Lehrstuhl für Sicherheit in der Informationstechnik
Fakultät für Elektrotechnik und Informationstechnik
Technische Universität München

A Hardware Benchmarking Platform for the Standardization of Authenticated Encryption Algorithms

Michael Theodor Tempelmeier

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:
Apl. Prof. Dr.-Ing. Walter Stechele

Prüfer der Dissertation:
1. Prof. Dr.-Ing. Georg Sigl
2. Assoc. Prof. Jens-Peter Kaps, Ph.D.
   George Mason University

Die Dissertation wurde am 11.01.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 03.05.2021 angenommen.

Quidquid agis, prudenter agas et respice finem!

```ada
with Ada.Text_IO;

procedure Thanks is
begin
   Ada.Text_IO.Put_Line("Thanks, Dad!");
end Thanks;
```

In memory of
Prof. Dr. Theodor Tempelmeier
(1952 – 2019)

Abstract

This thesis describes and solves the challenges of evaluating and benchmarking cryptographic algorithms during the standardization process of Authenticated Encryption with Associated Data (AEAD). Candidates of the CAESAR competition (Competition for Authenticated Encryption: Security, Applicability, and Robustness) serve as the data basis. AEAD algorithms combine two properties: they can independently authenticate and encrypt/decrypt data. These properties are especially important for network protocols, where parts of the data, e.g. the header information, must be protected from manipulation, but at the same time must be readable, and thus must not be encrypted; other data, however, should not be readable during transmission and thus must be encrypted.

This work focuses on hardware implementations on FPGAs. In contrast to software implementations, which are usually embedded in a larger software context, hardware implementations often do not provide a uniform interface, since they are often embedded in application-specific devices. This hinders a fair comparison because different implementations are measured against different requirements. For this reason, this thesis first motivates a uniform interface for hardware implementations. At the same time, it critically examines the current implementations of such an interface and proposes an improved version.

In the course of this work, the previous practice of evaluating cryptographic hardware implementations during the CAESAR competition is also critically examined: Due to the lack of a clear evaluation process, implementation flaws remained undetected until late in the competition. This is demonstrated by means of a specification problem and an implementation problem.

To overcome these problems, a low-cost, developer-friendly evaluation platform is developed. It consists of a Xilinx PYNQ-board. The platform uses a common hardware interface and can be used for pure functional testing as well as for advanced performance analysis. Due to its low price and ease of use, it can be used by any hardware developer. This results in two advantages: First, the quality of the implementations increases, since every developer is both financially and technically able to use this platform to test his or her implementation against a uniform set of tests; second, the results are comparable since every implementation is measured against the same assumptions.

Subsequently, this platform is used to evaluate all the finalists and a large part of the prefinalists of the CAESAR competition. Here, functionality; specification conformance; resource, power and energy consumption; and runtime are evaluated. It is shown that the winners in the field of “lightweight applications” do not necessarily have “lightweight” properties and that, according to the measured data, other eliminated candidates should have been considered as finalists.

Furthermore, this work presents an alternative for the uniform hardware interface. It turns out that for very specific hardware software co-design applications, the presented unified hardware interface is suboptimal. Nevertheless, for the majority of applications the advantages of a uniform hardware interface outweigh. This is demonstrated on the basis of two selected examples: “Physical Unclonable Functions” and “Network-on-Chip” applications. In both cases the above mentioned implementations are integrated in other scientific research areas and the uniform hardware interface allows for a fast evaluation of a large number of different implementations.

Finally, this work results in a set of rules for comparing and creating cryptographic algorithms under the aspects of good hardware optimizability and using a general hardware interface. These rules are designed to fit the current standardization of Lightweight Crypography (LWC) by the National Institute of Standards and Technology (NIST).

Zusammenfassung

Diese Arbeit beschreibt und löst die Herausforderungen bei der Evaluierung und Leistungsbewertung von kryptographischen Algorithmen der Klasse „Authenticated Encryption with Assosiated Data (AEAD)“, die typischerweise während der Standardisierung auftreten. Als Datenbasis dienen hierbei Kandidaten des CAESAR-Wettbewerbs („Competition for Authenticated Encryption: Security, Applicability and Robustness“). AEAD Algorithmen vereinen dabei zwei Eigenschaften: Sie können unabhängig voneinander Daten verschlüsseln, bzw. entschlüsseln und authentifizieren. Diese Eigenschaften sind insbesondere bei Netzwerkprotokollen wichtig, da hier Teile der Daten, z.B. Metainformationen, vor Manipulation geschützt werden müssen, aber gleichzeitig im Klartext lesbar sein müssen; andere Daten hingegen sollen während der Übertragung nicht lesbar sein.

Diese Arbeit fokussiert sich auf Hardwareimplementierung auf Basis von FPGAs. Im Gegensatz zu Softwareimplementierungen, die meistens in einem größeren Softwarekontext eingebettet sind, ist bei Hardwareimplementierungen häufig kein einheitliches Interface vorgesehen, da sie oftmals als eigenständige, anwendungsspezifischen Bausteine bestehen. Aus diesem Grund wird in dieser Arbeit zuerst ein einheitliches Interface für Hardwareimplementierungen motiviert. Gleichzeitig wird sich kritisch mit dessen bisheriger Implementierung auseinandergesetzt und eine verbesserte Version des Selbigen vorgeschlagen.

In diesem Zuge wird sich auch kritisch mit der bisherigen Praxis der Bewertung von kryptographischen Hardwareimplementierungen während des CAESAR-Wettbewerbs befasst. Durch das Fehlen von klaren Bewertungsprozessen blieben Implementierungsprobleme lange Zeit unentdeckt. Dies wird Anhand eines Spezifizierungs- und eines Implementierungsproblem demonstriert.

Als Lösung dieser Probleme wird hierzu eine kostengünstige, entwicklerfreundliche Evaluierungsplattform entwickelt. Sie basiert auf einem Xilinx PYNQ-Board. Sie nutzt eine einheitliche Hardwareschnittstelle und kann sowohl zum reinen Funktionstest als auch zur erweiterten Leistungsanalyse verwendet werden. Wegen des günstigen Preises und der einfachen Bedienbarkeit kann sie von jedem Hardware-Entwickler eingesetzt werden. Dadurch entstehen zwei Vorteile: Erstens steigt die Qualität der Implementierungen, da jeder Entwickler sowohl finanziell, als auch technisch in der Lage ist, diese Plattform zu benutzen und somit frühzeitig Fehler im Design erkennen zu können; zweitens verbessert sich die Vergleichbarkeit der Ergebnisse, da jede Implementierung anhand der gleichen Voraussetzungen gemessen wird.

Anschließend wird diese Plattform genutzt um alle Finalisten und einen Großteil der Vorfinalisten des CAESAR-Wettbewerbs zu bewerten. Hierbei wird sowohl die Funktionalität, die Spezifikationskonformität, der Ressourcen-, Leistungs- und Energieverbrauch als auch die Laufzeit evaluiert. Dabei zeigt sich, dass die Gewinner im Bereich „lightweight applications“ nicht zwangsweise „lightweight“ Eigenschaften besitzen und dass gemäß der gemessenen Daten andere, ausgeschiedene Kandidaten als Finalisten berücksichtigt hätten werden müssen.

Weiterhin präsentiert diese Arbeit eine Alternative für die einheitliche Hardwareschnittstelle. Es zeigt sich, dass für sehr spezielle Hardware-Software-Codesign-Anwendungen, die präsentierte einheitliche Hardwareschnittstelle suboptimal ist. Trotzdem überwiegen für das Gros der Anwendungen die Vorteile einer einheitlichen Hardwareschnittstelle. Dies wird nochmals anhand von zwei ausgewählten Beispielen demonstriert, wobei die oben genannten Implementierungen in anderen wissenschaftlichen Kontexten („Physical Unclonable Functions“ und „Network-on-Chip“-Anwendungen) eingebunden werden. Die verwendete Hardwareschnittstelle erlaubt dabei eine schnelle Evaluation einer Vielzahl von verschiedenen Implementierungsvarianten.

Abschließend wird basierend auf den Ergebnissen dieser Arbeit eine Handreiche für das Vergleichen und das Erstellen von kryptographischen Algorithmen unter den Gesichtspunkten einer guten Hardwareoptimierbarkeit sowie unter Verwendung einer allgemeinen Hardwareschnittstelle gegeben. Diese Handreiche ist darauf ausgelegt bei der aktuellen Standardisierung von „leightweight Authenticated Encryption with Assosiated Data“ des National Institute of Standards and Technology (NIST) Verwendung zu finden.

Contents

1 Introduction

When in 1997 the National Institute of Standards and Technology (NIST) initiated a public competition to determine which block cipher algorithm should become the Advanced Encryption Standard (AES) [62], it broke new ground, since all previous cryptographic standards were determined behind closed doors [7].

In 2000, two similar research projects were launched: NESSIE (New European Schemes for Signatures, Integrity and Encryption) by the European Commission [50] and CRYPTREC (Cryptography Research and Evaluation Committee) by the Japanese government [15]. They are comparable in many respects to the AES competition run by NIST. Therefore, there were no new groundbreaking findings during these competitions. However, they strengthened the idea of publicly available cryptographic competitions.

Thus, subsequent contests for cryptographic standards became more and more popular, with more algorithms submitted each time:

In 2004, another European project, called eSTREAM (ECRYPT Stream Cipher Project), was launched by the European Network of Excellence in Cryptology (ECRYPT) [49]. In contrast to the previous projects, eSTREAM focused on stream ciphers.

In 2007, NIST announced its second cryptographic competition: the SHA3-hashing competition. While other projects like NESSIE already included hashing to their call for submissions, NIST chose to have a dedicated competition for hashing. Again, this competition formed a new standard: the SHA-3 standard [20].

One of the latest concluded contests is CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) [12], which was announced in 2013 and ended in February 2019. Its goal was to determine a portfolio of authenticated ciphers that offer advantages over AES in Galois Counter Mode (AES-GCM) in terms of performance, security, and ease of correct implementation. Throughout the contests, candidates were evaluated based on their security, efficiency in software, efficiency in hardware, and flexibility. In contrast to the other competition, CAESAR did not report to any governmental agency.

There are more competitions that are still ongoing: the NIST Post-quantum Cryptography (PQC) project [61], started in 2016, and the NIST Lightweight Crypography (LWC) project [60], started in 2018. While the first focuses on the new field of post-quantum cryptography, the latter combines very well known cryptographic primitives to form lightweight primitives which are suitable for the Internet-of-Things (IoT) era. The latter can also be seen as a successor of CAESAR, as the specification of the encryption primitives is a subset of CAESAR's.

In this thesis, we focus on hardware implementations of CAESAR and LWC candidates. One characteristic of both projects is the volume of different submissions compared to the AES competition: In 1998, there were only 15 candidates competing for the new Advanced Encryption Standard. Furthermore, 13 of them were based on well known structures like Feistel Networks, or substitution-permutation networks, and only two of them were of dedicated character. This made a fast decision possible, so that in 1999 five finalists could be presented. One year later, Rijndael was chosen to become the new Advanced Encryption Standard [82].

CAESAR on the other hand had 57 submissions, divided into more than six different categories [1] and the LWC project accepted 56 submissions [60]. This poses a serious challenge for their evaluation. Furthermore, the CAESAR and and LWC competition included hardware benchmarking into their evaluation process because hardware implementations gain more and more importance.

Today, high-speed co-processors are used to speed up cryptographic operations on server CPUs, but even desktop CPUs are equipped with AES instruction set extensions. Dedicated Field-programmable gate array (FPGA) or even Application-specific Integrated circuit (ASIC) clusters are used for blockchain operations. More and more memory elements (e.g. hard disks, or RAM) provide an on the fly memory encryption. Lightweight, power and energy optimized implementations are used in IoT devices. Side-channel and fault attack protected implementations are mandatory even for consumer products.

Taken the afore mentioned tremendous number of submissions into account, benchmarking becomes even more challenging because there are now multiple benchmarking dimensions, which must be applied to each submission, like security, software performance on different processors, hardware performance on FPGAs, hardware performance on ASICs, performance in mixed hardware software co-designs, ability for IoT applications, etc.

This thesis addresses the following points and thus contributes to the overall evaluation process:

- We will demonstrate that one of the main challenges when benchmarking different hardware implementations is the used interface. Thus, it is crucial to adopt a common hardware API for all implementations. However, even a common API is not enough. We will show that such an API can be misused to "optimize" the synthesis results of a cipher, but actually only the implementation of the API is improved. Therefore, we will closely analyze the CAESAR-API in terms of area cost and propose an improved

API, which could be used for the NIST-LWC competition.

- Next, we will motivate the need for an universal hardware testbed by showing that more than expected (official) hardware implementations are functionally incorrect.

- For the functionally correct implementations, we will extend this testbed to allow for benchmarking. We will present benchmarking results in terms of area, throughput, power, and energy consumption.

- We will compare those API compliant, pure hardware results with mixed hardware software implementations and show the advantages of both approaches.

- Finally, we will demonstrate the benefits of a fixed API implementation with an external and internal interface: Ciphers fully equipped with the API can be easily integrated on protocol level. We will demonstrate this at the example of a PUF-based IoT protocol. Having also a common internal interface between the cipher and API implementation will allow for a seamless integration of the Cipher into custom projects, without the (communication) overhead of the API. We will demonstrate this at the example of a NoC router.

Consequently, the thesis is organized as follows:

As preliminaries, we first explain Authenticated Encryption with Associated Data, introduce the "Competition for Authenticated Encryption: Security, Applicability, and Robustness", and present the hardware platform used throughout this thesis in the next chapter.

Having dealt with the preliminaries, we carve out the challenges and problems, when evaluating different hardware implementations in Chapter 3. In Chapter 4, we present and analyze a solution to make different implementations comparable by examining the CAESAR hardware API and present an improved LWC hardware API.

The actual analysis of the ciphers starts in Chapter 5. In this chapter, we focus on the functional behavior. We show, that many implementations are not functional correct. This emphasizes the need of the presented framework. In the next chapter, this framework is extended to allow for performance benchmarking.

Chapter 7 is an excursus to show the applicability of CAESAR ciphers in other research domains. We show how these domains benefit from the common API presented in Chapter 4.

In Chapter 8, we conclude the thesis with a very personal point of view on how things should be done.

2 Preliminaries

In the following, we introduce the concepts of Authenticated Encryption with Associated Data, the CAESAR competition, and Systems on Chips, as this thesis heavily relies on these concepts.

2.1 Authenticated Encryption with Associated Data

Secure communication is based on two main principles: *Secrecy* and *Authenticity*. Secrecy, the oldest cryptographic principle, ensures that a message can only be read (decrypted) by the appropriate recipient. However, in many cases secrecy is not the (only) concern. It is at least as important to verify that a message has not been altered intentionally or unintentionally during its transmission.

Therefore, cryptographic protocols rely on authentication and encryption to provide confidentiality, as well as integrity and authenticity. Traditionally, a Message Authentication Code (MAC) and an encryption scheme were combined at protocol level:

- *MAC-and-Encrypt*, where a message is independently encrypted and authenticated. This scheme is used amongst others by SSH [55].
- *MAC-then-Encrypt*, where a message is first authenticated and then together with the resulting MAC encrypted. This scheme is used amongst others by SSL/TLS [69].
- *Encrypt-then-MAC*, where a message is first encrypted and then the encrypted message is authenticated. This scheme is the recommended mode to combine an encryption algorithm with a MAC, as it is the only one that can provide indistinguishability under adaptive chosen ciphertext attacks (IND-CCA), non-malleability under chosen plaintext attacks (NM-CPA), and the integrity of ciphertexts (INT-CTXT) [5]. It is used amongst others in IPsec [52], but there are extensions for TLS and SSHv2 to use this mode [30].

While these composition approaches have obvious advantages in terms of re-usability, they suffer from being neither efficient nor always secure and prone to implementation errors [6, 5, 53]. Additionally, a key separation is needed, as MAC and encryption are two distinct cryptographic primitives and therefore need two distinct keys [51]. Otherwise the above mentioned security goals can not be guaranteed [5]. A more modern approach is to consider Authenticated Encryption (AE) as a cryptographic building block on its own, rather than as the mere combination of two. For this reason and others, the NIST-recommended modes like Counter with CBC-MAC (CCM) [18] and AES-GCM [19], as well as the ISO standard AES in Offset Codebook 2 Mode (AES-OCB2) [71] are used.

In the following the term AE is used for algorithms that combine both principles into one single algorithm. Typical use cases for AE are (embedded) software updates. In this scenario, the update is confidential, as it involves Intellectual Property (IP) and at the same time, it must be ensured that it is not altered.

In addition to authentication and encryption, several protocols require checking the integrity and authenticity of so called Associated Data (AD), which are public data that must be authenticated by the receiver, but do not contain confidential information. The most prominent example of AD is the header of a network protocol: While the header must not be encrypted, as it must be readable by the network router, it must be protected in terms of integrity and authenticity. This leads to Authenticated Encryption with Associated Data (AEAD) as a building block. The most recent example for AEAD is TLS-1.3 which only allows AE algorithms and purged all composition-based approaches, as they are considered legacy [68].

Formally, an AEAD scheme is defined as follows [1]:

Definition 1 *Let $k, \nu, t \geq 1$, $K \in \{0,1\}^k$ denote a secret key, $N \in \{0,1\}^\nu$ a nonce, $T \in \{0,1\}^t$ an authentication tag, $P \in \{0,1\}^*$ a plaintext, $AD \in \{0,1\}^*$ associated data, and $C \in \{0,1\}^*$ a ciphertext. An AEAD is a triple $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, with a key-generation procedure $\mathcal{K}$ that returns a randomly K, an encryption algorithm $\mathcal{E}_K(N, AD, P)$, and a decryption algorithm $\mathcal{D}_K(N, AD, C, T)$, where $\mathcal{E}$ outputs a pair (C, T), and $\mathcal{D}$ outputs either the plaintext P or the void symbol $\perp$ if the tag is invalid:*

$$\mathcal{E} : \{0,1\}^k \times \{0,1\}^\nu \times \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^t \tag{2.1}$$

$$\mathcal{D} : \{0,1\}^k \times \{0,1\}^\nu \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^t \rightarrow \{0,1\}^* \cup \{\perp\} \tag{2.2}$$

The formal definition of the decryption in Equation (2.2) is often relaxed for implementations: As the tag verification takes place after the complete decryption and as it is often infeasible to store the entire decrypted plaintext, the plaintext is released regardless whether the tag is valid or not. There is only a flag set to indicate the status of the verification and appropriate actions must be taken on system or protocol level.

2.2 Competition for Authenticated Encryption: Security, Applicability, and Robustness

In the past few years, some concerns about the performance and ease of side-channel secure implementations of AES-GCM, which had been the defacto standard for AEAD for many years, were raised. As a result, the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) was started with the goal to

> *"identify a portfolio of authenticated ciphers that (1) offer advantages over AES-GCM and (2) are suitable for widespread adoption".* [12]

Although funded by NIST grant 60NANB12D261 and supported by the University of Illinois at Chicago [7], it was organized by "the international cryptologic research community", a loose committee of 18 international researchers from eleven different countries. Secretary of the committee and most prominent spokesman was D. J. Bernstein [12].

2.2.1 Functional Requirements for Submissions

The call for submissions [12] had very limited requirements: The submissions must support AEAD, that provides integrity for the plaintext, the Associated Data, public message number, and the secret message number, as well as confidentiality for the plaintext and the secret message number. Both message numbers may impose a single-usage.

The plaintext and the secret message number had to be recoverable from the ciphertext, AD, public message number, and key. The submissions had to support a minimum length of 2^{16} bytes for plaintext/ciphertext and AD. However, designers were encouraged not to limit the maximum length. For practical reasons, most ciphers were designed to support maximum lengths up to 2^{64} bytes. The support for public and secret message numbers was optional. There are no prerequisites on these numbers, except that they are only used once with the same key. As there are no late-round candidates that made use of the secret message number, in the following, the term nonce (number used only once) always refers to the public message number. There was no definition for a minimum or maximum length of the key.

Additionally, each submission was allowed to specify a family of ciphers with different parameter sets for each family member.

2.2.2 Process for Determining the Final Portfolio

In 2014, more than 50 algorithms were submitted to the competition. The competition was designed to identify a portfolio of ciphers in three consecutive rounds. For each round, the committee

> *"will issue a report that, for each selected algorithm, cites the previously published analyses that led the algorithm to be selected. [...] an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision."* [12]

An appeal of the committee decision was not possible. However, submitters were "expected to promptly and publicly respond to those analyses" they disagree with [12].

In the first round, the cryptographic strength of the submitted algorithms was evaluated. Additionally, all first round candidates were required to publish a software reference implementation. Based on their findings, 28 candidates were eliminated with the announcement of the second round in 2015.

The second round was dedicated to software implementations. Authors of the ciphers and independent designers were invited to submit optimized software implementations for different platforms ranging from embedded devices to servers. SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives) was used to benchmark the different implementations [8]. It compiles the submitted code with different compilers, for different target platforms, and with different compiler options to find the best ones. Additionally, all authors of second round candidates were required to submit hardware implementations written in either VHDL or Verilog. 13 candidates were eliminated during this round in 2016.

The third round was dedicated to hardware implementations, but also software tweaks were allowed. Again, authors and independent designers were invited to submit optimized implementations. Planned to be the last round, it was the longest round and lasted until 2018. Nine candidates were eliminated.

With seven finalists remaining, it took another year, until the final portfolio was announced in 2019. It consists of a primary and a secondary recommendation for *lightweight* applications (use case 1), and *defense in depth* applications (use case 3) and lists two recommendations without a preference for *high-performance* applications (use case 2).

2.2.3 Examined Ciphers

In this thesis, we focus on fifteen different families of ciphers and in parts multiple implementations or versions of the same cipher. Those implementations cover all CAESAR finalists and half of the second round candidates. Table 2.1 lists their parameters. In the following, they are briefly explained to allow for a better classification throughout this thesis.

Like Abed et al. in [1], we classify them into the following groups[1]: *block cipher based*, *stream cipher based*, *sponge based*, and *dedicated* algorithms. In [1], Abed et al. also provide an overview of external cryptanalysis and security proofs of the ciphers.

Nomenclature

In the literature, there are two ways to refer to the same cipher: first, the common name of the cipher, which may include a version number e.g. “SILC v2.0”, a data path width e.g. “Acorn32”, or a key width e.g. “AEGIS-128”; second, the name of the (software reference) implementation. This can be simply the same name, but without any white spaces and upper cases or a more specific name like “aes128n12t8silcv2”. The latter contains additional information like the used Block Cipher (BC) (AES-128), the length of the nonce ($n = 12$ bytes), or the length of the tag ($\tau = 8$ bytes). The release version is indicated with a “v” suffix; dots in the versioning are ignored, i.e. “v141” stands for “release 1.41”. This nomenclature is often used, when the construction is very generic and allows for a variaty of different parameters. For the SILC example, there existed also “present80n6t4silcv2”, which uses PRESENT-80 with a six byte nonce and a four byte tag. If both names are given, the software reference implementation is notated in parenthesizes, e.g. “AEGIS-128 v1.1 (aegis128)”. A “_”-suffix can be used to specify to distinguish optimized implementations from the reference implementation, e.g. “aegis128_ref”, “aegis128_arm64”, etc.

Block Cipher

The following ciphers use a BC to form an AEAD scheme:

- AEZ [34] uses multiple, different instances of AES. According to the authors “writing software for AEZ is not easy, while doing a hardware design for AEZ is far worse.” Homsirikamol and Gaj confirmed that in their paper “Anything-but EaZy in Hardware” [35]. Throughout this thesis, only one – non working – implementation is used as a bad example.

[1] Abed et al. also used the groups *compression function based* and *(non-sponge) permutation based*, but non of the late round CAESAR candidates fall into those categories.

Table 2.1: Security parameters of used implementations

Cipher Name	Claimed Security[5,6]	Key Size[6]	Nonce Size[6]	Tag Size[6]	Block Size[6]	State Size[6]	Main security function	Use Case
AEZ	128	384	96	128	128	128	three AES-like structures	defense in depth
Acorn	128	128	128	128	1	293	non linear shift register	lightweight
AEGIS-128	128	128	128	128	128	640	5 rounds of AES-128[4]	high-performance
AEGIS-128l	128	128	128	128	128	1 024	8 rounds of AES-128[4]	high-performance
AEGIS-256	256[1]	256	256	128	128	768	6 rounds of AES-128[4]	high-performance
AES-GCM	128	128	128	128	128	128	10 rounds of AES-128[2] + "H"-mult.	–
ASCON	128	128	128	128	64	320	6 rounds of permutation + sponge	lightweight
CLOC	128	128	64	64	128	128	10 rounds of AES-128	lightweight
COLM	64	128	128	128	128	128	10 rounds of AES-128	defense in depth
Deoxys-I	128	128	64	128	128	128	10 rounds[3] of Deoxys-BC	high-performance
Deoxys-II	127	128	120	128	128	128	10 rounds[3] of Deoxys-BC	defense in depth
Icepole	128	128	128	128	1 024	1 280	6 rounds of permutation + sponge	High-performance
Jambu-AES	128	128	64	64	64	192	10 rounds of AES-128	defense in depth
Norx	256	256	256	256	768	1 024	4 rounds of permutation + sponge	high-performance
Morus	128	128	128	128	256	1 280	shift, rotate, AND, XOR	high-performance
OCB	128	128	128	128	128	128	10 rounds of AES-128	high-performance
SILC	128	128	96	64	128	128	10 rounds of AES-128	lightweight
Tiaoxin	128	128	128	128	256	1 664	6 rounds of AES-128	high-performance
dummy	–	128	128	128	128	128	XOR	–

[1] 128 bit for authenticity
[2] not for Associated Data
[3] 20 rounds for plaintext/ciphertext
[4] without key scheduling
[5] security decreases for large number of blocks
[6] numbers are in bits

- Jambu [85] uses the blocksize of a BC as its state. This state is divided into two even parts. One part absorbs the AD and plaintext by xoring it. The other part acts as a key generator. By xoring this block key with the data, the data is encrypted or decrypted. Finally, the BC is executed on the complete state and new data can be absorbed and encrypted/decrypted. For security reasons, there is also a forward path, that bypasses the BC and xors the absorbing halve of the state to the other halve after executing the BC. Throughout this thesis we use JAMBU-AES, i.e. AES-128 as the BC.

- CLOC [40] is a kind of cipher block chaining mode of operation for AES-128 and TWINE-80. In contrast to the original cipher block chaining, there are some additional function blocks before passing data to the block cipher. Throughout this thesis, the 64-bit nonce and AES-128 version is used.

- OCB [54] stands for "offset codebook" and is a patent-registered mode of operation for AES. In favor of the CAESAR competition, there is also a free license which is suitable for most non military use cases[2]. In contrast to other modes of operation, OCB only requires one pass of the AES BC per input and a linear masking of the input and output. This mask can be a precomputed and stored in a table. Throughout CAESAR OCB version 3 is used.

- SILC [41] is very similar to CLOC and also a kind of cipher block chaining mode of operation for AES128, Present-80, and LED-80. In contrast to the original cipher block chaining, there are some additional function blocks before passing data to the block cipher. Throughout this thesis, the 96-bit nonce and AES128 version is used.

- AES-GCM [19] is the oldest and well-known dedicated AEAD scheme. It uses the AES-128 BC for encryption, but heavily relies on the "H" point-multiplication for authentication. Throughout this thesis AES-GCM is used as reference, as the goal of CAESAR is to find algorithms that "offer advantages over AES-GCM".

- Deoxys-I [43] is based on the Deoxys-BC. For 128-bit keys it uses Deoxys-BC-256 and for 256-bit keys it uses Deoxys-BC-384. In both both cases a 64-bit nonce and 128-bit input blocks are used. Deoxys-BC is based on the AES functions SubBytes, ShiftRows and MixBytes. The AddRoundKey function is replaced with a AddRoundTweakey, which is basically a Linear Feedback Shift Register (LFSR) based key scheduling. The two Deoxys-BCs only differ in their Tweakey generation. The AEAD construction itself is very similar to OCB. Throughout this thesis we use deoxysi128v141.

- Deoxys-II [43] is a nonce-misuse resistant mode that uses the same BC and same general structure as Deoxys-I, but with a 120-bit nonce. To achieve its nonce-misuse resistance, it uses a two-pass structure: In the first pass an authentication checksum is generated. In the second pass this checksum is used as as a tweakey to encrypt/

[2]`https://www.cs.ucdavis.edu/~rogaway/ocb/license.htm`

decrypt the message. It is the only remaining two-pass cipher in the competition. Throughout this thesis we use deoxysii128v141.

- COLM [3] superseded AES-COPA and ELmD and combines the best of both. There were two versions COLM_{127} and COLM_0. However, only COLM_0 is of practical relevance and in the following referred to as COLM. Like Deoxys-II, it is nonce-misuse resistant. It has a Encrypt-Linearmix-Encrypt structure: Data is first encrypted using AES-128. For processing the AD the encrypted data is then xored. For processing the plaintext, the output is fed to a linear mixing function ρ. The result is then again encrypted with AES-128. ρ has a second input and a second output. This additional output is fed to the second input for the processing of the next input block. Additionally, like OCB, COLM applies a linear mask to its input and output.

Stream Cipher based

Acorn [84] is the only late round stream cipher. It uses a 128-bit key and a 128-bit nonce. Its state consists of a 293-bit shift register, which is a mix of six Galois and Fibonacci LFSRs. A feedback function uses 13 out of the 293 bits to generate an overall feedback bit. Eight out of these 13 bits are used to generate an intermediate key bit, which is used to encrypt or decrpyt one bit of the stream input. The overall feedback bit is xored with the input bit and fed back to update the shift register. After the initialization the register is shifted for 1535 cycles with a constant input of one. After processing the AD, and the plaintext/ciphertext, it is shifted for 512 cycles with a key dependent input.

Sponge-based

Figure 2.1 shows the general structure of a sponge construction. It consists of a state and a permutation function f. The state is divided into a capacity c and a rate r. The round function performs a permutation of the state. Typically this round function is applied several times to form f. The (padded) input m is xored to the rate. The output of the sponge construction is z. The individual ciphers differ in the used permutation, the width of r and c, the padding function, the way how the construction is used to encrypt data (i.e. how m is generated from the input (e.g. plaintext) and how the output (e.g. ciphertext) ist generated from z) and some minor tweaks. These differences are as follows:

- ICEPOLE [58] uses a duplex construction, where m and z are not divided into two consecutive phases, but are interleaved. The key and nonce are used to initialize the state. There are three variants of Icepole: Icepole128, Icepole128a, and Icepole256a. Icepole256a uses a 256-bit key, a 96-bit nonce, a 962-bit rate and a 318-bit capacity. Icepole128a uses a 128-bit key, a 128-bit nonce, a 1026-bit rate and a 254-bit capacity.

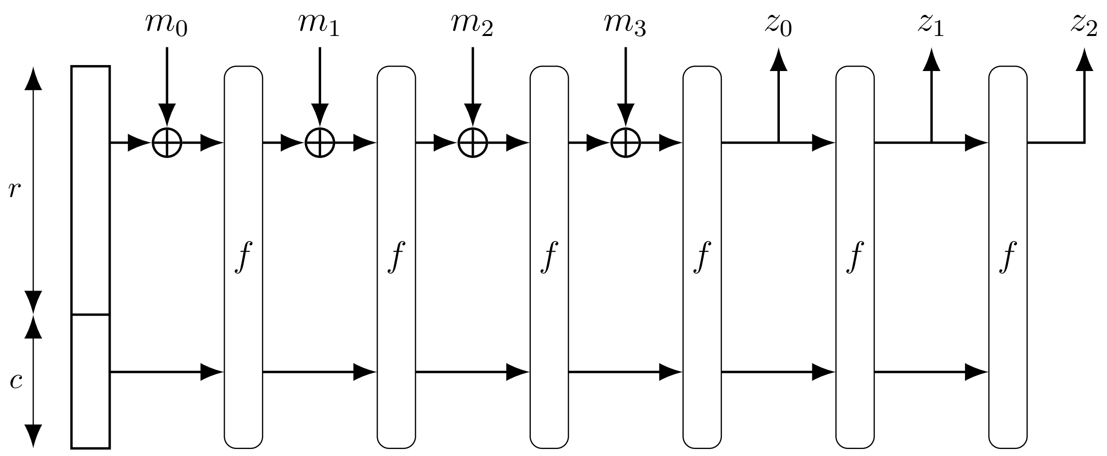


Figure 2.1: Sponge construction [42]

The odd choice of the rate is due to Icepole's special padding, which always needs two additional bits. Thus, subtracting the value two from the rate results in an even power of two. This also influence the capacity, which is built to form a complete state of 160 byte. Icepole128 uses the same parameter as Icepole128a but has additional support for a secret message number. Throughout this thesis we use icepole128av2.

- ASCON [17] again uses the duplex construction. The state is initialized with the key, the nonce, and a constant. Additionally, the key is used for domain separation. This means, that the capacity is xored with the padded key and a constant between different processing phases. There are two variants Ascon-128 (ascon-128-64) and Ascon-128a (ascon-128-128). Both use a 128-bit key, a 128-bit nonce, and a 320-bit state. Ascon-128a uses a 128-bit rate and a 192-bit capacity. Ascon-128 uses a 64-bit rate and a 256-bit capacity. Throughout this thesis we use ascon128v12 as this is the primary recommendation of the authors.

- NORX [4] is a highly parameterizable cipher. In this thesis we use norx6441v3, which refers to a wordsize of 64 bits, 4 rounds and a parallelism of degree 1. In this configuration it is a strait forward duplex construction with domain separation. It uses a 256-bit key, a 256-bit nonce, a rate of 768 bit and a capacity of 256 bits.

Dedicated structures

The following ciphers have a similar structure as a *Type-3 Feistel* scheme. They consist of a multi-block state $S = S_0||S_1||...||S_n$. A message (block) M is added to one or all sub-states, e.g. $S_0 := S_0 \oplus M$ and the individual sub-states are updated by adding their neighbor states e.g. $S_i := S_i \oplus f(S_{i-1})$.

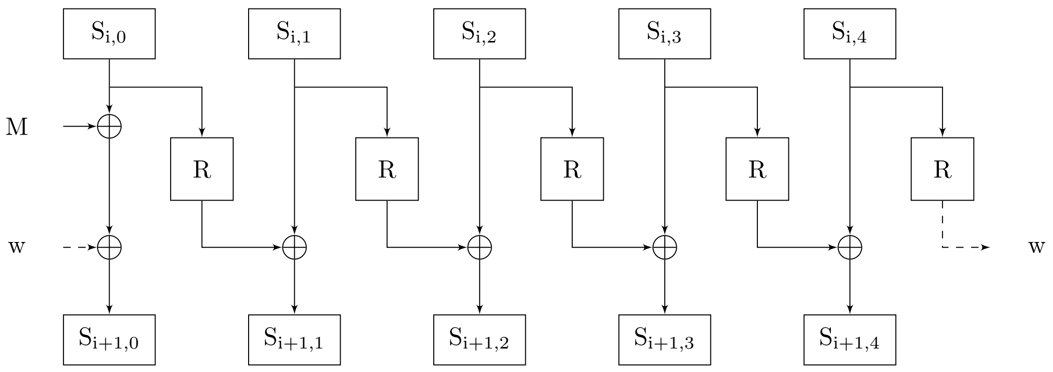


Figure 2.2: Aegis-128 state update: The state update function R consists of the AES functions SubBytes, ShiftRows and MixColums [88]. ©2019 IEEE [80]

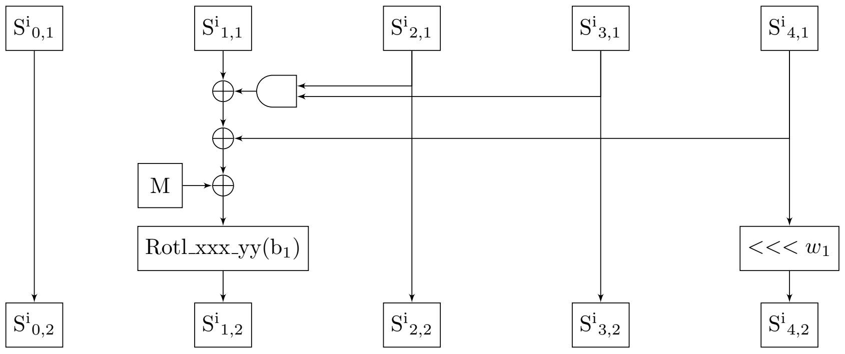


Figure 2.3: Second step of Morus's update function: The circuit is sequentially repeated for each substate [86]. ©2019 IEEE [80]

- AEGIS-128 (aegis128), AEGIS-128l (aegis128l), and AEGIS-256 (aegis256) [88] share the same structure as depicted in Figure 2.2, but differ in their parameters: Aegis-128 consists of five 16-byte sub-states, uses a 128-bit key and permutes the state ten times for initialization. Besides eight 16-byte sub-states, Aegis-128l has the same parameters as Aegis-128. Aegis-256 consists of six 16-byte sub-states, uses a 256-bit key and permutes the state 16 times for initialization.

- The Morus family [86] has two parameters: the state size and the key size. In this thesis morus1280128v2 is used. That means the state size is 1280 bits and the keysize is 128 bits. Other parameters would be a 640 bit state and a 256 bit key. This state is divided into five substates of 256 bits. Three at a time are used to update one substate by anding two and xoring them and remaining one to the updated one. Finally the results is rotated by a round constant. Figure 2.3 shows the update function of the second substate.

- Tiaoxin [63] also called Tiaoxin-346 – pronounced Tiaoxin three four six – uses a 128-

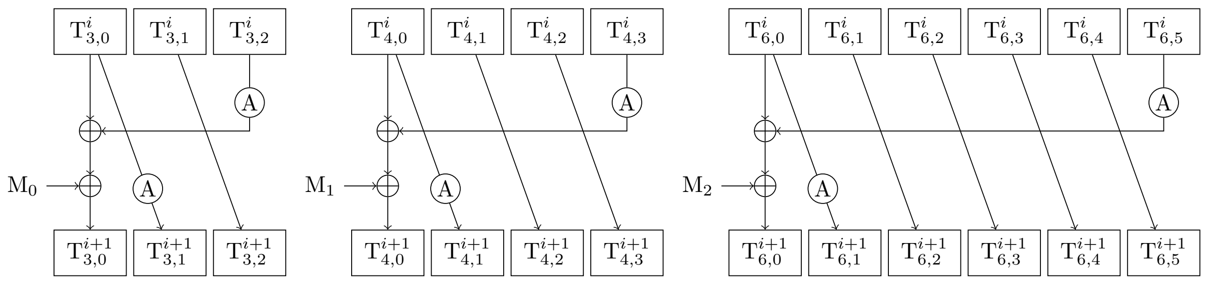


Figure 2.4: Tiaoxin-346 state update: The state update function Ⓐ consists of the AES functions SubBytes, ShiftRows and MixColums [63].

bit key, a 128-bit nonce and can encrypt six bytes per round. It has three states T_3, T_4, T_6. They are composed of 3, 4, and 6 16-byte words, respectively. Figure 2.4 shows the associated update function.

Dummy Cipher

The dummy cipher simply xors a 128-bit key, a 128-bit nonce, and a counter with a 128-bit input block. It is used to provide a lower boundary for ciphers and to allow for testing benchmarking frameworks. Other names are dummy1 [39] in the CAESAR competition[3] and dummy_lwc [79] in the LWC competition.

2.3 New Generation of Reconfigurable System-on-Chips

As mentioned, today's cyptographic algorithms must perform in a variety of different environments: in high-performance computing clusters, in embedded devices, in mixed hardware-software codesigns, and in low-powered IoT-devices. With cheap hardware available, hardware implementations gain more and more importance. Another factor for the increasing interest is the spread of re-configurable Systems on Chips (SoCs). They become more and more common, not only in ASICs, but also on FPGAs, as they combine the advantages of a fully Programmable Logic (PL) with the flexibility of a Processing System (PS) running arbitrary software. In Chapter 5 we will pointed out the advantages of using the PS of a SoC to pre- and post-process data, e.g. test vectors for the actual hardware under test, which is programmed into the PL.

[3]As part of the development package, there are also dummy2 to dummy5, but they are not used throughout this thesis.

Therefore, we first introduce the used SoC-platform. It integrates a single or dual core ARM-Cortex-A9 processor with an Artix-7 or Kintex-7 based .

PYNQ: Xilinx Zynq-7000 SoC

Throughout this thesis we use the Xilinx' PYNQ, which is a portmanteau of Python and Zynq. It combines the very successful Zynq-7000 SoC with an easy to use and program Python interface. The hardware consists mainly of a XC7Z020 FPGA, which combines an Artix-7 based FPGA fabric with an integrated dual-core ARM Cortex-A9 [93]. Additionally, 512 MB of DDR3 RAM, 16 MB of QUAD-SPI Flash, and a MicroSD-card slot are available. Furthermore, the XC7Z020 also features an analog-to-digital converter (XADC), which can be used to measure analog values like temperature or current draw. The board itself is striped to the basics, which enables a very low (academia) price. The Arduino and Pmod input/output interfaces can be used for additional measurement and control extensions. An Ethernet connector enables a fast way of communication with the board, such that a realistic amount of data can be processed without a bottleneck, like when using an UART. Since there are only very few additional interfaces such as HDMI (in and out), USB, and analog audio (in and out) on the board, it is very well suited for realistic power and performance measurements, as on one side, there is no exotic hardware that influences the measurement, yet on the other side, it has the characteristics of a complete embedded system.

The software side features a fully functional Linux, which runs a SAMBA server, a web-server with a Jupyter notebook, and an SSH-client. Thus, it enables a very broad spectrum of applications and use cases.

There are three particularly useful features of the PYNQ board:

On-the-fly FPGA programming

In order to use the Programmable Logic to its fullest potential at all times, the programmed logic inside has to be changed to the current requirements of the software running on the Processing System. This is a complex task and requires the user to either re-program the FPGA using the appropriate vendor toolchain, or know how to use partial reconfiguration for on-the-fly reconfiguration. PYNQ overcomes this problem by making it possible to load *overlays* at runtime using a simple Python command. Overlays are precompiled bitstreams of the desired FGPA configuration, along with a corresponding block design representation in TCL, that can be stored on the Linux file system of the SoC.

Python drivers

The python driver uses the information of the block design from the TCL-file to abstract the underlying complexity, for example, the bus communication, or the individual configuration steps needed for a DMA transfer, and allows the user to easily address the PL from the PS. This is especially useful for designers who want to test their design on real hardware, but are not aware of each and every detail in a SoC.

Web-based User Interface

Additionally the Python framework provides a Jupyter notebook interface that allows prototyping Python code directly from the web browser.

3 Challenges in Evaluating Cryptographic Competitions

In the following, we will discuss the challenges in evaluating cryptographic competitions. Our findings are based on the CAESAR competition, but can be transferred to the NIST-LWC competition, because –besides the official character– they are very similar in terms of requirements and submissions. We hope, that this discussion helps to improve the currently ongoing NIST-LWC-competition and helps to prevent some of the discovered flaws.

3.1 Review Process Flaws at the Example of CAESAR

The review process of the CAESAR competition was purely steered by the CAESAR committee. It consisted of 18 members; 17 of them with voting power. They made their selection decisions on the basis of published analyses. Besides the problem that "an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision" [12], there was also a risk of flaws in the review process itself:

As the cryptographic community is relatively small, a pure independent evaluation was rarely possible. Often only mutual evaluation was possible. This problem was further increased by the huge amount of heterogeneous submissions. As a consequence, there was an overlap between the decision-makers, the participants, and the evaluators. This resulted in four of seven finalists having at least one author being part of the CAESAR committee.

The CAESAR competition required the authors to submit both hardware and software reference implementations. Due to the nature of the different approaches, common verification platforms were hard to put into practice. First, there was only a common benchmarking and verification suit for software implementations. Second, even with this tool, there was no process to verify if the submitted reference implementation matches the specified requirements.

For hardware implementations there was no official verification suite, and only very late in the competition a mandatory hardware interface was put into practice. Thus, there were undetected flaws in the official hardware reference implementations of late round candidates.

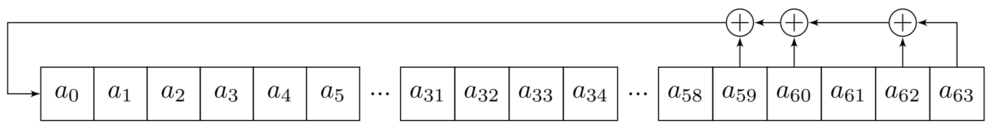


Figure 3.1: Intended LFSR

3.1.1 Algorithmic or Software Flaws

One example for a very long undiscovered mismatch between specification and the algorithm itself is ICEPOLE. It uses round constants in its κ function which are defined

> *"as the output of a simple 64-bit maximum-cycle Linear FeedbackShift Register (LFSR). The polynomial representation of LFSR is $x^{64} + x^{63} + x^{61} + x^{60} + 1$. The LFSR state is initialized with the 64-bit vector '0123456789ABCDEF' (hexadecimal format) and then each cycle generates a subsequent constant."* [58]

Figure 3.1 shows the intended hardware.

However, as we have shown in [76], this is not the case. The algorithm itself does not calculate them on the fly, but uses predefined constants, which were generated with the following C-code[1]

```
1 # include <stdint.h>
2 int main(void)
3 {
4   uint64_t lfsr = 0x0123456789ABCDEFu;
5   unsigned bit;
6   int i = 0;
7   int number_of_rounds = 12;
8
9 printf("Initial state: %016llX\n",lfsr);
10
11 for (i=0; i<number_of_rounds; i=i+1){
12
13   // taps: 64 63 61 60
14   // feedback polynomial: x^64 + x^63 + x^61 + x^60 + 1
15   bit  = ((lfsr >> 0) ^ (lfsr >> 1) ^ (lfsr >> 3) ^ (lfsr >> 4)) & 1;
16   lfsr =  (lfsr >> 1) | (bit << 63);
17
18 printf("constant[%2d] := %016llX\n",i,lfsr);
19 }
```

Unfortunately, this does not generate an LFSR but a non LFSR, due to an overflow in line 16, caused by the wrong datatype declaration in line 5. The variable `bit` is defined as `unsigned`.

[1]confirmed by the author Paweł Morawiecki <pawel.morawiecki@gmail.com> via email on 19.10.15, 22:24

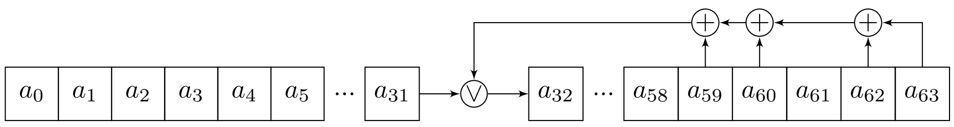


Figure 3.2: Implemented non LFSR

On most architectures this is equal to `uint32_t`. Thus, it can only contain 32 bits of data. In the right part of line 16 the value of `bit` is shifted to the left by 63. This results in an overflow. The real pity is that default compiler warnings even tell this:

```
lfsr.c:16:36: warning: left shift count >= width of type [-Wshift-count-overflow]
          lfsr =  (lfsr >> 1) | (bit << 63);
                                     ^~
```

Therefore, `bit` is not shifted by 63, but actually by $63 \mod 32 = 31$. Thus in line 16, the feedback bit is not added to the most left side of `lfsr` but is ored in the middle. Thus resulting in a non linear feedback shift register. Figure 3.2 shows the corresponding hardware.

Unfortunately, this was first discovered during the hardware benchmarking in the second round of the competition.

3.1.2 Hardware Flaws

One example for a very long undiscovered erroneous hardware reference implementation, is the CAESAR finalists Morus: The official reference hardware implementation is dated July 2016[2]. This implementation is also used to back up the claimed implementation and performance results.

However, even a brief look at the VHDL code reveals the following flaws:

```
19 entity CipherCore is

37     port (
38         --! Global
39         clk              : in  std_logic;

43         bdi              : in  std_logic_vector(G_DBLK_SIZE      -1 downto 0);

63         msg_auth_done    : out std_logic;
64         msg_auth_valid   : out std_logic
65     );
66 end entity CipherCore;
```

[2] https://www.ntu.edu.sg/home/wuhj/research/caesar/caesar.html

```
68 architecture structure of CipherCore is

73     signal tag_rev        : std_logic_vector(G_DBLK_SIZE     -1 downto 0);

125    signal msg_auth_done_s : std_logic;

144 begin

148    process(clk)
149    begin
150        if rising_edge(clk) then

204            if (msg_auth_done_s = '1') then
205                if (tag_rev(255 downto 128) = bdi(255 downto 128)) then
206                    msg_auth_valid <= '1';
207                else
208                    msg_auth_valid <= '0';
209                end if;
210                msg_auth_done <= '1';
211
212            end if;
213
214        end if;
215    end process;

441 end structure;
```

If the condition in line 204 is true, the tag is evaluated in line 205. If the tag is correct, the corresponding output is set to one (line 206), otherwise it is set to zero (line 208). In any case, the corresponding done-flag is set to one (line 210). These are the only lines where the outputs of lines 63 and 64 are set. There is no line in the entire architecture, where `msg_auth_done` is set to zero. Thus, if the condition of line 204 is false afterwards, e.g. during the processing of the next message, the output is not reset.

This is a very basic error in the output handling, nevertheless the candidate made it to the final round. Thus, hardware testing was not conducted carefully during the competition.

3.2 The Role of the Interface

When benchmarking (cryptographic) hardware implementations, the assumptions made on the interface between the primitive and the surrounding benchmarking framework is crucial. In the following we will demonstrate this at the example of ICEPOLE [58] a second round CAESAR candidate. For doing so, we first explain an optimized implementation of ICEPOLE. Next, we analyze the presented results and explain the pitfalls when using different interfaces.

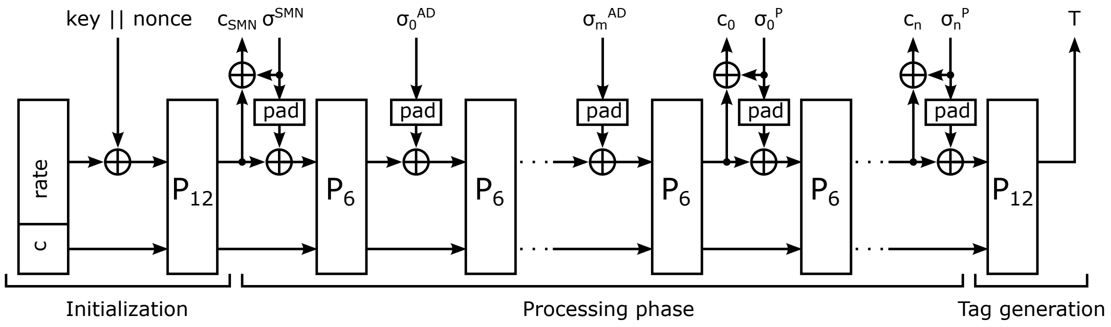


Figure 3.3: ICEPOLE-128 ©2016 IEEE [76]

Parts of the following analysis have already been pre-published in "Michael Tempelmeier, Fabrizio De Santis, Jens-Peter Kaps, and Georg Sigl. An area-optimized serial implementation of ICEPOLE authenticated encryption schemes. In 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 49–54, May 2016. doi: 10.1109/HST.2016.7495555".*

3.2.1 The Example of Icepole

ICEPOLE is a high-speed, hardware-oriented family of single-pass authenticated encryption schemes based on the duplex construction as introduced in [10]. In this example, only Icepole-128 is considered, but the results can be translated to any sponge-based block cipher with an arbitrary state size.

Icepole-128 works on a 1280-bit state S arranged into a three dimensional $4 \times 5 \times 64$-bit cube $S[x][y][z]$. According to [58], bits $S[x][y]$, which share the same axis z, are called *slices*, bits $S[y]$, which share the same axes x and z, are called *rows*, while bits $S[z]$, which share the same axes x and y, are called *words*. Therefore, the state can be seen alternatively as a collection of either 64×20-bit slices, or 256×5-bit rows or 20×64-bit words.

ICEPOLE-128 encrypts and authenticates variable-length messages in three phases: the initialization phase, the processing phase, and the tag generation phase, as illustrated in Figure 3.3. In each phase the state is transformed by a 1280-bit permutation P that is based on the successive execution of the round function R. In the initialization phase, the permutation P is denoted by P_{12} and transforms the initial state by applying a round function R 12 times. The initial state is obtained by filling the state with a "1280-bit pseudo-random constant[3]" and then adding the secret key and the nonces to the first four words of the state $S[i][0]$, $i \in \{0, 1, 2, 3\}$. All the successive invocations of P in the processing phase are denoted by P_6. The permutation P_6 transforms the current state by applying the round function R only 6 times.

[3]truncated result of applying Keccak-f[1600] to an all zero input [58]

The round function $R = \kappa \circ \psi \circ \pi \circ \rho \circ \mu$ consists of a composition of 4 linear steps denoted by μ, ρ, π and κ and a single non-linear step denoted by ψ. The μ step transforms each slice of the state using a Maximum Distance Separable (MDS) matrix. More formally, each slice of the state is viewed as a column vector of 5-bit words $(Z_i)_{0\leq i\leq 3}$ and is multiplied by a constant matrix defined as follows:

$$\begin{bmatrix} 2 & 1 & 1 & 1 \\ 1 & 1 & 18 & 2 \\ 1 & 2 & 1 & 18 \\ 1 & 18 & 2 & 1 \end{bmatrix} \begin{bmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \end{bmatrix} = \begin{bmatrix} 2Z_0 + Z_1 + Z_2 + Z_3 \\ Z_0 + Z_1 + 18Z_2 + 2Z_3 \\ Z_0 + 2Z_1 + Z_2 + 18Z_3 \\ Z_0 + 18Z_1 + 2Z_2 + Z_3 \end{bmatrix}$$

where all operations are performed in the finite field GF$(2^5)/(x^5+x^2+1)$. The ρ step bitwise rotates all 20 64-bit words of the state according to the mapping

$$S[z] \mapsto S[z + r(x,y) \bmod 64], \tag{3.1}$$

where the offsets $r(x,y)$ are defined as follows:

$$\begin{array}{llll} r(0,0)=0, & r(0,1)=36, & r(0,2)=3, & r(0,3)=41 \\ r(0,4)=18, & r(1,0)=1, & r(1,1)=44, & r(1,2)=10 \\ r(1,3)=45, & r(1,4)=2, & r(2,0)=62, & r(2,1)=6 \\ r(2,2)=43, & r(2,3)=15, & r(2,4)=61, & r(3,0)=28 \\ r(3,1)=55, & r(3,2)=25, & r(3,3)=21, & r(3,4)=56 \end{array}$$

The π step permutes the words within the state by replacing the word at position (x,y) with the word at the position (x',y') according to the following rule:

$$(x',y') = (x + y \bmod 4,\ x' + y + 1 \bmod 5) \tag{3.2}$$

The ψ step performs a non-linear transformation of each 5-bit row of the state $(M_k)_{0\leq k\leq 4}$ using the following boolean equations:

$$\begin{aligned} Z_k = M_k &\oplus (\neg M_{k+1} M_{k+2}) \oplus (M_0 M_1 M_2 M_3 M_4) \\ &\oplus (\neg M_0 \neg M_1 \neg M_2 \neg M_3 \neg M_4), \qquad 0 \leq k \leq 4 \end{aligned} \tag{3.3}$$

Finally, the κ step adds a round constant C_r to the word at position $(0,0)$ in each round $0 \leq r < 12$.

Processing of the Input

In the data processing phase, the permutation P_6 is iterated to wrap the 128-bit secret message number σ^{SMN}, the associated data blocks σ_i^{AD} and the plaintext blocks σ_i^P into the state and to generate the ciphertext blocks c_i. Figure 3.3 shows the interleaved absorbing and squeezing of data. Each block σ_i^{AD} and σ_i^P has a length between 0 and 1024 bits and is padded as follows: A frame bit is appended to the block, followed by "1" and a number of "0"s to reach the required length of 1026 bits. The frame bit is used for domain separation as follows: The frame bit is set to "1" for the last associated data block σ^{AD} and all the plaintext blocks σ_i^P but the last. A detailed view on the padding is not needed for understanding this thesis and therefore out of the scope. Finally, in the tag generation phase, the permutation P_{12} is iterated once more to generate a 128-bit tag $T = S[0][0]] \parallel S[1][0]$ given by the concatenation of the two 64-bit words $S[0][0]$ and $S[1][0]$.

Optimized Hardware Implementations

For optimized hardware implementations, we first analyze the optimization potential of the individual subfunctions or building blocks. Then, based on the findings, we combine them to form an implementation that has a high throughput, but still obtains a reasonable area footprint. This also means, that it can make sense to use a suboptimal implementation for a subfunction, because it enables a global optimization. Therefore, in the following, we discuss how the different parts of the round function can be implemented:

The π step solely changes the position of the individual words within the state. Therefore, the π step can be implemented through simple rewiring, which comes at no additional costs in hardware, but also offers no further potential for optimization in FPGA implementations.

The ρ step performs a circular shift of each word of the state. This can be implemented either as rewiring in parallel architectures or as shift registers in serial architectures. We use the latter.

The μ step consists of 64×20 equations that operate on one slice of the state at a time. One way to implement the μ step is to instantiate these 20 equations and iterate over them for 64 clock cycles, thus trading area for time.

The ψ step applies the ICEPOLE S-box 256 times on all 5-bit rows of the state. The ICEPOLE S-box can be implemented either in combinational logic by instantiating the five boolean equations of Equation (3.3) or as a Look Up Tables (LUTs).

Finally, the κ step adds the round constants C_r. It can be implemented either as LUTs or as a particular shift register[4].

[4]As shown in Section 3.1.1 on page 30, κ cannot be realized by a LFSR as claimed in the original specification [58] because it contains a nonlinearity.

An optimized 20-bit Slice-Serial Implementation

All but the κ and ρ steps need to access at most one bit of each word within the state at the same time. Therefore, a very natural choice for a serialized implementation is a 20-bit sliced-serial architecture based on the 20-bit slice representation of the state S. In the following, every slice $S[x][y]$ is mapped to the linear vector v by the formula $v[4x + y] = S[x][y]$. Figure 3.4 illustrates the architecture diagram of our 20-bit serial architecture.

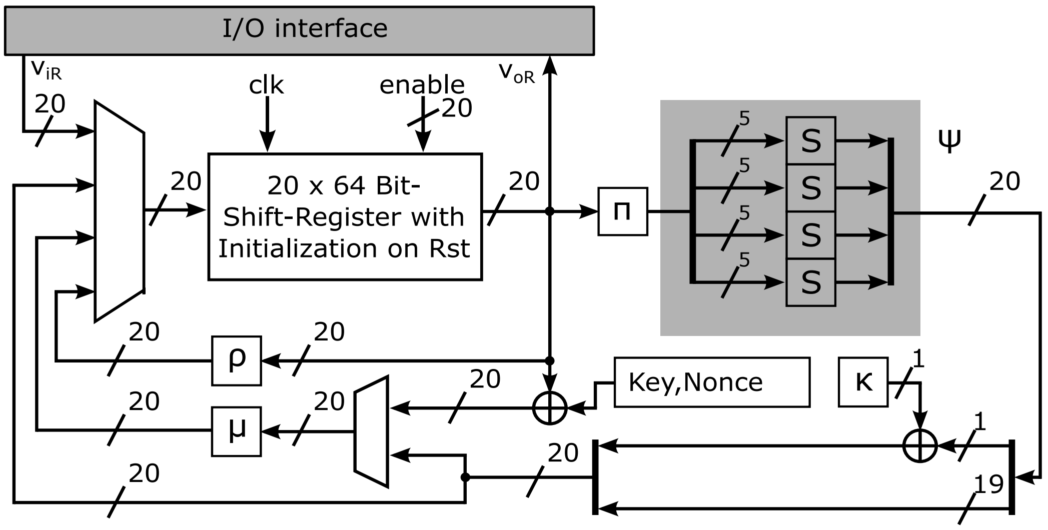


Figure 3.4: ICEPOLE: 20-bit architecture ©2016 IEEE [76]

The μ step is a straight forward serial implementation as explained in subsubsection 3.2.1.

With a 20-bit slice data path, the π step becomes trivial. It is a fixed rewiring of the 20 bits as described in Equation (3.2). After 64 cycles all words of the state are correctly reordered.

The implementation of the ρ step is a little more complicated, as it does not change the position of the bits within a slice, but within a word. In our slice-wise architecture the implementation of the ρ step is performed by the control logic:

The 1280-bit state is implemented using D-Flip-Flops (FFs) with input enable to control the shifting operations. Hence, 64 FFs are cascaded to form 20 64-bit simple shift registers[5]. The 4-to-1 20-bit multiplexer placed before the state register selects either a new input message from outside or one of the three parts (ρ, μ, or $\kappa \circ \psi \circ \pi$) of the round function R.

Each shift register stores one word of the state and has its own enable signal. The enabled FFs of the state are shifted by one position in each clock cycle. Thus, the ρ step takes 62 clock cycles to complete, as this is the largest offset given in Equation (3.1).

The ψ step applies the ICEPOLE S-box to each 5-bit row of the slice. Therefore, 4 instances of the ICEPOLE S-box are needed to process all 20 bits of one slice. Each S-Box takes 5 consecutive bits of the vector v as input. The output of the S-box is concatenated to form a 20-bit vector again. In a 20-bit slice architecture, the constants of κ must be serialized as the state words can only be addressed bit-wise. Alternatively, the constant can be generated on-the-fly with a non linear shift register as discussed in [76]. The output of κ is XORed with $v[0] = S[0][0]$ one bit per clock cycle.

The π, ψ and κ steps can be performed in the same clock cycle, as they all need 64 clock cycles and are consecutive steps in the round function R. The μ step also needs 64 clock cycles, but is separated from the π, ψ and κ steps by the ρ step, which only needs 62 cycles and requires access to the input and the output of the state register at the same time. The 2-to-1 multiplexer before the μ step provides a shortcut between two consecutive rounds of R. Hence, the steps $\mu_n \circ \kappa_{n-1} \circ \psi_{n-1} \circ \pi_{n-1}$ can be performed at the same time after the first round, thus resulting in a speedup of $5 \cdot 64$ cycles in P_6 and $11 \cdot 64$ cycles in P_{12}.

The first round takes $64 + 62 + 64 = 190$ clock cycles, while all the remaining rounds take $62 + 64 = 126$ clock cycles. In total, P_6 takes $190 + 5 \cdot 126 = 820$ clock cycles to complete, while P_{12} takes 1576 clock cycles.

Sugarcoated Results

We synthesized our design for a low-end Xilinx Spartan-E3 FPGA (*xc3s1200e-5fg400*), a high-end Xilinx Virtex 6 (*xc6vcx75t-3ff784*), and low-power Xilinx Artix 7 (*xc7a100t-3fgg676*). For fair comparison with other FPGA and ASIC implementations no hard-coded fabric, e.g. block RAMs, were extracted during the synthesis. As the original ICEPOLE implementation was optimized for speed [58], we also set the optimization goal to speed for all our targets. All results are after place-and-route with Xilinx ISE 14.7 and optimization using ATHENa

[5]In contrast to the original work in [76] and as an improvement, these shift registers could also be implemented using LUTRAM and LUTROM. On a Xilinx Virtex-6 or Artix-7 FPGA a 32-bit shift register can be implemented into one LUT using the SRL primitive [90, 92]. However, these shift registers cannot be initialized. Therefore, an additional LUTROM is needed to store the initial value of the state. On a Xilinx Virtex-6 or Artix-7 FPGA a 64-bit ROM can be implemented into one LUT [90, 92]. Thus, $2 \times 20 + 20 = 60$ LUTs and 64 additional clock cycle are needed to replace the 1280 D-FFs. On a Spartan-3E 80 LUTs are needed [89].

Table 3.1: Comparison with other implementations on a Xilinx Virtex-6

Design	**Area [Slices]**	**Freq. [MHZ]**	**TP [Gbps]**	**TP/Area [Mbps/Slice]**
ICEPOLE-128 (20-bit)	274	374.95	0.434	1.584
AES-GCM [94]	350	142.76	0.127	0.363
AES-GCM [31]	217	475.00	0.278	1.281

(Automated Tool for Hardware EvaluatioN) [27]. ATHENa uses a heuristic process to test for different synthesis options in order to archive the highest throughput, while maintaining a reasonable size. This results in 274 slices on the Virtex 6.

We computed the throughput according to Equation (3.4), where $B = 1024$ is the bit-size of the input blocks[6] and C is the number of cycles which is required to process one block:

$$T = \frac{B \cdot f_{clk}}{C} \tag{3.4}$$

Our design has a maximal obtainable frequency of 374.95 MHz. For large data the initialization and tag generation phase (P_{12}) can be neglected. Our 20-bit serial implementation needs $190 + 5 \cdot 126 = 820$ cycles to process P_6. Additional 64 cycles are needed to load and store the state, at which loading block$_{i+1}$ and storing block$_i$ take place at the same time. Four our 20 bit implementation, this results in:

$$T_{20} = \frac{1024\,\text{Bits} \cdot 374.95\,\text{MHz}}{820 + 64} = 434\,\text{Mbit/s} \tag{3.5}$$

Table 3.1 shows our results compared to two state-of-the-art impelmentations of AES-GCM. As it can be seen, our implementation outperforms the others in terms of throughput (TP) and competes in terms of area.

3.2.2 Is it fair?

The results presented in Section 3.2.1 lack comparability although they are based on assumptions that are perfectly fine with the guidelines of CAESAR. However, they cannot be generalized:

[6]The rate r equals $B + 2$, as each block is 10^{+} padded.

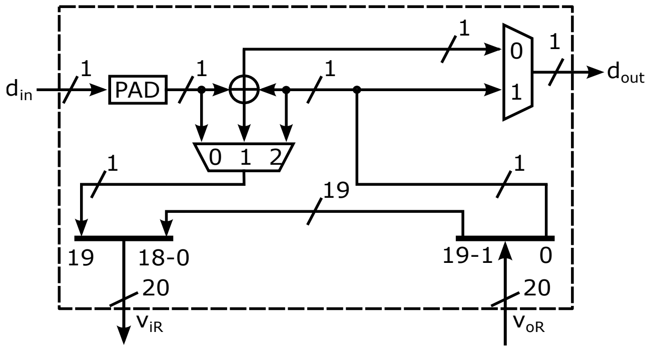


Figure 3.5: ICEPOLE: 1-bit I/O Interface ©2016 IEEE [76]

First, it is assumed that data arrive in chunks of 1024 bits, sliced and reordered into 20 bits. This could be argued by using a standard 32-bit interface, where only 20 bits are used. However, each of the 20 bits belongs not only to a different byte, but also a different 64-bit word. This might be justifiable for very special hardware software codesigns, but not for general applications.

In general applications, data arrives ordered in words with aligned bytes[7]. Thus, the data must be reordered. This can either be achieved implicitly by a 2D state access or explicitly by a dedicated circuit. The first option contradicts the design as it requires $r = 1026$ 2-to-1 multiplexers before each flip-flop and thus, eliminates the use of the shift registers. In this case, the complete design should be revised.

Figure 3.5 shows the second option. This circuit assumes a one-bit input stream, like used on many serial communication interfaces. The idea of this interface is to concatenate the 20 64-bit shift registers to one 1280-bit shift register.

This is depicted in the lower part of Figure 3.5: The vector $v_{oR}[1-19]$ of the output of the round function R is assigned to the vector $v_{iR}[0-18]$ of the input of the round function R, forming a $20 \times 64 = 1280$ bit shift register. Thus, the loading of the state takes 1280 cycles. 1026 cycles for loading the new input including padding (input mux = 0) and 254 cycles for shifting the capacity and aligning the input (input mux = 2).

Subsequent blocks are xored to the state (input mux = 1). As the state acts as a keystream that is xored to the input for encryption and decryption, this xored result is also output for decryption/encryption (output mux = 0). For the tag generation 128 bits of the state are output (output mux = 1).

Taking this interface into account, it must be distinguished between the raw throughput T_{20} of the round function, shown in Equation (3.5), and the normalized throughput T_n that takes

[7]either in big or little endianness

also the reordering into account:

$$T_n = \frac{1024\,\text{Bits} \cdot 374.95\,\text{MHz}}{820 + 1280} = 182\,\text{Mbit/s} \tag{3.6}$$

As the reordering takes longer than the actual processing of P_6, the throughput decreases dramatically[8].

[8]The time for loading can be halved as the 64 bit shift registers can be split into two 32 bit shift registers at no costs. It is then possible to form two 640 shift registers at the cost of 40 2-to-1 multiplexers. Assuming the input is derived from a serializer and the input width can be doubled a throughput of 263 Mbit/s can be achieved. However, this is still far less than T_{20}.

4 Hardware APIs

In this Chapter, we introduce and explain two hardware APIs. We compare them in terms of functionality and area; and cave out the pitfalls. Finally, we demonstrate their applicability on real world examples.

4.1 The CAESAR-API

To overcome the mentioned problems of sugarcoated results, which are technically fair but lack inter-cipher comparability, the CAESAR committee decided, as proposed and recommended by us in [76], to promote the GMU-Hardware-API [36] to be the official CAESAR Hardware-API [38].

The top level interface consists of two input ports, one for public data input (PDI) and one for secret data input (SDI); and one output port for data output (DO): The sole purpose of the SDI port is to transfer the (secret) key. All other data (instructions, plaintext, ciphertext, AD, etc.) are transfered over PDI and DO [39]. Each port consists of a data bus and two handshake signals to control the dataflow, as shown in Figure 4.1. This toplevel is to be defined as "AEAD". Data is transmitted when both handshaking signals (valid and ready) are high.

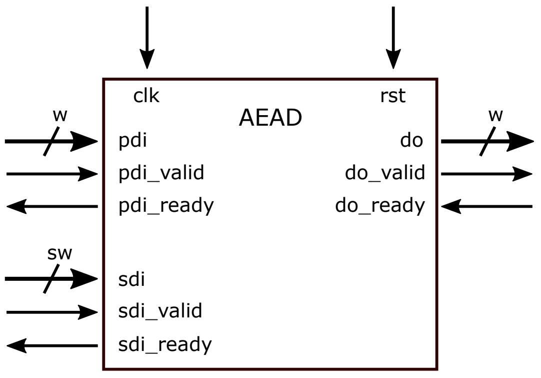


Figure 4.1: AEAD-Toplevel: PDI signals are used for sending public data input to the AEAD core, SDI signals are used to send the key to the AEAD core, and DO signals are used to receive data output from the AEAD core. ©2018 IEEE [77]

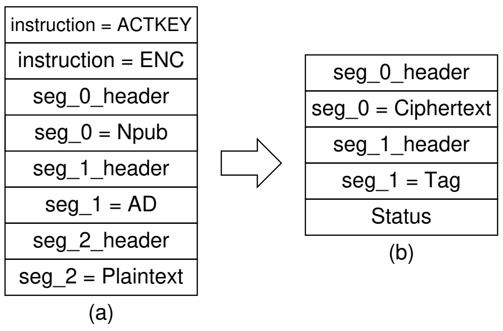


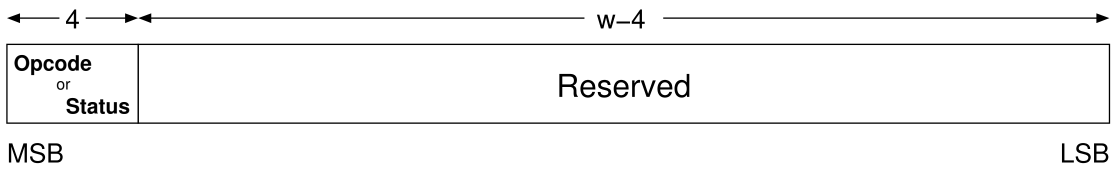
Figure 4.2: Sequential arrangement of PDI (a) and DO (b) for an authenticated encryption

This equates to the AXI4-Stream (AXIS) [91] protocol. With small adoptions, it can also be integrated in FIFO-based communication channels. The bus widths w and sw are defined as $w, sw \in \{8, 16, 32\}$ for lightweight implementations and $w \in [32, 256] \wedge sw \in [32, 64]$ for highspeed implementations [38].

The formats of SDI and PDI have a similar structure: they are composed of an instruction and one or several data segments. The DO port supplies the corresponding data segments followed by a status command. Figure 4.2 shows the sequential arrangement of the PDI and DO port for an authenticated encryption.

Instruction and Status Format: All instructions and statuses are defined to be 16 bits wide and include four bits of opcode or status and twelve additional reserved bits. Lightweight implementations, using a port width smaller than 16 bit, require the division of the instruction code into several words. Quite contrary to this definition, all hardware implementations based on the official Development Package [39] use the format shown in Figure 4.3. Thus, for $w < 16$, that is $w = 8$, the instruction code is *not* divided into several words, but only *one* word is sent! Figure 4.3 also lists all possible instructions and statuses.

The only possible opcode for SDI is loading a key (LDKEY), whereas PDI instructions begin with activating the key (ACTKEY) followed by information about the encryption mode (ENC for encryption or DEC for decryption). Each encryption or decryption is concluded by a status word sent to the DO port. For decryption, this indicates the success or failure of the tag verification. For encryption, this should always be success. Failures might occur in the event of a detected fault (attack).



Opcode:

0010 – Authenticated Encryption (ENC)
0011 – Authenticated Decryption (DEC)
0100 – Load Key (LDKEY)
0111 – Activate Key (ACTKEY)

Status:

1110 – Success
1111 – Failure

Others – Reserved

Figure 4.3: Instruction/Status format as it is used in the official Development Package [39].
Note: In the specification the "Reserved" field is fixed to 12 bits. For $w < 16$, multiple words were demanded [38], but never used in any implementation. For $w > 16$, the lower bits are undefined, but expected to be zero in [39].

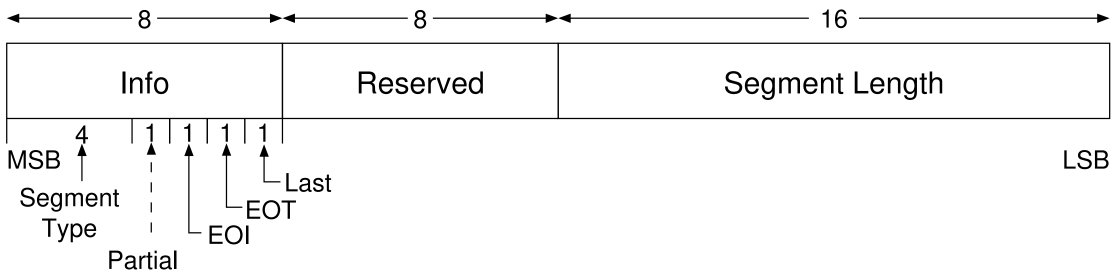


Figure 4.4: Segment header format as depicted in [39].
Note: For $w > 32$ – although not explicitly defined – only the highest 32 bits are used and the lower bits are treated as undefined, but expected to be zero in the official testbench [39].

Table 4.1: Segment type encoding

Encoding	Type	Encoding	Type
0000	*Reserved*	1000	Tag
0001	AD	1001	*Reserved*[1]
0010	Npub||AD[3]	1010	Length[2]
0011	AD||Npub[3]	1011	*Reserved*
0100	Plaintext	1100	Key
0101	Ciphertext	1101	Npub
0110	Ciphertext||Tag[3]	1110	Nsec[2]
0111	*Reserved*[1]	1111	Enc Nsec[2]

[1] later used in NIST-LWC competition
[2] not used by any last round CAESAR candidates
[3] not used in any CAESAR implementation

Data Segments: There are 16 different segment types. Table 4.1 lists the definitions of these types. However, only six of them are actually used. Each data segment starts with a 32 bit header, as shown in Figure 4.4, followed by up to $2^{16} - 1$ bytes of data. For more data, multiple data segments must be sent. Four flags indicate if the current segment is:

- "partial", i.e. the current segment contains an incomplete block,
- the "end of the input" (EOI), i.e. all *following* segments, except the one containing the tag, will be of size zero,
- the "end of the type" (EOT), i.e. no more segments of the current type are following,
- the "last" segment of the current instruction.

The partial flag is optional and was only required by the second round CAESAR candidate AES-COPA. The remaining three flags were intended to provide greater flexibility: The EOI flag provides the look-ahead information that the only following input, a cipher must process, is the tag. Thus, all other received (empty) segments can be skipped. The "last" flag indicates that no optional (empty) segments are following. However, no hardware implementation made use of segment types that can be omitted.

Thus, this flexibility seems to be a little bit overambitious, as the correct usage of these flags is not obvious, and the only practical usage is a reduced latency for highspeed ciphers in the case they process an empty input. However, this is a very rare use case at the cost of implementation complexity.

4.2 The LWC-API

As the NIST LWC competition is based on CAESAR, having a similar Application Programming Interface (API) is reasonable. Therefore, we proposed a similar API [45, 46]. Our LWC-API [47] is based on and fully backwards compatible to the CAESAR-API. The most notable extension is the support for hash algorithms.

It is tailored to lightweight cryptography and suggested to be used for benchmarking [46] hardware implementations of algorithms competing in the NIST-LWC competition [60]. Thus, it only supports widths $w, sw \in \{8, 16, 32\}$. Next, it clarifies the definitions of status/instruction segments, as depicted in Figure 4.3, for 8 and 16 bit implementation and of header segments, as depicted in Figure 4.4. Furthermore, it adds two optional features: an additional output `do_last` to be compliant with Xilinx's implementation of the AXIS protocol and a random data input (RDI) for random data, which can be used for remasking in side-channel resistant hardware implementations. The RDI port provides rw bits of random data. There is no specific protocol needed and the designer can chose rw arbitrarily. Figure 4.7 shows the toplevel of this API. It is renamed from "AEAD" to "LWC", as hashing is supported.

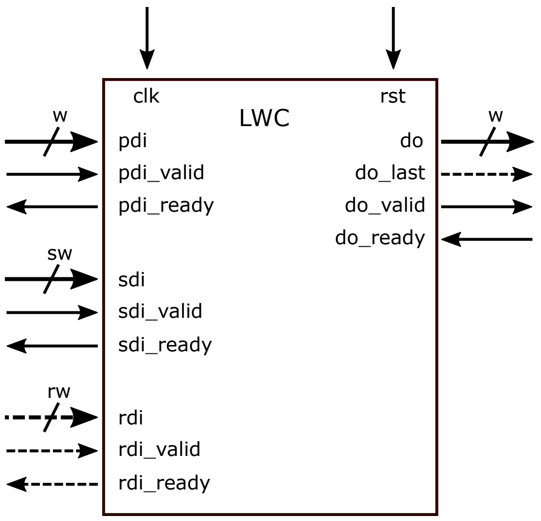


Figure 4.5: LWC-API

Protocol Extensions: The reserved encodings in the CAESAR-API are used to add support for hashing. Figure 4.6 depicts the new sequential arrangement for hashing. The new opcode HASH ("1000") is added to the opcodes defined in Figure 4.3. The reserved segment encodings, shown in Table 4.1, are used for hash messages ("0111") and the corresponding hash value ("1001").

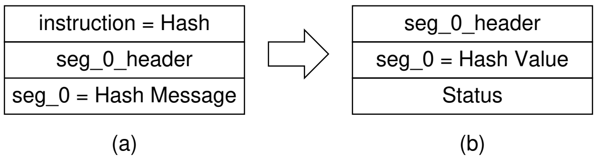


Figure 4.6: Sequential arrangement of PDI (a) and DO (b) for hashing

4.3 Comparison and Evaluation of the APIs

In 2016, the first CAESAR implementations that made use of the new CAESAR Hardware API, were published. They are based on the "Development Package for Hardware Implementations Compliant with the CAESAR Hardware API" [26]. Although written by the same authors as the specification of the API [38], it was stressed in [37] that

> *"the implementations of authenticated ciphers compliant with the CAESAR Hardware API can be also developed without using any resources described in this document, by just following directly the specification of the CAESAR Hardware API."*

Nevertheless, this development package became the de facto standard of implementing the hardware API, as hardly any CAESAR implementation used its own. Thus, the term *API* is widely used for both, the specification and the actual implementation of the protocol.

4.3.1 API Compliant Development Packages

In order to speed up the design process of hardware implementations, the CAESAR and LWC API both feature a corresponding development package and an Implementer's guide [39, 79]. As the LWC development package emerged from the CAESAR package, both designs consist of the same modules, i.e. the *PreProcessor*, a *HeaderFifo*, the *PostProcessor* and the *CipherCore*/*CryptoCore*[1]. All modules use the AXIS `_valid-_ready` handshaking protocol, which will be discussed in more detail in Figure 6.2 on page 76.

Figure 4.7 provides an overview of the structure and the modules in the CAESAR and LWC development packages. Each of the Pre- and PostProcessor consists of a Mealy-FSM that takes care of the protocol parsing and allows for forwarding the data directly to the CryptoCore and the DO port. Thus, control signals are registered, but data signals are not. To avoid a combinatorial path from the input (e.g. PDI) to the output (DO) data are registered in the HeaderFifo and the CryptoCore.

[1]The CipherCore was renamed to CrypoCore in LWC to also incorporate the hash functionality.

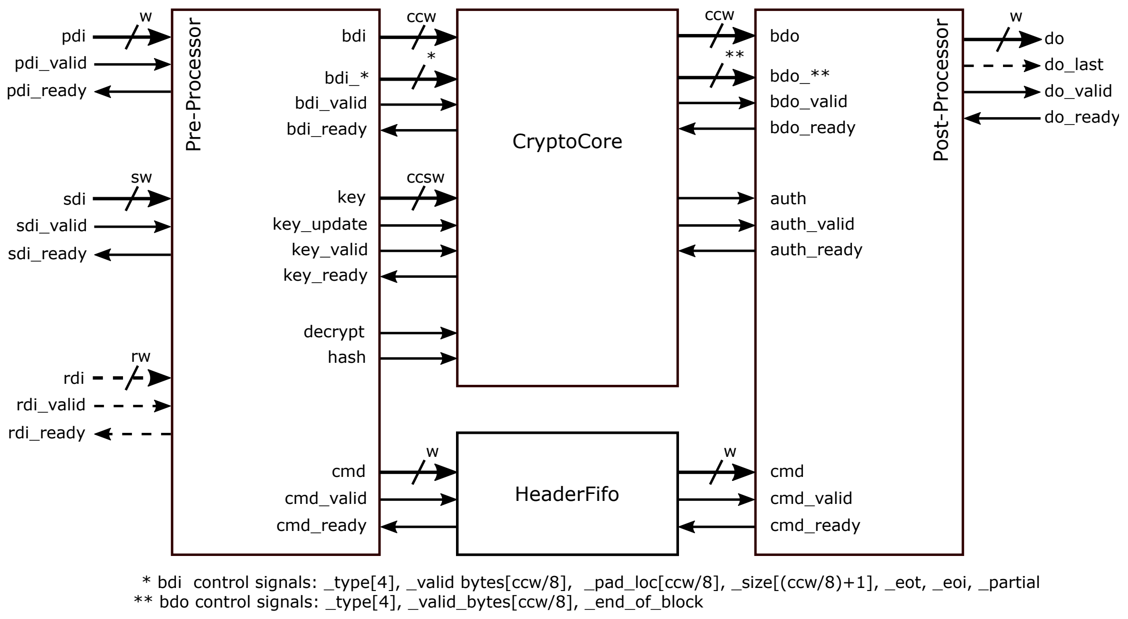


Figure 4.7: Modules and overview of the CAESAR/LWC Development Package

PreProcessor The main purpose of the PreProcessor is parsing segment headers to control the CryptoCore. Therefore, it receives data via the PDI and SDI ports. It then removes the header information and stimulates the CryptoCore: The public input, like plaintext or ciphertext, AD, or nonce is provided via the block data input (BDI) port; new keys via the key port. In some implementations the PreProcessor also takes care of the padding and width conversion between PDI and BDI and between SDI and key.

HeaderFifo The FIFO briefly stores and transmits instructions and segment headers to the PostProcessor. Thus, it also provides a clock boundary. Furthermore, in some implementations, the tag is forwarded to the PostProcessor.

PostProcessor The main purpose of the PostProcessor is generating the package format for the output. Therefore, it receives header information from the PreProcessor and data from the CipherCore. It then combines them, sends them to the DO port and adds the status. Furthermore, in some implementations the PostProcessor takes care of the tag comparison, the clearing of unused portion of the ciphertext and provides a width conversion between block data output (BDO) and DO.

Definition 2 *In the following, we will distinguish between the* external *and* internal *interface. The external interface is composed of the PDI, SDI, and DO port. The internal interface*

Table 4.2: Feature comparison of the different development packages

	CAESAR v1	**CAESAR v2**		**LWC**	
	HS	HS	LW	default	LW based
w	≥ 32	≥ 32	32, 16, 8	32	16, 8
sw	≥ 32	≥ 32	w	32	w
ccw	w	w	w	32, 16, 8	w
ccsw	sw	sw	w	32, 16, 8	w
multisegments	✓	✓	✗	✓	
stall save	✗	✓		✓	
do_last	✗	✗		✓	

refers to the connections between the CryptoCore and the PreProcessor (BDI, key, and control signals) and between the CryptoCore and the PostProcessor (BDO, msg_auth, and control signals).

Although the CAESAR and LWC packages have the same structure, they differ not only in the implemented features, but also in the way they are implemented. In the following, we will distinguish between three different packages. Table 4.2 lists the features of the different implementations and the parameter ranges of Figure 4.7.

CAESAR v1 (highspeed)

The very first "Development Package" [37] is based on the "Supporting Files for High-Speed Implementations v1.2" by the Cryptographic Engineering Research Group at George Mason University. It targets high-speed implementations and thus, it only supports bus widths ≥ 32. It provides a synchronous and an asynchronous mode. Furthermore, the Pre- and PostProcessors are not limited to protocol parsing, but can optionally also take over tasks, typically performed by the cipher itself:

- The PreProcessor can upscale multiple input words to form a complete message block as required by the cipher. It can also be configured to take care of padding incomplete input blocks.
- The tag comparison can be offloaded to the PostProcessor. Thus, for decryption, the CipherCore only needs to recalculate the tag. The received tag is forwarded from the PreProcessor to the PostProcessor, where it is compared with the one, computed by the CipherCore.

We showed that this version does not comply with the specifications of the CAESAR API. Besides several small problems, the design does not properly handle backpressure from the DO port. If there is a backpressure, e.i. `do_ready` is false during tag comparison, this backpreassure is not propagated to the CipherCore, as the `msg_ready` is omitted. Thus, information is lost and the complete design stalls [39].

As a consequence, this version was superseded by the CAESAR Hardware API version 2.

CAESAR v2 (lightweight)

This version [39] fixes the bugs reported in version 1. In order to fix the backpressure issue, the internal interface between the PostProcessor and the CipherCore needed to be replaced. Thus, this version is not fully backwards compatible to its predecessor.

This version also adds support for lightweight ciphers. However, it only allows for widths $sw = w \in \{8, 16, 32\}$. Additionally, the widths of the internal and external interfaces must not differ. Again, this version is not compliant with the specifications of the CAESAR Hardware API: While it properly realigns the 32-bit segment header from two 16-bit or four 8-bit words, it fails to do so for the opcode and status format. As in any case, the opcode and status information fits in one word, only one word is sent, regardless of the word size. This violates the API specification, but still allows for functional correctness of the cipher.

Additionally, the lightweight version does not support multiple segments of the same type. Thus, it only supports messages smaller than $2^{16} - 1$ bytes. This might be enough for most lightweight applications, but must be considered when comparing implementations that are fully compliant with the specification.

LWC

With the LWC package [79], we added support for hashing and extended the support for lightweight cryptography, as required by the NIST LWC project. There are two versions:

Default: Although, besides the support for hashing, the LWC package implements the same functionally as the lightweight CAESAR package, it is a complete rework, which is more resource efficient and better to maintain. We clearly distinguish between the Crypto-Core and the surrounding protocol parsing modules. Thus, the support for external tag comparison and padding was removed from the PostProcessor and the PreProcessor. All crpyto implementations must take care of these themselves. Furthermore, the FIFO was replaced by a more efficient version.

In addition to that, we provide a conversion feature[2]. Thus, a designer can configure different widths for the internal and external interface. As a consequence, the same test framework can be used for CryptoCore implementations with different data path widths. Width conversion also allows integrating 8- or 16-bit ciphers into –in embedded systems widely used– 32-bit designs.

LW based: As the community demanded also support for 8- and 16-bit external interfaces, there is also a version that supports them. This version is based on the lightweight CAESAR v2; thus having the same limitations regarding the bus widths.

4.3.2 Resource Analysis of Lightweight APIs

Parts of the following analysis have already been pre-published in "Patrick Karl and Michael Tempelmeier. A detailed report on the overhead of hardware APIs for lightweight cryptography. Cryptology ePrint Archive, Report 2020/112, 2020. https://eprint.iacr.org/2020/112". They were presented to the National Institute of Standards and Technology at the Lightweight Cryptography Workshop in October 2020.

With the additional features implemented, the question of how the development packages compare in terms of resource consumption arises. On the one hand, more features require more resources. On the other hand, an API for lightweight applications should not dominate the resource costs of the actual cipher implementation.

In order to evaluate the effects of the additional features and the lightweight rework of the development package, we synthesized the underlying modules of both support packages with different I/O widths in standalone mode. That means, only the module itself without the possibility of optimizations across module border is synthesized. For synthesis we used the Xilinx Vivado Design Suite v.2018.3 with default synthesis parameters for an Artix-7 FPGA. The resource consumption of FPGA designs is stated in terms of (Slice-) LUTs and (Slice-) registers, i.e. FFs. However, there are two types of Slice-LUTs. Whereas both types can be used for implementing logic, only one of them can be used as a memory element, i.e. LUTRAM. Therefore, we will abbreviate the overall number of occupied Slice-LUTs as LUTs, of which a subset will be explicitly used as LUTRAM.

[2]In contrast to the high-speed CAESAR package, this width conversion reduces the external interface width to match the internal one.

PreProcessor

The PreProcessor synthesis results are shown in the upper part of Table 4.3. For the 32-bit PreProcessor version it shows that the LWC version requires less LUTs, but two additional FFs compared to the CAESAR design. The additional registers implement flags used for hash support in the LWC version. The 16- and 8-bit versions both consume more LUTs. The 16-bit version consumes one additional FF, whereas the 8-bit version saves 22 FFs. By using the width conversions feature, additional FF are required. However, a conversion from 32-bit to 16- or 8-bit still consumes less LUTs than a plain 16- or 8-bit version. This is due to the fact that the 32-bit FSM implementation requires less logic than the 16- or 8-bit implementation.

PostProcessor

The LWC PostProcessor can be implemented to make use of two different flags, either the last_flit or the end_of_block signal, to determine the end of a transmission. They are both passed from the CryptoCore to the PostProcessor. The last_flit version allows using multiple data segments per message, thus allowing message sizes $> 2^{16} - 1$ bytes, which is required to fulfill the NIST requirement to support at least messages of $2^{50} - 1$ bytes. The end_of_block version, however, is more resource efficient, as it saves one 16-bit counter, but only allows transmitting messages of size $< 2^{16} - 1$ bytes, which is sufficient for most use cases. Although the PostProcessor in the lightweight version of the CAESAR package could make similar differentiations, the corresponding PreProcessor does not support splitting messages over multiple segments. Therefore, a different analysis of the PostProcessor is omitted and the default implementation with the last_flit flag is used. The PostProcessor synthesis results are shown in the lower part of Table 4.3.

For the last_flit configuration, the LWC PostProcessor requires in general more LUTs and FFs compared to the CAESAR version. The only exception is the FF requirement in the 8-bit case, where the LWC version saves 11 FFs. As one can expect, using the width conversion functionality results in an additional overhead, because the PostProcessor requires additional resources for data alignment.

When using the end_of_block flag, the LWC PostProcessor saves a significant amount of FFs compared to the CAESAR version, while still implementing the same functionality regarding message sizes. Even if the PostProcessor is configured to convert between internal and external widths, the amount of required FFs is reduced compared to the last_flit version. The 32- and 16-bit versions without conversion save LUTs, whereas the 8-bit version and the ones with width conversion require more LUTs compared to the plain CAESAR versions. As the message size of lightweight ciphers is unlikely to exceed $2^{16} - 1$ bytes, the end_of_block version is the appropriate configuration in terms of efficiency.

Table 4.3: Resource comparison of Pre- and PostProcessor

Module	I/O Width		CAESAR		LWC	
	ext.	int.	LUT	FF	LUT	FF
PreProcessor	32	32	95	33	88	35
	16	16	111	24	124	25
	8	8	137	56	159	34
	32	16	–	–	114	37
	32	8	–	–	111	39
PostProcessor (last_flit)	32	32	87	20	92	28
	16	16	77	21	86	22
	8	8	67	42	111	31
	32	16	–	–	105	45
	32	8	–	–	112	54
PostProcessor (end_of_block)	32	32	–	–	67	12
	16	16	–	–	62	6
	8	8	–	–	78	15
	32	16	–	–	81	29
	32	8	–	–	88	38

HeaderFifo

For the LWC package, the design of the HeaderFifo has been replaced after routing problems occurred on some hardware platforms. The FIFO is designed to be implemented as a ring buffer using distributed RAM. The development package of the CAESAR API supports a feature where the tag verification is performed inside the PostProcessor and thus, the tag is passed from the PreProcessor to the PostProcessor. That means, the HeaderFifo must be large enough to buffer the tag plus the header information. For the provided dummy ciphers the tag is 128 bits. Therefore, the minimum size of the FIFO would be 128 bits plus the size of two additional words for the header. In other words, a 32-bit implementation of a cipher with 128-bit tag requires a FIFO with a depth of 6 words. As the RAM is addressed in power of two, a FIFO buffering 8 words would suffice. On Artix-7 FPGA fabrics, distributed RAM is implemented with the "RAM32M" primitive, which requires four LUTs per primitive [92] and is used in the 6×32 dual port RAM configuration.

Still, the default configuration of the HeaderFifo in the CAESAR package sets the depth to

1024 and thus differs by a factor of 128 which leads to a suboptimal design optimization[3]. In the LWC package the default configuration of the HeaderFifo is set to 4, as there is no tag comparison in the PostProcessor. To allow for a fair comparison in Table 4.4, the CAESAR and LWC FIFO were set to a depth of 4 words, which is the default configuration in the LWC package. It shows that the LWC FIFO needs significantly less resources than the CAESAR FIFO.

Table 4.4: Synthesis results for the HeaderFifo, 4 words (LWC default)

W	**CAESAR**			**LWC**		
	LUT	FF	LUTRAM	LUT	FF	LUTRAM
32	110	39	64	32	7	24
16	61	23	32	20	7	12
8	37	15	16	16	7	8

CipherCore / CryptoCore

All development packages provide a dummy cipher implementation to demonstrate the designs functionality. For the AEAD scheme, they implement the same specification; whereas the dummy cipher of the LWC package implements additional hash support. Just as for the other modules, we synthesized the CipherCore/CryptoCore as a standalone module in order to compare them. Since the LWC API supports hash functionality, the corresponding core was synthesized twice. The first version is unchanged, i.e. with hash support, whereas in the second run the CryptoCore's `hash_in` port was removed. Internally tying the `hash_in` flag to zero allows the synthesis tool to remove most of the logic required for hash support.

Table 4.5 shows the results for all three word sizes. It shows, that the savings of the LWC core depend on the configuration. In the 32-bit version, the LWC saves resource even if hash functionality is implemented. For the 8-bit case, the LWC core uses around 50 extra LUTs when implementing hash functionality and 7 extra LUTs when deactivating it. The reason for that difference is the impact of the LUTRAM. Whereas the CAESAR core makes use of 5 internal RAMs, the LWC core only uses 3 RAMs, i.e. 60%. As the absolute number of allocated LUTRAM cells decreases, the savings of the LWC core also decrease.

Comparing the LWC versions with and without hash support also shows that the absolute cost of the additional hash support stays relatively equal. This shows that in relative terms, additional features become more costly the smaller the overall design is.

[3]We will later show, that the default configuration of the HeaderFifo's depth in the CAESAR package is accountable for the bad performance of some CAESAR implementations and the problems when comparing different implementations.

Table 4.5: Synthesis results of the provided CipherCore (CAESAR) and CryptoCore (LWC)

ccw/	**CAESAR**			**LWC (without hash)**			**LWC (with hash)**		
ccsw	LUT	FF	LUTRAM	LUT	FF	LUTRAM	LUT	FF	LUTRAM
32	524	119	160	405	94	96	458	96	96
16	317	113	80	301	95	48	347	97	48
8	209	113	40	216	95	24	258	98	24

Bringing it together

Standalone synthesis prevents tools from optimizing logic across module borders. In order to compare the development packages of both APIs, we synthesized the whole package with its corresponding dummy cipher implementations. All synthesis results were obtained using Xilinx's Vivado-2018.3 on a Artix-7 FPGA fabric with default synthesis options. In particular, this means '-flatten_hierarchy = rebuilt', which "allows the QoR [Quality of Result] benefit of cross-boundary optimizations, with a final hierarchy that is similar to the RTL for ease of analysis[4]." This also means that in case of resource sharing the complete structure is added to one of the sharing modules.

First, both HeaderFifos were set to a depth of 1024, which is the default value in the CAESAR package. The synthesis results are shown in Table 4.6a. For every width, the LWC design consumes less resources. While only a few LUTs are saved in the 8-bit version, the 32-bit version saves 622 LUTs, which is saving of about 31% compared to CAESAR's LUTs. Reason for that is again, the huge amount of LUTRAM allocation in CAESAR's HeaderFifo. The LWC PostProcessor was configured to make use of the last_flit flag, supporting multi-segment messages[5]. Hash support was enabled in the LWC version as a mechanism for disabling hash was not intended in the original package.

In the second run, we reduced the FIFOs' sizes, because the HeaderFifo dominated the size of the whole design. Since in the LWC package the default value is four words, we configure the CAESAR FIFO to its minimum possible size, which is the size of the tag plus four additional words. As mentioned before, the depth must be a power of two. This results in depths of 8, 16 and 32 words for the 32-, 16- and 8-bit CAESAR versions. Table 4.6b shows the synthesis results. Now as the HeaderFifo is not the dominant factor anymore, the resource comparison follows the observations in Table 4.5. As the design becomes smaller, the CAESAR version's overhead due to increased LUTRAM consumption decreases and the additional hash support becomes an increasing factor.

[4]https://www.xilinx.com/support/answers/52257.html

[5]This is the default LWC configuration. As the lightweight CAESAR package does not support for multi-segment messages, the LWC implementation is not only smaller, but also supports more features.

Table 4.6: Synthesis results of the development packages implementing the dummy cipher for different HeaderFifo dimensions.

(a) Equally sized HeaderFifo: Depth is 1024 words.

W	CAESAR			LWC		
	LUT	FF	LUTRAM	LUT	FF	LUTRAM
32	2009	266	1184	1387	209	672
16	1226	212	592	1025	188	400
8	793	250	296	786	194	216

(b) Minimum sized HeaderFifo: CAESAR's depth is 8, 16, 32 words; LWC's depth is 4 words each.

W	CAESAR			LWC		
	LUT	FF	LUTRAM	LUT	FF	LUTRAM
32	771	213	224	705	166	116
16	588	185	112	564	151	60
8	458	232	56	518	170	32

Comparing the default configurations i.e. Table 4.6a for CAESAR and Table 4.6b for LWC, respectively, it shows that the 32-bit LWC version saves around 1304 LUTs due to a large saving in LUTRAM. For the 16- and 8-bit versions, the savings decrease but still are significant. The same holds true for the FF savings. Table 4.7 summarizes these numbers for the 32-bit dummy implementation using default parameters. Using the CAESAR package with default configuration for lightweight cipher comparison might therefore distort the comparison of ciphers because the package is likely to dominate the resource requirements.

4.3.3 Exemplary Analysis of Published Implementations

In the following, the impact of the API packages on the resource consumption of different cipher implementations is analyzed. For evaluating the CAESAR package, the Ascon128 implementation from [29] and the SpoC-64 implementation from [81] are taken. For the LWC package, we took the Ascon128 implementation (without hash support) from [81] and the SpoC-64 implementation from [81]. The CAESAR and LWC variants for both, the Ascon128 and SpoC-64 implementations were configured with the same parameter set. However, the CAESAR Ascon128 was implemented using the development package version 1.0.3, which is a high-speed variant. The lightweight support for the CAESAR package was first introduced in the current release 2.0. Taking the author's claimed numbers into account, the Ascon128 LWC version requires more cycles per associated data block (factor of 1.5) and message blocks (factor of 1.7) compared to the CAESAR high-speed version.

Table 4.7: Synthesis results for the 32-bit dummy with '-flatten_hierarchy = rebuilt'

	CAESAR			**LWC (without hash)**			**LWC (with hash)**		
	LUT	FF	LUTRAM	LUT	FF	LUTRAM	LUT	FF	LUTRAM
Toplevel	2 009	266	1 184	704	163	116	705	166	116
PreProcessor	124	33	0	128	34	0	134	35	0
PostProcessor	43	20	0	56	28	0	56	28	0
HeaderFifo	1 354	94	1 024	51	7	20	51	7	20
CryptoCore	488	119	160	469	94	96	464	96	96

In addition to that, the Gimli implementation from [75] was included to demonstrate the package overhead for extremely constrained implementations. The hash support for the Gimli implementation was manually deactivated[6] for comparability.

Table 4.8 lists the synthesis results for the 32-bit implementations. For the CAESAR implementations, Ascon128 requires less resources than SpoC-64, especially in terms of LUTRAM consumption. This is due to the different parameterization of the HeaderFifo. For SpoC-64, the FIFO was configured to a word width of 32-bits and depth of 512 words. In the Ascon128 implementation, however, the FIFO width was trimmed to 24 bits[7] and the depth set to 4 words.

For the LWC implementations with equally sized HeaderFifos, Ascon128 requires more LUTs than the SpoC-64 implementation, but less FFs. As the Gimli implementation is specifically designed for resource optimization, it requires less LUTs and FF than the other implementations. The increased LUTRAM requirements come from the fact that the 384-bit Gimli-state is implemented in LUTRAM. This reduces the amount of required FFs but decreases performance because only parts of the state are accessible in each clock cycle.

As previously stated, the sizes of the HeaderFifos for the CAESAR implementations were manually adjusted. The implementation from [29] for example, reduced the FIFO to its minimum size such that functional correctness is guaranteed. Due to the different parameterization, Table 4.8 does not allow a fair comparison of the cipher implementations itself, as the overhead added by the API package's modules is not comparable. To show that, the LUT requirements of the implementations, shown in Table 4.8, are either assigned to the CryptoCore or to the API package modules[8]. For Ascon128 and SpoC-64, the assignments are depicted in Figure 4.8. For Gimli, the assignments are depicted in Figure 4.9.

[6]The CryptoCore's `hash_in` port was removed and an internal `hash_in` flag was tied to zero. This allows the synthesis tool to trim most of the hash-logic in the CryptoCore.

[7]Only 24 bits of the 32-bit header word are actually used.

[8]This is not 100% accurate, as the hierarchy is rebuilt after optimization, but the best Xilinx's vivado offers. cf. `https://www.xilinx.com/support/answers/52257.html`

Table 4.8: Resources of different ciphers implemented with the CAESAR and LWC package

	Cipher	LUT	FF	LUTRAM
CAESAR	Ascon128[1] [29]	1595	818	42
	SpoC-64[2] [81]	2136	876	416
LWC	Ascon128[3] [81]	1802	539	20
	SpoC-64[3] [81]	1565	728	20
	Gimli[4] [75]	946	235	84

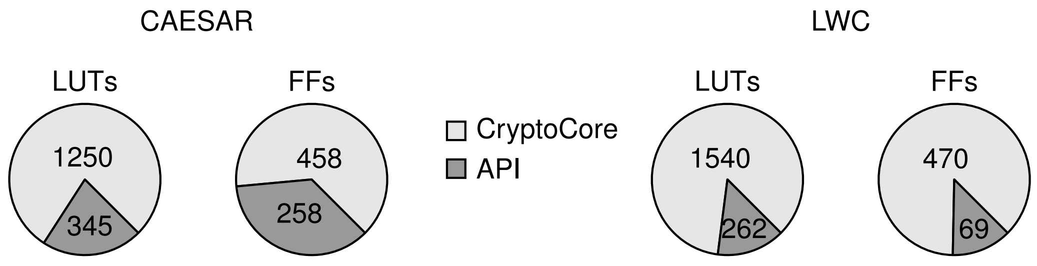
[1] HeaderFifo: 24×4
[2] HeaderFifo: 32×512
[3] HeaderFifo gets optimized from 32×4 to 24×4. State is implemented with FFs.
[4] Hash is deactivated for comparability.

For the Ascon128 cipher, Figure 4.8a shows that the LWC CryptoCore requires more resources than the CAESAR version. This overhead is mitigated by the LWC API package, which saves around 80 LUTs and 190 FFs. Nevertheless, the CAESAR version is a high-speed implementation, which of course adds additional overhead to the API package. Considering the whole design, the CAESAR implementation delivers more performance at less resource consumption than the LWC implementation.

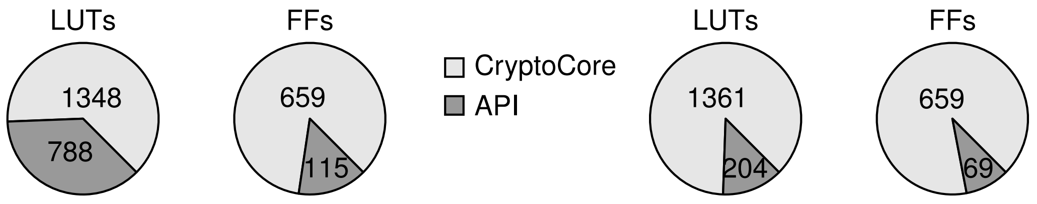
The comparison of the SpoC-64 implementations in Figure 4.8b shows a significant difference in the impact of the API modules; whereas both CryptoCore implementations are roughly of equal size, the development packages almost differ by a factor of 4 for LUTs and a factor of 1.7 for FFs. By taking only the CryptoCore into account, the difference in terms of LUTs between the CAESAR implementations of Ascon128 and SpoC-64 is not as drastic as it seemed in Table 4.8, but worse in terms of FFs.

In Figure 4.9, the same separation is done for the LWC Gimli implementation. It shows that for small ciphers the API package has a significant impact. In this specific case, the API package makes up around 28% of the overall LUT and 29% of the overall FF requirements. Nevertheless, the number of resources allocated by the LWC package is the same for the Ascon128 and the Gimli implementation. Thus, the API package adds the same overhead for both ciphers and allows for a fair comparison.

For the SpoC-64 implementation, however, the LWC package requires less resources. One might think that SpoC-64 allows for a better optimization, but the opposite is the case: 204 LUTs and 69 FFs for the API is very close to the sum (212 LUTs and 70 FFs) of the resources needed by the PreProcessor, PostProcessoor and HeaderFifo. Looking at the schematic of the synthesized SpoC-64 implementation confirms that the internal interface is preserved. The difference of 58 LUTs is located in the PreProcessor; more specific in



(a) Ascon128 with CAESAR (left) and LWC (right)



(b) SpoC-64 with CAESAR (left) and LWC (right)

Figure 4.8: Distribution of allocated LUTs for SpoC (4.8b) and Ascon128 (4.8a): The CAESAR package is shown on the left, whereas the LWC package is depicted on the right.

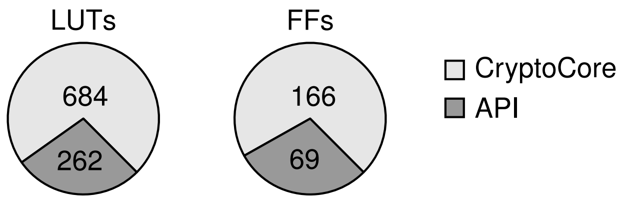


Figure 4.9: Distribution allocated LUTs for a constrained Gimli-AEAD LWC implementation

Table 4.9:
LWC development package without HeaderFifo and with the provided dummy CryptoCore

W	LUT	FF	LUTRAM
32	684	159	96
16	550	144	48
8	504	163	24

Table 4.10:
Resource savings without HeaderFifo and with the provided dummy CryptoCore

W	LUT	FF	LUTRAM
32	21	7	20
16	14	7	12
8	14	7	8

the logic of a 16-bit counter that holds the information about the remaining segments of the current type. Looking at the corresponding schematic, this information is shared with the CryptoCore. A out-of-context synthesis of the CryptoCore needs 62 LUTs more than the shared CryptoCore. Thus, in total 4 LUTs are saved at the costs of attributability.

There is no clear reason why this optimization cannot be performed for Spoc-64, as 4 LUTs are too little to manually assign them to their highlevel representation, nor does Xilinx's Vivado tell this. An educated guess is that cross-boundary optimizations are harder if this boundary is also the boundary between VHDL and Verilog, which is the case for the SpoC-64 implementation.

4.3.4 Tweaking the Development Package

The fact that different HeaderFifo dimensionings can lead to different impressions when comparing cipher implementations brings up the question whether the FIFO is required at all. The drawbacks of removing the FIFO is that there is a combinatorial path from the input of the PreProcessor to the output of the PostProcessor. However, for lightweight implementations where high frequencies are not necessarily a concern, removing the FIFO would save additional resources and improve comparability with respect to resource consumption.

When removing the HeaderFifo, it turned out that there are some implementation flaws in the PreProcessor regarding the valid/ready handshaking. We submitted a patch[9] for the PreProcessor that fixes this issue such that the HeaderFifo can be removed. The modification in the PreProcessor did not change the resource requirements significantly. Table 4.9 shows the synthesis results of the LWC development package without HeaderFifo and with modified PreProcessor. Comparing these numbers with the LWC default configuration in Table 4.6b shows that additional resources can be saved. Table 4.10 lists these resources. Although the savings in that case are not very large, it further reduces the impact of the API package and increases accuracy when comparing cipher implementations.

[9]`https://github.com/GMUCERG/LWC/commit/b97c9b8`

4.4 Summary

In this chapter, we introduced the CAESAR-API and presented an enhanced version that is suitable for the NIST-LWC-competition, called LWC-API. We had a closer look at the actual implementations of these APIs and analyzed their area footprints, both out-of-context and embedded in actual implementations of current ciphers. We clearly showed that the LWC-API outperforms the CAESAR-API and that the LWC-API is suitable for lightweight implementations.

Despite the benefits of a uniform API, it also poses a threat for misuse or misinterpretation of the results. Thus, when presenting optimized implementation results, it is crucial to provide information about the used API-implementation and its parameters. Even though this might not be enough for a 100 % accurate analysis, it is enough to provide a fair analysis, on which sound decision can be made. Unfortunately, this was not always the case during the CAESAR competition. Therefore, it is a fortiori important for the LWC competition to force designers to list implementation results together with the details about their API implementation. We presented this position at the Lightweight Cryptography Workshop hosted by the National Institute of Standards and Technology in October 2020.

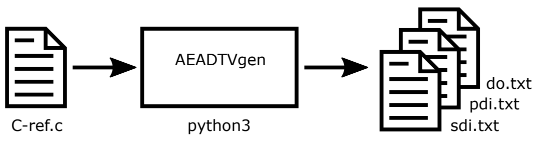
5 A Framework for Testing CAESAR-Hardware-Implementations

Having a uniform hardware API is the first step for a fair benchmarking. However, this is not sufficient. Testing the implementations of both the algorithms and the API is also crucial. Especially, if the challenges of Section 3.1 were taken into account, a uniform testing concept would be needed.

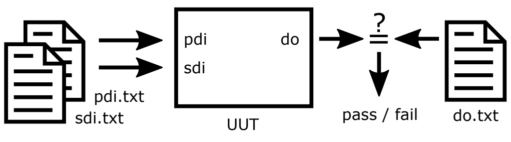
In the following we will show, why a pure simulation-based testbench is not enough. We will further propose a uniform FPGA-SoC-based test framework that overcomes the problems of the simulation-based approach.

5.1 Functional Verification using Simulations

The development packages feature a simulation testbench that reads test vectors (TVs) generated by the AEADTVgen[1], sends them to the Unit Under Test (UUT) and compares the received results with the ones specified in the TVs. This process is depicted in Figures 5.1a and 5.1b.



(a) AEADTVgen



(b) VHDL-Testbench

Figure 5.1: Software verification setup

We showed in [77] that this testbench is not sufficient for hardware testing: First, the testbench is very inchoate. It solely relies on the quality of the TVs. Thus, even if the TVs are well chosen and cover all corner cases, they only test the behavior of the CipherCore. They can by design not test the behavior of the API in case of input or output stalls. As a consequence, errors in the implementation of the API remained undiscovered until late in the

[1] `https://cryptography.gmu.edu/athena/CAESAR_HW_API/caesar_hw_devpkg-1.0-3.zip` renamend to CryptoTVgen in the LWC package

second round of the competition. We were the first to discover in [77] that the CAESAR-API does not handle backpreasure accurately [39].

Additionally, the TVs generated by AEADTVgen are actually only known-answer-tests (KATs). As they are assumed to be correct, there are no tests for faulty (e.g. altered) inputs. Thus, the authentication part in authenticated encryption is never tested. It would be easy for the test bench to alter the input by purpose and check whether the tag verification fails accurately.

While the problems mentioned above might have been caught by an improved testbench, the following problems cannot be solved in a pure simulation environment, as they at least need a post synthesis simulation, which might be too hard to achieve.

Synthesis and Simulation Mismatch: Perfectly fine VHDL-Code that passes any testbench may still fail on real hardware as it may contain non-synthesizable constructs, or –even worse– the same piece of code can be treated differently in synthesis and simulation. While non-synthesizable code can be found relatively easy by a synthesis run, an actual mismatch between simulation and synthesis is harder to catch. The most prominent example of a synthesis and simulation mismatch are omitted signals in the sensitivity list: In simulation, a process is only triggered if there is an `'event` on a signal which is also listed in the sensitivity list. During synthesis however, this list is ignored by most synthesis tools. As a result simulation and synthesis behave differently.

Latches and Timing: Although latches are valid components, which can be synthesized on both FPGAs and ASICs, their post-synthesis simulation might still not be accurate enough or not account for temperature or process variations. Thus, especially on FPGAs unexpected spikes can lead to an erroneous behavior. One example is the suboptimal Header-Fifo from the CAESAR package.

5.2 Functional Verification using Hardware Testbeds

In the following, we will show that although tested on the official testbench, there are many CAESAR implementations that are not actually functional, but nevertheless used in official benchmarkings [27].

Parts of the following analysis have already been pre-published in "Michael Tempelmeier, Fabrizio De Santis, Georg Sigl, and Jens-Peter Kaps. The CAESAR-API in the real world – towards a fair evaluation of hardware CAESAR candidates. In 2018 IEEE International

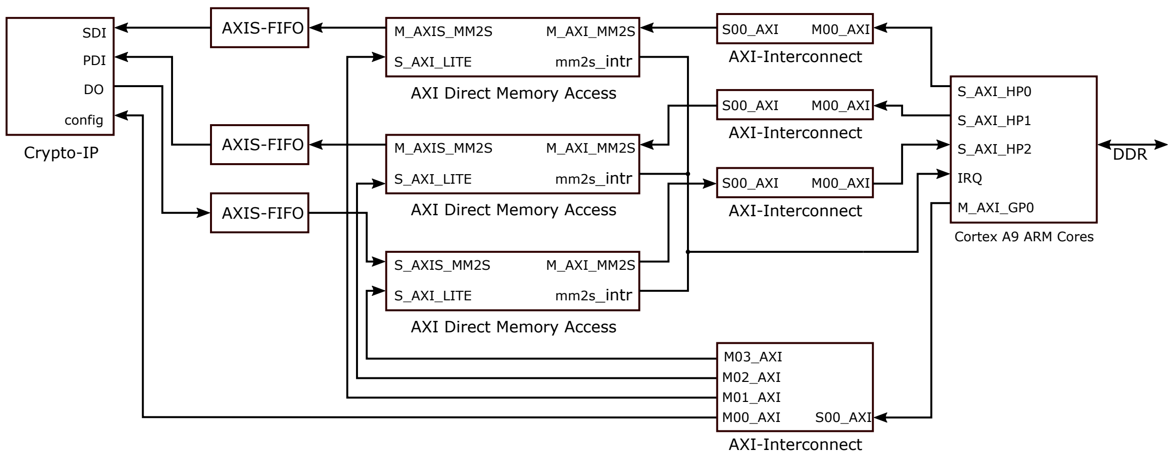


Figure 5.2: Block diagram of the evaluation framework including the ARM Cores in the PS, the Crypto-IP (c.f. Figure 5.3) in the PL and the interconnects ©2018 IEEE [77]

Symposium on Hardware Oriented Security and Trust (HOST)*, pages 73–80. IEEE, 2018". This is a extended version that takes more ciphers into account than the published one.*

5.2.1 System Design

Our evaluation framework is based on Xilinx' PYNQ-SoC and consists of two components. First, there is the Programmable Logic where the Crypto-IP, AXIS-FIFOs, DMA controllers and measurement cores are instantiated. Second, there is the Processing System which is dominated by the two ARM Cortex A9 cores hosting a Linux.

The complete system is shown in Figure 5.2. On the left side the Crypto-IP-Core is displayed and on the right side the Cortex-A9 processor system. In between, three AXIS communication channels with FIFOs for PDI, SDI, and DO are visible. They are mapped to three exclusively used, high-speed Advanced eXtensible Interface Bus (AXI) ports (HP0, HP1, HP2) of the Cortex-A9 by the DMA controllers in the middle. At the bottom, a shared 32 bit AXI4-Lite (AXIL) control bus connected to GP0 is used to configure the modules. Finally, the DMA controllers can request an interrupt via the IRQ port to signal that a DMA transfer is complete.

CAESAR-Candidate under Test

The internal structure of the Crypto-IP is shown in Figure 5.3. In the lower part of the figure, the normal structure of the CAESAR-API is depicted. A user can plug in the desired

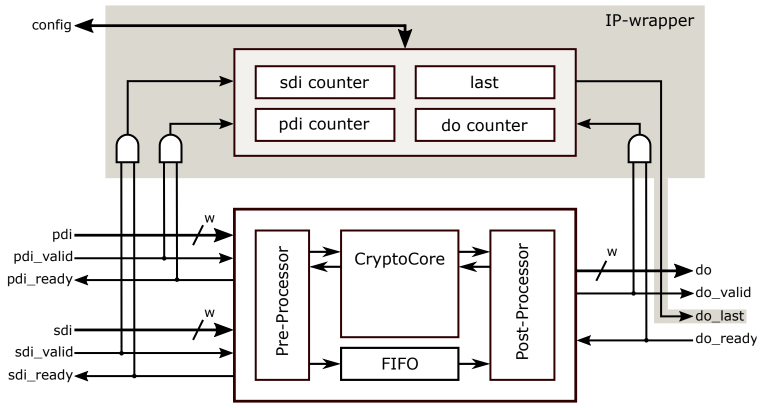


Figure 5.3: CryptoCore with IP wrapper

implementation of a CAESAR candidate by simply copying the corresponding VHDL (or Verilog) files to the source folder of the IP-Core. Two AXIS interfaces act as slaves for the public and secret data input, and one acts as a master for the public data output. All AXIS interfaces are set up in minimal configuration as defined in the CAESAR-API. This means that besides the data bus, only the two handshaking signals `ready` and `valid` are implemented. However, due to limitations of the Xilinx AXI-DMA-Controller, the AXIS DO master needs to eventually generate a strobe pulse on the `do_last` signal[2]. As this signal is not implemented in the CAESAR-API, an additional wrapper logic is needed. This logic is depicted at the top of Figure 5.3. The DO counter counts the number of sent DO words. If the number equals the expected “last” value, a strobe on the `do_last` line is generated[3].

Two additional counters count the number of received PDI and SDI words. This is for debugging only, but turned out to be quite handy. An AXIL interface is used to store the expected value of “last” and to read out the values of the counters.

Communication Channel

The interconnection between the PS and the PL is realized with the AXI protocol. As the Crypto-Core has FIFO-based interfaces, AXIS is used to communicate with the Crypto. AXIS is a point to point interconnect with a flexible data width and two handshaking signals. Since the PS does not directly support AXIS, an AXIS to AXI converter is needed. This is accomplished by the Xilinx’s AXI-DMA-Controller. It enables direct memory access to PYNQ’s DDR3 RAM through an AXI directly connected to the memory controller inside the PS on the one side, and a AXIS on the other side. For configuration of the DMA controller (e.g. initiate a DMA transfer), an AXIL interconnect is used. The AXI and AXIS both support

[2]Xilinx AR# 60053 `https://www.xilinx.com/support/answers/60053.html`
[3]The LWC-API incorporates this features into its core such that this counter is not needed anymore.

bus widths of 32, 64, 128 and 256 bits. Therefore, no width conversion is needed for the different CAESAR candidates, only a different option during synthesis has to be selected.

Additionally, a Xilinx AXIS-FIFO IP-core is inserted in each AXIS connection between the Crypto-IP-Core and the AXI-DMA-Controllers. They have two purposes: First, they are needed for efficient DMA transfers, as they enable sending of multiple data to the Crypto-IP Core in advance without the need of a separate DMA transfer for each encryption or decryption. Second, they allow us to separate the clock domains between the Crypto-IP and the rest of the PL.

Software Interface

A Python3.6 framework abstracts the communication with the FPGA and the configuration of the SoC, so that the designer of a CAESAR implementation only needs to copy his/her sources of the CAESAR-API-compatible implementation to our framework, synthesize the project and copy the bit-file and tcl-file to the SD-card. Afterwards, the evaluation can either be done in the web browser using Jupyter, by the provided scripts, or by customized scripts.

5.2.2 Validation and Testing

It is easy to write test benches in a high-level design language such as Python or SystemC and to integrate them in the design process. However, most of these test benches are non-synthesizable and designed to run in a software-based simulator. The problem with this approach is that they cannot be reused for testing the circuit on real hardware. Unfortunately, writing a test bench that is synthesizable is very complex and also very inefficient, as the synthesizable subset of the HDL is not designed for tasks like file parsing and generating test vectors.

We overcome this problem by combining the flexibility of a software running on Zynq's processor with the test accuracy of a real hardware, running on the FPGA fabric, as shown in Figure 5.4. Like in the software verification setup of Figure 5.1, the AEADTVgen script is used to generate the TVs. Instead of a pure VHDL testbench, we use a python script to configure and verify the UUT. These parts are located in the PS.

The UUT itself is located in the PL. PYNQ's onbaord DDR-RAM is used to transfer data between PS and PL using Direct Memory Access (DMA). Additionally the PS's General Purpose Input/Output (GPIO) pins are used for additional control logic like Interrupt Request (IRQ) and AXIL. The shaded part of Figure 5.4 is displayed in more detail in the left part of Figure 5.2.

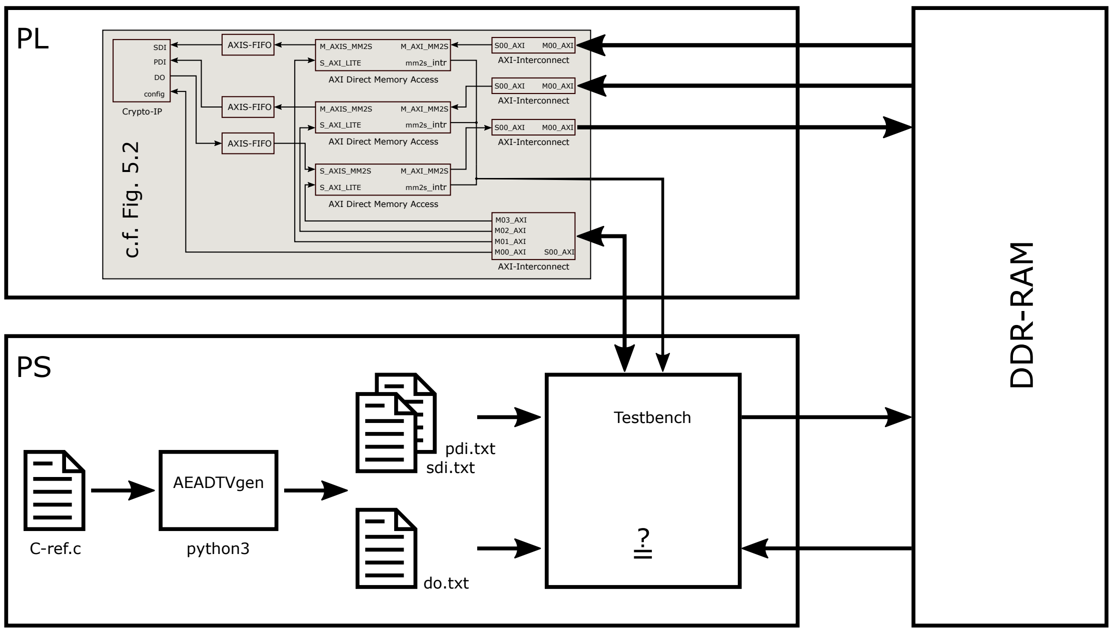


Figure 5.4: Test vector generation on hardware

We synthesized all eleven CAESAR-candidates provided by George Mason University [26], as they were the first that had been extended with the CAESAR-API and fulfill the requirements of the third round. In order to cover all CAESAR finalists, we also examined our own implementations [75], and the official hardware reference implementations from [59]. For better comparison, we also synthesized the provided dummy-cipher, which basically implements an XOR functionality, and thus shows the hardware overhead for the Pre- and Post-Processor, the Cipher Core control logic, the FIFOs, and the AXI controllers.

5.2.3 Functional Behavior

The first thing we discovered during our tests was that only two out of the eleven implementations from [26] worked as expected. Nine of them, plus the provided dummy, behave differently on real hardware than in simulation. The same holds true for the the official hardware reference implementations, taken from [59]: Both ciphers failed on our hardware tests. Table 5.1 summarizes the status of the synthesized ciphers. Further investigations showed that there are two main problems.

First and most important, the implementations from [26] use the Pre- and Post-Processor of the official Development Package for the CAESAR-API in version 1.0-3 that, as presented in Section 4.3.1, have some problems when the inputs and outputs are not transmitted in one continuous stream, but when there are stalls. The input stall can be fixed with a patched

Table 5.1: Examined ciphers in simulation and on PYNQ

Implementation (software reference)	**Research Group**	**Simulation**	**On PYNQ**
Acorn32 v1.1 (acorn128v2)	NTU [59]	✓	✓†
AEGIS-128 v1.1 (aegis128)	TUM [75]	✓	✓
AEGIS-128l v1.1 (aegis128l)	GMU [26]	✓	✓*
AEGIS-256 v1.1 (aegis256)	TUM [75]	✓	✓
AES-GCM v1.0 (aes128gcmv1)	GMU [26]	✓	✓*
AEZ v2.1 (aezv4)	GMU [26]	✓	✗
ASCON v2.0 (ascon128v12)	GMU [26]	✓	✓*
CLOC v1.2 (aes128n8t8clocv2)	GMU [26]	✓	✗
COLM v1.0 (colm_0)	GMU [26]	✓	✓*
Deoxys-I v3.0 (deoxysi128v141)	GMU [26]	✓	✓*
Deoxys-II v3.0 (deoxysii128v141)	TUM [75]	✓	✓
JAMBU-AES v1.0 (aesjambuv2)	GMU [26]	✓	✓*
NORX v3.0 (norx6441v3)	GMU [26]	✓	✓
Morus v1.0 (morus1280128v1)	NTU [59]	✗	✗
Morus v2.0 (morus1280128v2)	TUM [75]	✓	✓
OCB v1.0 (aeadaes128ocbtaglen128v1)	GMU [26]	✓	✓
SILC v2.0 (aes128n12t8silcv2)	GMU [26]	✓	✓*
Tiaoxin v2 (tioxinv2)	GMU [26]	✓	✓*
dummy1 (dummy1)	GMU [26]	✓	✓*

* Works only after some patches in the Cipher Core or API.
† Empty input (ϵ-string) and inputs lager than 2048 bytes are not working.

version of the Pre- and Post-Processor. To fix the output stall, a change in the handshaking protocol between the CipherCore and the Post-Processor would be needed, and thus an update of the package [39]. As this is a major change in the implementations, we did not patch this, but mitigated the problem. As a workaround for our experiments, a larger DO-FIFO that can hold up to the last but one block of a message was sufficient.

Second, some ciphers had internal problems within their implementation. Most of these are very minor and could be fixed by the authors after we had informed them. They are marked with an asterisk ($*$). However, this does not hold true for implementations from [59]. Acorn32 stalls on an empty (ϵ-string) input and produces wrong results for inputs that are larger than 2048 bytes. Unfortunately, the problems with Morus – described in Section 3.1.2 – were not fixed. Even after informing the authors of the problems, they have not published fixed versions yet. Therefore, we included our own implementation for Morus [75] for benchmarking. For Ascon32 we stuck to the official reference implementation, as there are only a few bytes wrong, and the expected amount of data matches. Thus, we don't expect these performance results to significantly change once these problems are fixed.

5.3 Resource Implications

Table 5.2 shows the synthesis results of the running implementations including the minimal infrastructure shown in Figure 5.2. The three FIFOs have a depth of 64 each. For a better comparison the used bus widths are included in the table. Table 5.3 shows the resources needed for the Crypto-IP as depicted in Figure 5.3. They were obtained from an out-of-context synthesis. As it can be seen, there is no common bus width neither for the external nor for the internal interface. Next, some implementations make use of the up-scaling functionality of the high-speed variant of the CAESAR-API. This makes a fair comparison hard.

Notwithstanding the above, we will try to give a brief interpretation of the results, based on Table 5.3: By far the smallest designs are Jambu-AES, Acorn, and Ascon. In terms of LUTs, Deoxys-I can also compete. They are all based on different cryptographic primitives: Jambu-AES is a block cipher and uses AES-128 as its core; Acorn is a stream-cipher and based on a non LFSR; and Ascon is based on a sponge construction. All met the requirement to be smaller than AES-GCM. The results show that non of the different cryptographic primitives is superior by design. Considering designs that are based on AES-128, it shows that there is a wide range of results: While Jambu-AES and Deoxys-I are (slightly) smaller than AES-GCM, Silc is about the same size; OCB is slightly larger, and Deoxys-II and Colm are much larger than AES-GCM. These differences are not only based on the different AEAD-constructions but also heavily depend on the design choices for the AES core. Implementations targeting high-performance use a faster and thus larger implementation of the AES core than implementations targeting lightweight applications.

Table 5.2: Resource utilization of the PL as depicted in Figure 5.2 including the corresponding Crypto-IP.

Implementation	**Key / BDI width**		**SDI / PDI width**[2]		**Slices**[1] **(total)**	**LUTs**[1] **logic**	**Slice**[1] **Registers**	**BRAM Tile**
Acorn32 v1.1	128	32	32	32	1 927	4 534	6 125	6
AEGIS-128 v1.1	128	128	128	128	4 602	13 740	8 845	15
AEGIS-128l v1.1	32	256	32	256	4 733	13 277	10 501	21
AEGIS-256 v1.1	256	128	128	128	5 251	15 623	9 621	15
AES GCM v1.0	32	128	32	32	2 477	6 349	6 621	10
ASCON v2.0	32	64	32	32	2 047	4 922	6 155	6
COLM v1.0	128	128	32	32	3 765	11 052	7 946	10
Deoxys-I v3.0	32	128	32	32	2 615	6 656	6 855	6
Deoxys-II v3.0	128	128	32	32	2 800	7 850	7 071	6
JAMBU-AES v2.0	32	64	32	32	2 039	4 884	5 956	10
NORX v3.0	32	768	32	256	4 249	10 707	12 077	21
Morus v2.0	128	256	128	256	4 904	13 465	11 225	24
OCB v1.0	32	128	32	32	2 900	7 600	6 878	10
SILC v2.0	128	128	32	32	2 602	6 557	6 479	6
Tiaoxin v2	32	256	32	256	4 827	13 076	11 024	21
dummy1	128	128	32	32	2 067	4 540	6 682	6

[1] The numbers of used resources (slices, LUTs and registers) are lower than the ones presented in [77] because the block design has been further optimized.
[2] The DO width is omitted because it is the same as the PDI width.

Table 5.3: Crypto-IP resource utilization

Implementation	**Key / BDI width**		**SDI / PDI width**		**LUTs**		**Reg.**	**Mult.**		**BRAM**
					logic	**mem**	**FF**	**F7**	**F8**	**Tile**
Acorn32	128	32	32	32	1 139	42	1 044	5	1	0
AEGIS-128	128	128	128	128	9 006	42	1 618	3 008	1 440	0
AEGIS-128l	32	256	32	256	7 639	42	1 748	2 051	1 024	1
AEGIS-256	256	128	128	128	10 889	42	2 394	3 472	1 728	0
AES GCM	32	128	32	32	3 030	42	1 551	17	0	4
ASCON	32	64	32	32	1 519	42	1 085	0	0	0
COLM	128	128	32	32	7 683	42	2 876	1 137	256	4
Deoxys-I	32	128	32	32	1 224	42	1 612	1	0	0
Deoxys-II	128	128	32	32	3 483	1 066	2 001	769	128	0
JAMBU-AES	32	64	32	32	1 497	42	886	66	0	4
NORX	32	768	32	256	4 915	42	3 324	0	0	1
Morus	128	256	128	256	7 431	42	2 003	0	0	1
OCB	32	128	32	32	4 261	42	1 804*	618	160	4
SILC	128	128	32	32	3 225	42	1 409	717	256	0
Tiaoxin	32	256	32	256	7 369	42	2 271	1 539	768	1
dummy1	128	128	32	32	1 224	42	1 612	1	0	0

* The design contains 4 additional latches.

Table 5.4: Resource utilization of the interconnects

Implementation	**Key / BDI**		**SDI / PDI**		**LUTs**		**Reg.**	**Mult.**		**BRAM**
	width		**width**		**logic**[1]	**mem**[1]	**FF**[1]	**F7**[1]	**F8**[1]	**Tile**[1]
Acorn32	128	32	32	32	3 103	250	5 081	1	0	6
AEGIS-128	128	128	128	128	4 410	282	7 227	1	0	15
AEGIS-128l	32	256	32	256	5 283	313	8 753	1	0	20
AEGIS-256	256	128	128	128	4 410	282	7 227	1	0	15
AES GCM	32	128	32	32	3 027	250	5 070	1	0	6
ASCON	32	64	32	32	3 111	250	5 070	1	0	6
COLM	128	128	32	32	3 077	250	5 070	1	0	6
Deoxys-I	32	128	32	32	3 024	250	5 070	1	0	6
Deoxys-II	128	128	32	32	3 051	250	5 070	1	0	6
JAMBU-AES	32	64	32	32	3 095	250	5 070	1	0	6
NORX	32	768	32	256	5 437	313	8 753	1	0	20
Morus	128	256	128	256	5 673	319	9 222	1	0	23
OCB	32	128	32	32	3 049	250	5 070	1	0	6
SILC	128	128	32	32	3 040	250	5 070	1	0	6
Tiaoxin	32	256	32	256	5 352	313	8 753	1	0	20
dummy1	128	128	32	32	3 024	250	5 070	1	0	6

[1] Values are retrieved by subtracting the out-of-context synthesis results of the Crypto-IP (depicted in Table 5.3) from the ones of the complete system (depicted in Table 5.2).

Since it is obvious that a larger data path requires more resources than a smaller one, synthesis results of related works are often presented in relation to the block size, data path width, or throughput. However, these results are often limited to the cryptographic implementation itself, which is too simplistic.

Table 5.4 shows the resources needed for the interconnect structure of Figure 5.2 without the resources needed for the Crypto-IP. As can be seen, a standard 32 bit interconnect infrastructure needs about 250 LUTs, 5070 registers, and 6 block ram tiles for storage and 3000 LUTs for logic. Correspondingly, a larger interface requires a larger interconnect structure. For a 256 bit interface, this culminates in about 6000 LUTs, 9222 registers and 23 block ram tiles for a minimal interconnect structure. For a larger interconnect structures this effect increases. However, even for a minimal interconnect structure, these additional costs (2600 logic LUTs, 70 memory LUTs, 4150 registers, 17 block ram tiles) are in the same dimension as an average cipher core and larger than lightweight cipher core implementations. Therefore, for a fair comparision, this additional costs would need to be accounted for the cipher.

As a consequence, we recommend that hardware benchmarking of NIST-LWC-implementations should be performed on a common 32 bit interface.

6 Benchmarking of CAESAR-Implementations

Starting with the ECRYPT Stream Cipher Project (eSTREAM) [70] in 2004, software benchmarking suites became available for a fair and comprehensive evaluation of candidates, culminating in the SUPERCOP [8]. SUPERCOP evaluates hundreds of compilation options to find those which result in the best performance. The purpose of ATHENa [24] and Minverva [22] is to accomplish the same goals, but for hardware implementations on FPGAs. However, the inputs to these tools are only verified in simulations and never tested on actual hardware. As presented in the previous chapter, simulation-based results can lead to a false impression. Therefore, in the following, we will go one step further and present a low cost benchmarking framework that runs on actual hardware. We extend the framework presented in Chapter 5 such that it can be used to benchmark the performance as well as the power and energy consumption of any implementation compliant with the CAESAR or LWC API.

This chapter is structured as follows: In Section 6.1 we recap the overall structure of the setup and briefly introduce the new hardware blocks. In Section 6.2 we explain how the measurement setup uses these blocks to obtain reliable numbers. The actual results are presented in Section 6.3. In Section 6.4 we interpret these results and point out the consequences for the individual cipher categories.

Parts of the following analysis have already been pre-published in “Michael Tempelmeier, Georg Sigl, and Jens-Peter Kaps. Experimental power and performance evaluation of caesar hardware finalists. In 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2018” and “Michael Tempelmeier, Fabrizio De Santis, Georg Sigl, and Jens-Peter Kaps. The CAESAR-API in the real world – towards a fair evaluation of hardware CAESAR candidates. In* 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 73–80. IEEE, 2018”. This is an extended version that takes more ciphers into account than the published one and provides a more sophisticated evaluation.*

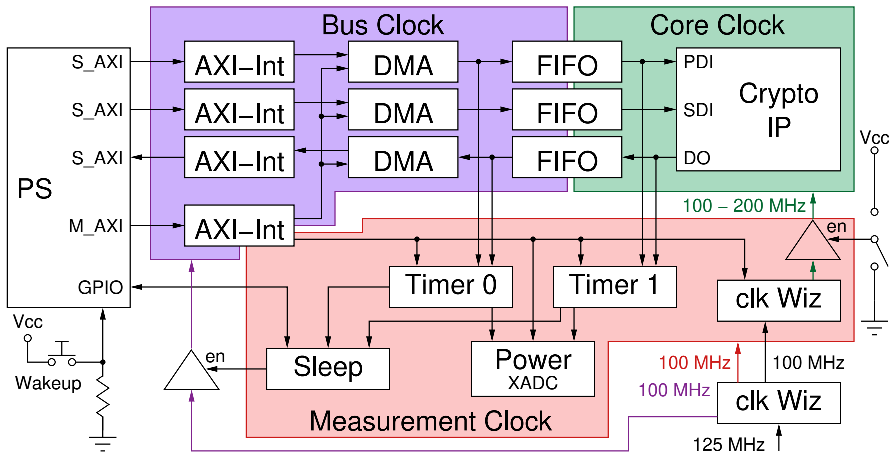


Figure 6.1: Block diagram with three clock domains ©2018 IEEE [78]

6.1 Hardware Architecture

Like in Chapter 5, our hardware architecture is based on Xilinx' PYNQ-board. Besides the mentioned advantages of a SoC, we will make particular use of the integrated Xilinx analog mixed signal module (XADC), which can be used to measure analog values in the range of 0 – 1 volt with a precision of 12 bits. Additionally, the PYNQ board features only very little distracting hardware.

Figure 6.1 shows the block diagram of our hardware architecture. It extends the one depicted in Figure 5.2 by defining three separate clock domains, two timers to measure the processing time, two clock generators (clk Wiz) to specify the clock for the Crypto-IP, and new blocks for sleep logic and power measurement logic.

The Sleep module monitors the clocks of the PS and PL and coordinates the fall asleep and wake-up process of the PS. It samples and filters the clock provided by the PS. This clock has a frequency of 100 MHz during normal PS operation. When the PS is asleep, this frequency drops to about 1 kHz and thus indicates that the PS has reached its sleep mode. During wake-up this process is reversed: After sending a wake-up signal to the GPIO, it waits until the PS clock is again stable at 100 MHz.

Since the clock provided by the PS depends on the power state of the PS, our design uses the external clock provided by the PYNQ board. In contrast to the PS clock, this clock is fixed to 125 MHz. Thus, a clock generator is needed to provide the 100 MHz system clock. This clock is spread across multiple clock regions in order to gate certain components (green and purple clock domain). A second clock generator is used to dynamically adjust the clock frequency of the Crypto-IP (green clock domain).

The Power module logic hosts the Xilinx's XADC macro and a logic to calculate the current average and the maximum of the digitized values. It is explained in more detail in Section 6.2.2.

The two timers measure the time needed to process an authenticated encryption. They are also used to trigger the Power and Sleep module. Their exact behavior is explained in more detail in the following sections.

6.2 Measurement Setup

6.2.1 Runtime

There are two timers in the setup: Timer 1 measures the time needed by the Crypto-IP and its triggers are therefore located between the FIFOs and the Crypto-IP; Timer 0 additionally includes the time needed for the bus transfers as well as possible wait cycles and thus its triggers are located between the FIFOs and the DMA-Controllers.

Timer 1 is started by both `pdi_valid` and `pdi_ready` being high. This indicates that the first word on the bus has been transferred (see blue data word in Figure 6.2). It is stopped by both `do_last`[1] and `do_ready` being high (see orange data word in Figure 6.2, which indicates that the last word on the bus was successfully transmitted.

Timer 0 is started by a rising edge of the `pdi_valid`, which indicates that there is data available, but not necessarily transmitted (see white data word in Figure 6.2). It is again stopped by both `do_last` and `do_ready` being high.

6.2.2 Power and Energy

The power consumed by a device can be computed from the current I that it draws and its supply voltage $\mathrm{V_D}$ as $P = I \cdot V_D$. The energy it consumes for executing a particular task is the integral of P over the run time. The current is measured by sensing the voltage drop across a small shunt resistor R_S. If the voltage drop is very small it can be assumed that V_D is constant. The most convenient way to measure the power consumption of the PYNQ board is the *Power Select* jumper (JP5), where the user can select whether the board is powered by the USB port or from an external source. We measure this voltage drop on an external Printed Circuit Board (PCB), as displayed in Figure 6.3. Next, this analog value is digitized and averaged in a dedicated measurement logic within the PL. Finally, these values

[1] `do_last` is always asserted together with `do_valid`

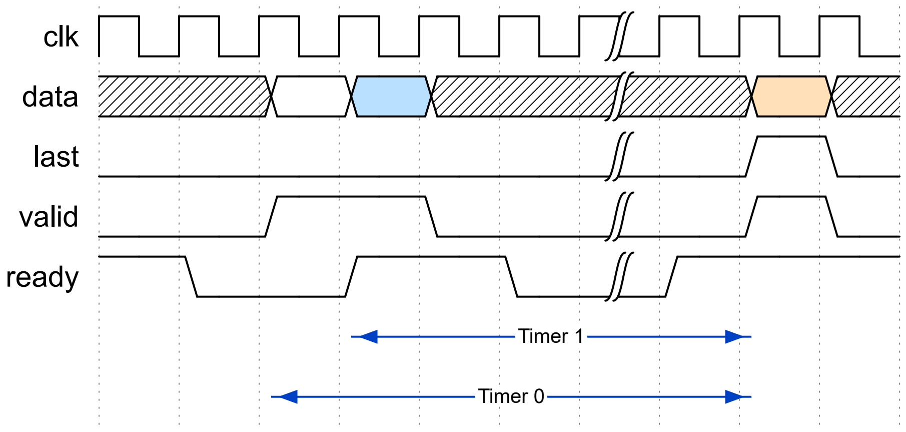


Figure 6.2: AXIS handshaking: Data is transmitted, when both lines "valid" and "ready" are high.

are transferred to the PS and analyzed. Measuring only the power consumption of the Zynq SoC is not possible without major modifications to the board. Unfortunately, this means we are measuring the power consumption of all devices on the PYNQ board including the DDR memory, Ethernet controller, FTDI USB chip, audio amplifier, etc. In the following we describe the mentioned steps in more detail and show how to deal with that error and how it influences the measurement.

Measurement Circuit

We use the XXBX Power Shim (XBP) [44], connected to JP5, which selects the source of the 5 Volt supply voltage (see Figure 6.3), to measure the current drawn by PYNQ. The XBP has a 0.1 Ohm shunt resistor R_S. However, that means that voltage drop across the shunt will be very small and has to be amplified before it can be measured by an analog to digital converter (ADC). Furthermore, the low input resistance of ADCs makes a direct measurement unfeasible. Hence, the XBP uses a current-shunt monitor (CSM), i.e. the INA225 from Texas Instruments. It has a programmable gain setting between 25 and 200, a buffered output so that it can drive an ADC input, a bandwidth of 125 kHz and supports the high-side measurements. We use ZYNQ's own XADC to measure the output of the CSM V_{CSM} and set the desired gain δ using ZYNQ's GPIO pins. The maximum analog input voltage on PYNQ is 3.3 Volt, hence the gain must be set carefully in order not to damage the XADC. The resolution V_{res} of the XADC is

$$V_{res} = \frac{V_{CSMmax}}{2^{ADCbits}} = \frac{3.3\,\mathrm{V}}{2^{12}} = 0.8\,\mathrm{mV}. \tag{6.1}$$

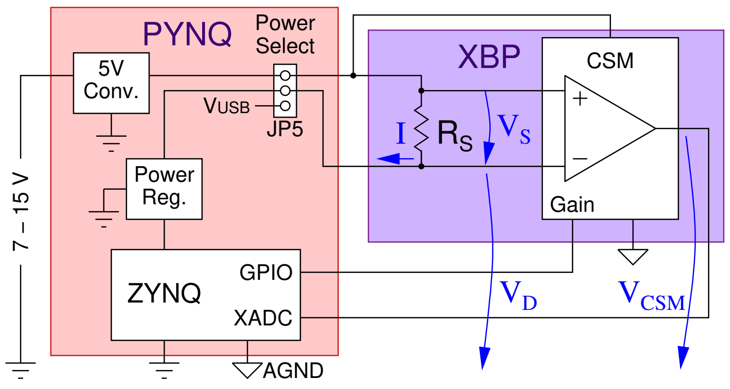


Figure 6.3: Block diagramm with external measurement circuit ©2018 IEEE [78]

Using a 0.1 Ohm shunt and a gain of 50, the current can be at most

$$I_{max} = \frac{V_{CSMmax}}{R_S \cdot \delta_{CSM}} = \frac{3.3\,\mathrm{V}}{0.1\,\Omega \cdot 50} = 660\,\mathrm{mA} \tag{6.2}$$

which is equivalent to a maximum power consumption of 3.3 Watt and measured at a resolution of

$$I_{res} = \frac{V_{res}}{R_S \cdot \delta_{CSM}} = \frac{0.8\,\mathrm{mV}}{0.1\,\Omega \cdot 50} = 160\,\mu\mathrm{A} \tag{6.3}$$

which equates to 0.8 mW.

Measurement Logic (Power IP)

A VHDL module was designed to instantiate the Xilinx XADC, capture the maximum power consumption, compute the average, and count the number of samples taken. This power measurement IP interfaces with the PS through AXIL. Measurements can be triggered through the hardware timers (section 6.2.1), software commands, and a push button.

The XADC is capable of approximately 1 mega samples per second (MSPS), which means, if the Crypto-IP runs at 100 MHz, every 100 clock cycles a measurement is taken. The power measurement IP core runs at 100 MHz which allows it to compute a new average after every measurement as

$$M_k = M_{k-1} + \frac{x_k - M_{k-1}}{k}, \tag{6.4}$$

where k is the measurement count, M_k is the current average and x_k is the current measurement. The computation of the average takes 37 clock cycles and uses a non restoring unsigned division. As the divider uses 32-bit numbers, up to 2^{20} measurements of 12 bits each can be sequentially averaged without loss of precision.

Error Minimization

There are two kinds of biases in our setup: First, there is the additional power consumption of the devices on the PYNQ board such as the Ethernet controller, USB chip, etc. As can be seen in Figure 6.3, the 5 Volt input converter and the CSM are not part of that bias. Second, also the SoC itself consumes some additional power, which again can be divided into two parts: the PS and the rest of the PL.

PS power consumption The largest amount can be mitigated by suspending the PS. We use the normal linux suspend-to-memory mode for that. Doing so, also disables parts of the additional components of the board such as the Ethernet controller. When in sleep mode, only the DDR3-RAM is kept awake for refreshing the data. It must be noted that the stock PYNQ-Linux kernel does not support sleep mode and has to be re-compiled.

PL power consumption The second largest amount of unwanted power consumption is drawn by the PL. We first deactivated the two data counter, introduced for debugging in Chapter 5, page 64, and all status LEDs. Second, we gated or reduced all clocks that are not needed during the power measurement. Figure 6.1 shows the different clock domains. The PL uses the external 125 MHz clock provided by the PYNQ-board, as this clock source is also stable when the PS is in sleep mode. This clock is fed to a Xilinx clocking wizard (i.e. a clock generator), which provides three constant, synchronized 100 MHz clocks:

- The *bus clock* domain (shown in dark purple) used for communication, contains the AXI-Interconnects, the DMA engines and the input part of the FIFOs. As there is no communication between the PS and PL, when the PS is in sleep mode during our measurement, this clock domain is gated, thus, reducing unwanted power consumption.
- The *measurement clock* domain (shown in red) contains the components for the measurements: two Xilinx timing-IP-cores as in [77] for timing measurements, the power module, the sleep control unit for managing the sleep mode from the hardware perspective, and the clocking wizard for the Crypto-IP.
- The third clock is used by this second clocking wizard.

This second clocking wizard creates the *core clock* that feeds the green clock domain, containing the Crypto-IP. This clock domain can dynamically be reconfigured from the PS, which is particularity useful as it enables a fast way to, first, determine the maximum working frequency of a given implementation of a cipher, and second, to measure the power consumption at different frequencies, thus also reducing the relative influence of the red clock domain, as the time of measurement can be reduced and the SNR can be increased.

Reference Measurement and Error Correction For those components, that cannot be switched off, or suspended, we made reference measurements to be able to subtract the amount of power consumed by them from the total power consumption.

For the first reference measurement we measured the total power consumed by the PS in sleep mode, the bus clock domain (purple) gated and core clock domain (green) also manually gated. This gives us the power consumption of our measurement clock domain (red) plus the static power of both bus clock and core clock domains as well as power consumption of rest of the system. This measurement is called “Avg Idle off” (brown ▼) in Figure 6.10.

For the second reference measurement we activated the core clock domain (green), but did not process any data in the Crypto-IP. This adds the dynamic power consumption of an idle Crypto-IP to our measurement. This measurement is called “Avg Idle on” (green ▲) in Figure 6.10.

Measurement process

We tested all candidates in a temperature controlled room after a warm-up phase of one hour with two different sets of test vectors. The first set contains five test vectors with each 1 kB of associated data and 2 kB of plaintext. The second set again contains five test vectors with each 1 kB of associated data, but 4 kB of plaintext. The values are chosen to be compatible with the software evaluation framework SUPERCOP [8]. Additionally, larger values allow more samples for the XADC.

A measurement cycle is started by the PS initiating a PDI (and if needed an SDI) DMA transfer to the PL. It then sends itself to sleep, indicating that to the PL with a dedicated GPIO-Pin. The sleep control unit notices that, and waits until the PS has reached its low power mode. It then stops the bus clock domain, and enables the core clock domain ports of the FIFOs. By doing so the Crypto-IP starts processing the data, Timer 1 and the power measurement module get triggered and the measurement is started. While data is processed, the power measurement module continuously calculates the average and maximum power consumption. After the data has been processed, Timer 1 and the power measurement module get triggered again, the measurement is stopped, the sleep control enables the bus clock domain and wakes up the PS. When the PS is fully running again, it initiates the DO DMA transfer, verifies the data and reads out the values of the power measurement and timer module.

The measurement cycles are repeated for each test vector. Subsequently, the two reference measurements are taken. We repeated this measurement process at least 10 times for each CAESAR algorithm and each frequency.

6.3 Results

We evaluated all working examples from Section 5.2.3 with the parameters (bus widths) described in Section 5.3. In the following we present the performance of the ciphers in four dimensions: area, throughput, power, and energy. While the first two are straight-forward reports, the results for power are divided into two main categories: no data is being processed (i.e. idle or stand-by power consumption), and power consumption during an authenticated encryption. The energy efficiency of a cipher is the result of its power consumption and its throughput.

6.3.1 Synthesis

Again, we synthesized all implementations using Xilinx Vivado 2017.2 with standard optimization settings but with a target frequency for the core clock domain (green) of 100 MHz and 200 MHz. The other clock domains were constraint to 100 MHz.

Except for Norx, all implementations met the timing requirements for 100 MHz. Norx failed to meet the timing requirements due to high net delays in 10 out of its 1024 state registers due to high congestion. As there is now a dedicated clock for the Crypto-IP, the usable area is limited to the coresponding clock regions. Thus, in contrast to Chapter 5, the logic cannot be spread across the PL, which makes routing more difficult. Nevertheless, the experimental results showed that Norx runs stable at 200 MHz and therefore can be included into our benchmarking. This also supports the results of [21].

Table 6.1 shows the resource utilization in the PL, optimized for an core clock of 200 MHz and 100 MHz. As can be seen, the difference between the 100 MHz and 200 MHz optimization is negligible. The overhead of our measurement setup compared to the minimal setup in Section 5.3 is about 3000 LUTs and 4000 registers.

6.3.2 Runtime

As shown in Section 3.2.2, the interface has a huge influence on the implementation results in terms of area and speed. For an indication of whether a hardware implementation, especially considering the overhead of the API and the data transfer, is suitable for real world scenarios, we compare the runtime of the non optimized software reference implementations executed on the PS with the runtime of the hardware implementations being executed in the PL and called from the PS for the same test vectors. The hardware performance measurement therefore includes all data transfers and function calls. All hardware clocks are fixed to 100 MHz. Testvectors include empty messages (ϵ-string), single and multi byte

Table 6.1: Resource utilization of the PL as depicted in Figure 6.1 including the corresponding CipherCore optimized for an AEAD clock of 100 MHz and 200 MHz.

	Resources 100 Mhz			Resources 200 Mhz		
Algorithm	**LUT**	**Regs.**	**BRAM**	**LUT**	**Regs.**	**BRAM**
Acorn32*	7 328	10 328	18.5	7 328	10 328	18.5
Aegis128	16 652	13 105	66.5	16 695	13 126	66.5
Aegis128l	16 360	14 746	126.5	16 377	14 746	126.5
Aegis256	18 539	13 881	66.5	18 574	13 902	66.5
AES GCM	9 152	10 824	22.5	9 168	10 824	22.5
Ascon	7 713	10 358	18.5	7 713	10 358	18.5
Colm	13 846	12 149	22.5	13 844	12 149	22.5
Deoxys-I	9 440	11 058	18.5	9 454	11 058	18.5
Deoxys-II	10 662	11 274	18.5	10 681	11 274	18.5
JAMBU-AES	7 690	10 159	22.5	7 713	10 159	22.5
NORX	13 792	16 322	126.5	13 788	16 322	126.5
Morus	16 580	15 542	128.0	16 595	15 542	128.0
OCB†	10 390	11 079	22.5	10 424	11 079	22.5
SILC	9 353	10 682	18.5	9 368	10 682	18.5
Tiaoxin	16 152	15 269	126.5	16 169	15 269	126.5
dummy1	7 341	10 885	18.5	7 343	10 885	18.5

* stalls on empty input
† including 4 latches

messages up to 3072 bytes. As the maximum transmission units for Ethernet frames is 1500 bytes this seems reasonable. The testvectors are composed of an equal amount of plaintext and AD.

Software Reference Figure 6.4 shows the performance of the reference implementation in C of all ciphers; to better distinguish between the fast ciphers Figure 6.5 zooms in by a factor of 10. As can be seen, the runtime of the reference implementation increases almost linearly with the amount of processed data. This was expected. The results can be grouped into three subgroups:

First, and most notable in Figure 6.4 are the slow Acorn32 and the AES-GCM reference implementations. AES-GCM relies heavily on the (galois field) multiplication. In the reference implementation this is implemented strait forward and not as t-tables. The reason for the poor performance of the lightweight cipher Acorn32 is its bit-serial algorithm. As the reference implementation does not make use of slice-wise calculations, only one bit at a time can be calculated.

Next, the Deoxys family can be grouped in Figure 6.4 and Figure 6.5. Both members of the family need multiple instances of an AES-based block cipher, but with a simplified key-schedule. As Deoyxs-II is a two-pass cipher, it needs twice as long as Deoxys-I for processing the plaintext. The time needed for processing the AD stays the same.

The remaining ciphers form the last group. They are very similar in terms of their runtime; only Ascon and Colm are slightly slower.

SUPERCOP [8] reports results in terms of cycles/byte processed on a Cortex-A9 (*h2tegra*) processor for the same algorithms and sizes of test vectors (64 kB, 3072 kB), which are between 2.45 and 7.42 times better than ours. On the Cortex-A9+NEON (*odroid*) platform they are between 3.14 and 14.62 times better. Upon closer inspection of the SUPERCOP results, it became clear that these are results of 64-bit optimized or, in the case of *odroid*, NEON optimized codes.

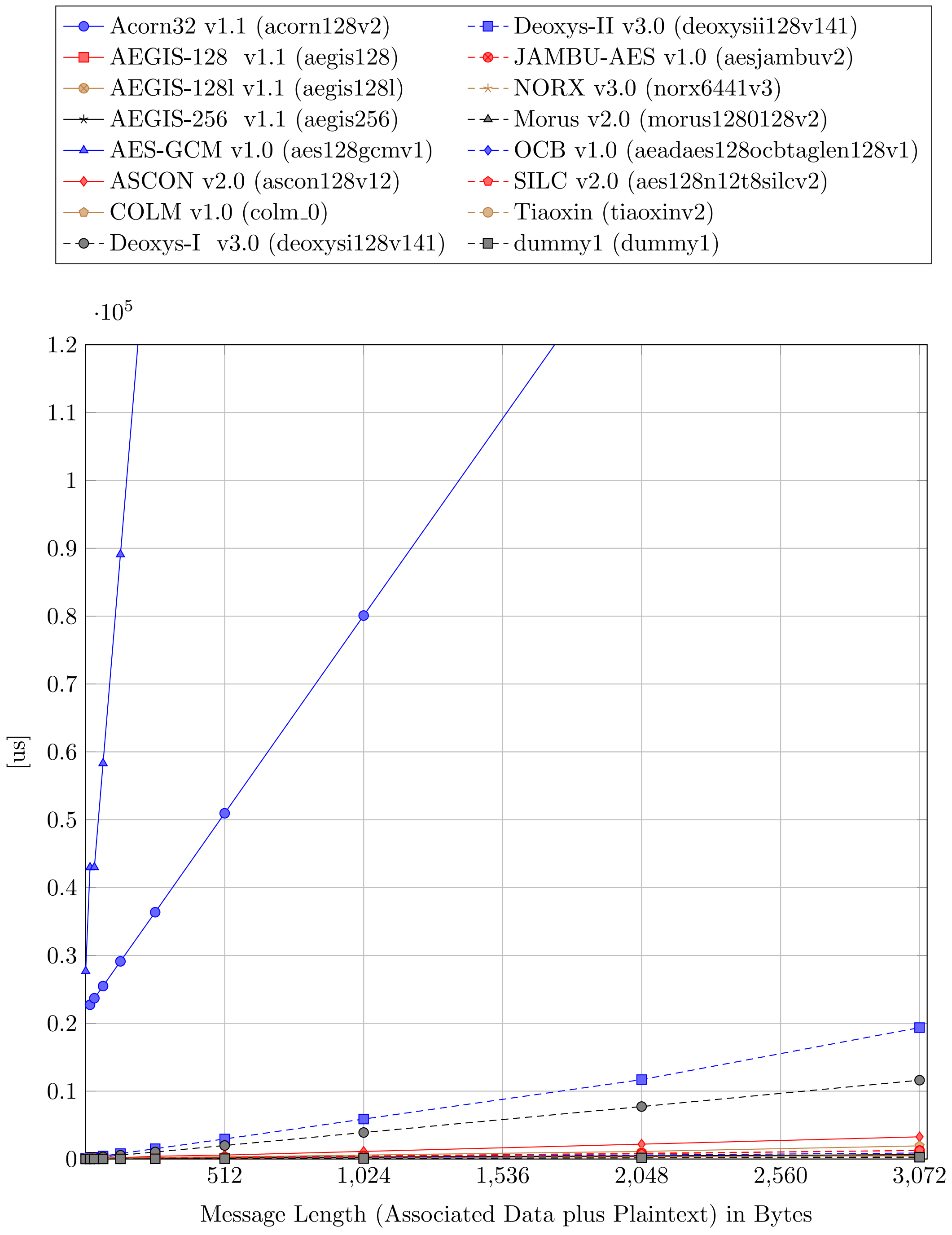


Figure 6.4: Latency of AEAD function called in software: Encryption is performed in software. Time is measured on the PS including all function calls.

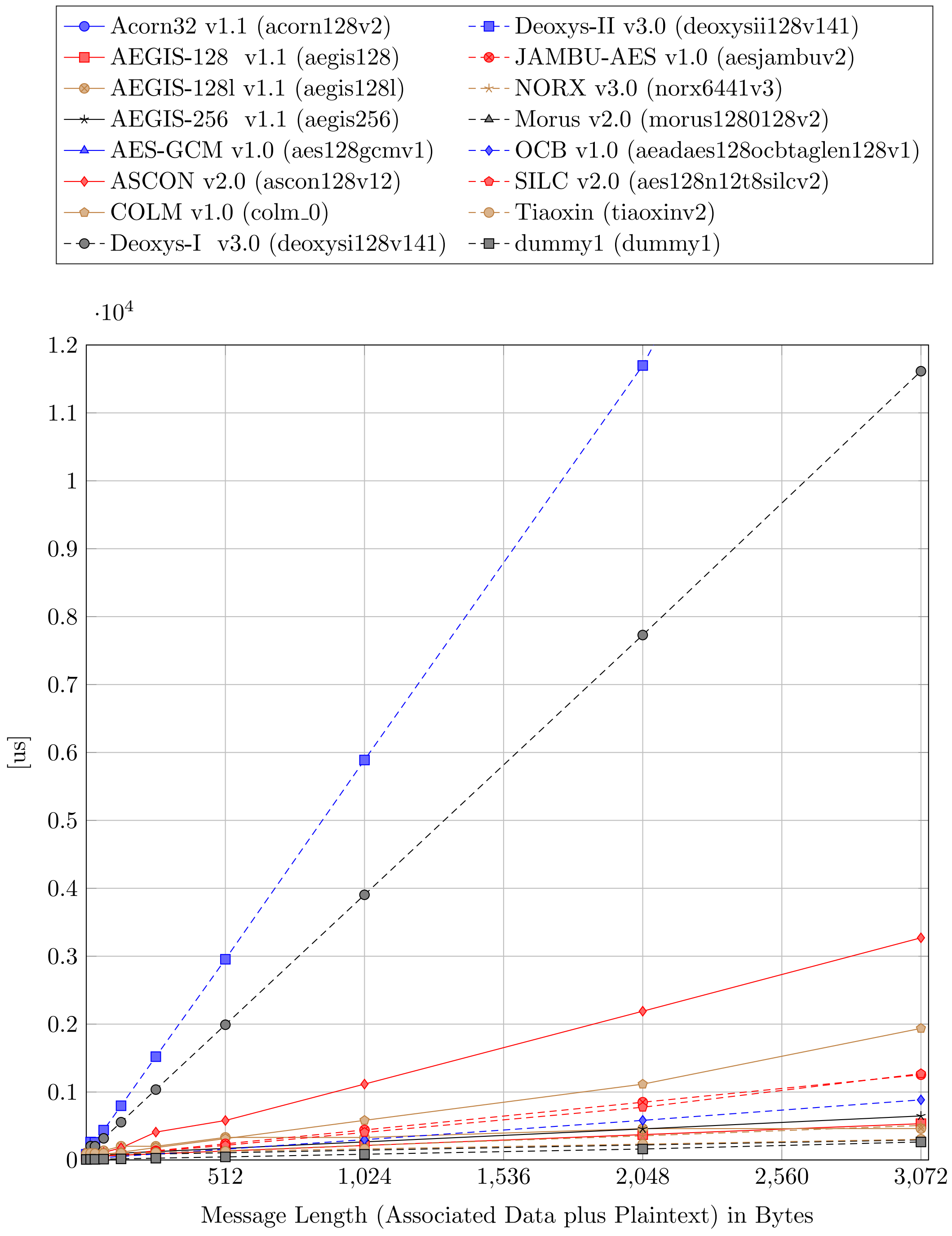


Figure 6.5: Latency of AEAD function called in software (zoomed in): Encryption is performed in software. Time is measured on the PS including all function calls.

Python Driver Figure 6.6 shows the time needed by the corresponding hardware implementation if called from PYNQ's python framework. The runtime of the hardware implementation is more or less independent of the amount of processed data and only depends on the used python driver.

This is because configuring the DMA controllers in python takes more time than processing the data in hardware. The difference between the old legacy driver used in [77] and the current DMA driver used in [78] is quite significant. The reason for this are more checks in the current driver. It therefore is more robust, but also slower [57].

Nevertheless, the hardware implementation combined legacy driver outperforms most reference implementations when more than 128 bytes are processed. However, as the overhead is still the dominant factor, the python framework is only suitable for testing the PL and the different CipherCores for functionality, but not suitable for a high-performance hardware accelerator or benchmarking.

C Driver Therefore, at the costs of ease of implementation, we use a C driver for further benchmarking. Figure 6.7 shows the time needed for the same tasks as in Figure 6.6 but with a driver written in C. As can be seen, the time increases linearly with a constant offset.

To further evaluate the different implementations, we used the two timers in the PL: Timer 1 measures the time needed for authenticated encryption of a message in the AEAD-IP, from fetching data out of the PDI-FIFO to, and including, sending them back to the DO-FIFO. As it is decoupled from the PS by the FIFOs, there is no influence from the DMA controllers. The time measured confirms the execution time precalculated by the authors. The results are shown in Figure 6.8. Again, the processing time increases linearly with the amount of processed data, as expected.

Comparing the values from Figure 6.7 and 6.8 shows that the communication overhead is constant and about 7.2 us. Thus, the CAESAR Hardware API is also suitable for a generic hardware accelerator.

Timer 0 measures the time from DMA controller to DMA controller. Thus, it includes the communication overhead from the perspective of the PL. As the difference between the two timers is neglectable, only the values of of Timer 1 are shown in the diagram. The difference is the expected delay of a few clock cycles caused by the FIFOs. The two values will only differ significantly, if there is a backpreassure from either side. As this is not the case, our setup is suitable for benchmarking the cipher implementations.

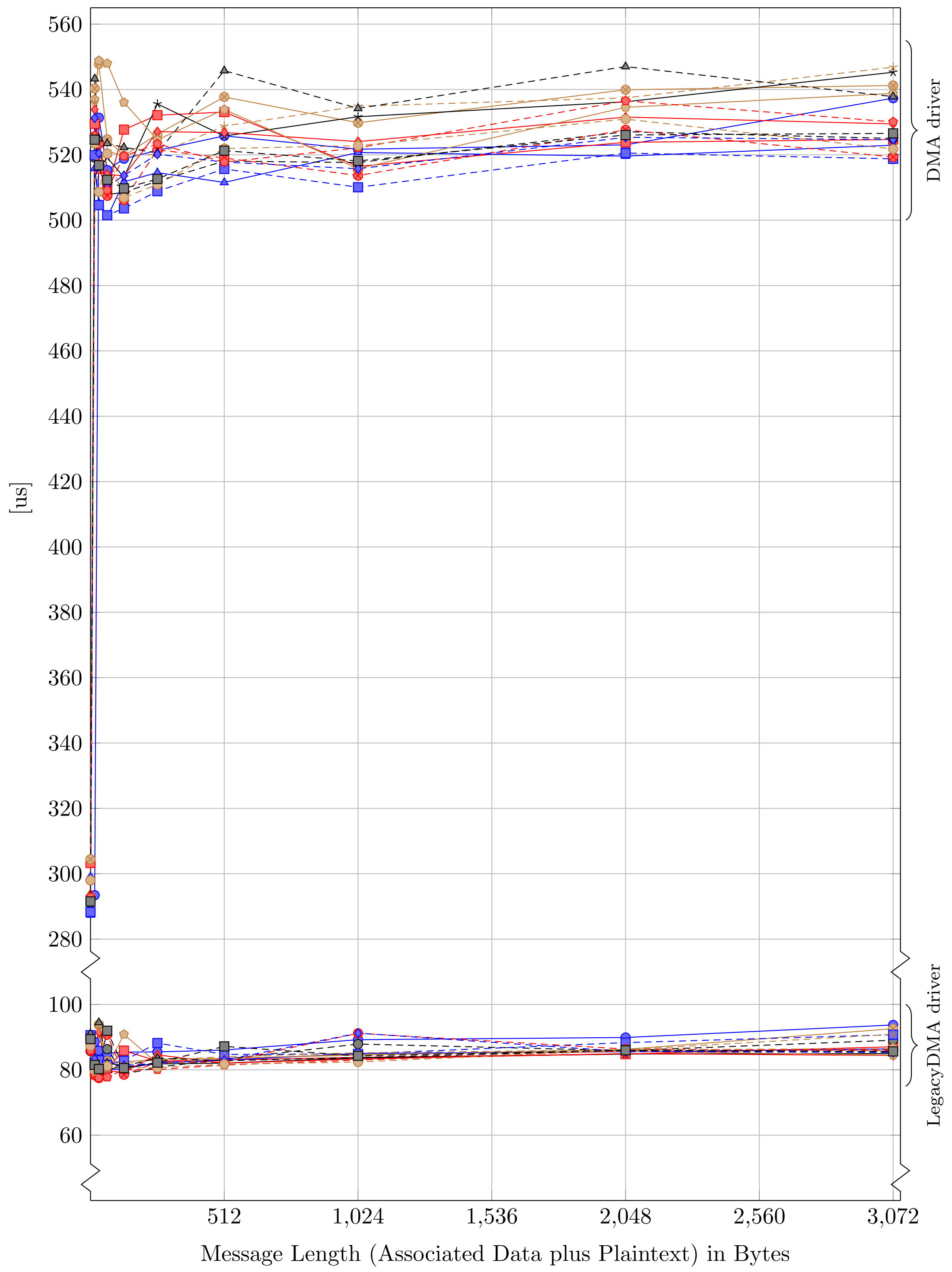


Figure 6.6: Latency of AEAD hardware with FIFOs, DMA, and function calls in in Python: Encryption is performed in hardware at 100 MHz. Time is measured on the PS including the time needed for DMA transfers and function calls.

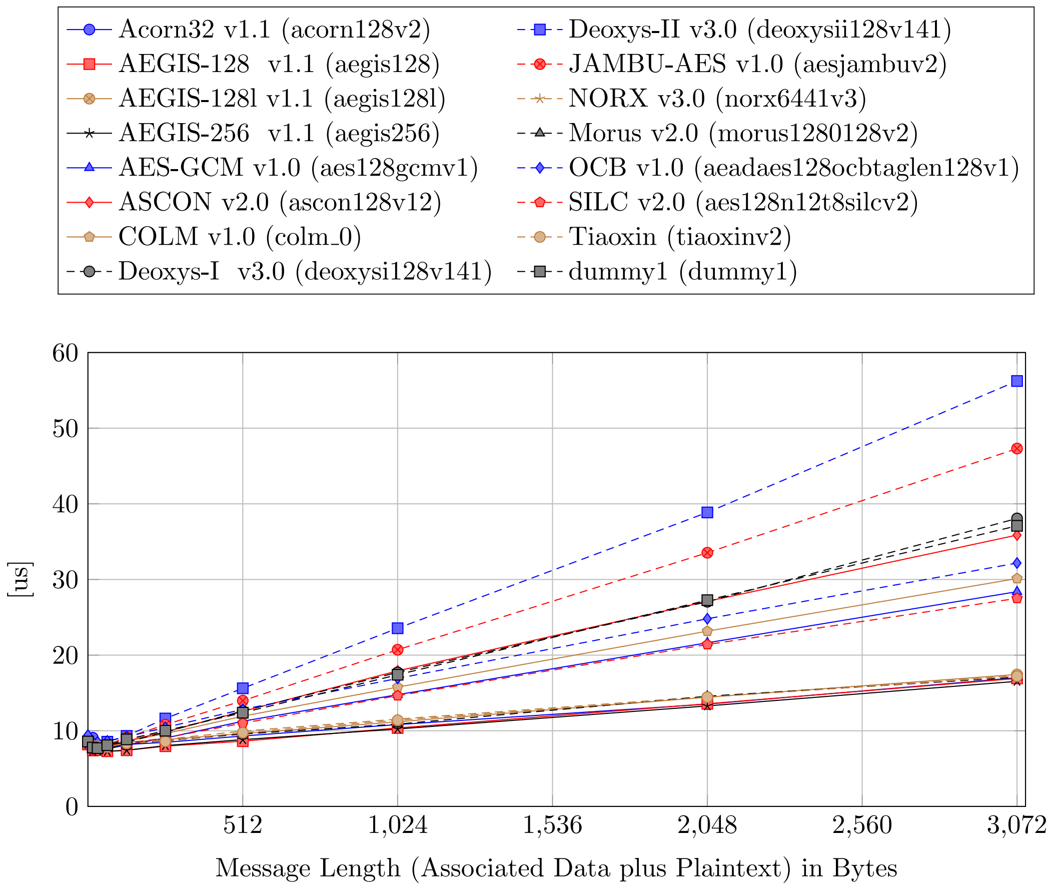


Figure 6.7: Latency of AEAD hardware with FIFOs, DMA, and function calls in in C: Encryption is performed in hardware at 100 MHz. Time is measured on the PS including the time needed for DMA transfers and function calls.

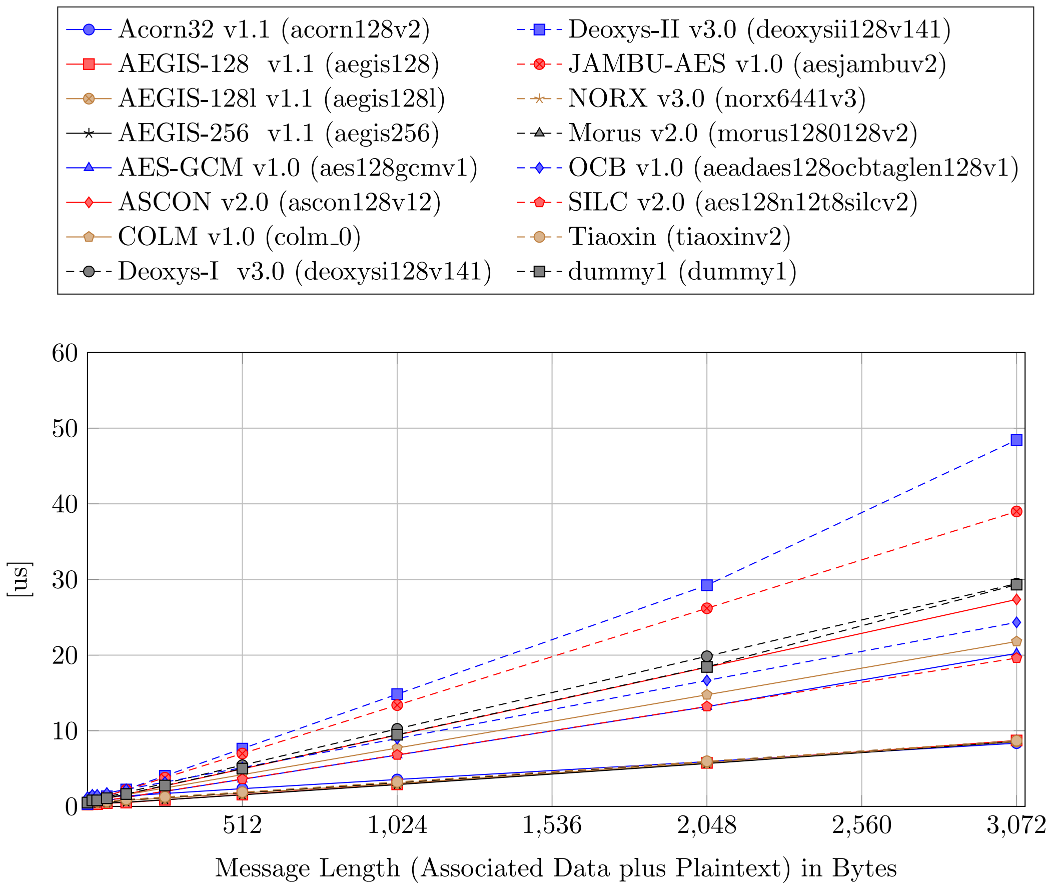


Figure 6.8: Latency of AEAD hardware: Encryption is performed in hardware at 100 MHz. Time is measured on the PL taking only the latency in the AEAD-IP into account (Timer 1).

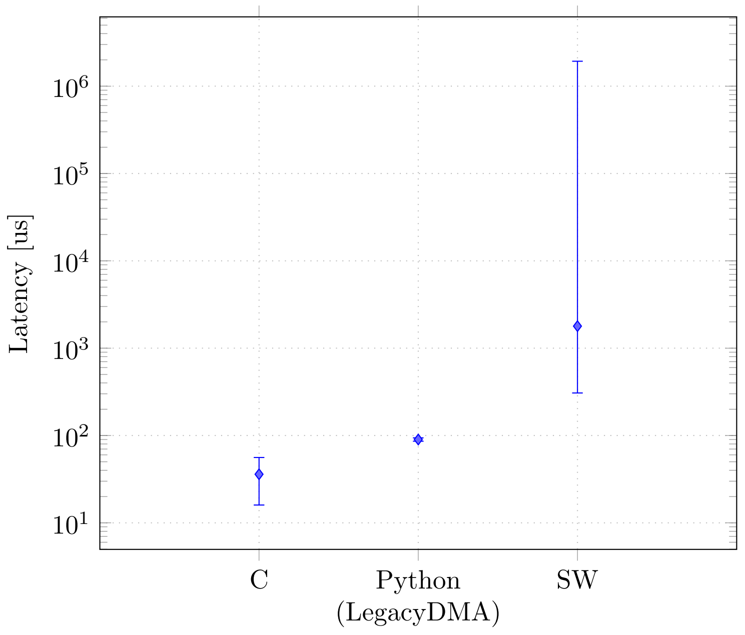


Figure 6.9: Latency for encrypting 1 kB of Associated Data and 2 kB of plaintext in hardware with the C DMA driver and the python LatencyDMA driver; and in software (SW)

Comparison Figure 6.9 shows the latency for encrypting 1 kB of Associated Data and 2 kB of plaintext with different configurations: As previously shown, using a hardware implementation together with the C-DMA driver results in the lowest latency. Using PYNQ's LatencyDMA Python driver still provides a notable speed-up compared to the software implementations. However, there is no variance between the ciphers because the Python calls are the dominant factor. The software (reference) implementation is by far the slowest. The asymmetric skew is based on the outliers of the AES-GCM, Acron, and the Deoxys family.

6.3.3 Power Consumption

Figure 6.10 shows the measured maximum and average power consumption of the whole PYNQ board (system power) when the Crypto-IP is performing authenticated encryptions at 200 MHz and the ARM core is asleep. Only the core and the measurement clocks are on. The values are based on at least 10 measurement cycles each containing 10 authenticated encryption processes. The ciphers can be clearly distinguished according to their power consumption.

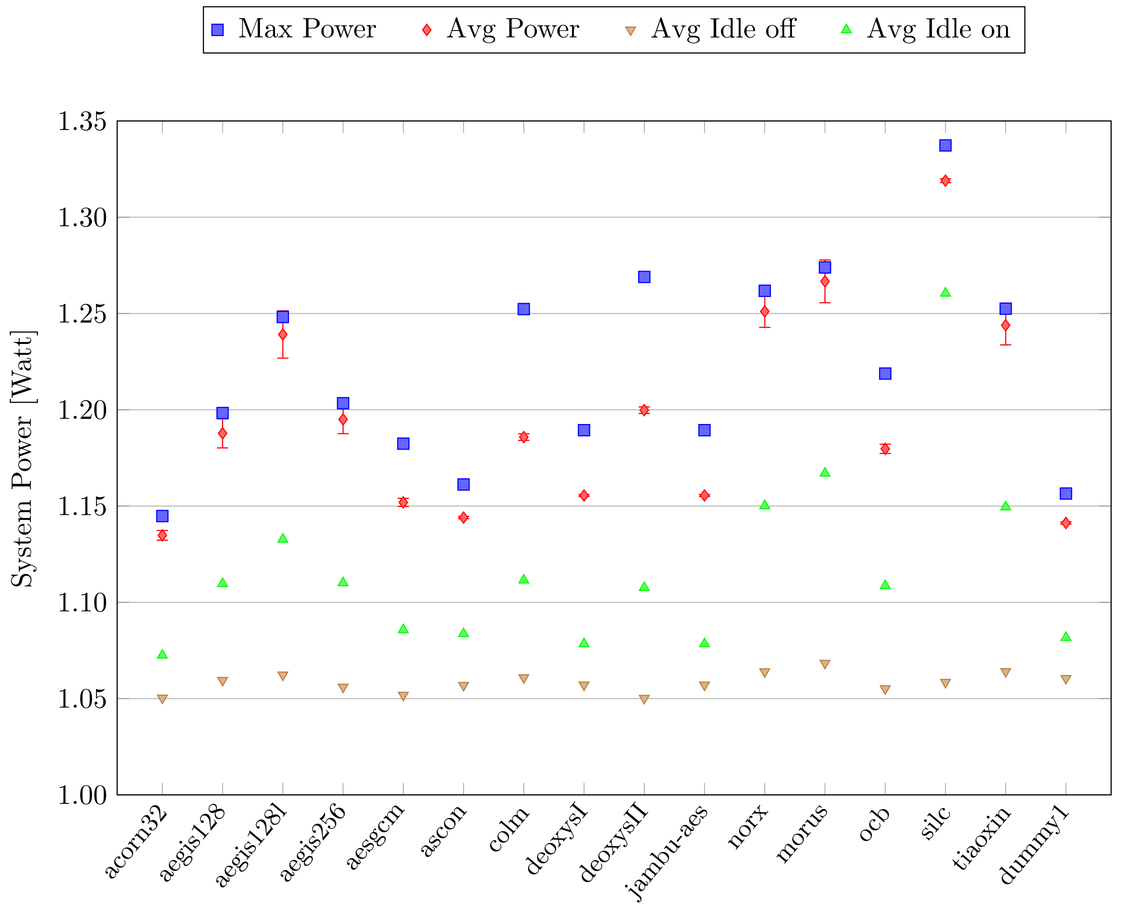


Figure 6.10: Measured maximum and average system power during authenticated encryptions at 200 MHz and reference measurement with core clock turned off and on

Idle Power Consumption In the figure, the average power consumption when the Crypto-IP is idle is displayed in green (▲) and the average power consumption when the core clock ia gated is shown in brown (▼). It can be seen, that with a disabled core clock, the power consumption of the system is roughly constant and only slightly depends on the different bus sizes. This means that our presented measurement setup is reliable and the differences in the other measurements is based on the individual cipher implementations. If the core clock is turned on, we expected a correlation between the power consumption and the area. When comparing Figure 6.10 with Table 5.2, this seems true for most of the ciphers. However, there is an exception for Silc, which has a notably higher idle power consumption than the rest. This effect can also be seen in simulation, as the 128 bit `bdo` line is toggling regardless whether there are valid data on it or not. Presumably the underlying round function is not stopped. While this seems like a good optimization in terms of resources, it is bad for power and energy consumption. Figure 6.11 visualizes this effect. It shows the calculated power dissipation $P_{\text{core}} = P_{\text{idle, on}} - P_{\text{idle, off}}$ of the individual ciphers in relation to the number of its FFs.

Power Consumption during an Authenticated Encryption The average system power and the maximum system power are displayed in red and blue in Figure 6.10. This is the power the systems draws during an authenticated encryption and it is a metric for the efficiency of a cipher and its implementation. The error bars show the Standard Errors of Means (SEMs):

$$\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}, \tag{6.5}$$

where σ is the standard deviation, $\bar{x}$ is the sample's mean and n is the sample size. It measures the sample-to-sample variability of the sample means and predicts the accuracy of the sample means, which is the average power consumption. As described in Section 6.2.2, the XADC has a sample rate of 1 MSPS, which means that if the Crypto-IP runs at 200 MHz, every 200 clock cycles a measurement can be taken. This seemed enough for the original publication in [78], as the examined implementations were slow enough: In average 10 to 40 measurements were taken. Unfortunately, for the highspeed implementations of the Aegis family only up to four measurements and for Norx, Morus and Tiaoxin only one measurement per run were possible. Therefore, their SEMs are higher, but still small enough to allow for a meaningful comparison.

Figure 6.12 shows the raw average system power consumption at different core frequencies. While generally there is an increasing trend, the overall power consumption has its minimum at 150 MHz. This is based on the fact, that the complete system including the clock generators with the external 125 MHz clock seems to work most efficient at 150 MHz. To show the actual power needed by the AEAD-Core itself, we subtracted our idle reference measurement with the core clock off (brown ▼) from the average power consumption. Thereby, the power consumption of the measurement setup, the buses, the PS, and the clock generators cancels out. Figure 6.13 shows this calculated core power consumption in relation to the frequencies. Now, the expected increasing trend is visible.

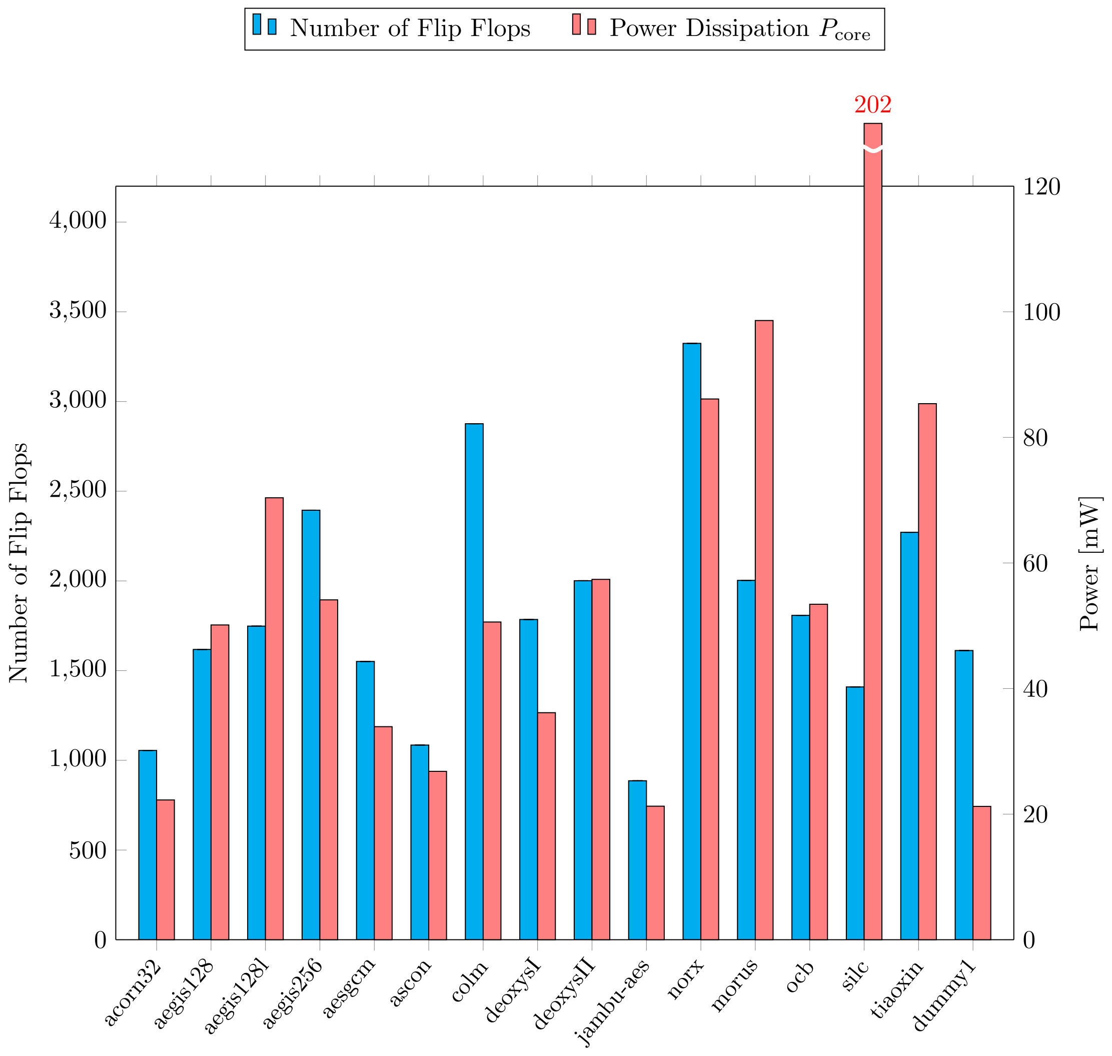


Figure 6.11: Flip-Flop and core power relation

It can be seen, that Silc again has the highest power consumption and the steepest slope. This is expected as its power consumption is dominated by (unneeded) switching activities. Next, Mours, Norx, Tiaoxin and Aegis128l have a notable high power consumption. The rest of the Aegis family, COLM, OCB and the Deoxys family perform better, but still have a higher power consumption than AES-GCM. Only Jambu-AES, Ascon, Acorn (and the dummy cipher) perform better than AES-GCM – the gauge of the CAESAR competition

6.3.4 Energy Consumption

Especially for lightweight applications not only the power consumption is crucial, but also the total energy needed. It depends on the power consumption and the runtime. Thus, it is a measure for the efficiency of a cipher implementation. As Timer 1 measures the time t for performing an authenticated encryption with 4096 bits of plaintext and 1024 bits of AD, the energy E can be calculated as:

$$E = P_{\text{avg}} \cdot t \tag{6.6}$$

To be comparable with other works, we normalized the energy and calculate the energy per bit ratio:

$$E_{\text{bit}} = \frac{E}{5120} = \frac{P_{\text{avg}} \cdot t}{5120} \tag{6.7}$$

Figure 6.14 shows the energy per bit E_{bit} consumed at different frequencies. As can be seen, the complete system works more efficient at higher frequencies, as the runtime decreases. This was expected: First, like shown in Table 6.1, the utilization of the PL is roughly the same for a target frequency of 100 MHz and 200 MHz. Second, as shown in Figure 6.13, the power consumption does not double for a doubling of the frequency.

Comparing the individual ciphers, it is notable that Deoxys-II has a very high energy per bit ratio compared to the other ciphers. This was expected, as Deoxys-II first authenticates the complete message (including associated data and plaintext) and uses the generated tag as the key for the encryption of the plaintext. (It was the only remaining two-pass algorithm of the contest.) Thus, Deoxys-II has a very long runtime, compared to the other ciphers. Next, Jambu's Energy per bit ratio is also very high. This is based on the fact, that the underlying cryptographic primitive is a complete AES core, but only half of the AES block-size can be used to process the input. This trend can also be seen in Figure 6.13. Concerning the CAESAR finalists, the aegis family performs best in terms of energy efficiency. Deoxys-II, Colm, and Ascon have a higher energy per bit ratio than AES-GCM. Especially for a lightweight winner, this is not ideal. Although the power consumption for Ascon is very low, the high energy per bit ratio results from the long runtime of Ascon. The best energy per bit ratio has Morus, followed by Tiaoxin and Norx. However, none of the latter candidates made it into the final CAESAR portfolio.

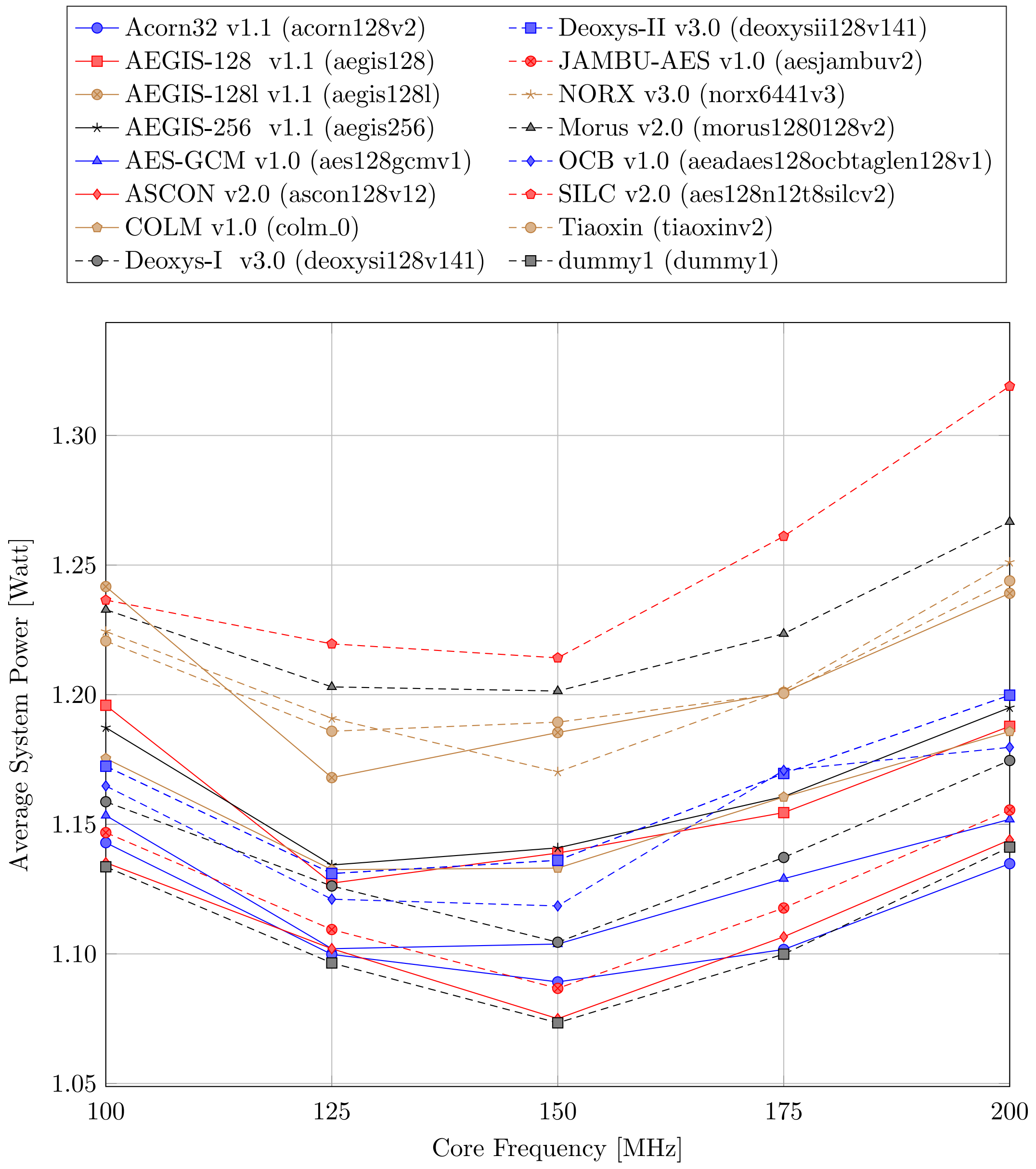


Figure 6.12: Measured average system power consumption

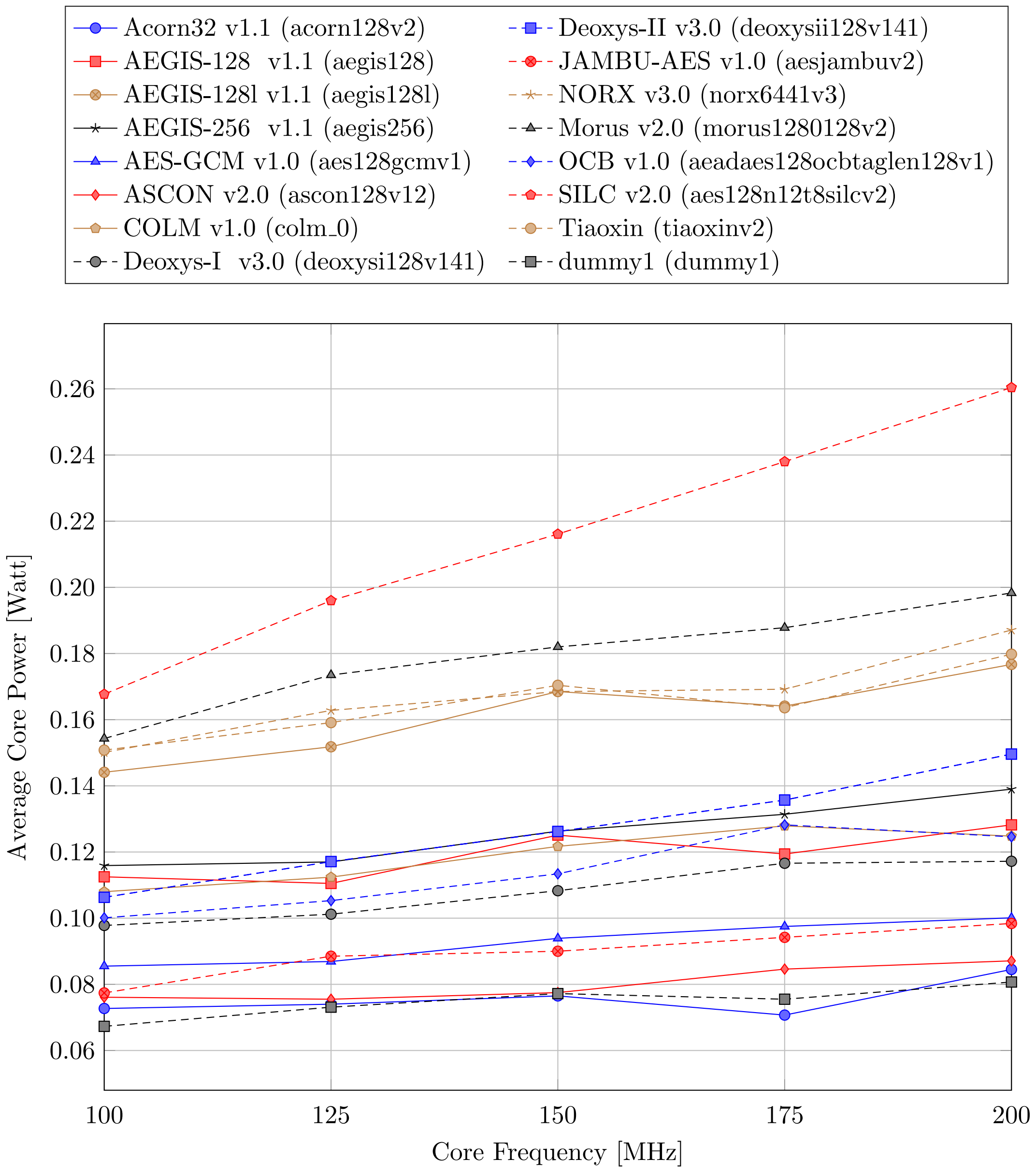


Figure 6.13: Calculated core power consumption

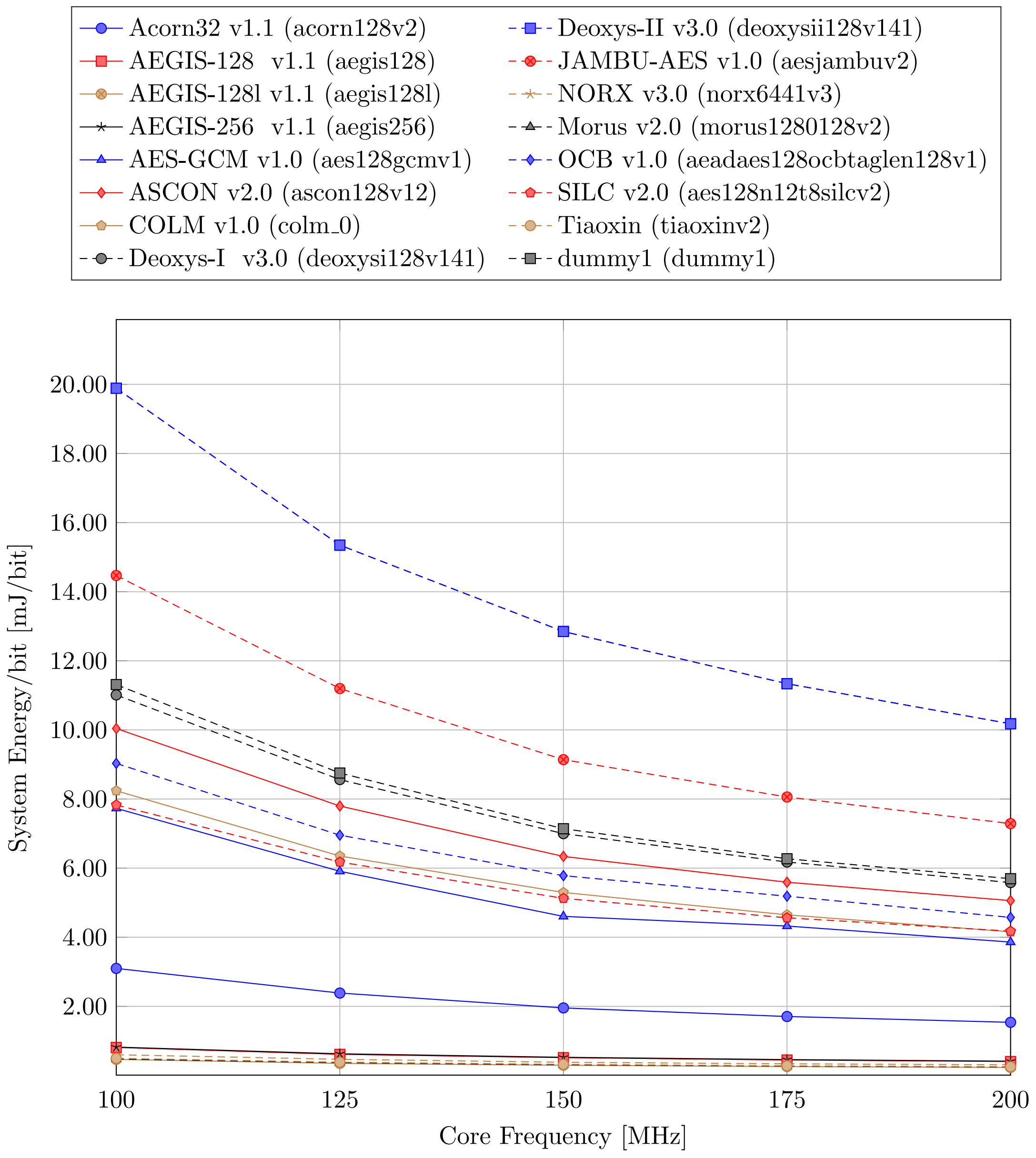


Figure 6.14: Total energy per bit

6.4 Consequences for Ciphers

The final portfolio for the CAESAR competition consists of Ascon and Acorn as primary and secondary choice for lightweight applications; Aegis128 and OCB without a preference for high-performance; and Deoxys-II and COLM as primary and secondary choice for defense in depth applications. Unfortunately, there is no explanation of the CAESAR committee on how and why they choose these candidates in 2019. The results of this chapter were prepublished in 2018. In the following, we will analyze, whether the decisions made, are backed up by our data from 2018.

Table 6.2 summarizes the synthesis results, the runtime (time t), the estimated power (Est. Power) which Vivado reports under the assumption that the PS uses 85% of the power, the measured power (Act. Power) with the PS asleep and only the core and measurement clocks running, and the calculated energy per bit E_{bit} for an authenticated encryption of 1024 bytes of AD and 4096 bytes of plaintext.

The choice for the lightweight applications seems reasonable regarding the resources. The only other candidate that needs a similar amount of resources is Jambu-AES, which performs much worse in terms of runtime. However, from a pure hardware perspective, the ranking in primary and secondary choice is not justifiable, as Acorn performs better than Ascon in all categories; especially regarding power and runtime, and thus also energy, Acorn seems the better choice for a lightweight candidate. Further, they are based on two totally different primitives: Acorn is based on a stream cipher construction, while Ascon is based on a sponge construction. However, considering the fact that the used hardware implementation of Acorn stalls for empty inputs and –more severe– has problems for inputs larger than 2048 bytes, should disqualify Acorn to be a finalist. Even though these problems are most certainly solvable without changing the results very much, it still contradicts the committee's selection criteria. Thus, considering that its author is part of the committee, Acorn becoming a finalists left a stale aftertaste.

The choice for the high-performance applications is not clearly comprehensible either. Aegis-128 performs very well in terms of time, power, and energy, while still being reasonable in terms of resources. However, the other family members perform similarly. On the contrary, OCB is a magnitude slower, and thus not as energy efficient as the Aegis family. Its advantage is the smaller amount of needed resources, but for high-performance applications, the runtime should be key. Morus was dropped because there was no running implementation. However, this does not hold true for Norx and Tiaoxin, which also would have been very promising candidates. It remains unclear, why they were dropped, as at least Norx is sponge-based and thus, very well understood.

The focus for the defense in depth choice was the nonce-misuse mode of Deoxys-II and Colm, which comes at the cost of resources and time.

Table 6.2: Summary of the results for the PYNQ-Overlay including crypto cores optimized for 200 MHz and the corresponding results for processing 5120 bytes

	Resources			Power [W]		Time	Energy/bit
Algorithm	**LUT**	**Regs.**	**BRAM**	**Est.**	**Act.**	**[us]**	**[mJ/bit]**
Acorn32*	7 328	10 328	18.5	1.624	1.135	6.94	1.538
Aegis128	16 695	13 126	66.5	2.127	1.188	1.74	0.404
Aegis128l	16 377	14 746	126.5	1.819	1.239	0.96	0.232
Aegis256	18 574	13 902	66.5	1.737	1.195	1.77	0.413
AES GCM	9 168	10 824	22.5	1.671	1.152	17.16	3.861
Ascon	7 713	10 358	18.5	1.628	1.144	22.65	5.061
Colm	13 844	12 149	22.5	1.798	1.186	17.95	4.157
Deoxys-I	9 454	11 058	18.5	1.74	1.175	24.33	5.582
Deoxys-II	10 681	11 274	18.5	1.746	1.200	43.43	10.177
JAMBU-AES	7 713	10 159	22.5	1.642	1.156	32.30	7.290
NORX	13 788	16 322	126.5	3.019	1.251	1.26	0.308
Morus	16 595	15 542	128.0	1.761	1.267	0.99	0.245
OCB†	10 424	11 079	22.5	1.687	1.180	19.85	4.574
SILC	9 368	10 682	18.5	1.757	1.319	16.21	4.176
Tiaoxin	16 169	15 269	126.5	1.969	1.244	1.04	0.253
dummy1	7 343	10 885	18.5	1.632	1.141	25.55	5.695

* stalls on empty input
† including 4 latches

7 Case Studies

In the following, we want to go one step beyond. First, we present a mixed hardware software codesign setup, as an alternative to the API. We show, that this technique allows to reduce the hardware overhead, as one communication channel can be saved. Additionally, we point out, that on loosely coupled systems like SoCs, the possible savings in hardware are limited to the basic building blocks of the cipher, as otherwise the communication overhead becomes the dominated factor. Next, we demonstrate the advantages of the internal interface of the API. We embed the CipherCores of two selected examples into the communication structure of a Network on a Chip (NoC). Finally, we combine ciphers equipped with the CAESAR and LWC API with a Physical Unclonable Function (PUF). There, we demonstrate the flexibility of having different ciphers or implementations of them in IoT applications.

7.1 Enhancing Authenticated Encryption with Hardware Software Codesign Techniques

The following section is a summary of "Michael Tempelmeier, Maximilian Werner, and Georg Sigl. Using hardware software codesign for optimised implementations of high-speed and defence in depth CAESAR finalists. In 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 228–237, 2019".* In this work, we present five optimized implementations on a Xilinx-Zynq7200 SoC for the high-speed and defense in depth finalists of the CAESAR competition. We eliminated the standard interfaces used during the competition. Through optimized interfaces between hardware and software we reduced the used programmable logic, while maintaining high performance.

7.1.1 Setup

Figure 7.1 shows the setup. In contrast to the CAESAR-API, there are no dedicated channels for SDI, PDI, and DO, but only one data channel per direction and one configuration channel. As one Xilinx's DMA controller can serve memory-mapped-to-stream (MM2S) reads and stream-to-memory-mapped (S2MM) writes, we only need one controller for our

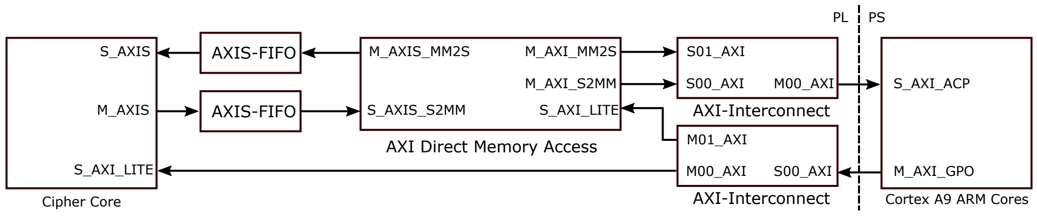


Figure 7.1: Hardware software co-design setup ©2019 IEEE [80]

two communication channels. The DMA controller acts as master (M_AXI) for the communication with the ARM Cores (memory-mapped side). The read and write channels are combined in an AXI-interconnect module and connected to the Accelerator Coherency Port (ACP), which is the fastest communication port of the ARM Cortex. The Cipher Core receives data on its slave port (S_AXIS) from the DMA controller and sends data back on its master port (M_AXIS). An AXIL interconnect is used to configure the DMA controller and the Cipher Core. It is controlled by the general purpose output (GPO) of the ARM Cortex.

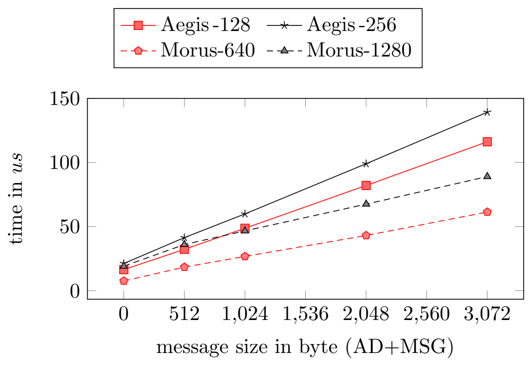
Table 7.1: Resource utilization of the setup depicted in Figure 7.1

Module	**LUTs**	**FFs**	**BRAM**
DMA	1 666	2 281	3
Interconnect 0	519	693	0
Interconnect 1	561	654	0
FIFO 0	49	126	1.5
FIFO 1	50	126	1.5
$\sum$	2 845	3 880	6

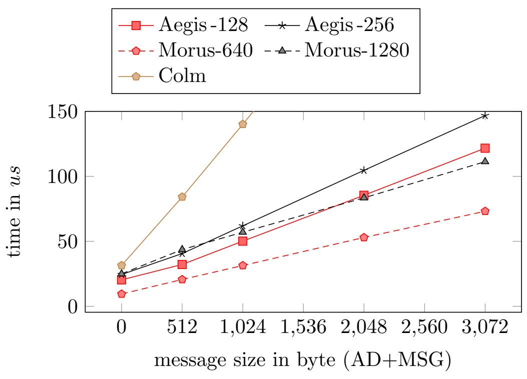
Table 7.1 shows the resource utilization needed for the interconnects. As can be seen, only $2/3$ of the resources of the minimal 32 bit setup, as presented in Table 5.4 in Section 5.3 are needed.

For the software side we used the official *software* API. By doing so, the used implementation (software vs. hardware software co-design) is abstracted for the user and either implementation can be plugged in. Due to the loosely coupled PS-PL-communication, the communication overhead plays an important role. To minimize the communication overhead, we decided to process the basic building blocks of a cipher either in hardware or in software, and not to split off individual subfunctions. Thus, we transfer data only once and not multiple times between the PL and PS.

In particular, the basic building blocks of the ciphers are the *round function*, the *tag generation and verification* and the *padding*. The round function is the computational most expensive part of a cipher and therefore assigned to the hardware. Additionally, parallelisms can be exploited much better in hardware. While the padding rules of the ciphers are very



(a) Bare-metal results, running at 650 MHz



(b) Linux results, running at 650 MHz

Figure 7.2: ARM CORTEX-A9 software performance at 650 MHz

simple, their implementations in hardware are quite expensive, which is why this part is more suitable for software than for hardware. For the tag generation and verification it depends on the cipher: the tag can be a part of the state after the final round and comes for free; it can involve another pass of the block cipher, or there might be the need for some dedicated function. Therefore, we implement it depending on the cipher either in hardware or in software.

7.1.2 Results

We implemented two high-speed families and one defense in depth candidate of CAESAR using hardware software co-design techniques to minimize the overhead of the hardware API and to optimize the overall performance on a Xilinx Zynq-7020 SoC. As a reference, we use the already presented software and hardware implementations of Chapter 6. Figure 7.2 shows the runtime performance of the Aegis and Morus family as a bare-metal and linux application on the PS of the PYNQ Z1 board. The performance of Colm is only displayed for the linux application, as it relies on software libraries that are not available for ARM Cortex A9 bare-metal applications. Figure 7.3 shows the performance of the available, corresponding hardware implementations. The numbers are the same as in Figure 6.5.

Table 7.2 lists the resources needed for our implementations. They are all designed to support a frequency of at least 150 Mhz. The hardware software co-design implementation of the Ageis and Morus family needs only about 50 % of the slices of a pure hardware implementation. The amount of FFs is slightly increased, while there are drastic savings in the number of LUTs.

Due to the high degree of parallelism, the hardware software co-design implementation of Colm has a significant higher area footprint compared to the one presented in Section 5.3.

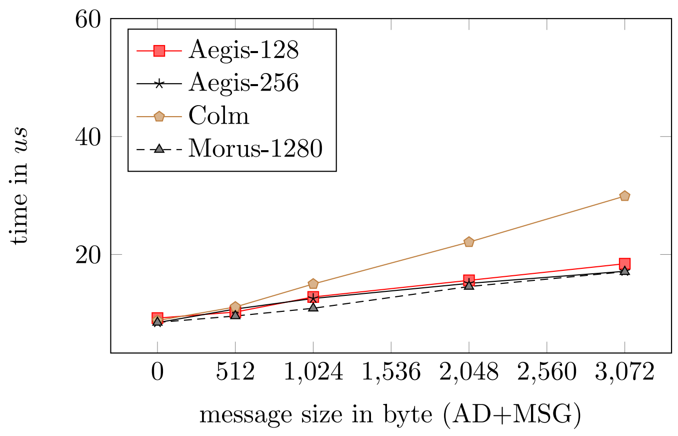


Figure 7.3: Latency of hardware implementations including the CAESAR-Hardware-API, running at 100 MHz on a Zynq-7200; DMA calls are performed in C.
Note: The plot for Morus was not presented in [80] because there was no published working version of Morus at that time.

Table 7.2: Resource utilization of the CipherCores

Implementation	**HW/SW**		**Hardware**	
	LUT	**FF**	**LUT**	**FF**
Aegis-128	5 199	2 656	9 048	1 618
Aegis-256	6 247	3 044	10 931	2 394
Morus-640	1 995	2 016	n.a.	n.a.
Morus-1280	3 558	3 693	7 473	2 003
Colm	46 760	19 715	7 725	2 876

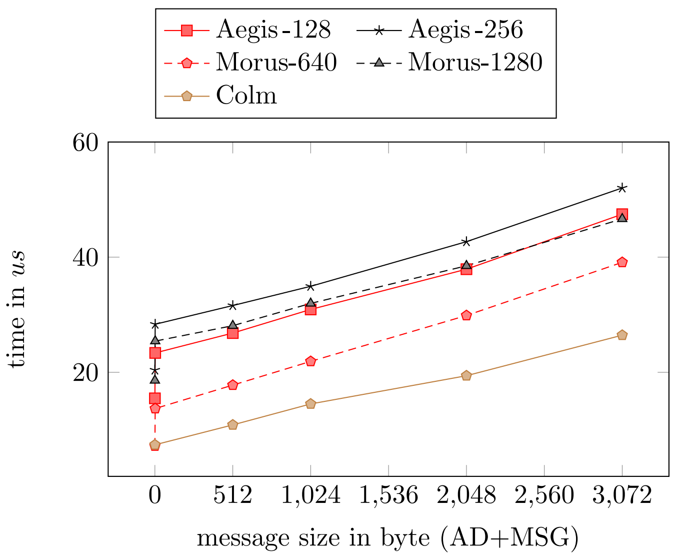


Figure 7.4: Latency of HW/SW implementations: Hardware is running at 150 MHz, software at 650 MHz.

Figure 7.4 summarizes the times needed by all our implementations for an authenticated encryption at different message sizes. Compared to pure software implementations, this is a speed-up of factor 15 for Colm, a factor of up to 1.9 for Morus, and a factor of up to 2.7 for Aegis.

7.2 Securing Network-on-Chip Applications

The following section is a summary of "Siavoosh Payandeh Azad, Michael Tempelmeier, Gert Jervan, and Johanna Sepúlveda. CAESAR-MPSoC: Dynamic and efficient MPSoC security zones. In 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 477–482, 2019".*

Dynamic security zones in Multiprocessor System on Chip (MPSoC) have been used to isolate sensitive applications from possible attackers. These physical wrappers are usually configured through programmable hardware firewalls. Previous works have shown the efficiency of this security mechanism against a wide variety of attacks. However, the security zone configuration is performed in an unprotected way, exposing the system to attacks caused by rogue firewall update. In this work we propose CAESAR-MPSoC, an enhanced MPSoC able to ensure the protected configuration of the firewalls through encrypted and authenticated reconfiguration packets. To this end, we present two contributions. First, we integrate two CAESAR hardware IP cores, ASCON and AEGIS, into a MPSoC. Finally,

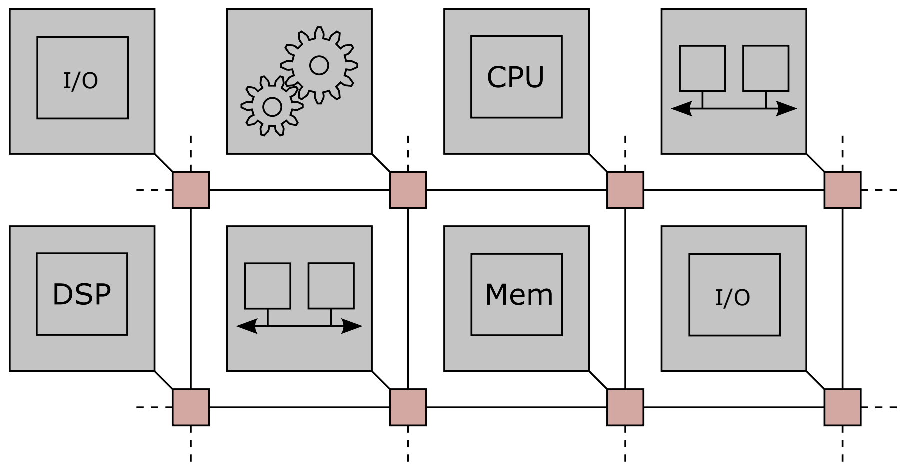


Figure 7.5: MPSoC: The individual tiles are connected to a 2D mesh NoC.

we evaluate the area and cost of CAESAR-MPSoC. The results show that our solution is feasible and effective to allow the protected and efficient security zone configuration.

7.2.1 Multiprocessor Systems on Chips

MPSoCs are an extension of SoCs: A typical FPGA-SoC consists of a PL and a PS. Figure 7.5 shows the structure of a typical MPSoC. It consists of a number of heterogeneous PSs, on-chip memories, Inputs/Outputs (IOs), and dedicated hardware accelerators. Typically, there is no dedicated PL. User specific logic must be integrated during manufacturing. These individual components are called *tiles* connected by a NoC. There are different typologies for NoCs. The most prominent ones are meshes, rings, tori, and octagons. However, more complex structures like stacked NoCs are possible [32]. Each tile has a Network Interface (NI) to connect to a router of the NoC.

7.2.2 Attacks on Networks on Chips

User specific tiles, or user specific software running on one of the tiles, pose a serious threat to the whole MPSoC. Malicious code can allow an adversary to take over the complete system. Most works assume the NoC to be trusted and instantiate firewalls between the routers and tiles [72, 73, 14, 28].

In more sophisticated NoCs, the routers (especially their routing schemes) can be reconfigured to improve overall performance. This introduces a new attack vector. Miss-configured routers can be exploited in various ways: simplest the routing scheme can be deactivated.

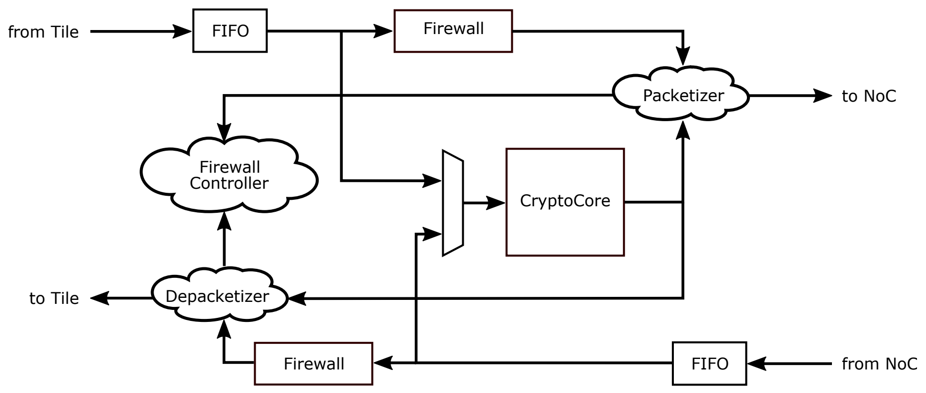


Figure 7.6: CAESAR equipped Network Interface

As a consequence the complete system stalls. Only the works of [2], [11], [74] consider the NoC as a possible malicious component. However, the proposed security mechanisms of [2] and [11] are based on XORing the plain-text with a static key throughout the lifetime of the NoC and thus must considered as beeing broken by design. In [74] an encrypted tunnel-based communication was introduced, but without authentication. In contrast, we propose a CAESAR-based security concept for MPSoCs that allows for both: authenticity and secrecy. Furthermore, we use the internal interface of the CAESAR-API. Thus, the used cryptographic algorithm and implementation is exchangeable. Finally, we present a proof-of-concept analysis with the CAESAR finalists aegis and ascon and show the applicability for high-speed and lightweight MPSoCs.

7.2.3 CAESAR Network Interface Reconfiguration

Figure 7.6 depicts the block diagram of the CAESAR-NI. We only use the CryptoCore without the API. The CryptoCore is shared between the packetizer and depacketizer units. This means that the same NI cannot be receiving and sending reconfiguration packets at the same time. As reconfiguration occurs rarely, this is not an issue. Upon receiving a reconfiguration packet, the NI decrypts and authenticates the reconfiguration messages and updates its firewalls.

In this work we considered a 4×4 wormhole switching network based on Bonfire-framework [67]. Each router's FIFO has a 4 flit deep buffer and the network interface has a 32 flit deep buffer in each direction. The network has been partitioned into three security zones, where each zone is managed by a single security manager. In each zone, a master node broadcasts the reconfiguration package to its zone. The size of the packet payload is constrained to 128 bits which reconfigures only a single firewall entry.

Table 7.3: Area overhead evaluation

	Combinatorial (μm^2)	**Sequential** (μm^2)	**Total** (μm^2)	**Overhead** (%)
Baseline	$27\,316.8$	$29\,185.2$	$56\,502.0$	–
CAESAR (AEGIS)	$176\,001.3$	$37\,389.2$	$213\,390.6$	277.6
CAESAR (ASCON)	$34\,134.5$	$33\,008.6$	$67\,143.2$	18

Table 7.4: Critical path overhead evaluation

	Critical Path Delay (ns)	**Overhead (%)**
Baseline	4.08	–
CAESAR (AEGIS)	4.81	17.8
CAESAR (ASCON)	4.04	–

7.2.4 Results

The experiments show that reconfiguration of the baseline[1] system takes 1840 clock cycles on average. However using AEGIS core, it takes 2064 cycles on average to reconfigure the network, considering all different traffic patterns. At the same time the average network reconfiguration time for ASCON is 2260 cycles. Which means that ASCON is on average 10 % slower than AEGIS.

Using high-speed Aegis, the overall latency for normal traffic and reconfiguration traffic increases by 41 % and 12 %, respectively. Using lightweight Ascon, the overall latency for normal traffic and reconfiguration traffic increases by 84 % and 23 %, respectively. For other traffic pattern, the difference between the two ciphers decreases[2]. Table 7.3 shows the area overhead for a 4×4 MPSoC. As expected, the overhead of the high-speed cipher Aegis is far grater than the lightweight cipher Ascon. However, it also shows that the security overhead – especially for Ascon – is still tolerable. Table 7.4 shows the impact of both ciphers on the critical path. Ascon has no impact. Aegis's impact is acceptable.

Using the internal interface of the CAESAR-API other CAESAR or LWC implementations can be tested in future works.

[1]This means no other traffic is involved.
[2]Exact numbers can be found in [64].

7.3 Authenticated Encryption based on Physical Unclonable Functions

The following section is a summary of "Christoph Frisch, Michael Tempelmeier, and Michael Pehl. PAG-IoT: A PUF and AEAD enabled trusted hardware gateway for IoT devices. In 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 500–505, 2020".*

Low-cost cryptographic security solutions are essential to protect resource-constrained devices or their respective data in the emerging IoT. By connecting an ever-increasing amount of "things" such as sensors or various embedded devices, the IoT has emerged along with new benefits and attack vectors. In this network of devices, IoT gateways serve as a connection point between the various end nodes like sensors and actuators on one side, and a cloud or users on the other side. The gateways control the data flow while also providing security features such as authentication and encryption. Hence, it is important to find secure realizations for such gateways which at the same time are low-cost.

Besides lightweight ciphers, PUFs can also be valuable in the IoT setting [13]. They are promising security primitives because they provide a low-cost and universally feasible anchor of trust in hardware, e.g. by storing device-unique keys [56]. In comparison to secure non-volatile memory (NVM), PUFs additionally are not susceptible to attacks when turned off and they are even feasible for small technology nodes.

We propose a novel low-cost realization of a secure gateway that combines PUF with AEAD. Unlike [25], which proposes a PUF protocol based on an AEAD scheme, this work utilizes the associated data in conjunction with a PUF.

7.3.1 Physical Unclonable Function

During the manufacturing process of a device there are unpredictable and uncontrollable variations. This causes a fingerprint-like, unique, and unpredictable characteristic for each device. A PUF utilizes this aspect by measuring a device-specific response. The measurement result, and thus the response, depends on the intrinsic process variations as well as on a possibly applied challenge which configures the PUF. For the approach presented in this work, the PUF is not limited to a specific PUF primitive or a certain amount of possible challenges. We only require that the PUF responses are different for each user who connects to our trusted hardware gateway. These different responses per user can be achieved by a kind of virtualization of the PUF: Either a separation of the challenge-response space for different users for Multi-Challenge PUF (MCPUF) or a segmentation of the address space of a Single-Challenge PUF (SCPUF) or a combination of both solves this problem.

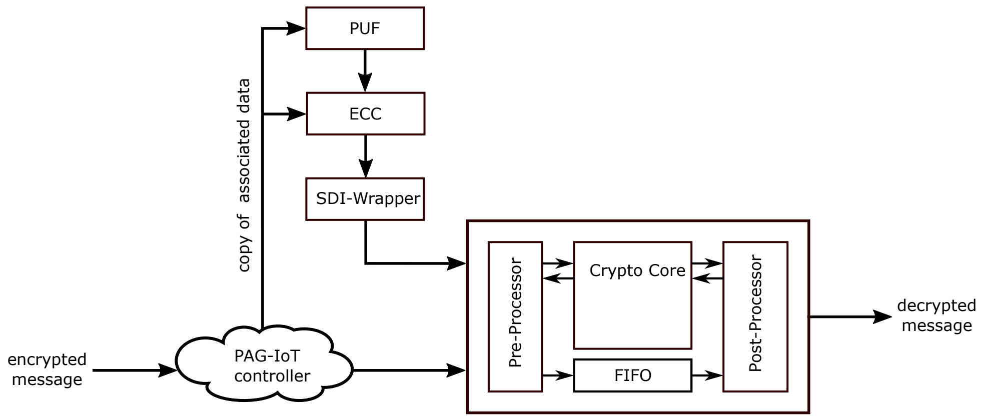


Figure 7.7: PAG-IoT hardware setup

We select an XOR SUM-PUF [95] for our proof-of-concept implementation, which is an MCPUF. Because a PUF response varies slightly due to noise, operation conditions, and aging, usually a helper data algorithm with an error correcting code (ECC) is employed to reliably reproduce the original PUF response. This post-processing step is necessary for PAG-IoT, such that we briefly recall the most important aspects:

1. A sufficient amount of PUF response bits must be generated. For MCPUF, a challenge is used as a seed, e.g., of a suitable LFSR to derive more challenges. This sequences of challenges defines the overall PUF response.

2. In the enrollment phase, helper data are determined. Basically, the helper data ensure that later on a key can be reproduced reliably. In [65], several designs for helper data algorithm (HDA) are discussed.

3. In the reproduction phase, the helper data map the errors in the reproduced PUF response to a codeword, such that error correction can be applied. Finally, the original secret is derived.

7.3.2 Hardware Setup

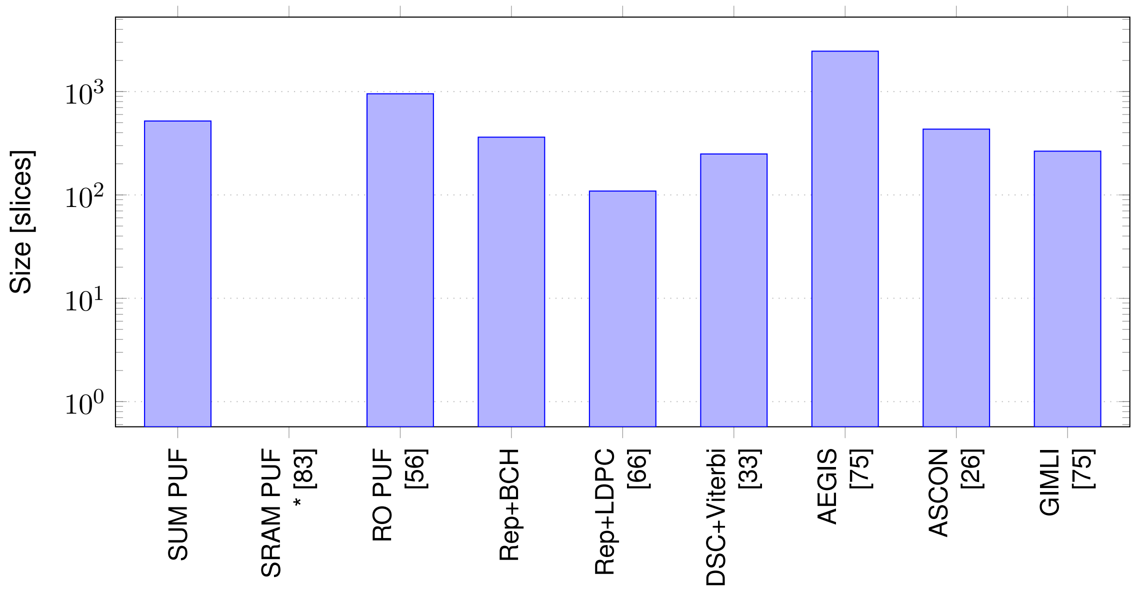
The cryptographic heart of the PAG-IoT (PUF and AEAD enabled trusted hardware gateway for IoT devices) consists of a PUF for the key generation and an AEAD module. The purpose of the AEAD module is twofold: First, it is used for normal encryption, decryption, and authentication of messages. Second, the challenge and the helper data act as AD. As helper data are the only input an attacker can control, their integrity must be enforced to mitigate attacks on the PUF.

Again, the CAESAR-API is used in our design. By doing so, the AEAD algorithm is interchangeable. The PAG-IoT controller extracts the AD from the data stream. It then uses the AD to derivative a challenge for the PUF and to provide the helper data for the error correction module. This modules generates the cryptographic key from the PUF's response. The SDI-Wrapper adds the Load Key instruction and sends the package to the AEAD module. The AEAD module decrypts and authenticates the complete message and releases the messages. If the helper data were tampered, the tag will not match and appropriate actions can be taken.

7.3.3 Results

To demonstrate the flexibility of the CAESAR-API, we synthesized the design with three different ciphers on a PYNQ Z1 board at 100 Mhz: Aegis [87] and Ascon [16] were chosen as they are the high-speed and lightweight finalists of the CAESAR competition. Gimli [9] is a round 2 NIST-LWC candidate. The latter demonstrates the benefits of the backwards compatibility of the LWC-API. Aegis and Gimli are our own implementations from [75], Ascon is a bug-fixed version from [26].

As can be seen in Figure 7.8, all three ciphers are a reasonable choice. They pose a trade-off between area and latency and are in the same dimension as the other components of the PAG-IoT.



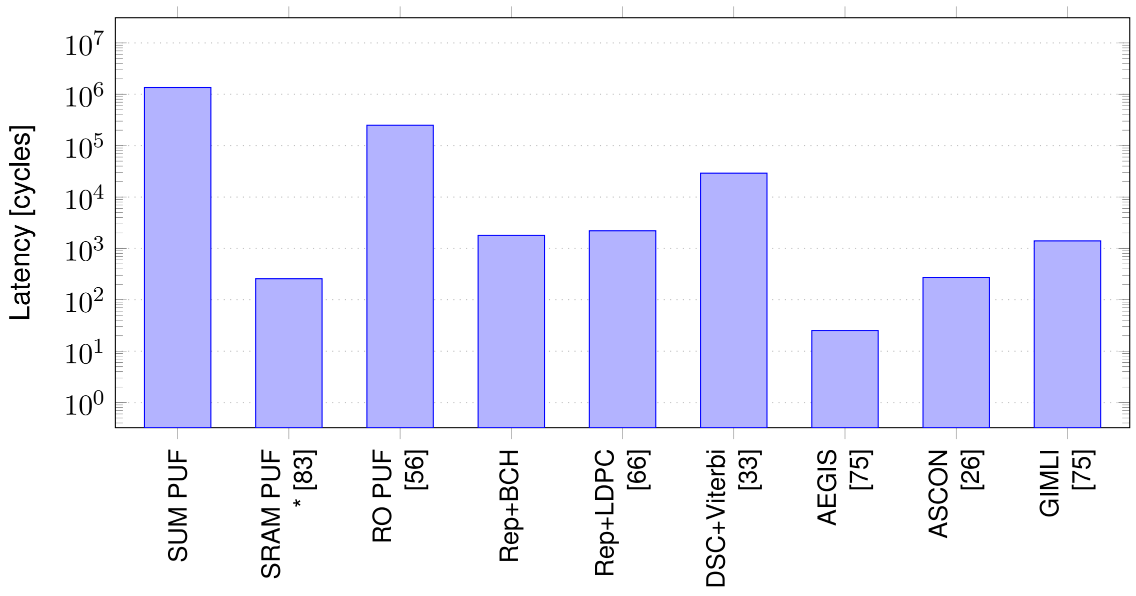


Figure 7.8: Size and latency for possible realizations of PUF, ECC, and AEAD.
*Note that an SRAM PUF [83] is not implemented in slices but in uninitialized BRAM on FPGAs.

8 The Implementer's Point of View

Considering our experiences with implementing new cryptographic algorithms on hardware, especially in the face of competitions with a lot of heterogeneous submissions, there are a few points worth mentioning.

Comparing different algorithms

1. Define a common interface with a fixed, but mandatory set of functions. This means one common bus width and keeping cryptographic functions (like padding or message authentication) separate from protocol functions. As shown in Section 4.3.2, this is mandatory for a clear assignment of the resources to either the CryptoCore or the API, and thus, an independent comparison of the individual components. Further, as shown in Section 5.3, a common bus widths prevents that different implementations require different interconnect structures, and thus ensures that all implementations require the same amount of external resources.

2. Define one or multiple common hardware architectures on which benchmarking is performed. A loosely coupled SoC like the Zynq-7000 series has different requirements for mixed hardware software implementations than a tightly coupled RISC-V environment. Thus, inter-platform results are hard to compare.

3. Either provide a thoroughly designed and tested (hardware) testbench, or ensure that (hardware) implementations are actually working. Finalists that have obvious implementations flaws, like shown in Section 3.1, disqualify themselves for any kind of benchmarking.

4. When comparing different (hardware) implementations, carefully analyze whether the differences are in the actual cryptographic primitive, in the surrounding protocol parsing, or in the system assumptions. As shown in Section 4.3.3, absolute numbers might lead to a false impression about the (in)efficiency of an implementation.

Designing algorithms

1. Algorithms that need the absolute size of a message are problematic if this number must be either determined by the implementation itself (for example, if the length information is incorporated into the tag), or if there are algorithmic dependencies on the messages size. Both result in a counter. Considering state-of-the-art message sizes [60], this counter must be up to 2^{64} bits. Things get even worse, if the same information is also needed for the Associated Data.

2. Two-pass algorithms are hard to benchmark; especially if they provide a significant security enhancement. For a fair benchmarking, the message must either be buffered or retransmitted for the second pass. Thus, either the size of this additional buffer (up to 2^{64} bytes) would have to be accounted to the implementation size, or the time needed for retransmission would have to be accounted to the overall runtime. In the CAESAR and LWC competition this was often not the case or implementations limited the length of the message [46, 38].

3. If possible avoid designing algorithms that need a two dimensional state access. This is for example the case if one subfunction of the round function needs to access the same bits of different words and another subfunction needs to access one complete word. This is particularly important for small, but fast serial implementations.

4. As shown in Section 5.3, the width of the data path does not only affect the cipher itself (size, performance), but also the surrounding elements like FIFOs, DMA-controllers, interconnects, etc. Thus, this parameter accounts multiple times for the overall system size and performance.

Implementing and using algorithms with respect to the CAESAR-API or LWC-API

1. Having a fixed set of control signals for the CryptoCore, i.e. the internal interface, allows for a fast prototyping, as it is clear which information is provided and which information must be calculated from the inputs. However, the information encoded in "eot", "eoi", "last", and "partial" should be enhanced. We suggest to either use more control signals for the look-ahead information of omitted, empty segments, or –preferred– not to omit any empty segments. Although, this introduces a possible latency of two clock cycles under rare conditions, it simplifies understanding the behavior of the interface, which is more significant than making the most of the performance. As discussed in Section 6.4, even the official hardware reference implementation of the CAESAR finalist Acorn struggles with empty inputs.

2. Having a compact external interface allows for an easy integration into larger systems. Especially the dedicated key interface (SDI) is very useful for connecting on-chip key

storage (e.g. a PUF or a secure on chip memory) with the CryptoCore, as there is no need for an input multiplexing. This was demonstrated in Section 7.3.

3. The disadvantage of the API is that it does not allow for mixed hardware software co-designs, or highly optimized, tightly coupled co-processors. However, this is not needed in the first benchmarking phase of cryptographic competitions. Here, it is more important to narrow down the number of candidates. Thus, the advantages of the API predominate.

9 Conclusion

In this thesis, we demonstrated the problems and the need of defining a hardware API for a fair benchmarking of hardware (cryptographic) implementations. We proposed some extensions to the CAESAR-API to fulfill the requirements of the NIST-LWC-competition. Further, we analyzed different support packages, that implement the CAESAR API and our proposed LWC API, and showed that the implementation of our hardware API is of reasonable size. The LWC development package is open source and available on GitHub[1].

Next, we evaluated the implementation of eleven third-round CAESAR candidates. We showed that there are some (partly major) problems in those implementations. Thus, pointing out that independent, mutual testing is crucial not only in simulation but on real hardware, as even some implementations of the third-round candidates behave differently on real hardware than in simulation. Therefore, we proposed a Zynq-based hardware evaluation framework for candidates participating in the CAESAR competition. This framework can not only be used as an evaluation platform, but also as dynamically reconfigurable hardware accelerator for CAESAR candidates. The framework is also open source and freely available[2] to allow interested designers to test and validate their designs based on the PYNQ evaluation board from Xilinx.

Finally, we showed how ciphers, equipped with the CAESAR or LWC-API, can be integrated into other research domains and that the advantages of the API predominate its disadvantages.

Although, the evaluation was mostly based on the CAESAR competition, the results and especially the pitfalls can be transferred to the NIST-LWC-competition. We hope, that especially our works regarding the LWC-API and regarding a fair benchmarking process are considered during the NIST-LWC-competition. Some of the results like the API itself are most likely endorsed by NIST for its final round of the LWC competition.

[1] `https://github.com/GMUCERG/LWC`
[2] `https://gitlab.lrz.de/tueisec/PYNQ-CAESAR`

List of Figures

List of Tables

Acronyms

AD	Associated Data
AEAD	Authenticated Encryption with Associated Data
AE	Authenticated Encryption
AES	Advanced Encryption Standard
AES-GCM	AES in Galois Counter Mode
AES-OCB2	AES in Offset Codebook 2 Mode
API	Application Programming Interface
ASIC	Application-specific Integrated circuit
ATHENa	Automated Tool for Hardware EvaluatioN
AXI	Advanced eXtensible Interface Bus
AXIL	AXI4-Lite
AXIS	AXI4-Stream
BC	Block Cipher
BDI	block data input
BDO	block data output
CAESAR	Competition for Authenticated Encryption: Security, Applicability, and Robustness
CCM	Counter with CBC-MAC

CRYPTREC Cryptography Research and Evaluation Committee

CSM current-shunt monitor

DMA Direct Memory Access

DO data output

ECC error correcting code

ECRYPT European Network of Excellence in Cryptology

eSTREAM ECRYPT Stream Cipher Project

FF Flip-Flop

FPGA Field-programmable gate array

GPIO General Purpose Input/Output

HDA helper data algorithm

IO Input/Output

IoT Internet-of-Things

IP Intellectual Property

IPsec Internet Protocol Security

IRQ Interrupt Request

ISO International Organization for Standardization

KAT known-answer-test

LFSR Linear Feedback Shift Register

LUT Look Up Table

LWC Lightweight Crypography

MAC Message Authentication Code

MCPUF	Multi-Challenge PUF
MPSoC	Multiprocessor System on Chip
MSPS	mega samples per second
NESSIE	New European Schemes for Signatures, Integrity and Encryption
NI	Network Interface
NIST	National Institute of Standards and Technology
NoC	Network on a Chip
nonce	number used only once
NVM	non-volatile memory
PAG-IoT	PUF and AEAD enabled trusted hardware gateway for IoT devices
PCB	Printed Circuit Board
PDI	public data input
PL	Programmable Logic
PQC	Post-quantum Cryptography
PS	Processing System
PUF	Physical Unclonable Function
RDI	random data input
SCPUF	Single-Challenge PUF
SDI	secret data input
SEM	Standard Error of Mean
SoC	System on Chip
SSH	Secure Shell

SSL Secure Sockets Layer

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

TLS Transport Layer Security

TP throughput

TV test vector

UUT Unit Under Test

XADC Xilinx analog mixed signal module

XBP XXBX Power Shim

Bibliography

[1] Farzaneh Abed, Christian Forler, and Stefan Lucks. General classification of the authenticated encryption schemes for the CAESAR competition. Cryptology ePrint Archive, Report 2014/792, 2014. https://eprint.iacr.org/2014/792.

[2] Dean Michael Ancajas, Koushik Chakraborty, and Sanghamitra Roy. Fort-NoCs: Mitigating the threat of a compromised NoC. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2730-5. doi: 10.1145/2593069.2593144. URL http://doi.acm.org/10.1145/2593069.2593144.

[3] Elena Andreeva, Atul Luykx, and Bart Mennink. COLM v1. *Submission to the CAESAR competition*, 2016.

[4] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. NORX v3. 0. submission to CAESAR (2016). *Submission to the CAESAR Competition*, 2015.

[5] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21 (4):469–491, Oct 2008. ISSN 1432-1378. doi: 10.1007/s00145-008-9026-x. URL https://doi.org/10.1007/s00145-008-9026-x.

[6] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):206–241, May 2004.

[7] Daniel J. Bernstein. Failures of secret-key cryptography. Invited Talk at FSE 2013 (20th International Workshop on Fast Software Encryption), 2013.

[8] Daniel J. Bernstein and Tanja Lange. SUPERCOP: System for unified performance evaluation related to cryptographic operations and primitives. Online, 2006. http://bench.cr.yp.to/supercop.html.

[9] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. Gimli, 2019. URL https://gimli.cr.yp.to/.

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography (SAC 2012)*, volume 7118 of *LNCS*, pages 320–337. Springer, 2012.

[11] Travis Boraten and Avinash Karanth Kodi. Packet security with path sensitization for NoCs. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016.

[12] CAESAR secretary. CAESAR: Competition for authenticated encryption: Security, applicability, and robustness. http://competitions.cr.yp.to/caesar.html, July 2012. Project Website.

[13] Baibhab Chatterjee, Debayan Das, and Shreyas Sen. RF-PUF: IoT security enhancement through authentication of wireless nodes using in-situ machine learning. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 205–208. IEEE, 2018.

[14] Pascal Cotret, Guy Gogniat, and Johanna Sepúlveda. Protection of heterogeneous architectures on FPGAs: An approach based on hardware firewalls. *Microprocessors and Microsystems*, 42, 2016.

[15] CRYPTREC Secretariat. CRYPTREC: Cryptography research and evaluation committee. https://www.cryptrec.go.jp/en/development.html, 2020. Project Website.

[16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. *Submission to the CAESAR Competition*, 2016. https://ascon.iaik.tugraz.at/index.html,.

[17] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1. 2. *Submission to the CAESAR Competition*, 2016.

[18] Morris Dworkin. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38c.pdf, May 2004. Online.

[19] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) for confidentiality and authentication, Nov 2007.

[20] Morris J. Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.

[21] Farnoud Farahmand, Ekawat Homsirikamol, and Kris Gaj. A Zynq-based testbed for the experimental benchmarking of algorithms competing in cryptographic contests. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig*, December 2016. doi: 10.1109/ReConFig.2016.7857148.

[22] Farnoud Farahmand, Ahmed Ferozpuri, William Diehl, and Kris Gaj. Minerva: Automated hardware optimization tool. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2017.

[23] Christoph Frisch, Michael Tempelmeier, and Michael Pehl. PAG-IoT: A PUF and AEAD enabled trusted hardware gateway for IoT devices. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 500–505, 2020.

[24] Kris Gaj, Jens-Peter Kaps, Venkata Amirineni, Marcin Rogawski, Ekawat Homsirikamol, and Benjamin Y. Brewster. ATHENa – automated tool for hardware evaluation: Toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs. In *20th International Conference on Field Programmable Logic and Applications - FPL 2010*, pages 414–421. IEEE, 2010.

[25] Gerben Geltink. Concealing KETJE: A lightweight PUF-based privacy preserving authentication protocol. In *International Workshop on Lightweight Cryptography for Security and Privacy*. Springer, 2016.

[26] George Mason University, Cryptographic Engineering Research Group. CERG source code webpage. https://cryptography.gmu.edu/athena/index.php?id=CAESAR_source_codes, snapshot from October 2019.

[27] George Mason University, Cryptographic Engineering Research Group. ATHENa Automated Tool for Hardware EvaluatioN. http://cryptography.gmu.edu/athena/, 2019. Project Website.

[28] Miltos D. Grammatikakis, Kyprianos Papadimitriou, Polydoros Petrakis, Antonis Papagrigoriou, George Kornaros, Ioannis Christoforakis, Othon Tomoutzoglou, George Tsamis, and Marcello Coppola. Security in MPSoCs: A NoC firewall and an evaluation framework. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1344–1357, Aug 2015. ISSN 0278-0070. doi: 10.1109/TCAD.2015.2448684.

[29] Graz University of Technology, Institute of Applied Information Processing and Communications. IAIK GITHUB repository. https://github.com/IAIK/ascon_hardware/tree/master/caesar_hardware_api_v_1_0_3/ASCON_ASCON, snapshot from January 15, 2020.

[30] Peter Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366, September 2014. URL https://rfc-editor.org/rfc/rfc7366.txt.

[31] *AES-GCM Core family for Xilinx FPGA*. Helion Technology, 2011. http://www.heliontech.com/downloads/aes_gcm_8bit_xilinx_datasheet.pdf.

[32] Andreas Herkersdorf, Stefan Wallentowitz, and Thomas Wild. Integrated multicore processors, 2012.

[33] Matthias Hiller, Meng-Day Yu, and Georg Sigl. Cherry-picking reliable PUF bits with differential sequence coding. *IEEE Transactions on Information Forensics and Security*, 11(9):2065–2076, 2016.

[34] VT Hoang, T Krovetz, and P Rogaway. AEZ v4. 2: Authenticated encryption by enciphering. *Submission to the CAESAR competition*, 2016.

[35] Ekawat Homsirikamol and Kris Gaj. AEZ: Anything-but EaZy in hardware. In *International Conference on Cryptology in India*, pages 207–224. Springer, 2016.

[36] Ekawat Homsirikamol, William Diehl, Ahmed Ferozpuri, Farnoud Farahmand, Malik Umar Sharif, and Kris Gaj. GMU Hardware API for Authenticated Ciphers. Cryptology ePrint Archive, Report 2015/669, 2015. http://eprint.iacr.org/.

[37] Ekawat Homsirikamol, William Diehl, Ahmed Ferozpuri, Farnoud Farahmand, and Kris Gaj. Implementer's guide to the CAESAR hardware API. http://cryptography.gmu.edu/athena/CAESAR_HW_API/CAESAR_HW_Implementers_Guide_v1.0.pdf, May 2016. Online.

[38] Ekawat Homsirikamol, William Diehl, Ahmed Ferozpuri, Farnoud Farahmand, Panasayya Yalla, Jens-Peter Kaps, and Kris Gaj. CAESAR hardware API. Cryptology ePrint Archive, Report 2016/626, 2016. https://eprint.iacr.org/2016/626.

[39] Ekawat Homsirikamol, Panasayya Yalla, Farnoud Farahmand, William Diehl, Ahmed Ferozpuri, Jens-Peter Kaps, and Kris Gaj. Implementer's guide to hardware implementations compliant with the CAESAR hardware API version 2.0, 2017. URL https://cryptography.gmu.edu/athena/CAESAR_HW_API/CAESAR_HW_Implementers_Guide_v2.0.pdf.

[40] Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, Sumio Morioka, and E Kobayashi. CLOC: Compact low-overhead CFB. *Submission to the CAESAR competition*, 2014.

[41] Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, Sumio Morioka, and Eita Kobayashi. SILC: simple lightweight CFB. *Submission to the CAESAR competition*, 2014.

[42] Jérémy Jean. TikZ for Cryptographers. https://www.iacr.org/authors/tikz/, 2016.

[43] Jérémy Jean, Ivica Nikolic, Thomas Peyrin, and Yannick Seurin. Deoxys v1. 41. *Submission to the CAESAR competition*, 2016.

[44] Jens-Peter Kaps. eXtended eXternal Benchmarking eXtension (XXBX). SPEED-B - Software performance enhancement for encryption and decryption, and benchmarking, Oct. 2016. Utrecht, invited talk.

[45] Jens-Peter Kaps. Hardware benchmarking framework for lightweight cryptography, October 2019. URL https://groups.google.com/a/list.nist.gov/d/msg/lwc-forum/ho1OiRPGRLE/hpIIeJR7CwAJ.

[46] Jens-Peter Kaps, William Diehl, Michael Tempelmeier, Farnoud Farahmand, Ekawat Homsirikamol, and Kris Gaj. A comprehensive framework for fair and efficient benchmarking of hardware implementations of lightweight cryptography. Cryptology ePrint Archive, Report 2019/1273, 2019. https://eprint.iacr.org/2019/1273.

[47] Jens-Peter Kaps, William Diehl, Michael Tempelmeier, Ekawat Homsirikamol, and Kris Gaj. Hardware API for lightweight cryptography. Technical report, Tech. Rep., Oct, 2019.

[48] Patrick Karl and Michael Tempelmeier. A detailed report on the overhead of hardware APIs for lightweight cryptography. Cryptology ePrint Archive, Report 2020/112, 2020. https://eprint.iacr.org/2020/112.

[49] Katholieke Universiteit Leuven, Department Electrical Engineering, Division SCD/-COSIC.

[50] Katholieke Universiteit Leuven, Department Electrical Engineering, Division SISTA/-COSIC. NESSIE: New european schemes for signatures, integrity, and encryption. https://www.cosic.esat.kuleuven.be/nessie/, 2020. Project Website.

[51] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014. ISBN 1466570261.

[52] Stephen Kent. IP Encapsulating Security Payload (ESP). RFC 4303, December 2005. URL https://rfc-editor.org/rfc/rfc4303.txt.

[53] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: how secure is SSL?). Cryptology ePrint Archive, Report 2001/045, 2001. http://eprint.iacr.org/2001/045.

[54] Ted Krovetz and Phillip Rogaway. OCB (v1. 1). *Submission to the CAESAR Competition*, 2016.

[55] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, January 2006. URL https://rfc-editor.org/rfc/rfc4253.txt.

[56] Roel Maes, Anthony Van Herrewege, and Ingrid Verbauwhede. PUFKY: A fully functional PUF-based cryptographic key generator. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012.

[57] Cathal McCabe. Performance of transfer()-method of legacydma vs. dma, December 2017. URL https://groups.google.com/d/msg/pynq_project/ZNglipv4yog/fD2FedHBDAAJ. Xilinx University Program Manager EMEA.

[58] Paweł Morawiecki, Kris Gaj, Ekawat Homsirikamol, Krystian Matusiewicz, Josef Pieprzyk, Marcin Rogawski, Marian Srebrny, and Marcin Wójcik. Icepole v2. *Submission to the CAESAR competition*, 2015.

[59] Nanyang Technological University, School of Physical and Mathematical Sciences, Division of Mathematical Sciences. NTU CAESAR submission webpage. http://www3.ntu.edu.sg/home/wuhj/research/caesar/caesar.html, snapshot from June 2018, 2018.

[60] National Institute of Standards and Technology. LWC project website. https://csrc.nist.gov/projects/lightweight-cryptography, 2020. Lightweight Cryptography.

[61] National Institute of Standards and Technology. PQC project website. https://csrc.nist.gov/projects/post-quantum-cryptography, 2020. Post-Quantum Cryptography.

[62] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. Report on the development of the Advanced Encryption Standard (AES). Technical report, Computer Security Division Information, Technology Laboratory, NIST, Oct 2000.

[63] Ivica Nikolic. Tiaoxin-346. *Submission to the CAESAR Competition*, 2014.

[64] Siavoosh Payandeh Azad, Michael Tempelmeier, Gert Jervan, and Johanna Sepúlveda. CAESAR-MPSoC: Dynamic and efficient MPSoC security zones. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 477–482, 2019.

[65] Michael Pehl, Matthias Hiller, and Georg Sigl. *Information Theoretic Security and Privacy of Information Systems*, chapter Secret Key Generation for Physical Unclonable Functions. Cambridge University Press, 2017.

[66] Michael Pehl, Christoph Frisch, Peter Christian Feist, and Georg Sigl. KeLiPUF: a key-distribution protocol for lightweight devices using physical unclonable functions. In *17th escar Europe : embedded security in cars*. 2019. doi: 10.13154/294-6676.

[67] Project Bonfire. https://github.com/Project-Bonfire/Bonfire, 2019. Accessed: 2019-02-03.

[68] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018. URL https://rfc-editor.org/rfc/rfc8446.txt.

[69] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008. URL https://rfc-editor.org/rfc/rfc5246.txt.

[70] Matthew Robshaw and Olivier Billet. *New Stream Cipher Designs, The eSTREAM Finalists*, volume 4986 of *LNCS*. Springer, 2008.

[71] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *Advances in Cryptology – ASIACRYPT 2004*, volume 3329 of *LNCS*, pages 16–31. Springer, 2004.

[72] Johanna Sepúlveda, Guy Gogniat, Daniel Flórez, Jean-Philippe Diguet, Cesar Zeferino, and Marius Strum. Elastic security zones for NoC-based 3D-MPSoCs. In *Electronics, Circuits and Systems (ICECS), 2014 21st IEEE International Conference on*, pages 506–509. IEEE, 2014.

[73] Johanna Sepúlveda, Daniel Flórez, and Guy Gogniat. Reconfigurable security architecture for disrupted protection zones in NoC-based MPSoCs. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2015 10th International Symposium on*, pages 1–8. IEEE, 2015.

[74] Johanna Sepúlveda, Andreas Zankl, Daniel Flórez, and Georg Sigl. Towards protected MPSoC communication for information protection against a malicious NoC. *Procedia Computer Science*, 108:1103 – 1112, 2017. ISSN 1877-0509. URL http:

//www.sciencedirect.com/science/article/pii/S1877050917307068. International Conference on Computational Science, ICCS 2017.

[75] Technical University of Munich, Department of Electrical and Computer Engineering, Chair of Security in Information Technology. TUMEISEC crypto implementation repository. https://gitlab.lrz.de/tueisec/crypto-implementations/, git checkout 4f054dc5.

[76] Michael Tempelmeier, Fabrizio De Santis, Jens-Peter Kaps, and Georg Sigl. An area-optimized serial implementation of ICEPOLE authenticated encryption schemes. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 49–54, May 2016. doi: 10.1109/HST.2016.7495555.

[77] Michael Tempelmeier, Fabrizio De Santis, Georg Sigl, and Jens-Peter Kaps. The CAESAR-API in the real world – towards a fair evaluation of hardware CAESAR candidates. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 73–80. IEEE, 2018.

[78] Michael Tempelmeier, Georg Sigl, and Jens-Peter Kaps. Experimental power and performance evaluation of caesar hardware finalists. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2018.

[79] Michael Tempelmeier, Farnoud Farahmand, Ekawat Homsirikamol, William Diehl, Jens-Peter Kaps, and Kris Gaj. Implementer's guide to hardware implementations compliant with the hardware API for lightweight cryptography, 2019. URL https://cryptography.gmu.edu/athena/LWC/LWC_HW_Implementers_Guide.pdf.

[80] Michael Tempelmeier, Maximilian Werner, and Georg Sigl. Using hardware software codesign for optimised implementations of high-speed and defence in depth CAESAR finalists. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 228–237, 2019.

[81] Virginia Tech, Signatures Analysis Laboratory. VTSAL GITHUB repository. https://github.com/vtsal?tab=repositories, snapshot from January 13, 2020.

[82] Rüdiger Weis. Der Advanced Encryption Standard (AES), 2008.

[83] Alexander Wild and Tim Güneysu. Enabling SRAM-PUFs on Xilinx FPGAs. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2014.

[84] Hongjun Wu. ACORN: a lightweight authenticated cipher (v3). *Submission to the CAESAR competition*, 2016.

[85] Hongjun Wu and Tao Huang. The JAMBU lightweight authentication encryption mode (v2. 1). *Submission to the CAESAR competition*, 2016.

[86] Hongjun Wu and Tao Huang. The authenticated cipher MORUS (v2). *Submission to the CAESAR competition*, 2016.

[87] Hongjun Wu and Bart Preneel. AEGIS: A fast authenticated encryption algorithm. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-43414-7.

[88] Hongjun Wu and Bart Preneel. AEGIS: a fast authenticated encryption algorithm (v1.1). *Submission to the CAESAR competition*, 2014.

[89] Xilinx. Spartan-3 generation FPGA user guide (UG331), ver. 1.8, 2011.

[90] Xilinx. Virtex FPGA configurable logic block (UG364), ver. 1.2, 2012.

[91] Xilinx. AXI reference guide (UG761), ver. 13.4, 2012.

[92] Xilinx. 7 series FPGAs configurable logic block (UG474), ver. 1.8, 2016.

[93] *Zynq-7000 SoC Data Sheet: Overview*. XILINX, 7 2018. v1.11.1.

[94] Panasayya Yalla, Ekawat Homsirikamol, and Jens-Peter Kaps. Comparison of multi-purpose cores of Keccak and AES. In *DATE 2015*, Mar 2015.

[95] Meng-Day Yu and Srinivas Devadas. Recombination of Physical Unclonable Functions. *35th Annual GOMACTech Conference*, March 2010.

Credits:

Figures 2.2, 2.3, 3.3, 4.1, 6.1, 6.3 and 7.1 reprinted with permission of IEEE. Sections 3.2, 5.2 and 7.1 to 7.3, Chapter 6, and Figures 3.4, 3.5 and 5.2 partially reprinted with permission of IEEE.