAR Competition*, 2016.

[125] M. Montes and D. Penazzi. Yarará and Coral v1. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[126] K. R. Driscoll. Bleep64. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[127] K. Driscoll. Beepbeep: Embedded real-time encryption. In *International Workshop on Fast Software Encryption*, pages 164–178. Springer, 2002.

[128] B. Zhang. Fountain: A lightweight authenticated cipher (v1). *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[129] K. C. Gupta and I. G. Ray. On constructions of MDS matrices from companion matrices for lightweight cryptography. In *International Conference on Availability, Reliability, and Security*, pages 29–43. Springer, 2013.

[130] M. Hell, T. Johansson, W. Meier, J. Sönnerup, and H. Yoshida. Grain-128AEAD - a lightweight AEAD stream cipher. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[131] M. Ågren, M. Hell, T. Johansson, and W. Meier. Grain-128a: A new version of Grain-128 with optional authentication. *International Journal of Wireless and Mobile Computing*, 5(1):48–59, 2011.

[132] D. Ye, D. Shi, Y. Ma, and P. Wang. HERN & HERON: Lightweight AEAD and hash constructions based on thin sponge (v1). *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[133] B. Zhang. Quartet: A lightweight authenticated cipher. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[134] S. Banik, T. Isobe, W. Meier, Y. Todo, and B. Zhang. TRIAD v1. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[135] C. D. Cannière and B. Preneel. Trivium. In *New Stream Cipher Designs*, pages 244–266. Springer, 2008.

[136] M. Aagaard, R. AlTawy, G. Gong, K. Mandal, R. Rohit, and N. Zidaric. WAGE: An authenticated cipher. *A Submission to NIST Lightweight Cryptography Standardization Process*, 2019.

[137] Y. Nawaz and G. Gong. The WG stream cipher. *ECRYPT Stream Cipher Project Report 2005*, 33, 2005.

[138] D. Liu, S. Nepal, J. Pieprzyk, and W. Susilo. CLAE: A lightweight AEAD scheme of resisting side channel attacks. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.

[139] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[140] S. Renner, E. Pozzobon, and J. Mottok. A hardware in the loop benchmark suite to evaluate NIST LWC ciphers on microcontrollers. In *International Conference on Information and Communications Security*, pages 495–509. Springer, 2020.

[141] R. Ankele and R. Ankele. Software benchmarking of the 2nd round CAESAR candidates. Cryptology ePrint Archive, 2016. `https://eprint.iacr.org/2016/740` (visited on March 25, 2023).

[142] D. J. Bernstein and T. Lange. eBACS: ECRYPT benchmarking of cryptographic systems. `http://bench.cr.yp.to` (visited on March 25, 2023).

[143] D. Dinu, A. Biryukov, J. Großschädl, D. Khovratovich, Y. L. Corre, and L. Perrin. FELICS - Fair Evaluation of Lightweight Cryptographic Systems. *1st Lightweight Cryptography Workshop at NIST*, 2015.

[144] D. Dinu, Y. Le Corre, D. Khovratovich, L. Perrin, J. Großschädl, and A. Biryukov. Triathlon of lightweight block ciphers for the internet of things. *Journal of Cryptographic Engineering*, 07 2015. `doi:10.1007/s13389-018-0193-x`.

[145] K. Le Gouguec and P. Huynh. FELICS-AE: A framework to benchmark lightweight autheticated block ciphers. In *3rd Lightweight Cryptography Workshop at NIST*, 2019.

[146] R. Weatherley. Lightweight cryptography primitives, 2021. `https://rweather.github.io/lightweight-crypto/performance.html` (visited on March 25, 2023).

[147] Y. Nir and A. Langley. Chacha20 and Poly1305 for IETF protocols. Technical report, 2018.

[148] F. Campos, L. Jellema, M. Lemmen, L. Müller, D. Sprenkels, and B. Viguier. Assembly or optimized C for lightweight cryptography on RISC-V? In *International Conference on Cryptology and Network Security*, pages 526–545. Springer, 2020.

[149] NIST. Microcontroller Benchmarking, 2021. `https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking/` (visited on March 25, 2023).

[150] O. Hyncica, P. Kucera, P. Honzik, and P. Fiedler. Performance evaluation of symmetric cryptography in embedded systems. In *Proceedings of the 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems*, volume 1, pages 277–282, Sep. 2011. `doi:10.1109/IDAACS.2011.6072756`.

[151] H. Tschofenig and M. Pegourie-Gonnard. Performance of state-of-the-art cryptography on ARM-based microprocessors. *1st Lightweight Cryptography Workshop at NIST*, 2015.

[152] J.-P. Kaps, W. Diehl, M. Tempelmeier, F. Farahmand, E. Homsirikamol, and K. Gaj. A comprehensive framework for fair and efficient benchmarking of hardware implementations of lightweight cryptography. Cryptology ePrint Archive, 2019. `https://eprint.iacr.org/2019/1273` (visited on March 25, 2023).

[153] M. Khairallah. Lightweight Cryptography ASIC Benchmarking, 2021. `https://github.com/mustafa-khairallah/lwc-aead-rtl` (visited on March 25, 2023).

[154] M. Aagaard and N. Zidaric. ASIC benchmarking of round 2 candidates in the NIST lightweight cryptography standardization process: (preliminary results). Cryptology ePrint Archive, 2021. `https://eprint.iacr.org/2021/049` (visited on March 25, 2023).

[155] C. Rupp. Requirements Engineering-der Einsatz einer natürlichsprachlichen Methode bei der Ermittlung und Qualitätsprüfung von Anforderungen, 2000.

[156] Atmel Corporation. 8-bit AVR microcontroller with 32K bytes in-system programmable flash. `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf` (visited on March 25, 2023).

[157] STMicroelectronics. STM32F103x8 datasheet. `https://www.st.com/resource/en/datasheet/stm32f103c8.pdf` (visited on March 25, 2023).

[158] STMicroelectronics. STM32F746xx datasheet. `https://www.st.com/resource/en/datasheet/stm32f746ng.pdf` (visited on March 25, 2023).

[159] Espressif Systems. ESP32WROOM32 datasheet. `https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf` (visited on March 25, 2023).

[160] Seeed Studio. Sipeed Maixduino specifications v1.0. `https://www.mouser.de/pdfDocs/SipeedMaixduinoSpecifications_ENV10.pdf` (visited on March 25, 2023).

[161] S. Renner, E. Pozzobon, and J. Mottok. The final round: Benchmarking NIST LWC ciphers on microcontrollers. In *International Workshop on Attacks and Defenses for Internet-of-Things at ESORICS*, pages 1–20. Springer, 2022.

[162] T. Ding, W. Zhang, C. Zhou, and F. Ji. An automatic search tool for iterative trails and its application to estimation of differentials and linear hulls. Cryptology ePrint Archive, Paper 2020/1152, 2020. `https://eprint.iacr.org/2020/1152` (visited on March 25, 2023.

[163] S. Wang, S. Hou, M. Liu, and D. Lin. Differential-linear cryptanalysis of the lightweight cryptographic algorithm knot. In *International Conference on Information Security and Cryptology*, pages 171–190. Springer, 2021.

[164] S. Chen, Z. Xiang, X. Zeng, and S. Zhang. Conditional cube attacks on full members of knot-aead family. In *International Conference on Information and Communications Security*, pages 89–108. Springer International Publishing, 2022.

[165] M. S. Turan, K. McKay, D. Chang, C. Calik, L. Bassham, J. Kang, J. Kelsey, et al. Status report on the second round of the NIST lightweight cryptography standardization process. *National Institute of Standards and Technology Internal Report*, 8369(10.6028), 2021.

[166] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[167] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[168] S. Mangard. A simple power-analysis (SPA) attack on implementations of the AES key expansion. In *International Conference on Information Security and Cryptology*, pages 343–358. Springer, 2002.

[169] H. G. Molter, M. Stöttinger, A. Shoufan, and F. Strenzke. A simple power analysis attack on a McEliece cryptoprocessor. *Journal of Cryptographic Engineering*, 1(1):29–36, 2011.

[170] E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 16–29. Springer, 2004.

[171] F. Durvaux and F.-X. Standaert. From improved leakage detection to the detection of points of interests in leakage traces. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 240–262. Springer, 2016.

[172] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[173] B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel. Mutual information analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 426–442. Springer, 2008.

[174] C. E. Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[175] B. Gierlichs, L. Batina, B. Preneel, and I. Verbauwhede. Revisiting higher-order DPA attacks. In *Cryptographers' Track at the RSA Conference*, pages 221–234. Springer, 2010.

[176] S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 13–28. Springer, 2002.

[177] C. Herbst, E. Oswald, and S. Mangard. An AES smart card implementation resistant to power analysis attacks. In *International Conference on Applied Cryptography and Network Security*, pages 239–252. Springer, 2006.

[178] K. Tiri and I. Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 246–251. IEEE, 2004.

[179] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Annual International Cryptology Conference*, pages 398–412. Springer, 1999.

[180] L. Goubin. A sound method for switching between Boolean and arithmetic masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 3–15. Springer, 2001.

[181] J. Coron and L. Goubin. On Boolean and arithmetic masking against differential power analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 231–237. Springer, 2000.

[182] S. Nikova, C. Rechberger, and V. Rijmen. Threshold implementations against side-channel attacks and glitches. In *International Conference on Information and Communications Security*, pages 529–545. Springer, 2006.

[183] S. Nikova, V. Rijmen, and M. Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, 2011.

[184] O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede. Consolidating masking schemes. In *Annual Cryptology Conference*, pages 764–783. Springer, 2015.

[185] H. Groß, S. Mangard, and T. Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. *Cryptology ePrint Archive*, 2016.

[186] B. L. Welch. The generalization of 'student's'problem when several different population varlances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[187] B. J. Gilbert Goodwill, J. Jaffe, P. Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[188] F.-X. Standaert. How (not) to use Welch's t-test in side-channel security evaluations. In *International Conference on Smart Card Research and Advanced Applications*, pages 65–79. Springer, 2018.

[189] A. Moradi, B. Richter, T. Schneider, and F.-X. Standaert. Leakage detection with the x2-test. In *Transactions on Cryptographic Hardware and Embedded Systems*, pages 209–237. IACR, 2018.

[190] S. Renner, E. Pozzobon, and J. Mottok. Evolving a Boolean masked adder using neuroevolution. In *International Workshop on Attacks and Defenses for Internet-of-Things at ESORICS*, pages 21–40. Springer, 2022.

[191] E. Pozzobon, S. Renner, J. Mottok, and V. Matoušek. An optimized bitsliced masked adder for ARM Thumb-2 controllers. In *International Conference on Applied Electronics*, pages 1–4. IEEE, 2022.

[192] F. De Santis, A. Schauer, and G. Sigl. Chacha20-poly1305 authenticated encryption for high-speed embedded iot applications. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 692–697. IEEE, 2017.

[193] A. Biryukov, D. Dinu, Y. L. Corre, and A. Udovenko. Optimal first-order Boolean Masking for embedded IoT devices. In *Smart Card Research and Advanced Application Conference*, pages 22–41. Springer, 2017.

[194] B. Jungk, R. Petri, and M. Stöttinger. Efficient side-channel protections of ARX ciphers. In *Transactions on Cryptographic Hardware and Embedded Systems*, pages 627–653. IACR, 2018.

[195] D. Dinu, J. Großschädl, and Y. L. Corre. Efficient masking of ARX-based block ciphers using carry-save addition on Boolean shares. In *Information Security Conference*, pages 39–57. Springer, 2017.

[196] P. Schwabe and K. Stoffelen. All the AES you need on Cortex-M3 and M4. In *Selected Areas in Cryptology*, pages 180–194. Springer, 2017.

[197] A. Adomnicai and T. Peyrin. Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. In *Transactions on Cryptographic Hardware and Embedded Systems*, page 402–425. IACR, 2020.

[198] J. Coron, J. Großschädl, M. Tibouchi, and P. K. Vadnala. Conversion from arithmetic to Boolean masking with logarithmic complexity. In *International Workshop on Fast Software Encryption*, pages 130–149. Springer, 2015.

[199] T. Schneider, A. Moradi, and T. Güneysu. Arithmetic addition over Boolean masking. In *Applied Cryptography and Network Security*, pages 559–578. Springer, 2015.

[200] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999. doi:10.1109/5.784219.

[201] D. Floreano, P. Dürr, and C. Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary intelligence*, 1(1):47–62, 2008.

[202] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone. A neuroevolution approach to general Atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014. doi:10.1109/TCIAIG.2013.2294713.

[203] K. Stanley, B. Bryant, and R. Miikkulainen. Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–668, 2005. doi:10.1109/TEVC.2005.856210.

[204] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, 2002. doi:10.1162/106365602320169811.

[205] K. O. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2004.

[206] L. Cardamone, D. Loiacono, and P. L. Lanzi. Evolving competitive car controllers for racing games with neuroevolution. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, page 1179–1186. Association for Computing Machinery, 2009.

[207] J. Nadkarni and R. Ferreira Neves. Combining neuroevolution and principal component analysis to trade in the financial markets. *Expert Systems with Applications*, 103:184–195, 2018. doi:10.1016/j.eswa.2018.03.012.

[208] A. McIntyre, M. Kallada, C. G. Miguel, and C. F. da Silva. neat-python. https://github.com/CodeReclaimers/neat-python (visited on March 25, 2023).

[209] H. Groß, K. Stoffelen, L. D. Meyer, M. Krenn, and S. Mangard. First-order masking with only two random bits. In *ACM Workshop on Theory of Implementation Security*, pages 10–23. ACM, 2019.

[210] L. Zadeh. Optimality and non-scalar-valued performance criteria. *IEEE Transactions on Automatic Control*, 8(1):59–60, 1963. doi:10.1109/TAC.1963.1105511.

[211] E. Asadi, M. G. da Silva, C. H. Antunes, and L. Dias. Multi-objective optimization for building retrofit strategies: A model and an application. *Energy and Buildings*, 44:81–87, 2012. doi:10.1016/j.enbuild.2011.10.016.

[212] A. Konak, D. W. Coit, and A. E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, 2006. Special Issue - Genetic Algorithms and Reliability. doi:10.1016/j.ress.2005.11.018.

[213] I. Giagkiozis and P. J. Fleming. Methods for multi-objective optimization: An analysis. *Information Sciences*, 293:338–350, 2015.

[214] S. Risi, C. E. Hughes, and K. O. Stanley. Evolving plastic neural networks with novelty search. *Adapt. Behav.*, 18(6):470–491, 2010. doi:10.1177/1059712310379923.

[215] H. Gross. Sharing is caring—on the protection of arithmetic logic units against passive physical attacks. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 68–84. Springer, 2015.

[216] E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems*, pages 1–17. Springer, 2009.

[217] C. Beierle, A. Biryukov, L. C. dos Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, and Q. Wang. Alzette: A 64-bit ARX-box - (feat. CRAX and TRAX). In *Annual International Cryptology Conference*, pages 419–448. Springer, 2020.

[218] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018. `doi:10.17487/RFC8439`.

[219] DIEHL Metering. Passive drive-by meter reading, 2020. `https://www.diehl.com/metering/en/customer-cases/belfort-passive-drive-by/` (visited on March 25, 2023).

[220] German Institute for Standardization. DIN EN 13757-4:2019 - communication systems for meters - part 4: Wireless M-Bus communication, 2019.

[221] Open Metering System Group. OMS specification volume 2, 2016.

[222] J. Großschädl, S. Tillich, C. Rechberger, M. Hofmann, and M. Medwed. Energy evaluation of software implementations of block ciphers under memory constraints. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 1–6. IEEE, 2007.

[223] B. Buhrow, P. Riemer, M. Shea, B. Gilbert, and E. Daniel. Block cipher speed and energy efficiency records on the MSP430: System design trade-offs for 16-bit embedded applications. In *International Conference on Cryptology and Information Security in Latin America*, pages 104–123. Springer, 2014.

[224] S. Maitra, D. Richards, A. Abdelgawad, and K. Yelamarthi. Performance evaluation of IoT encryption algorithms: memory, timing, and energy. In *Sensors Applications Symposium*, pages 1–6. IEEE, 2019.

A Overview of the Characteristics of NIST LWC candidates

The following table displays key properties of the primary variants of all NIST LWC algorithms. The state, key, block/rate (abbreviated as *Bl./Ra.*), tag and security margin are presented in bits. The number of rounds is sometimes dependent on variable factors such as the input size. In this case, we mention the minimum number of rounds in the main operation. When more than one number is listed in this column, this refers to a call of the round function in different phases of the algorithm (e.g. during initialization and finalization).

Name	Variant	Type	Primitive	Mode	Rounds	State	Key	Bl./Ra.	Tag	Security
ACE	ACE-AE-128	Sponge	sLiSCP	Duplex	8	320	128	64	128	128
ASCON	**ASCON-128**	**Sponge**	**ASCON**	**Monkey Duplex**	**12, 6**	**320**	**128**	**64**	**128**	**128**
Bleep64	Bleep64	Stream	BeepBeep			127	128		64	128
CiliPadi	CiliPadi-Mild	Sponge	LED	Monkey Duplex	18, 16	256	128	64	64	128
CLAE	CLAE	LWE				16	128			128
CLX	CLX-128	Sponge	CLX	Duplex	1152	160	128	160	64	112
COMET	COMET-128_AES-128/128	Block	AES	CTR/Beetle	10	128	128	128	128	128
DryGASCON	DryGASCON128	Sponge	ASCON	Dry Sponge	11, 7	320	128	128	128	128
Elephant	**Dumbo**	**Block**	**Spongent**	**CTR**	**80**	**160**	**128**	**160**	**64**	**112**
ESTATE	ESTATE-TweAES-128	Block	Tweakable AES	OFB	8	128	128	128	128	128
FlexAEAD	FlexAEAD128b064	Block	FlexAE		8	64	128	64	64	168
ForkAE	PAEF-ForkSkinny-128-288	Block	SKINNY	PAEF	25, 31, 31	128	128	128	128	112
Fountain	Fountain-AE	Stream			384	256	128	1	128	112
GAGE/InGAGE	InGAGE	Sponge	ASCON	Monkey Duplex	32, 16	232	128	8	128	116
GIFT-COFB	**GIFT-COFB**	**Block**	**GIFT-128**	**COFB**	**40**	**192**	**128**	**128**	**128**	**128**
Gimli	Gimli	Sponge	Gimli-CIPHER	Duplex	24	384	256	128	128	128
Grain-128AEAD	**Grain-128AEADv2**	**Stream**			**384**	**256**	**128**	**1**	**64**	**128**
HERN/HERON	HERN	Stream			512, 128	256	128	1	128	128
HyENA	HyENA	Block	GIFT-128	CFB/PFB	40	128	128	128	128	128
ISAP	**ISAP-A-128A**	**Sponge**	**ASCON**	**ISAP**	**12, 1, 6, 12**	**320**	**128**	**64**	**128**	**128**
KNOT	KNOT-AEAD	Sponge	KNOT	Monkey Duplex	52, 28, 32	256	128	64	64	128

Name	Variant	Type	Primitive	Mode	Rounds	State	Key	Bl./Ra.	Tag	Security
LAEM	LAEM-SIMON-128/128	Block	SIMON	ECB	68	128	128	128	128	126
Lilliput-AE	Lilliput-TBC-II-128	Block	Lilliput-TBC	SCT-2	32	128	128	128	128	127
LOTUS/LOCUS	TweGIFT-64 LOTUS-AEAD	Block	Tweakable GIFT	OTR	28	64	128	64	64	128
mixFeed	mixFeed	Block	AES	mixFeed	10	128	128	128	128	112
ORANGE	ORANGE	Sponge	PHOTON	ORANGE-Zest	12	384	128	256	128	128
Oribatida	Oribatida-256-64	Sponge	SIMON	Duplex	34	256	128	128	128	128
PHOTON-Beetle	**PHOTON-Beetle-AEAD-128**	**Sponge**	**PHOTON**	**Beetle**	**12**	**256**	**128**	**128**	**256**	**121**
Pyjamask	Pyjamask	Block	Pyjamask	OCB	14	128	128	128	128	128
Qameleon	qameleon12812896gpv1	Block	Tweakable QARMA	PANORAMA	30	128	256	128	128	224
Quartet	Quartet	Stream	ASCON		24	256	128	1	128	112
REMUS	REMUS-N1	Block	Tweakable SKINNY	COFB	40	128	128	128	128	128
Romulus	**Romulus-N**	**Block**	**Tweakable SKINNY**	**COFB**	**40**	**384**	**128**	**128**	**128**	**128**
SAEAES	SAEAES128-64-128	Block	AES	SAEB	10	128	128	128	128	112
Saturnin	Saturnin	Block	Saturnin	CTR	20	256	256	256	256	224
Shamash	Shamash	Sponge	Shamash	Duplex	12, 9	320	128	128	128	126
SIMPLE	SIMPLE128-GIFT	Block	GIFT-128	CTR	40	128	128	128	128	128
SIV-Rijndael256	SIV-Rijndael256-AEAD	Block	Rijndael256	SIV	14	256	128	256	256	128
SIV-TEM-PHOTON	SIV-TEM-PHOTON-AEAD	Block	Tweakable PHOTON	SIV	20	256	128	256	256	128
SKINNY	SKINNY-128-384	Block	SKINNY	OCB	56	128	128	128	128	128
SNEIK	SNEIKEN128	Sponge	SNEIK	BLNK2	6	512	128	384	128	128
SPARKLE	**Schwaemm256-128**	**Sponge**	**SPARKLE**	**Beetle**	**11, 7**	**384**	**128**	**256**	**128**	**120**
SPIX	SPIX	Sponge	sLiSCP	Monkey Duplex	18, 9	256	128	64	128	128
SpoC	SpoC-64	Sponge	sLiSCP	Beetle	6, 18	192	128	64	64	120
Spook	Spook	Sponge	Shadow-512	Sponge One-Pass	12	512	128	256	128	121
Subterranean 2.0	Subterranean-SAE	Sponge	Subterranean	Duplex	8	257	128	33	128	128
SUNDAE-GIFT	SUNDAE-GIFT	Block	GIFT-128	SUNDAE	40	128	128	128	128	128
Sycon	Sycon-AEAD-128-r64	Sponge	Sycon	Monkey Duplex	14, 7	320	128	64	128	128
TGIF	TGIF-N1	Block	Twekable TGIF	COFB	72	128	128	128	128	128
TinyJAMBU	**TinyJAMBU-128**	**Block**	**TinyJAMBU**	**JAMBU**	**1024**	**128**	**128**	**32**	**64**	**112**
Triad	Triad-AE	Stream			1024	256	128	1	64	128
TRIFLE	TRIFLE-BC	Block	TRIFLE	TRIFLE	50	128	128	128	128	128
WAGE	WAGE-AE-128	Stream			111	259	128	1	128	128
XOODYAK	**XOODYAK**	**Sponge**	**XOODOO**	**Cyclist**	**12**	**384**	**128**	**352**	**128**	**128**
Yarará	Yarará	Sponge	Yarará	Duplex	10, 6	256	128	64	128	128

B Configuration File Used by neat-python

```
[NEAT]
fitness_criterion     = max
add_fit_goal          = 0
leak_fit_goal         = 0
pop_size              = 300
reset_on_extinction   = True

[BNGenome]
# use aggregation to realize gates
aggregation_default = xor
aggregation_options = nand nor and or
aggregation_mutate_rate = 0.05

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 2.0

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# node add/remove rates
node_add_prob           = 0.5
node_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.05

# Do not allow feed back in network
feed_forward              = True
initial_connection        = neat_double

# network parameters
num_inputs              = 6
num_hidden              = 5
num_outputs             = 4

[DefaultSpeciesSet]
compatibility_threshold = 3

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 15
```

```
45  species_elitism       = 2
46
47  [DefaultReproduction]
48  elitism               = 2
49  survival_threshold    = 0.1
```

C ARM Assembly Implementation of the Shared Bitsliced 32-bit Adder

```
1  .section .text
2  .cpu cortex-m3
3  .thumb
4  .syntax unified
5
6  // Macros for clearing the pipeline.
7  // Both require r0 to have no cryptographic material.
8  .macro  CLEAR_PIPELINE_1
9      orr.n r0,r0
10 .endm
11
12 .macro  CLEAR_PIPELINE_2
13     orr.n r0,r0
14     nop.n
15 .endm
16
17 .align 4
18 .global bitsliced_full_adder
19 .type bitsliced_full_adder, %function
20 .thumb_func
21 // [r0, #(4*n)]          for 0 <= n < 32   is the (n+1)-th slice of share 0 of A
22 // [r0, #(128 + 4*n)]    for 0 <= n < 32   is the (n+1)-th slice of share 1 of A
23 // [r0, #(256 + 4*n)]    for 0 <= n < 32   is the (n+1)-th slice of share 0 of B
24 // [r0, #(384 + 4*n)]    for 0 <= n < 32   is the (n+1)-th slice of share 1 of B
25 // r1 is the random refresh mask
26 bitsliced_full_adder:
27     push.w  {r4-r9,lr}
28 // Zero all temporary registers
29 // (this can be optimized out if it is ensured that the initial values
30 //  of these registers do not lead to pipeline or register-reuse leakage)
31     mov.n   r2,#0
32     mov.n   r3,#32
33     mov.n   r4,#0
34     mov.n   r5,#0
35     mov.n   r6,#0
36     mov.n   r7,#0
37     mov.n   r9,r3
38     sub.n   sp,#4
39     str.n   r0,[sp,#0]  // Clear write MDR
40 .align 2
41     mov.n   r5,r1
42 _add_next_slice:
```

```
43      mov.n   r8,r0
44      ldr.w   r3,[r0,#128]
45      ldr.w   r4,[r0,#384]
46      ldr.n   r6,[r0,#0]
47      ldr.w   r7,[r0,#256]
48      ldr.n   r0,[sp,#0]  // Clear read MDR
49
50  // Full adder implementations starts here
51  // This can be replaced with alternate
52  // full adder implementations
53
54  // r6, r3 are shares of A
55  // r7, r4 are shares of B
56  // r1, r5 are shares of C
57  // r2 is used as scratch space
58  // r8 will output the new share
59      eor.w   r8,r6,r7
60      CLEAR_PIPELINE_1
61      and.n   r6,r7
62      CLEAR_PIPELINE_2
63      eor.n   r7,r3
64      CLEAR_PIPELINE_2
65      eor.n   r3,r4
66      CLEAR_PIPELINE_2
67      eor.n   r7,r5
68      CLEAR_PIPELINE_2
69      eor.n   r4,r1
70      CLEAR_PIPELINE_2
71      orr.n   r1,r3
72      eor.n   r2,r2
73      eor.w   r2,r4,r5
74      CLEAR_PIPELINE_2
75      and.n   r7,r3
76      CLEAR_PIPELINE_2
77      and.w   r5,r8,r2
78      CLEAR_PIPELINE_2
79      bic.n   r1,r7
80      CLEAR_PIPELINE_2
81      orr.n   r5,r6
82      CLEAR_PIPELINE_2
83      eor.w   r8,r2
84      CLEAR_PIPELINE_2
85      eor.n   r5,r4
86      CLEAR_PIPELINE_2
87
88  // r1, r5 are shares of the output carry
89  // r8 is the second share of the output sum
90
91  // End of full adder implementation
92
93      str.w   r8,[r0],#4
94      str.n   r0,[sp,#0]  // Clear write MDR
95      cmp.w   r9,#1
96      sub.w   r9,#1
```

```
 97  .align 2
 98      bne.n    _add_next_slice
 99      nop.w
100      add.n    sp,#4
101      pop.w    {r4-r9,pc}
```