KU LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN

Energy Efficient Hardware Implementations of CAESAR Submissions

Michael Fivez

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
elektrotechniek, optie Ingebedde
systemen en multimedia

Promotoren:
Prof. dr. ir. Ingrid Verbauwhede
Prof. dr. ir. Vincent Rijmen

Assessoren:
Prof. dr. ir. R. Lauwereins
dr. B. Bilgin

Begeleiders:
ir. P. Maene
ir. B. Yang

Academiejaar 2015 – 2016

Preface

The realization of this thesis would not have been possible without the support and guidance of several individuals.

Firstly, I would like to thank my promotors, prof. Ingrid Verbauwhede and Prof. Vincent Rijmen, for providing the interesting thesis topic and giving me the chance to complete my thesis under their direction.

I would also like to thank my three daily advisers, dr. Begül Bilgin, ir. Pieter Maene, and ir. Bohan Yang. The door to their office was always open, and I could not have imagined any better supervisors. Their extensive feedback on the final thesis paper was also much appreciated.

A special thanks also goes out to the last jury member, prof. Rudy Lauwereins.

Lastly I would like to thank my parents for supporting me throughout the five years of my study, and for reading my thesis and providing me valuable feedback.

Writing this master thesis has been an enriching experience that I feel I will carry with me in the future.

Michael Fivez

Contents

Abstract

In this thesis, three lightweight cryptographic ciphers from *The Competition for Authenticated Encryption: Security, Applicability, and Robustness* (CAESAR) were evaluated on Field-programmable gate array (FPGA), namely Joltik, Morus and Ascon. The main focus of the evaluation was their energy consumption and area requirement.

Several hardware implementations were made of each cipher. The round-based structure of the three ciphers was used to create implementations that vary largely in size. Their energy consumption was determined through live measurements. Together with the area and maximum clock speed, obtained through synthesis, they could be compared.

Because of the high variability of hardware implementations and their measurements, special care was taken to ensure that the comparisons were accurate. By using the same external *Application programming interface* (API), the implementations were forced in to a similar structure. The core of the implementations was then isolated for the measurements. Measures were taken to reduce any overhead on the FPGA, as well as external influences like the temperature.

The comparisons yielded three main results. The general performance of each cipher was quantified, and it was compared to the claims of their authors. The trade-off between the area and energy consumption was determined for each cipher. This trade-off is relevant in low-power, embedded applications. And as a third result, the performance of the implementations was generalized to draw conclusions about effective optimization strategies for different types of ciphers.

Samenvatting

In deze thesis zijn drie lichtgewicht cryptografische algoritmes uit *The Competition for Authenticated Encryption: Security, Applicability, and Robustness* (CAESAR) geëvalueerd op *Field-programmable gate array* (FPGA). Deze algoritmes zijn Joltik, Morus en Ascon. De hoofdfocus van de evaluatie was hun energieverbruik en hardware-grootte.

Meerdere implementaties zijn gemaakt van elk algoritme. Er is gebruik gemaakt van hun *ronde gebaseerde* structuur om implementaties te maken van variërende groottes. Hun energieverbruik was gemeten via directe metingen. Samen met hun hardware-grootte en maximale kloksnelheid, beiden verkregen uit hun synthetisatie, konden ze worden vergeleken.

Omwille van de grote variabiliteit van hardware implementaties en hun metingen, is er intensief aandacht besteed aan de correctheid van de vergelijkingen. Door voor alle algoritmes dezelfde externe *application programming interface* (API) te gebruiken, kregen alle implementaties dezelfde structuur. De kern van de implementaties werd dan geïsoleerd en gemeten. Maatregelen zijn genomen om de invloed van externe factoren, zoals de temperatuur en externe communicatie, te elimineren.

De vergelijkingen leidden tot drie voorname conclusies. De algemene prestaties van elk algoritme zijn gekwantificeerd, en vergeleken met de originele beweringen van hun makers. Voor ieder algoritme is het compromis tussen de hardware-grootte en het energieverbruik besproken. Dit compromis is van belang in lage-energie toepassingen. En als derde resultaat zijn de prestaties van de implementaties veralgemeend om effectieve optimalisatie strategieën te bekomen voor verschillende types cryptografische algoritmes.

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

NSA	National Security Agency
CAESAR	Competition for Authenticated Encryption: Security, Applicability, and Robustness
FPGA	Field-programmable gate array
NIST	National Institute of Standards and Technology
AES	Advanced Encryption Standard
AE	Authenticated Encryption
AD	Associated Data
API	Application programming interface
XPE	Xilinx Power Estimator
ASIC	Application-specific integrated circuit
S-box	Substitution-box
LUT	Look-Up Table
XPA	Xilinx Power Analyzer
FSM	Finite-state machine
CLB	Configurable logic block
AC	Alternating current
DC	Direct current
MDS	Maximum Distance Separable

Chapter 1

Introduction

The amount of personal data that is stored in digital form is increasing rapidly. Together with high speed internet this enables hackers to steal millions of personal files at once [3]. The importance of encryption is increasing and millions can be gained by cracking encryption. Large organizations such as the National Security Agency (NSA) invest billions for this purpose, so the smallest mistakes in security implementations are capitalized upon [4].

At the same time, following Moore's law, the density and speed of integrated circuits have been increasing exponentially. This has allowed electronic devices to get smaller and smarter and eventually become portable, making the energy consumption of integrated circuits a design criteria. It is predicted that with the Internet of Things [5, 6] energy efficiency will become an even more important design factor in the future. Small and cheap chips, lasting years on a single small battery, require every component to consume as little energy as possible.

These two evolutions demand new, more robust, and more efficient encryption algorithms. This thesis explores the energy consumption of some possible candidates. It does this through the *Competition for Authenticated Encryption: Security, Applicability, and Robustness* (CAESAR). The ciphers in the competition are authenticated ciphers. They ensure the user that the information comes from the right person (authenticity), that only he can read it (confidentiality) and that it has not been tampered with (integrity) [7]. Three ciphers from the competition which have potential to be energy efficient in hardware are chosen and analyzed.

The aim of this thesis is twofold. First to see how the three chosen ciphers perform energy-wise, to contribute to the evaluation of the chosen CAESAR candidates. Multiple implementations of each cipher are made, with different areas and speeds, to analyze the area-energy-speed trade-off. The second goal is to choose diverse enough algorithms, so that relevant conclusions can be made about the energy consumption and useful optimization strategies of different cipher families.

Field-programmable gate array (FPGA) implementations of the algorithms and their versions are made in VHDL, and then optimized to be as energy efficient as possible. All implementations follow a similar approach and have a common interface, and focus on one FPGA in particular (Spartan 6). This is done to facilitate their comparison.

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

In 1970 public developments made high quality cryptography accessible to the general public. Governments tried to keep their monopoly on it, and until this day there are still laws limiting the export of cryptography. For example, the UK government recently called for banning secure messaging applications that do not build in *backdoor encryption* for governement access [8].

However, around the year 2000, the United States relaxed their laws on the export of cryptography [9]. At the time, the US National Institute of Standards and Technology (NIST) announced an open competition to find "an unclassified, publicly disclosed encryption algorithm capable of protecting sensitive government information well into the next century" [10]. The winning algorithm of that competition (renamed Advanced Encryption Standard or AES), is still one of the most used symmetric-key algorithms today.

Several other competitions, based on the same open principles, were organized in the new millennium. eSTREAM, organized from 2004 to 2008, resulted in a portfolio of seven stream ciphers [11]. Later, the SHA-3 competition aimed to find "a new hash algorithm to augment and revise" the SHA-1 and SHA-2 standards [12]. The competition was created because of fears that SHA-2 would be broken. Due to successful attacks on MD5 and theoretical attacks on SHA-1 [13, 14], there was no other backup algorithm. The competition went from 2007 to 2012.

The ongoing CAESAR competition follows in this tradition. The aim of the competition is to find Authenticated Encryption (AE) ciphers that offer advantages over the popular AES-GCM, and are suited for widespread adoption [15]. The competition will identify a portfolio of authenticated ciphers, good for a wide rage of applications. It will have a balance between security and performance (including criteria like robustness against implementation errors).

56 submissions entered the competition. Each submission proposes a family of authenticated ciphers, where the individual ciphers of the families can vary in external or internal parameters, like key length or the number of *rounds*. The first round of the competition, which went from March 2014 to July 2015, focused mostly on the security of the ciphers. While the next rounds will also focus on the performance, both in software and hardware. Software and hardware implementations of the ciphers are required and compared which each other.

The competition is based on the principle of public evaluation. A committee with members from several universities judge the ciphers [16], basing their decisions on published analyses by independent researchers.

1.2 Authenticated Encryption

Authenticated ciphers are ciphers that provide both authentication, confidentiality and integrity. This ensures the user that the information comes from the right person

Input	**Output**
Associated data	*Associated data* (unchanged)
Plaintext	*Ciphertext* (= encrypted plaintext + authentication tag)
Secret Key	
Public nonce	
Secret nonce	

Table 1.1: The inputs and output of authenticated ciphers participating in the CAESAR competition

(authenticity), that only he can read it (confidentiality) and that it has not been tampered with (integrity)[7].

It has been found that separating the confidentiality and integrity mode (for example with a block cipher and a hash function), easily leads to algorithms that are not robust against implementation errors [17]. This called for the development of good authenticated ciphers that combine both goals.

The ciphers participating in the CAESAR competition are required to accept variable length Associated Data (AD) and plaintext, and convert these into a ciphertext with the help of a fixed-length public nonce, secret nonce and key (the use of a secret nonce is optional). Integrity is provided for the Associated Data, and both integrity and confidentiality are provided for the plaintext [15]. This is displayed in Table 1.1.

AD is a part of a message that should be authenticated, but doesn't require confidentiality. For example the payload of a packet should by authenticated and encrypted (this is the plaintext), but the header should be only authenticated. Integrity is important for both parts, so no attacker can fool the receiver in to thinking that he is communicating with someone else.

The nonce is an extra number aside from the key that is used in the cipher. For example it prevents that when the same message is encrypted twice with the same key, the same ciphertext would be obtained. Often it can not be reused more than once for the same key, without the cipher losing its security.

1.3 Outline

Chapter 1 contains the introduction of the thesis. The background of the thesis is introduced. It defines the goals of the research of this thesis. It then lists the outline of the thesis.

In *Chapter 2* the ciphers in the CAESAR competition are presented. Three ciphers from the competition are selected for analysis. After that, the three chosen algorithms are presented in more depth.

Chapter 3 contains the literature review. Design methodologies are researched. Hardware implementations of other ciphers are analyzed, and some optimization techniques are discussed. Finally, some methods to estimate and measure energy of FPGA designs are found.

Chapter 4 lists the implementation approach. A general approach is established, with a common external Application Programming Interface (API) for all the implementations. A more specific structure is designed to facilitate the comparison of the ciphers. Then the structure and features of the used FPGA, Spartan 6, are listed. Finally, a concrete measuring method is described to measure the energy consumption of the implementations.

Chapter 5 presents the implementations of the first cipher, Joltik. The performance of the implementations is measured and the results are presented and discussed.

Chapter 6 and 7 repeat the same process for the other two ciphers. Respectively Morus and Ascon.

In *Chapter 8* the measuring results of the different ciphers are compared. The absolute performance and strengths and weaknesses of each cipher are identified. The global optimization strategies between the implementations are discussed, and conclusions about the benefits of certain strategies for certain types of ciphers are made.

The final chapter contains the conclusion of the thesis. The research questions are restated and the empirical findings in the thesis are summarized to give an answer to those questions. The limitations of the research and possible further research are discussed.

Chapter 2

The Candidate Ciphers

As stated in Chapter 1, this thesis aims to evaluate ciphers that, when implemented on FPGA, can be used for low energy applications. The ciphers should also be diverse, so that interesting conclusions can be made by comparing different cipher families. To achieve these goals, the choice of ciphers is important.

In this chapter, the chosen ciphers and their selection are given and commented. First, a short summary of the algorithms present in the competition is given, including the motivation for their selection for analysis. After that each of the three chosen ciphers are presented more in-depth with a short global overview of their features. In the end a short summary of the ciphers side by side is given.

2.1 Ciphers in the Competition

Thirty candidate ciphers made it to the second round of the CAESAR competition. Of these, 13 are block cipher-based, three stream cipher-based, eight are based on sponge functions, two on permutations, one on a compression function (hash) and three have their own dedicated scheme [18]. The vast majority of the block cipher-based algorithms is either using AES or uses a primitive based on AES [18]. In this thesis, three algorithms are picked for analysis, one block cipher based, one with a dedicated scheme that is partially based on a Type-3 Feistel scheme [19], and one that is based on sponge-functions.

A block cipher is a cipher that takes a block of data and converts it in to an encrypted block of data with the use of a secret key. If the same block of data is encrypted with the same key, the result will also be the same. To achieve confidentiality and authenticity of a message, extra operations need to be performed apart from the block cipher. This is called a *mode of operation* [20].

A Type-3 Feistel scheme changes a block of data iteratively through a number of rounds. The rounds have a structure as shown in Figure 2.1. F can be any function. A classical sponge construction is displayed in Figure 2.2. Blocks of data are xor'ed with an internal state, that is updated with a sponge function f.

No AES-based cipher is selected for analysis. Since AES is such a popular block cipher, a great amount of research has been already done on it. It is more useful to analyze less studied schemes and ciphers from the competition.

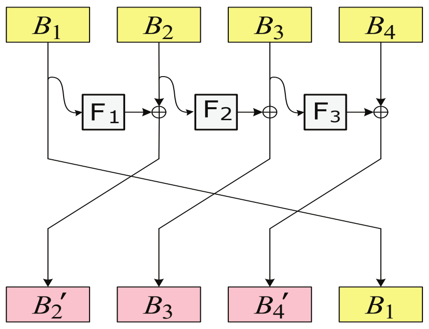


Figure 2.1: The type-3 Feistel scheme [1]

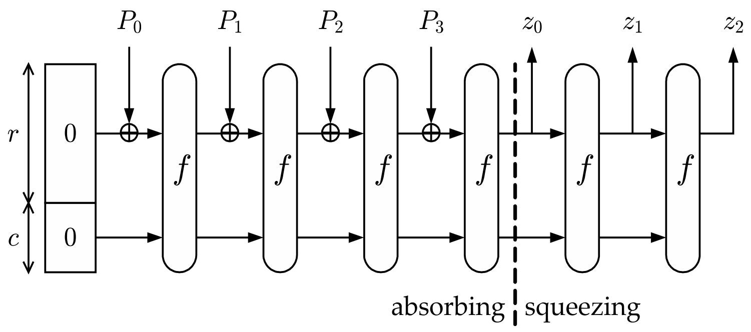


Figure 2.2: A sponge construction [2]

The first cipher chosen for analysis is Joltik [21], a block cipher based cipher. It uses a block cipher that is based on the TWEAKEY framework [22]. It has an AES-like round function, is hardware oriented with a low area footprint, and has 64-bit blocks with a key-size of 64, 80, 96 or 128 bit.

The second is Morus [23]. It uses its own dedicated function based on a Type-3 Feistel scheme to update its state, and is hardware oriented as well, using only shifts, ands and xors. Its internal state is either 640 or 1280 bits, and it has 128-bit blocks, with a key size of 128 or 256 bit.

The third and final cipher is Ascon [24], which is based on duplex sponge modes, similar to MonkeyDuplex [25]. The internal permutations use simple operations, that are easy and efficient to implement in hardware and software. These permutations work on a sponge state size of 320 bit. It has 64- or 128-bit blocks and a key-size of 128 bits.

Not many studies have been done on the energy consumption of the candidates, so the initial assumption of energy efficiency is based on the area footprint and the algorithm's complexity. The three algorithms have in common that they are fairly lightweight and should be fast and efficient in hardware. They have at least one set of parameters in common, so their performance can be compared easily. All three took different approaches for the core of their algorithm, so potentially interesting conclusions regarding energy usage of different cipher families can be made.

2.2 Joltik

Joltik [21] is developed at the Nanyang Technological University in Singapore. It is a lightweight authenticated encryption scheme oriented specifically at hardware applications. It has a small area footprint and performs well for short messages, because it has no initialization phase, using only n+1 block cipher calls for a n-block message. Making it suitable for low-power applications that only have to send short messages, like autonomous sensors.

Joltik is built around a tweakable block cipher called Joltik-BC, which is a particular instantiation of the general TWEAKEY framework [22]. Joltik-BC is an AES-based primitive, and can thus use and benefit from the extensive research on AES. This block cipher is used in modes based on OCB3 (the nonce-respecting mode) and COPA mode (the nonce-misuse resistant mode). The variants are quite similar. The focus of the thesis will be on the nonce-respecting mode (this means that a nonce can not be reused in different messages). It is more of interest for lightweight hardware, because it is the most efficient mode (it uses less calls to the Joltik-BC primitive), and the authors also recommend this mode [21]. A detailed illustration of this mode and the block cipher can be found in Chapter 5, the implementation chapter of Joltik.

This nonce-respecting mode has four different recommended parameter sizes (Table 2.1). Where the first two use a smaller version of the tweakable block cipher primitive (Joltik-BC-128), and the last two use a bigger version (Joltik-BC-192). The 64-64 (64-bit key and 64-bit block size), and 128-64 (128-bit key and 64-bit block size) versions are analyzed in this thesis. The 64-64 version is one of the smallest versions of the competition, and has the same key and block size as Ascon-128. The 128-64 version has the same key and block size as Ascon-128a and Morus-640-128 (section 2.4 and 2.3).

Name	**key-size**	**block-size**	**public message number size**	**tag-size**
$Joltik - 64 - 64$	64	64	32	64
$Joltik - 80 - 48$	80	64	24	64
$Joltik - 96 - 96$	96	64	48	64
$Joltik - 128 - 64$	128	64	32	64

Table 2.1: The recommended parameter sizes for the nonce-respecting modes of Joltik

The nonce-respecting mode offers full block security, beyond the birthday bound. So the 64-bit block size still offers 64-bit authenticity and integrity. It offers confidentiality up to the key size, more specifically 64-bit of confidentiality in the first version that will be analyzed, and 128-bit in the second version. This is the same security that AES-GCM offers. Joltik benefits from the research on AES and OCB3 to back up these claims.

To summarize, good features of Joltik are the high security relative to the parameters (integrity up to the block size and confidentiality up to the key size). The efficiency for small message, It is lightweight in hardware and software. Because it is based on AES for the block primitive and OCB3 for the nonce respecting mode, it can benefit from the extensive analyses and literature on these modes.

2.3 Morus

Morus [23] is also developed at the Nanyang Technological University in Singapore. It is designed to be fast in hardware by having a very short critical path, and is also very efficient in software. Due to its big state it has a relatively big area footprint. A long initialization and finalization phase makes it less efficient for short messages.

Morus uses a scheme similar to a type 3 Feistel scheme to update its state [18]. It has a multi-block state and updates using a number of rounds of a round function, generating a random keystream that is then XORed to the message to produce the ciphertext. It injects the message blocks into the state during the round function, to provide authentication as well. A detailed illustration of this structure can be found in Chapter 6, the implementation chapter of Morus.

Morus has three recommended parameter sizes. These are shown in Table 2.2. The first has a smaller state (640-bits) than the other two versions (1280-bit). The smallest version, with a 640-bit state, and 128-bit blocks, will be discussed in this thesis. The key size and block size of 128-bits is the same as in Joltik-128-64 and Ascon-128a (section 2.2 and 2.4).

Name	**key-size**	**block-size**	**public message number size**	**tag-size**
$Morus - 640 - 128$	128	128	128	128
$Morus - 1028 - 128$	128	256	128	128
$Morus - 1028 - 256$	256	256	128	128

Table 2.2: The recommended parameter sizes for Morus

This version offers 128-bit integrity, authenticity and confidentiality. Double the authentication security of the popular AES-GCM. These assumptions hold as long as the nonce is not reused.

To summarize, good features of Morus are its short critical path and easy operations in hardware. As well as its high authentication security (128-bits). It performs really good in software as well, but that is not in the scope of this thesis. A disadvantage is that Morus is not so efficient for short messages, having to run 16 times for the initialization and 8 times for the finalization.

2.4 Ascon

Ascon [24] is developed at the University of Technology in Graz, Austria. Its main goal is to facilitate easy implementation of side-channel resistance features, and to have a moderate size in hardware combined with a moderate speed. Placing it in the middle between Joltik and Morus, concerning speed and area. It has a short initialization and finalization stage.

Ascon is based on duplex sponge modes, similar to MonkeyDuplex [25]. But has a stronger initialization and key phase, which have more rounds compared to the data processing phase. Data is xored to a part of the internal state and this state then goes through a number of rounds, after which new data is added to the state. The ciphertext is also generated during this operation. A detailed illustration of this structure can be found in Chapter 7, the implementation chapter of Ascon.

There are two recommended parameters for Ascon, with the block size as the difference between the two (Table 2.3). Both versions have the same internal state of 320 bits, and they are also both analyzed in this thesis. Ascon-128 has a similar block and key size as Joltik-64-64, and Ascon-128a as Joltik-128-64 and Morus-640-128 (section 2.2 and 2.3).

Name	**key-size**	**block-size**	**public message number size**	**tag-size**
$Ascon - 128$	128	64	128	128
$Ascon - 128a$	128	128	128	128

Table 2.3: The recommended parameter sizes for Ascon

Both versions offer 128-bit confidentiality, integrity and authenticity. Similar to the Morus cipher. This only holds if the nonce is not reused.

To summarize, good features of Ascon are that it is lightweight in hardware and software, while still being reasonably fast. The same circuit that is used for encryption can be fully reused for decryption. The initialization and finalization stages are quite short, so it is still reasonably efficient for short messages.

2.5 Conclusion

The three ciphers that will be analyzed in the thesis were discussed in this chapter. They are all lightweight in hardware and have some parameters like key size and block size in common. Their internal structure are vastly different from each other, and their security goals also differ on some points. An overview is given in Table 2.4.

Their similar external parameters, but different internal structures, should make it possible to draw interesting conclusions when comparing their implementations.

Cipher	**Cipher family**	**Area**	**Speed**	**Message Overhead**	**Security**
Joltik	Blockcipher	smallest	slow	smallest	64-bit (128-bit confidentiality for Joltik-128-64)
Morus	Dedicated (based on type-3 Feistel)	biggest	fastest	biggest	128-bit
Ascon	Duplex Sponge Modes	small	fast	moderate	128-bit

Table 2.4: Features of the ciphers Joltik, Morus and Ascon

Chapter 3

Literature Review

This chapter contains the literature review of energy efficient FPGA designs. Both the implementation and the measurement of such designs are discussed. This research will serve as a basis for the implementations of the three ciphers described in the previous chapter.

Section 1 contains a summary of some general techniques used in the development of energy efficient FPGA implementations. It identifies some methods that are relevant for this thesis. *Section 2* studies FPGA implementations of similar cryptographic ciphers as those of the thesis. These implementations are mainly low-area implementations, since not much research has been done on energy-efficient implementations of cryptographic ciphers on FPGA. *Section 3* focuses on low level optimizations and coding practices that can be used to create energy efficient designs. *Section 4* researches methods to estimate and measure the energy of FPGA designs.

3.1 High Level Design Methodology

A difficulty in energy efficient FPGA design, is that the design space is really big [26]. Many different algorithms and architectures can be used to achieve the same functionality. FPGAs offer countless trade-offs and features, like the degree of parallelism, choice of memory, built-in blocks,... [27].

Research in [28] has shown that these high-level decisions have a big impact on the final energy use. The impact of optimization on algorithmic, register and circuit level are found to be respectively 20:2.5:1. Unfortunately these optimizations on algorithmic level are also the hardest, because a lot of time investment is needed to create or model the design in one alternative domain. Furthermore there are a lot of domains to compare. Still, because of the big impact of high level optimizations, a lot of methodologies for energy efficient FPGA design focus on these decisions.

One method is to first select all the *domains* with potential to be the most efficient design. Then a high-level model of these domains can be created, which can be modeled and its power usage can be estimated. With this information, the designer can then determine in which domains to focus his design and optimization efforts [26].

Another alternative is to identify the important parts of the design and determine a function that estimates their power usage in several different configurations. The full design can then be simulated by linking all these parts (called 'malleable algorithms' by the authors of [29]) together. This can be simulated, and the settings of each malleable algorithm can be tweaked, until the best global result is reached.

The most power intensive part of the designs in this thesis is the 'core' of the ciphers. For example, in the Joltik cipher this would be the block cipher 'Joltik-BC'. This core is not very big in complexity, which keeps the design space manageable. The viable versions of this core can be identified by studying energy efficient implementations of similar cryptographic algorithms. And since the number of alternatives remains managable, all these versions can be modeled or implemented in high detail. Which means their energy usage can be estimated using low-level tools like the *Xilinx power Estimator (XPE)* [30] or by direct measurements.

3.2 Hardware Implementations of Similar Cryptographic Ciphers

To have a basis for the implementations of the ciphers, implementations of similar ciphers can be studied. There has not been much research on energy efficient implementations of ciphers, especially for FPGA. Because of this, most of the focus of the reviewed implementations was on area. Some implementations of AES and Ascon are discussed in this section.

3.2.1 The AES Block Cipher

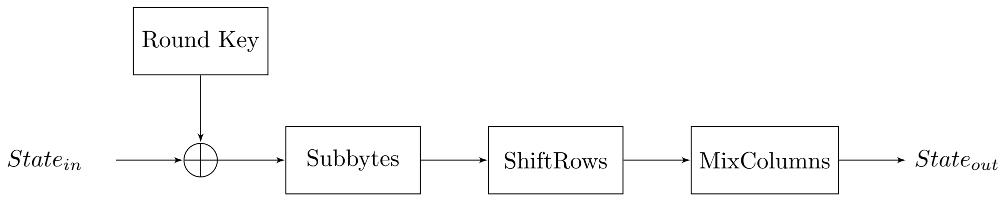


Figure 3.1: One round of the AES block cipher

AES [31] was announced in the year 2001, and is one of the most popular symmetric-key algorithm used today. It is very similar to the Joltik-BC block cipher used in Joltik [21]. A lot of research has been done on AES, so it is beneficial to look at some different FPGA implementations of it, to base the Joltik-BC design on.

The AES block cipher consists of rounds that are repeated 10, 12 or 14 times, depending on the version. Like most block ciphers with rounds, there are four popular approaches to implement it on FPGA or Application-specific integrated circuit (ASIC): iterated, pipelined, serialized and loop-unrolled architectures. In an iterated architecture the logic of a full round is implemented, and the processed block is cycled trough repeatedly. In the pipelined architecture there is logic of multiple rounds on the chip, with registers in between them, so multiple blocks can be processed at once. In the unrolled architecture, the logic of two or more rounds is executed in one clock cycle [32]. In the serialized architecture only a part of the round logic is placed on the chip, and a round is executed in multiple cycles by reusing it.

There is some parallelism in an AES-round, and it is possible to serialize a round of AES. A round can be relatively easily split in four, which mean one fourth of a round is implemented on chip and reused four times to execute a round. Some extra overhead is necessary to store some intermediary values, and to select the correct part of the state each round. To reduce the area even further, the FPGA logic elements (slices), can be configured as ram and used to store the state and the intermediary values. A big reduction in area is achieved this way [33].

The suboperations of a round of AES and those of a round of Joltik-BC are similar. Except for the keyschedule, the suboperations of an AES-round (displayed in Figure 3.1), are essentially bigger versions of those of a Joltik-BC round (which are described section 5.1.2). So there is benefit in looking at the implementations of these parts in AES ciphers. The Substitution-box (S-box) and the keyschedule are the most important parts that influence the design size [34]. On FPGA the S-box is often implemented in Lookup tables (LUT) [35]. And since the S-box of Joltik is a factor 8 smaller (4-bit inputs and outputs instead of the 8-bit in AES), this strategy is even more suitable for Joltik.

The subkeys of AES can be generated on the fly, or precomputed and, e.g. stored in the block ram of the FPGA. If the area is limited, it is generally better to compute the subkeys on the fly while the rounds are being executed [36]. The keyschedules of AES and Joltik-BC are different, but the same choice between precomputation or generation on the fly is available there. However since a big part of the key of Joltik-BC changes each block (the tweak part), only a limited part would be able to be precomputed.

3.2.2 The Ascon Cipher

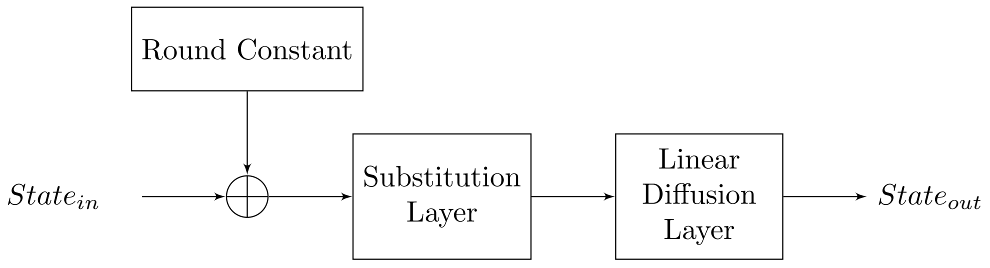


Figure 3.2: One round of the permutation used in the Ascon

Ascon is one of the three algorithms analyzed in this thesis (it is introduced in Section 2.4). Just like Joltik and Morus, the main part of the cipher is round-based (an Ascon round is displayed in Figure 3.2). And the hardware implementations make use of this round-based structure [37].

Just as in AES, an iterated and several unrolled implementations can be made (in [37] implementations with 1, 2, 3 and 6 rounds unrolled are made). The rounds are implemented in a straightforward way (first an addition of the round constant, followed by the S-boxes and the linear diffusion layer). The unrolled implementations are bigger in area, consume more power, but use slightly less energy per encryption. However the implementations were made for ASIC, so similar positive results on FPGA are not guaranteed.

The biggest parts of an Ascon round are the S-boxes and the linear diffusion layer. This second part can only be computed when the first part is done, so if a round is serialized, the S-boxes and the linear diffusion layer will have to be calculated in separate stages. These two parts can then be serialized to almost any degree with minimal overhead. In the most extreme case only one S-box is present, and the linear diffusion layer is calculated just one bit at a time. In this uncompromising low area implementation, one round takes 512 clock cycles, and the area is 35% of the area of the iterated implementation [37]. A compromise where, for example, 16 S-boxes are used, so 80 bits are calculated every clock cycle, is possible.

3.3 Low Level Energy Optimization's

After viable implementation versions of the ciphers are found, low-level optimizations can be applied on them. There have been some previous studies about these optimizations on FPGA, and some guidelines for energy efficient designs have been published. These will be discussed in this section.

The power consumption of an FPGA consists of two main parts: static and dynamic power consumption. The static power is mainly due to gate leakage and the complicated wiring in an FPGA [38]. This is not something the designer can influence, and it will not vary much between designs [39]. The dynamic power consumption is due to the switching activity, and the change of status of the wires. It can be improved with good design.

There are some general techniques that can be used to reduce the dynamic power consumption. One is to make use of the embedded blocks as much as possible, since they are optimized at the gate level instead of using less efficient LUTs. And a second technique is clock gating, to stop switching activity in certain parts of the design when it is not used. This last technique is not of much use in smaller designs [39].

Glitches also cause larger power consumptions. Glitches are unwanted switching activities that happen before a signal settles down to its correct value. They can be reduced by avoiding too long logic paths, for example by pipelining such paths. Rearranging the logic, can also help in some cases [39].

In [40], the authors try to find the link between the elements of the FPGA that are used in the design and the dynamic power consumption. They found that the power consumption depends linearly on the number of LUTs used if their switching activity is the same. Their measurements also confirmed that using embedded dedicated blocks is highly beneficial to reduce the power. Finally they found that the power also depends on the board. In their measurements, even identical models sometimes had a 10% different power consumption.

To have a better view of what parts of the FPGA will be used based on the written HDL code, it is beneficial to understand how an FPGA is built [41]. In Section 4.4, an overview of the structure of an Spartan 6, the FPGA on which the designs made in this thesis are analyzed, is described.

The tool used to synthethize and implement the design also has an effect on the power consumption. In [42], the effect of the optimization settings in Xilinx ISE on the power consumption of several cryptographic algorithms is measured. These settings include the optimization goal (either speed or area), the power reduction setting, and the optimization efforts. The best average improvement is around 10%, and AES achieves an improvement of 17% under the best settings.

3.4 Energy Estimation and Measurement

As mentioned in Section 3.1, the focus of the measurements and optimizations will be on a small part of the design, so low level estimation and measurement tools can be used. The energy can be estimated on fully completed designs, and no parts need to be modeled. The focus of this section is on these low level estimation and measurement methods. The energy consumption can be determined by using certain energy estimation tools or having a hardware setup to measure the power [42].

Two popular power estimation tools used for Xilinx boards are the Xilinx Power Estimator (XPE) and Xilinx Power Analyzer (XPA) [43]. These tools are easy and fast to use, because they are made to work together with Vivado or ISE, which is used to synthesize and generate the bitstream of designs on Xilinx FPGA's. XPE is used before the full HDL code is written, and XPA is used to analyze the design when the full HDL code is available. This last one is of interest for this thesis. Since the throughput per clock cycle of the implementations can be determined from simulation, the energy consumption can be deduced from the power consumption.

XPA uses the real design and parameters like the clock frequency, board voltage and the load on the output pins, to estimate the power consumption. In [42] the accuracy of this tool is analyzed for several cryptographic algorithms. They found the accuracy is highly dependent on the algorithm. The smallest error was 20% for their implementation of basic RSA and the biggest 190% for basic DES. For most algorithms, the error was under 50 %.

To measure the power consumption of the FPGA it is important to separate the voltage supplies of the different elements (like IO-ports and FPGA voltage) [44, 45], to measure only the current of the supply to the FPGA chip. By placing a small known resistor and measuring the voltage over it, this current can be found. This is the method used in a lot of commercially available boards [46, 45]. If the power supply is capable of measuring the current it supplies, this can be used as well.

These measurements also include static power and the power generated by supplying or generating the test vectors for the implementations. This power should be determined or reduced to get accurate and comparable measurements.

The static power can be determined approximately by measuring the power when the design is idle. This power is influenced by the state of the internal logic signals and the temperature of the board. However when the design is small this influence can be assumed negligible [44]. This assumption can be confirmed by measuring the same design at different frequencies and confirming that the dynamic power follows the frequency linearly.

To reduce the influence of the overhead on the FPGA that is used to test the design (communication over USB, testvector generation,...), some setups use two FPGA's. Their power consumption is measured separately and one of them handles all the overhead [46, 44]. If such a setup is not available, the designer can also use a setup that keeps the overhead on the FPGA to a minimum (for example by only supplying test vectors once and then reusing them).

3.5 Conclusion

This chapter contains the research background for the work in this thesis, which serves as a basis for the work described in the further chapters.

Some methods were found that will help with the general approach to the designs. A couple of different versions can be identified for each cipher and these can be implemented, measured and compared. Low area implementations of AES and Ascon were researched, which give direct ideas for the implementations in this thesis of Joltik and Ascon (since Joltik is very similar to AES).

After that, general coding techniques for energy efficient FPGA design were researched, the elements that lead to power consumption were identified, and the influence of several factors on the power consumption was analyzed. This includes previous research about the link between the power consumption and the number of used LUTs, the influence of long paths, the influence of using embedded blocks and the influence of the synthesis tool setting on the power consumption of some cryptographic algorithms.

The last section contains research about methods to measure and estimate the energy of the finished designs. The Xilinx Power Analyzer can be used to estimate the power consumption of the design, and is easy to use because it is built in into the synthesis tool. Its accuracy was examined for a number of cryptographic algorithms. Some ways to measure the power consumption of the FPGA were found, as well as methods to extract only the relevant portion of the power consumption (without all the overhead).

Chapter 4

The Implementation Approach

Based on the research from Chapter 3, and on experience achieved during the progress on the thesis, an implementation approach was formed for the implementations and measurements of the ciphers. The approach is described in this chapter.

In *Section 1*, the general approach for the implementation of the ciphers is described. The aim of having such a common approach is to get useful results. Concretely, this is achieved by developing the implementations in a way that their comparison is easier and more accurate.

One part of this, is the use of an API. This API and its advantages and disadvantages are described in *Section 2*. In *Section 3*, the strategy to implement the ciphers in this API is explained. The aim there is as well to facilitate the comparison of the ciphers and their implementations.

Section 4 presents the architecture of the Spartan-6 on which the implementations are placed and measured. Finally, in *Section 5*, the setup used to measure the energy consumption of the ciphers is explained. The physical setup and the measures taken to eliminate the inaccuracies introduced by several influencing factors are presented.

4.1 The General Approach

Multiple implementations of three different ciphers were made, totaling 24 implementations. Six of Joltik-128-64 and Joltik-64-64, three of Morus-640, four of Ascon-128, and five of Ascon-128a. The implementations were then compared with each other, and results from other research. The main goals of the comparisons are to compare the trade-off between the area and energy consumption for the individual ciphers, to determine the performance of the three ciphers, and to analyze how this trade-off between area and energy consumption compares over the different ciphers and cipher families. To make these comparisons more accurate, a common framework and strategy is used during the design of the ciphers.

This is visible in the structure of the implementations: the same external interface for the three algorithms is used (The GMU Hardware API described in Section 4.2), there is a common approach to the implementation of the core ciphers, and the different implementations of the ciphers only change the absolute core and leave the rest intact (this is explained in Section 4.3).

The implementations are written for the same FPGA, Spartan 6 (model xc6slx45 csg324-3), and the same measuring setup is used when measuring the power consumption of all the implementations. The implementations are made specifically with this FPGA in mind. Coding techniques that give good results on FPGAs are used (those are discussed in Section 3.3), and in some implementations intrinsic functions of the Spartan 6 are used to achieve a more efficient energy consumption or a lower area.

All implementations were written in VHDL, and they were simulated in Modelsim Student Edition [47]. Their functionality was tested using the published reference C-code of the ciphers together with the testbench provided with the GMU API. They were synthesized using Xilinx ISE [48], and their power consumption was measured with the Digilent Atlys board [49] (this setup is explained in Section 4.5).

4.2 The GMU Hardware API for Authenticated Ciphers

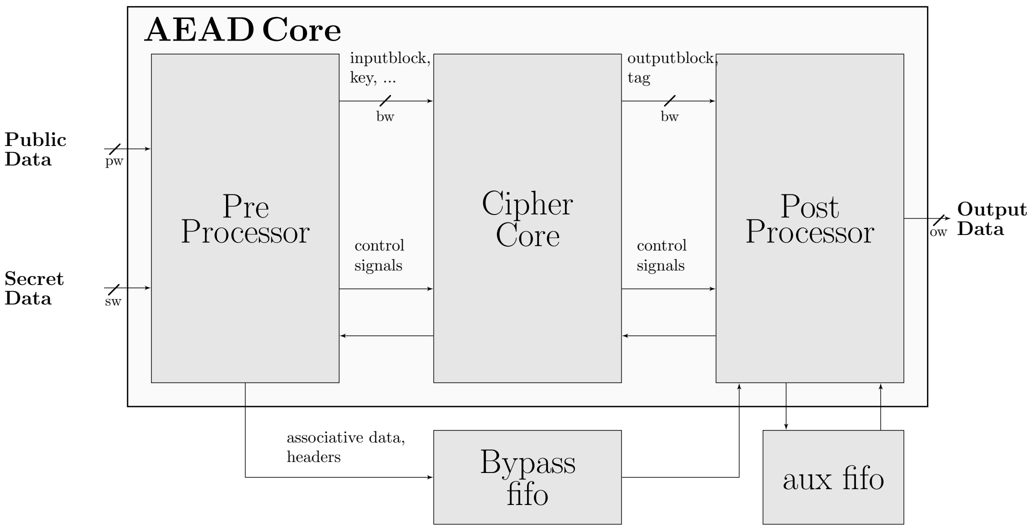


Figure 4.1: The structure of the GMU Hardware API

The GMU Hardware API, is a hardware API proposed to provide a common external interface for the hardware implementations of the ciphers participating in the CAESAR competition. It makes the comparison of different algorithms easier and fairer. The full specifications, features and the usage manual can be found in the API paper [50].

The GMU Hardware API separates the development of the *Core* (which is called *CipherCore*), containing the cipher specific part, and the external communication. One of the useful features is the support for a wide range of data port widths (ranging from 8 to 256 bytes), which are functionally completely separated from the CipherCore. Furthermore, it also supports an arbitrary length of the input stream. There is support for encryption and decryption with the same core. It is relative lightweight, and it can communicate with simple devices like FIFOs to use as memory.

Supporting VHDL code is provided, which fully takes care of external communication, the memory management and provides full width blocks to the CipherCore. A schematic is given in figure 4.1. The API uses a *PreProcessor* and a *PostProcessor*. The PreProcessor provides the key, public message number, secret message number, block data and tag to the CipherCore, with information about the expected operation. This indicates whether the data is AD or plaintext, whether the core has to encrypt of decrypt, if the current is the last block, and more. The PostProcessor then takes care of the output, accepting the crypted block from the CipherCore and delivering it to the output port.

In the implementations made in this thesis, the CipherCore is implemented for each cipher and fit in to this API. The area overhead introduced by the API is mainly determined by the size of the input words and the block size used by the ciphers. Since the three chosen ciphers have small block sizes (maximum 128-bit), and small word widths are used in the inputs (32-bit), this overhead in absolute terms is not that high. However, since the ciphers are all fairly small in area, the overhead in relative terms it is not negligible.

The implementations are verified by generating test vectors with Python and C-code provided with the GMU API. The test vectors are generated from the reference C-code of the ciphers, which had to be made available by the developers as part of the CAESAR competition [51]. These test vectors are then used by the test bench when simulating the implementation in Modelsim.

The energy measurements are done on a part of the CipherCore (see Section 4.3), because the overhead caused by this API is not of interest. When the ciphers will be used they will be integrated in a bigger design the majority of time, and not use this specific API. However the API forces the CipherCore of all ciphers to be structured a certain way, which makes the comparison of them more accurate. It is also useful to verify the correct functionality of the implementations with the API, and the structure makes it easier to understand and possibly reuse the implementations with different APIs in the future.

4.3 Approach for the Core

Now only the CipherCore has to be implemented to create a functional design. In the implementations in this thesis, one lower level of hierarchy is created. A kernel is placed inside the CipherCore that is a standalone entity that contains the *core* of the cipher. For example in the case of the Joltik cipher, this would be the block cipher Joltik-BC.

This kernel can function standalone, and has a start and busy signal so it can be controlled. The advantages of splitting the design like this, is that the majority of the area and energy will be consumed by this kernel, making it the focus of further

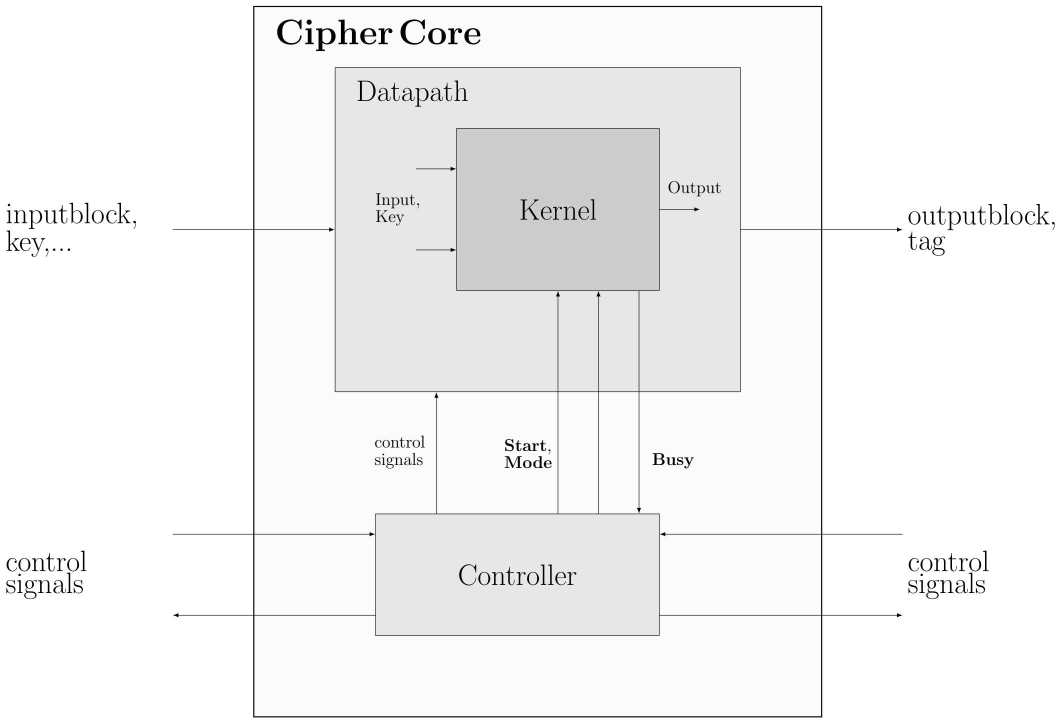


Figure 4.2: The structure of the implementations of the CipherCore

optimizations. The kernel can also be reused in other designs, even if the control signals to control the design are completely different from those in the GMU API. Everything outside the kernel can even be done in software, only using the kernel for hardware acceleration.

Some logic is then placed around the kernel to link it to the GMU API. It consists of a simple finite-state machine (FSM) and small datapath to convert the signals provided by the API to the signals required by the kernel. This structure is shown in Figure 4.2. The in- and outputs of the kernel are also visible on this figure. The different implementations of each cipher then only change this kernel, and keep the simple datapath and controller in the CipherCore the same. This kernel is the part that is energy optimized and measured, and its results are compared.

The implementations of the kernel use the round-based structure of the three ciphers, which is a common method to design these type of ciphers (see Section 3.2 in the literature review). The starting implementation of the ciphers makes one round fully combinational, and executes the cipher at the speed of one round per clock cycle. The other implementations then serialize or unroll this implementation in various degrees to find the energy-area-speed trade-off for the ciphers. The number of implementations are listed in Table 4.1. Especially for the serialized implementations specific optimizations can often be made.

Version	**Number of implementations**		
	Iterated	Serialized	Unrolled
$Joltik - 64 - 64$	1	2	3
$Joltik - 128 - 64$	1	2	3
$Morus - 640 - 128$	1	1	1
$Ascon - 128$	1	1	2
$Ascon - 128a$	1	1	3

Table 4.1: A summary of all the implementations made in this thesis

4.4 Spartan 6

FPGAs take advantage of the size and power efficiency of ASICs, while still providing flexibility in the form of reprogrammability. Making them cheaper and faster to develop on, but at the cost of worse specifications and cost-per-unit when made in large batches. Because of the exponential growth of transistor density through Moore's law, FPGAs have become feasible for more and more applications since their invention in 1982 [52].

The implementations in this thesis are made for a Spartan 6 FPGA, and the energy is also measured on this FPGA. It is the sixth generation FPGA in the Spartan FPGA Series, and was launched in 2009 [53]. The Spartan 6 is build in low-power 45nm technology, and is designed to be a cost-effective solution for a wide range of applications [54].

The specific model used in this thesis is the xc6slx45 csg324-3, which is a medium-sized Spartan 6, consisting of 6822 slices [55]. The most relevant specifications are listed in Table 4.2.

Device	**Slices**	**Flip-Flops**	**6-to-2 LUT's**	**Block ram**
XC6SLX45	6822	54576	27288	2088 Kb

Table 4.2: Features of Spartan 6 SLX45

The main logic resources for implementing the sequential and combinational logic are the Configurable Logic Blocks (CLB). In the Spartan 6 they contain two slices each, and each slice has four 6-to-2 LUT's and eight D Flip-Flops. There are three types of slices, where some can be configured as dedicated multiplexers, distributed ram or shift registers [41]. This can greatly reduce the resources needed for these elements. This is shown in Table 4.3.

These configurations can be inferred automatically from the VHDL code during synthesis, or can be initialized specifically using Xilinx's UNISIM package [56]. In one of the implementations of this thesis this is used to create 16-bit wide, 4-bit deep, distributed rams in just two slices.

Xilinx provides some guidelines about when elements should be manually initial-

Feature	SliceX	SliceL	SliceM
Occurence	50%	25%	25%
6-input LUTs	✓	✓	✓
8 Flip Flops	✓	✓	✓
Wide Multiplexers		✓	✓
Distributed RAM			✓
Shift Registers			✓

Table 4.3: Types of slices in the Spartan 6

ized and when they should be left to be inferred by the synthesis tool, and they give some rules of thumb to decide what element is worth using [57]. An example of this are guidelines about when to use block ram and when distributed ram. In Table 4.4 the resources used for the special elements are shown.

Element	Slices	LUT
6-to-2 LUT	n.a	1
7-to-2 LUT	n.a	2
8-to-2 LUT	n.a	4
256x1-bit distributed RAM	1	n.a
32x8-bit distributed RAM	1	n.a
64x4-bit distributed RAM	1	n.a
32-bit shift register	n.a	1
1-bit 8-to-1 multiplexer	n.a	2
1-bit 16-to-1 multiplexer	n.a	4

Table 4.4: The resources used when SliceL and SliceM are put in special configurations

Knowledge of this structure will make it easier to apply some energy saving methods from the literature review to the implementations. For example the number of LUTs used can be more easily predicted, which correlates to a lower power consumption.

4.5 The Measuring Setup

The first measurement setup that was tried, was the SAKURA-G board [46]. This setup did not work out as planned, so then the Digilent Atlys board [49] was used. Only the kernel is measured on the boards (see Section 4.3). The two setups are explained in this section, together with the measures taken to reduce influencing factors like the overhead from supplying the test vectors, and the temperature. The reasons why the SAKURA-G board was insufficient are discussed as well.

Since the final setup is easy and fast to use, there is no need to use energy estimation programs to determine the energy. These estimations can be really

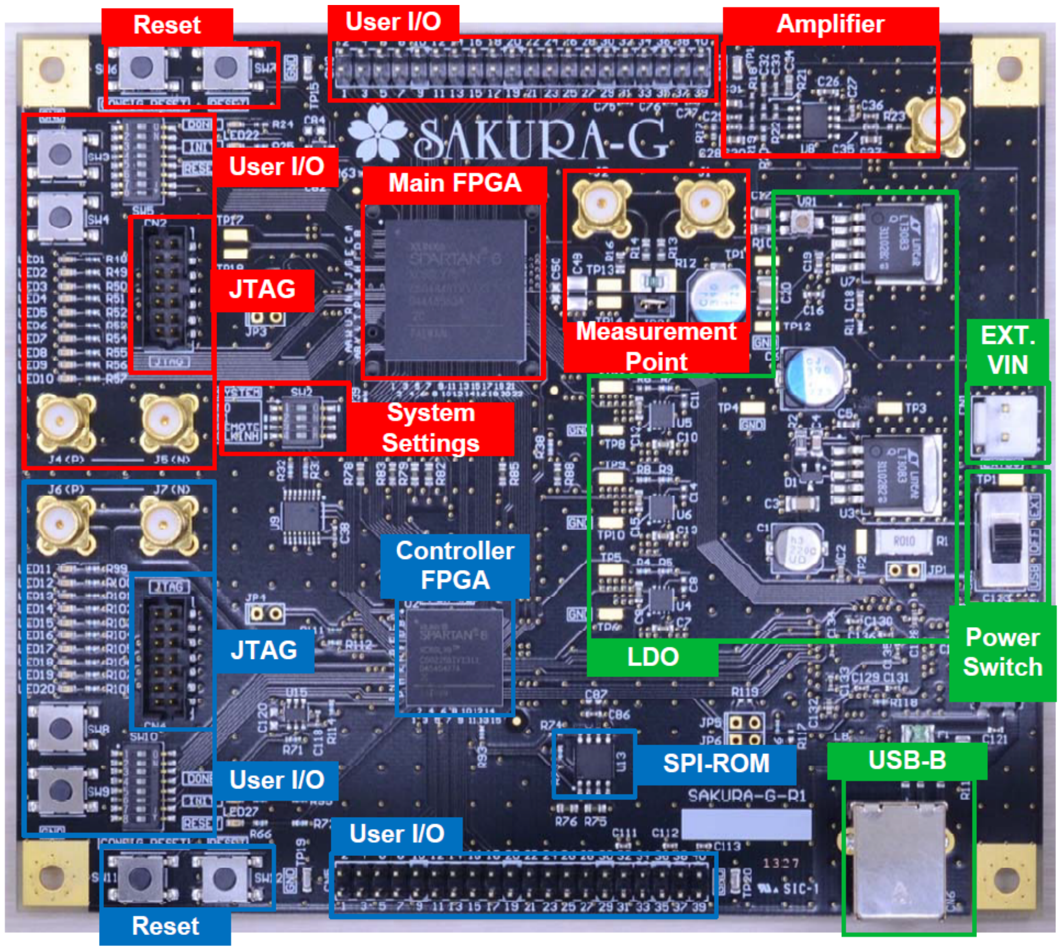


Figure 4.3: The SAKURA-G board

inaccurate (see Section 3.4) so it is better to measure the power immediately if that does not take too much time.

4.5.1 SAKURA-G Measurement Board

The SAKURA-G was developped at the Satoh Laboratory of the University of Electro-communications in Tokyo, in cooperation with the Morita Tech Coorperation in Taiwan [58]. It is designed to be used during R&D on hardware security, such as research on Side-Channel Attacks (SCA), Fault Injection Attacks (FIA), and Physical Unclonable Functions (PUF) [46].

The board is shown in Figure 4.3. It has two FPGAs, one main FPGA where the cipher is rUn, and one controller FPGA that will supply the test vectors to the main FPGA. Both FPGAs are Spartan 6's. The power consumption of the main FPGA is measured at the measurement point, where a small known resistor is placed between the main FPGA power supply and the main FPGA. The voltage over this resistor is measured to determine the power consumption (see Section 3.4). There is an amplifier on the board, but this only amplifies the alternating current (AC) power, while to determine the power consumption, the direct current (DC) power is needed. So this amplifier can not be used.

A small FSM was then placed on the controller FPGA. It can accept test vectors through a USB connection (using the RS-232 standard [59]). And it passes these to the main FPGA, which then activates one of the user I/O pins and starts the cipher. The cipher is repeated a number of times (in the order of thousands), and the output is always fed back into the input. Because the ciphers have the property that their output approximates a random distribution of bits, this ensures a random selection of inputs. The voltage is measured with an Agilent 6000 Series Osciloscope [60], that uses the pin that is activated when the cipher starts running, as a trigger pin.

Unfortunately the DC measurements were very inaccurate and it was not possible to get meaningful results out of this setup. The amplifier gave good outputs, but these AC measurements were not enough to get an accurate power consumption. That is why this setup was abandoned, but some elements were kept in the next measurement setup.

4.5.2 Digilent Atlys Measurement Board

After the SAKURA-G board measurements were not satisfactory, the Digilent Atlys measurement board was used. It is a ready-to-use development board which can measure the power consumption of the FPGA on the board when used with the Digilent Adept software [49].

The board is shown in Figure 4.4. It has only one FPGA, a Spartan-6 SLX45 with speed-grade 3. The current and power of the 3.3V, 2.5V, 1.8V and 1.2V supplies are measured separately (the 1.2 Voltage is the Voltage to the FPGA). This current is sampled at 5 Hz, and the margin of error is about 1%.

The main disadvantages of this board, compared to the SAKURA-G board, is the slow sampling frequency (the sampling frequency of the SAKURA-G is determined by the used oscilloscope and can be very high), and that there is only one FPGA. To offset the first disadvantage, the cipher can be run millions of times, and the measurement results will reflect the average power consumption of the ciphers. To offset the second disadvantage, a setup needs to be made that minimizes all the overhead on the FPGA. And this should be checked and confirmed through measurements.

A small FSM is placed on the FPGA that accepts test vectors through USB and supplies them to the cipher (on board). Just as in the SAKURA-G setup the output of the cipher is fed back to its input each time the cipher is ran. The cipher is then run millions of times (the number of times it should run is also passed through the USB connection). To minimize overhead, this small FSM stays in one state while the cipher is running, and almost no signals apart from the cipher are updated. It also pauses the FPGA for two seconds before and after running the cipher.

Some C-code that does the same operations that are performed on the FPGA was written. The final output of the ciphers on the FPGA is written back over the USB connection and compared with the result of the C-code. A couple of short runs

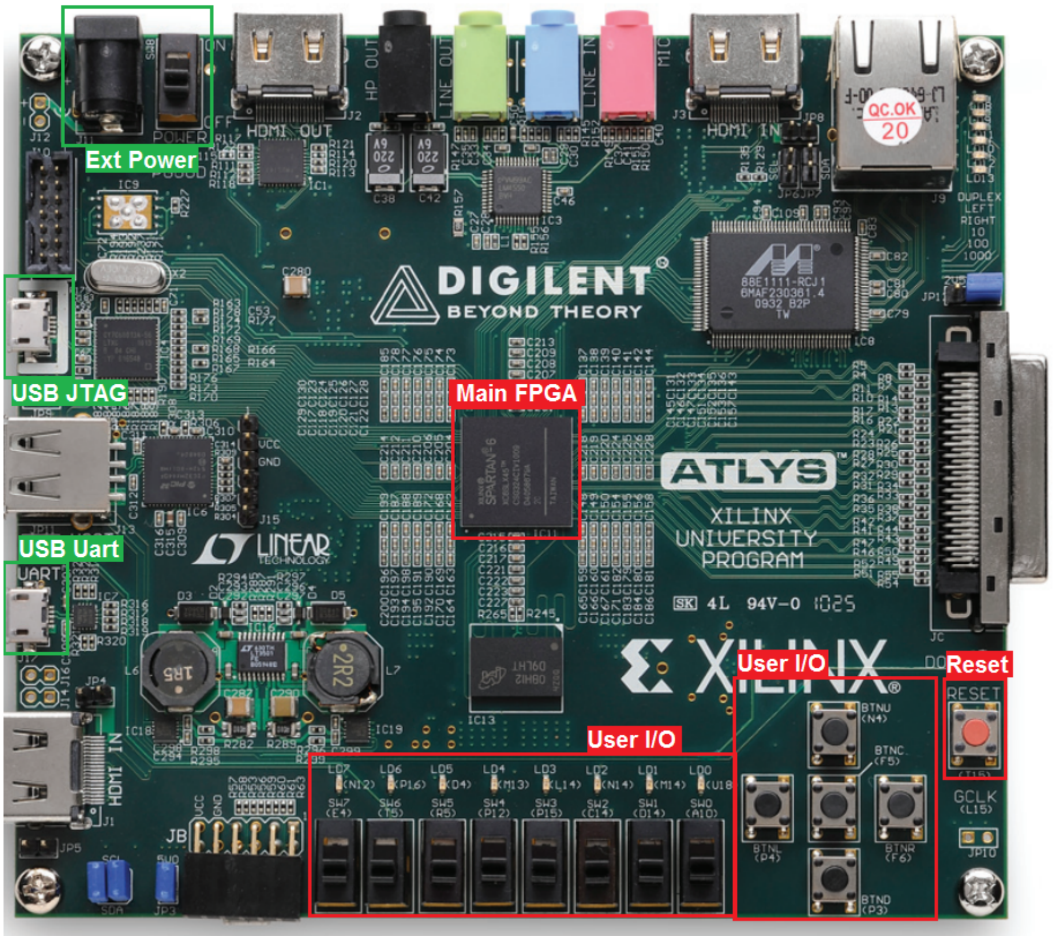


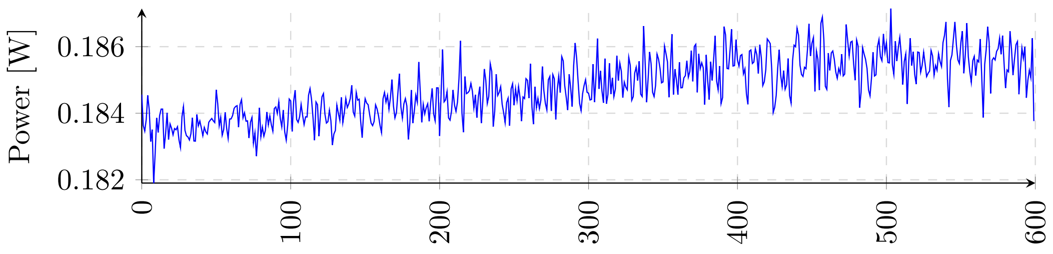
Figure 4.4: The Digilent Atlys board

test if the cipher functions correctly on the FPGA before the main measurements start. These main measurements can not be verified directly because the millions of runs on the FPGA would take hours to verify with the C-code.

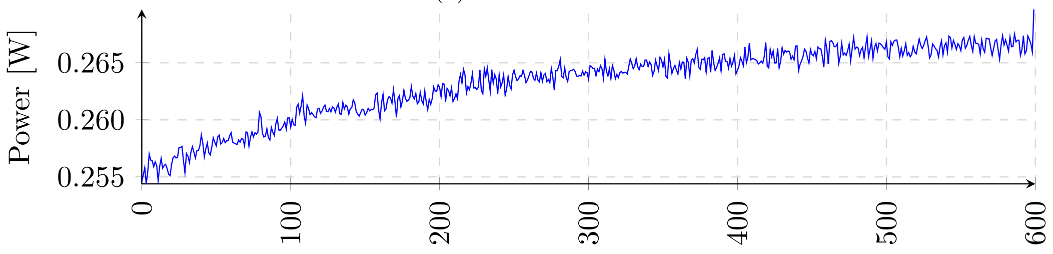
The static and overhead power are measured when the FSM is in an idle state. This is then subtracted from the measured power during operation. To confirm that this setup is correct, the ciphers are run at multiple frequencies and it is confirmed that the dynamic power follows a linear curve proportional to these frequencies.

There are between two and four clock cycles between the end and the start of a new run of the cipher. The exact amount of cycles one run takes, and these overhead cycles, are determined through simulation of the implementations in Modelsim. Together with the measured power, the energy consumption can be calculated.

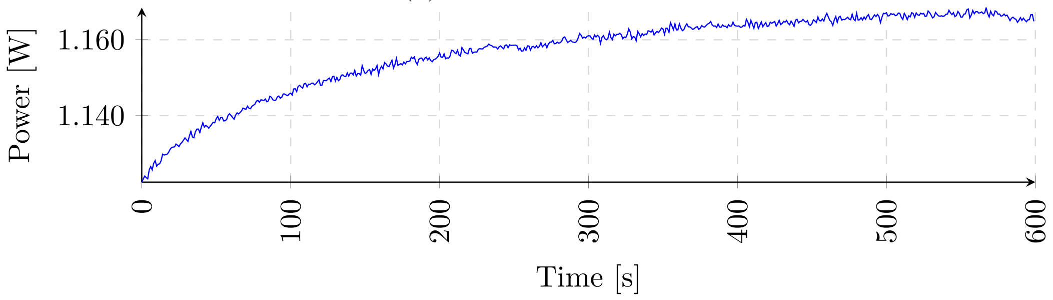
Since the power consumption of the ciphers is typically low, the FPGA does not heat up much. So the influence of heat is expected to be minimal. In Figure 4.5 the evolution of the power consumption over 10 minutes is shown when the FPGA is started from a cold start. The effect of heat is not completely negligible, especially in



(a) In idle state



(b) In moderate load



(c) In heavy load

Figure 4.5: Evolution of the power consumption when running Joltik-128 for 10 minutes

the most power hungry implementation. But if the operations are limited to under a minute they are still accurate, and special care for this will be taken during the measurements.

4.6 Conclusion

This chapter described the approach that is taken to implement the ciphers and how their structure will look. The three next chapters will then list the implementations and measuring results for the three chosen ciphers.

A hierarchy of the structure is determined for the implementations. An API is used to provide the external interface, and only a part where the cipher is placed (called CipherCore) has to be developed. The CipherCore will contain an additional layer of hierarchy, with a kernel that contains the core of the ciphers. This kernel is changed in each implementation, and the rest of the CipherCore is only changed for each cipher.

The approach to the implementation of these kernels is also discussed. They make use of the round-based structure of the ciphers, and different implementations of the same ciphers are implementations where one round is either serialized or unrolled to different degrees. This will facilitate finding the area-energy trade off for the ciphers.

Then the architecture of the Spartan-6 on which the implementations are placed and measured is presented. Knowledge of this structure will help apply the energy saving methods from Section 3.3. Some intrinsic functions of the FPGA can be used in the optimized implementations.

Finally the measurement method is discussed. It is easy and fast to use, making immediate measurement possible, and eliminating the need use inaccurate energy prediction programs. The measurements are done on an Atlys Digilent board, which can measure the power consumption of the FPGA separately from the consumption of other elements on the board. Some measures are also taken to determine and reduce the influence of external factors on the measurements.

Chapter 5

The Implementation of Joltik

Now that the implementation approach is determined, the implementations themselves are made. This chapter discusses the implementations of the first cipher, Joltik. The area, energy, and throughput of these implementations is measured and compared to each other. In the next two chapters the same will be done for the two other ciphers Morus and Ascon.

In *Section 1*, the specifications of the Joltik cipher are explained in more depth. This is important to understand the implementations, which are presented though Sections 2 to 4. In *Section 2*, the iterated implementation is described, which is also the reference implementation that the other implementations modify. In *Section 3*, the two serialized implementations are described, and in *Section 4*, the implementations with multiple unrolled rounds are discussed.

In *Section 5*, the most important measuring results are shown. The more detailed results can be found in Appendix A for reference. In *Section 6*, these results are discussed.

5.1 The Structure of Joltik

As mentioned in chapter 2.2, Joltik has a nonce-respecting and a nonce-misuse resistant mode. Both modes use the same block cipher primitive *Joltik-BC*, but differ in how they process the output from this block cipher. In this thesis, the nonce-respecting mode is implemented.

This section, presents the specification of Joltik. The structure of the nonce-respecting mode is explained first, followed by the structure of the tweakable block cipher Joltik-BC.

5.1.1 The Nonce-Respecting Mode

A diagram of an encryption in that nonce-respecting mode is shown in Figure 5.1. Confidentiality is achieved by a single encryption of each plaintext block. If the message does not fit perfectly in the 64-bit blocks, the last block will padded and processed in a slightly different way (this is visible on Figure 5.1). The padding is done by adding a single '1' after the last message bit, and then filling the rest of the last block with zeros.

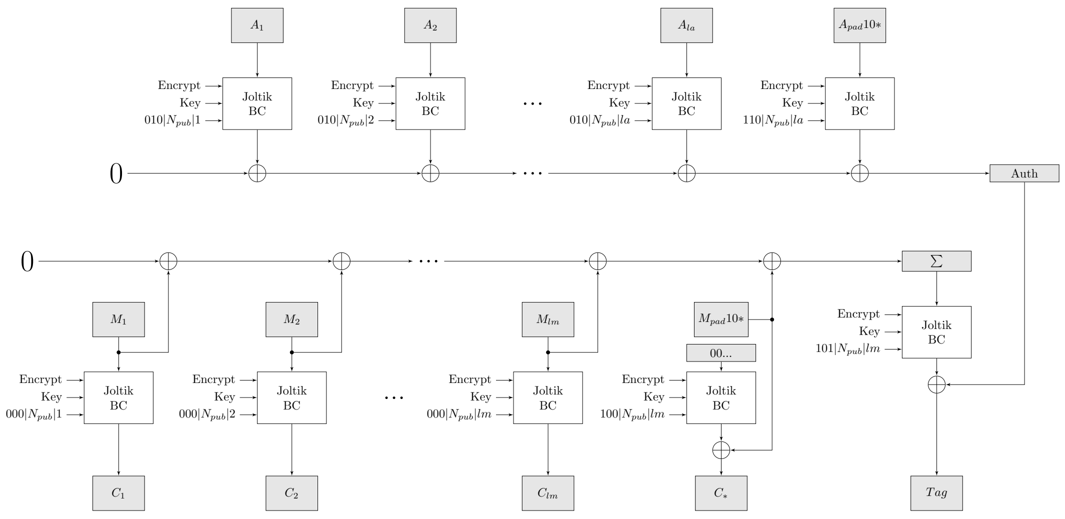


Figure 5.1: High level diagram of the nonce respecting mode of Joltik

The AD is also put through the block cipher, and the outputs are XORed together in to an authentication register. Together with a checksum of all the plaintext blocks, followed by another encryption with Joltik-BC, the tag is generated. This tag is added at the end of the ciphertext and is used to achieve integrity and authenticity of the message.

In each call to Joltik-BC a key, tweak, input and the mode are given. The key stays the same for the whole encryption or decryption of a message. However the tweak is changed on every call, and is composed of a 3-bit *tweak preamble*, the public nonce, and a block number. The sizes of these are variable, and are displayed in Table 5.1 for the two implemented Joltik versions. The mode is either encryption or decryption. Since the size of the block number is variable, the maximum length of a message differs between the versions of Joltik.

Name	**key-size**	**tweak-size**	**public message number size**	**blocknumber size**	**Joltik-BC version**
$Joltik - 64 - 64$	64	64	32	29	Joltik-BC-128
$Joltik - 128 - 64$	128	64	32	29	Joltik-BC-192

Table 5.1: The sizes of the parameters for the analyzed versions of Joltik

When Joltik is used in decryption mode, the AD is processed in the same way as for encryption, but the ciphertext is put through Joltik-BC in decryption mode, and the checksum is generated from the output. The resulting decrypted message should only be released when the provided tag is verified to be correct.

The time overhead for short messages is minimal, only a padding to a multiple of 64 bits and a single encryption to generate the tag is needed. Regarding resources, at least two additional 64-bit registers are needed for holding the authentication and checksum tag.

5.1.2 Joltik-BC

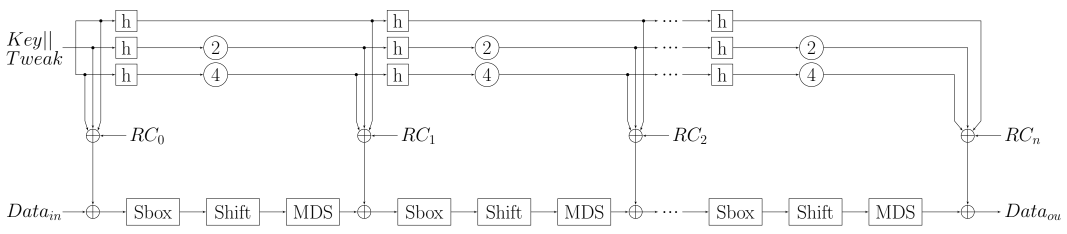


Figure 5.2: The rounds of the block cipher Joltik-BC-192

The tweakable block cipher Joltik-BC is very similar to the AES block cipher. It consists of AES-like rounds [61], with the biggest difference in the key schedule, and the state-size (64-bit blocks). The key schedule takes a key and a tweak as input to create the subkey. The subkey is then XORed to the state, and the result is modified in a number of steps. Depending on the version of Joltik-BC, this round is repeated 24 or 32 times. The subkey is also updated every round. There are two versions of Joltik-BC, which differ only in the length of the key schedule and the number of rounds (Table 5.2). The key-schedule input has the sum of the lengths of the key and tweak.

Name	**key-schedule input**	**state-size**	**number of rounds**
$Joltik - BC - 128$	128	64	24
$Joltik - BC - 192$	192	64	32

Table 5.2: Parameters for the two different versions of Joltik-BC

In Figure 5.2, the structure of a round is shown. The top part displays the key schedule and the bottom part displays the steps of a round. The Joltik-BC-192 version is shown in the figure, Joltik-BC-128 has the same structure except that the bottom part of the key schedule is not there. All the operations are done on 64-bit blocks, consisting of sixteen 4-bit *nibbles*, which are ordered as shown in Figure 5.3.

$$\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

Figure 5.3: Ordering of the 64-bit state of the block cipher Joltik-BC

The S-box operations consist of 4-bit S-boxes applied to each nibble. It is followed by a shift operation that changes the relative position of the nibbles in the 64-bit state. After that, the state matrix is multiplied by a Maximum Distance Separable (MDS) matrix [62], which mixes the columns of the state matrix with each other.

The keyschedule splits its 128 or 192-bit state in respectively two or three 64-bit parts. These are also ordered according to Figure 5.3. It updates by first changing the relative position of the nibbles through a permutation h, and then multiplying the nibbles by 1, 2 or 4. All operations are done in the Galois field, $GF(2^4)$. The exact details can be found in the submission paper of Joltik [21].

During decryption, these operations are performed in the opposite order. The S-box and shift operations are replaced by respectively, an inverse S-Box and inverse shift. The MDS matrix does not change because $MDS = MDS^-1$.

5.2 The Iterated Implementation

In this section, the iterated implementation of Joltik is described. This was also the reference implementation and it determines the base structure for the other implementations of Joltik in this thesis.

As described in chapter 4, the cipher is implemented with the GMU hardware API. The developer only has to implement the CipherCore, which contains the cryptographic cipher. The API handles the external interface and memory storage. A separate kernel is then made in this CipherCore that contains the block cipher Joltik-BC. This kernel is placed in the CipherCore, and a datapath and control FSM link the control and data signals of the GMU API to the kernel.

After an initial optimization of the implementation, the kernel is the part of the implementation that is further optimized for energy, and the part that will be adapted in the other implementations of Joltik. As mentioned in chapter 4, this will be the part that consumes the majority of energy, and its round-based structure can be used to build unrolled and serialized implementations to analyze the area-energy trade-off.

First, a short summary of the functioning of the GMU API is given, followed by an explanation of how the kernel is linked to this API, and finally the iterated implementation of the Kernel is described.

5.2.1 The GMU API

As can be seen in Figure 4.1 from the previous chapter, the API consists of a PreProcessor, a PostProcessor, some external memory, and of course the CipherCore itself.

The PreProcessor accepts user commands through two ports. A secret port, where the secret key is provided, and a public port where the data (like the public nonce, AD and plain- or ciphertext) and mode (encryption or decryption) are provided. It passes the key and the data to the CipherCore, with the plaintext and AD provided in blocks. The connections are full bit-width. Additional control signals are used to specify the operation that should be performed by the CipherCore. The PreProcessor can perform some supporting operations, such as padding the input blocks.

The PostProcessor accepts the output from the CipherCore, and stores the output of the decryption in the auxiliary FIFO until the tag is verified. It also provides the tag to the CipherCore in the last stage of decryption, or accepts this tag from the CipherCore during encryption.

The bypass FIFO is used to pass data from the PreProcessor to the PostProcessor that does not need to pass through the CipherCore. This is mainly the AD and the tag.

Several generics can be set in the VHDL code of the GMU API. With these generics the API can be customized to fit every cipher. This includes setting the block size, padding method or disabling the use of a public or secret message number.

5.2.2 The CipherCore

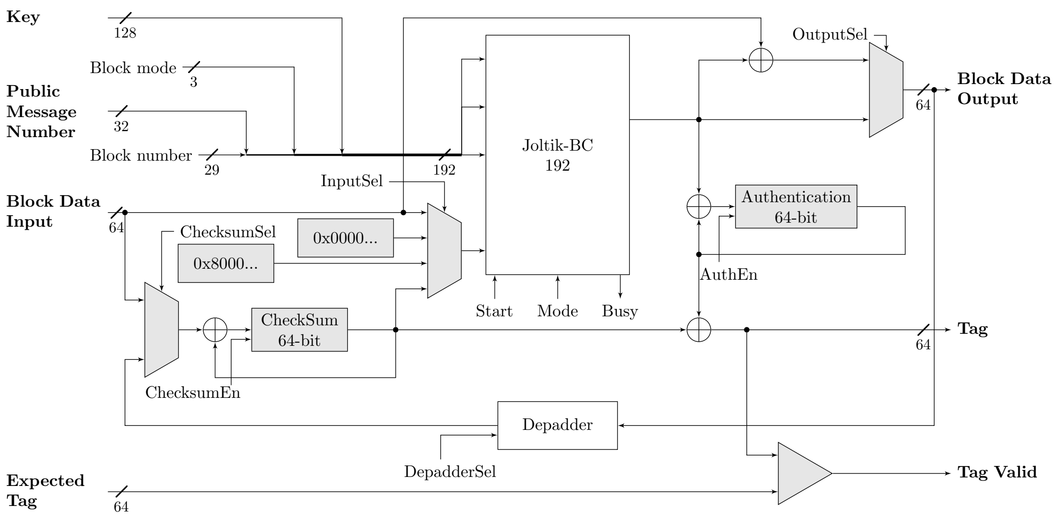


Figure 5.4: The datapath of the CipherCore of the Joltik

In Section 5.1.1, an overview of the Joltik cipher, and thus the functionality of the CipherCore of the Joltik implementations, is given. The block cipher *Joltik-BC* is implemented in a separate kernel with its own *start* and *busy* signal. This structure is shown in Figure 4.2.

A small datapath and FSM are needed to connect the GMU API to the kernel. These are displayed in Figure 5.4 and 5.5. The datapath of *Joltik-128-64* is displayed. The only difference with *Joltik-64-64* is the size of the key, tweak, and the version of the block cipher Joltik-BC.

The datapath contains an authentication and checksum register, needed to store the temporary values for generating the authentication tag in the end. Some additional multiplexers and constants are needed to be able to do the encryption and decryption for all cases. For example, if there is no AD, an empty padded block is encrypted before the message encryption is started. The datapath further connects the data signals given by the Preprocessor and PostProcessor to the Joltik-BC kernel.

The datapath is controlled by a small FSM (Figure 5.5). It only uses control signals from the Pre- and PostProcessor as input, and outputs certain control signals

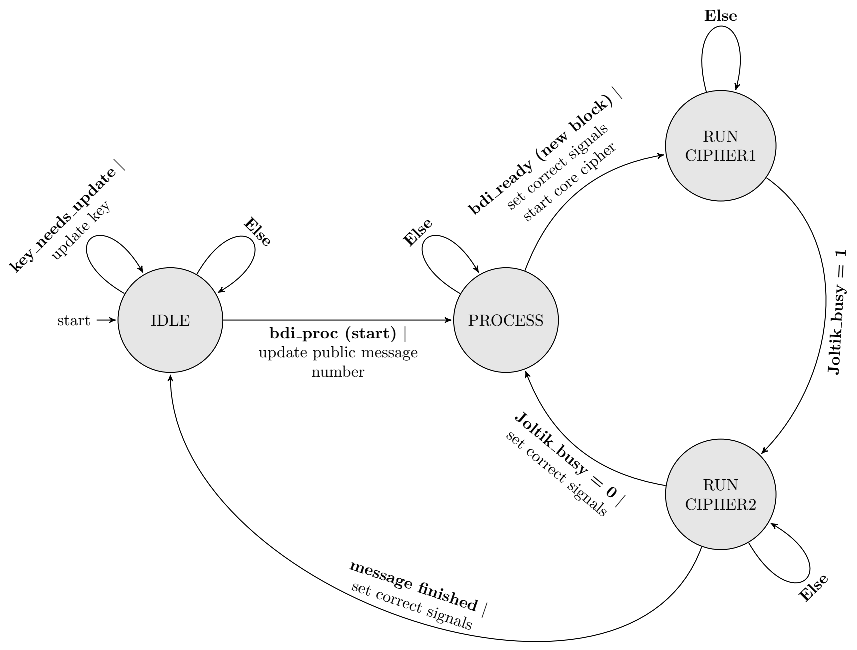

Else
RUN CIPHER1
bdi_ready (new block) | set correct signals start core cipher
key_needs_update | update key
Else
Else
start
IDLE
bdi_proc (start) | update public message number
PROCESS
Joltik_busy = 1
Joltik_busy = 0 | set correct signals
RUN CIPHER2
Else
message finished | set correct signals

Figure 5.5: The Finite State Machine of the CipherCore of the Joltik

back to them, as well as to the CipherCore datapath.

The controller waits in the *IDLE* state for a key update, or for a start signal (bdi_proc). The next state is the PROCESS state. There the controller waits until a block is ready to be processed (which is indicated with the control signal bdi_ready) and decides, based on other control signals from the API, how the next block should be processed. The processing is done in the *RUN_CIPHER* states. It is stored what type of block is being processed so the correct control signals can be passed to the Joltik-BC kernel, the CipherCore datapath and the GMU API.

The communication with the kernel happens with a handshake, which uses the *start* and *busy* signal of the kernel.

5.2.3 The Kernel

The kernel contains the block cipher Joltik-BC and will consume the vast majority of the energy and area. In Section 5.1.2 an overview of the structure of this block cipher is given.

The kernel takes a key, input data, mode (encryption or decryption), and a start signal as input. It outputs the output data and a busy signal indicating when it has finished encrypting or decrypting. The datapath and FSM of the iterated implementation of Joltik-BC-196 (used by $Joltik-128-64$) are displayed in Figures 5.4 and 5.5. Joltik-BC-128 (used by $Joltik-64-64$) is very similar, but the key schedule is only 128 bit big, instead of 192 bit. In the iterated implementation of Joltik, one round is calculated every clock cycle.

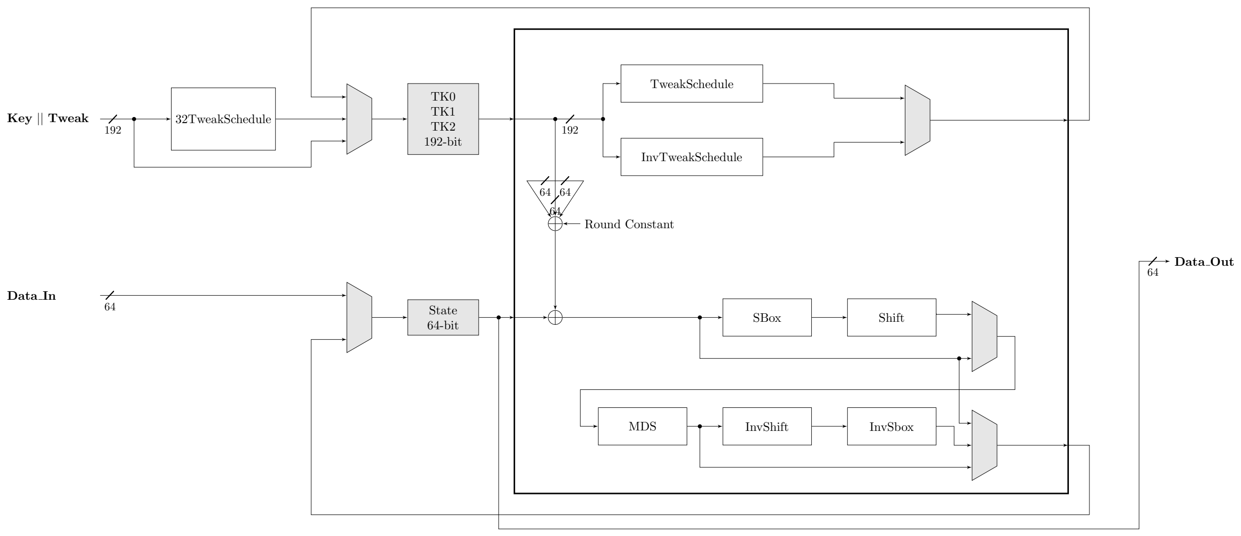


Figure 5.6: The Datapath of the iterated implementations of the kernel of Joltik (Joltik-BC-196)

The datapath has 192 bits of registers to store the subtweakeys (128-bit in the Joltik-BC-128 implementation), and a 64-bit register to store the state and final output. The subtweakeys are 64-bit values that can be combined to form the subkey of a round. The output of these registers passes through a hardware implementation of a full round every clock cycle, and the result are fed back to the input of the registers. The subkeys are not calculated beforehand, and are also updated every clock cycle. This is done to conserve area and keep the complexity of the design low (see Section 3.2.1).

The implementation is capable of both encryption and decryption. A multiplexer decides whether the encryption path (where the S-box is applied first, followed by the shift, and then the MDS multiplication), or decryption path (first the MDS multiplication, then the inverse shift, and then the inverse S-box) is followed. Another multiplexer decides whether the subtweakeys are updated for encryption or decryption.

During decryption, the starting subtweakeys are different than during encryption. The function of the *32tweakschedule* is to convert the input subtweakeys to their version 32 rounds later, so they can be used as the starting subtweakeys in the decryption mode. Because the operations mostly cancel themselves out in 32 cycles, the critical path stays manageable. The *32tweakschedule* is applied prior to the start of the rounds and makes the cipher take one additional clock cycle, for a total of 33 clock cycles per data block. This is the function of the state *STORE INPUTS* in the controller (Figure 5.7).

As described in Section 5.1.2, all sub operations are done on a matrix of sixteen 4-bit nibbles, and the operations individually either change the relative position of these nibbles, or they change the nibbles themselves. This change of the relative position is implemented by routing the 4-bit nibbles, and their value is changed by

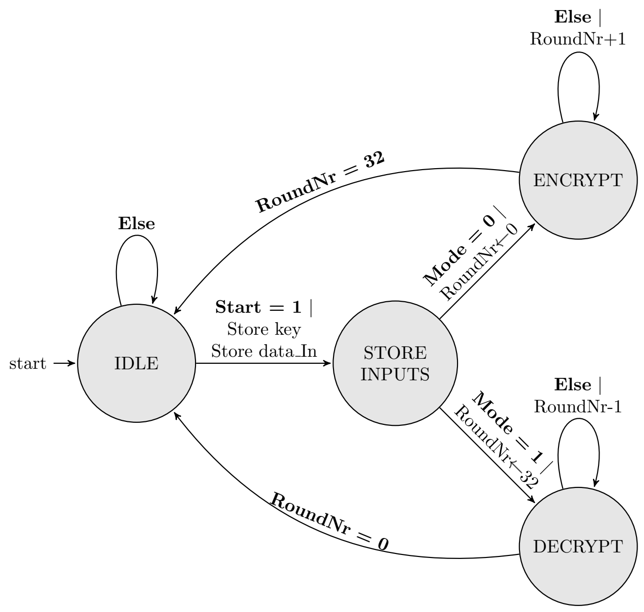


Figure 5.7: The Finite State Machine of the iterated implementations of the kernel of Joltik (Joltik-BC-196)

passing the nibbles through LUTs. In Table 5.3 this is listed for every subpart. The LUTs on the FPGA are 6-input, 2-output LUTs (see Section 4.4). The synthesis tool will combine the smaller LUTs in the HDL code to fit in the LUTs of the FPGA.

Part	2-to-1 LUT	3-to-2 LUT	4-to-4 LUT	Complicated interconnections
32TweakSchedule	0	0	16	
TweakSchedule	16	16	0	✓
InvTweakSchedule	16	16	0	✓
Sbox	0	0	16	
InvSbox	0	0	16	
Shift	0	0	0	✓
InvShift	0	0	0	✓
MDS multiplication	16	32	16	

Table 5.3: The LUTs and wiring used for the sub parts in the iterated implementation of the Joltik kernel (Joltik-BC-196)

5.3 The Serialized Implementations

In this section, the serialized implementations of Joltik are described. Compared to the iterated implementation, only the kernel is changed in these implementations. The description of the GMU API and the non-kernel part of the Ciphercore are therefor not repeated in this section.

First, the standard serialized implementation is presented, where a round is split in four parts, followed by an implementation where this is adapted to use distributed ram instead of registers to store the internal state.

5.3.1 The Standard Serialized Implementation

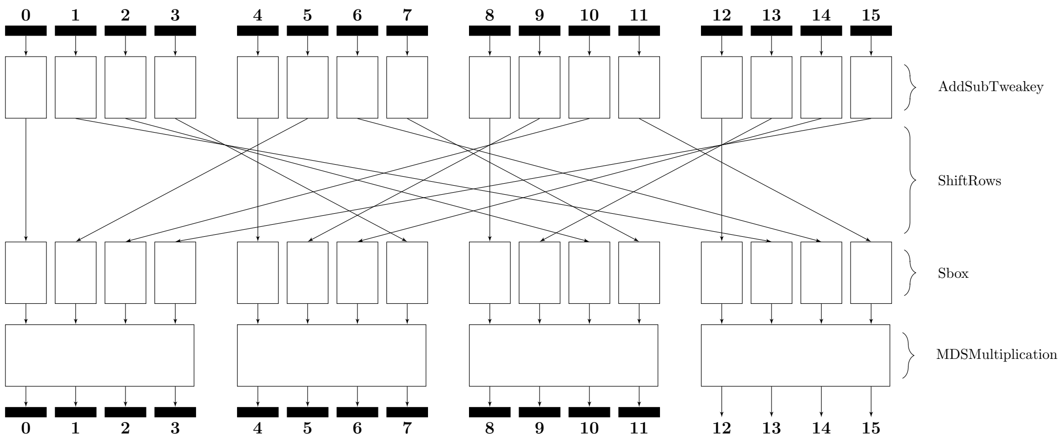


Figure 5.8: Parallelism in an encryption round of Joltik-BC

In the iterated implementation (see Section 5.2.3), one round was unrolled, so one round of the cipher was executed every clock cycle. When a round is analyzed, it is visible that it has a lot of parallelism. Its structure can be *split in four parts* relatively easy. Only a reordering of the *S-box* and *shiftrows* operations is necessary. This is possible because the S-box operates on each 4-bit nibble separately and the shiftrows operator only changes the order of the nibbles. The structure for encryption is shown in Figure 5.8. The idea is based on an implementation of AES (see Section 3.2.1 in the literature review) [33].

When applying this to encryption and decryption at once, the structure shown in Figure 5.9 is obtained. With an additional 48 bit of registers to store the intermediary state, a round can be serialized with a factor four. In theory, this saves 12 four-bit S-boxes, 12 inverse S-boxes and three MDS matrix multiplications. But it comes at the cost of additional registers and multiplexers. The wiring also gets more complicated, which makes optimizations harder for the synthesizer. The additional multiplexers are 8-way multiplexers, which take 2 LUTs per bit (see Table 4.4), making their influence not negligible.

The subtweakey generation is not split in four parts. The operations there are already quite simple (see Section 5.1.2). It would also introduce three additional registers of 48-bits, as well as complicated wiring and multiplexers, so it is not worth it.

The datapath of the serialized implementation is shown in Figure 5.10. The sub operations are implemented like in the iterated implementation, however only a fourth of the resources are needed for them. The two 8-way multiplexers on the left side of the datapath, take the 64-bit state and subkey and split it two times in four times 16-bits. It is done two times, because the decryption operation has the shift

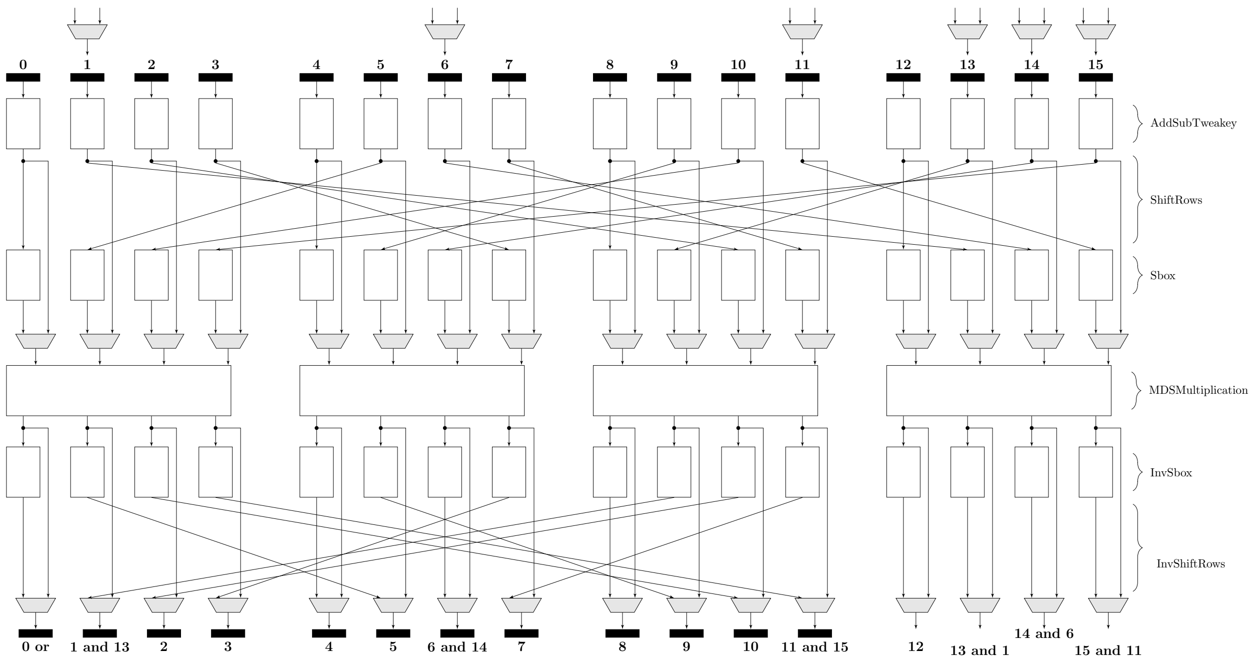


Figure 5.9: Parallelism in a round of Joltik-BC (capable of both encryption and decryption)

at the end so it needs different input nibbles in a round (see Figure 5.9). On the right side of the datapath, the 16-bit output is re-assembled to 64-bit (again in two different ways, one for encryption and one for decryption).

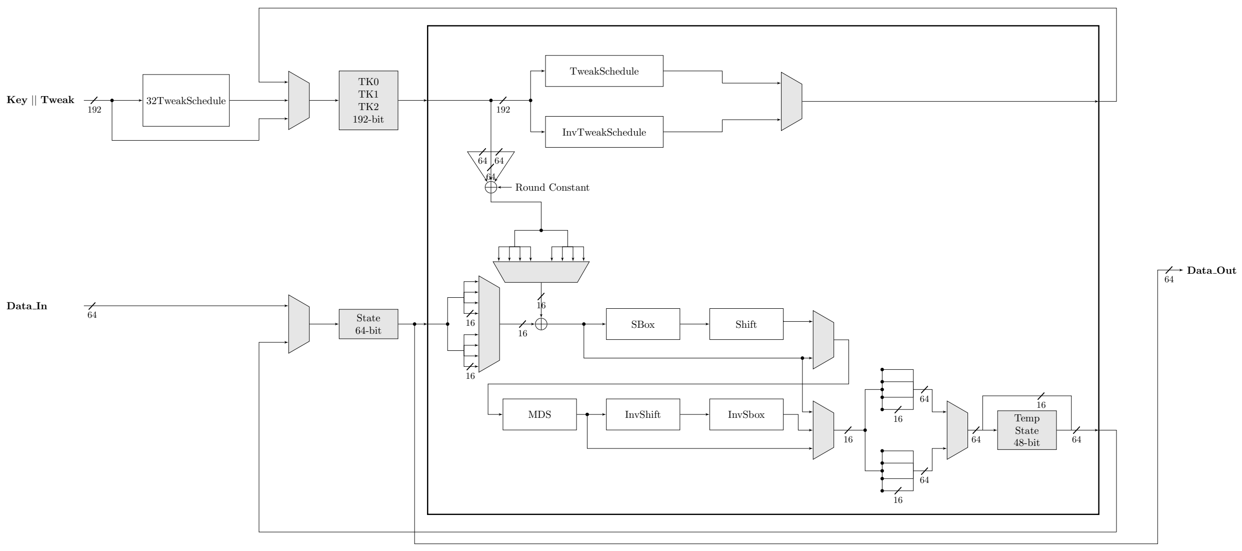


Figure 5.10: The Datapath of the serialized implementation of Joltik-BC-196

5.3.2 The serialized Implementation with distributed RAM

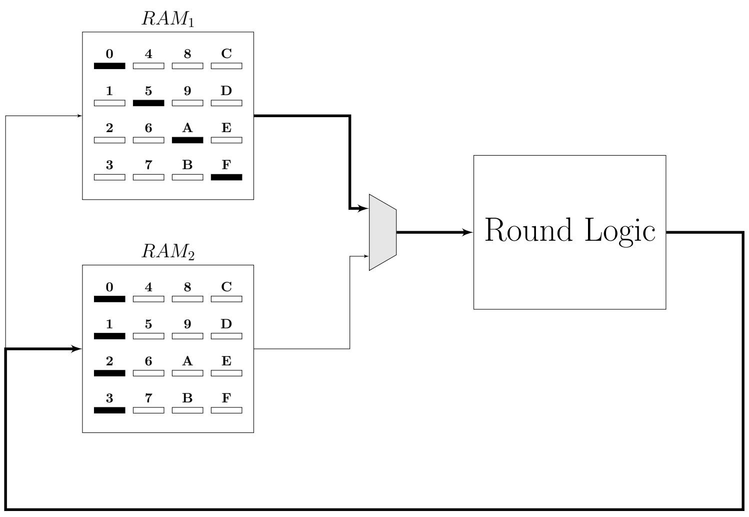


Figure 5.11: The structure of the serialized Joltik-BC implementation with distributed ram

The previous implementation uses 8-way multiplexers, which use a lot of resources. Even if the dedicated multiplexer configuration of the slices is used, the multiplexer still uses two LUTs per bit that is multiplexed.

To reduce the required resources, ram can be used to store the internal state. In that case, only the correct addresses need to be passed to the ram to select the correct data for the cycle, removing the large multiplexers. The same idea has successfully been applied to AES [33] (see Section 3.2.1 in the literature review). Since the state that needs to be stored is quite small, and a large part of it is needed each clock cycle, it is best to use distributed ram [57]. This is implemented by reconfiguring slices into ram. It was discussed in Section 4.4.

A simplified data arrangement for the state during encryption is shown on the left side of Figure 5.11. Two times 64 bit need to be stored in total, and four nibbles of 4 bit need to be read and written each clock cycle. The correct addresses are selected to connect the input nibbles to the output nibbles. In Figure 5.11, this is shown for the first encryption cycle (nibble 0, 5, 10 and 15 are connected to nibble 0, 1, 2 and 3 through the round logic). The path followed and memory used in the cycle is shown in bold on the figure.

Both memories can be used as input and output memory, and their function is swapped each round. During decryption, a different nibble arrangement is needed because the inverse shift is done after the MDS multiplication. In the first cycle nibbles 0, 1, 2 and 3 are written to 0, 5, 10 and 15, instead of the other way around. However, this can be achieved simply by giving different addresses tot the rams. Finally, a similar arrangement is used for the subtweakeys to eliminate the registers and 8-way multiplexers there.

In the actual implementation, the primitive RAM32M [63] is used. It configures four LUT's in to a 8-bit wide, 32-bit deep ram. So two of them, which occupy two slices, can be used for one of the memory banks displayed in Figure 5.11. For the state and subtweakeys together, a total of 16 of these are needed for Joltik-BC-196, and 12 for Joltik-BC-128.

5.4 The Unrolled Implementations

In the unrolled implementations multiple rounds are unrolled, which are then executed in one clock cycle. This leads to a bigger area needed for the logic, which will consume more power and lead to a lower maximum clock frequency. But because less clock cycles are needed per data block, and the overhead from writing to and reading from the registers is reduced, this might reduce the energy consumption and cause a higher throughput. The synthesizer will also have more opportunity for optimization. The purpose of these implementations is to analyze the area-speed-energy trade-off for Joltik.

The implementations are made by placing the round logic of the iterated implementation (described in Section 5.2.3), multiple times after each other. The round logic is the part in the big rectangle on Figure 5.6. Unrolled implementations of factor 2, 4 and 8 are made for both the Joltik-BC-128 and Joltik-BC-192 versions (referred to as unrolled2, unrolled4, and unrolled8 respectively in later sections).

No pipelining is used, because the aim is to look at the benefits of combining the logic of multiple rounds in one, and to find the moment when the additional power consumption from glitches and long paths has a bigger influence than the reduced overhead and benefits of the combined logic.

5.5 The Measurements Results

There are six different implementations for each of the two versions of Joltik (Joltik-BC-128 and Joltik-BC-192). All implementations had their power consumption measured according to the setup described in Section 4.5. The maximum frequency and area of the implementations is determined by synthesizing the implementation with a wrapper at different clock speed constraints. These results are presented in this section, but a more detailed report of the results can be found in Appendix A.

The dynamic power consumption of the Joltik-BC-192 implementations is shown in Figure 5.12. For reference, the static power consumption was around 200 mW. The measurements are done at 25 MHz, 50 MHz and 100 MHz, unless the maximum speed of the implementation was slower. The dynamic power consumption should follow a linear relationship with the frequency, and this was observed to be the case. This is a good indication that the dynamic power consumption was indeed obtained. Only the really low power measurements at 25 MHz were inconsistent at times. This error can be attributed to measuring inaccuracies, since their values are close to the measuring accuracy (the measuring accuracy is around 3 mW, and some measurements only consume 10 mW).

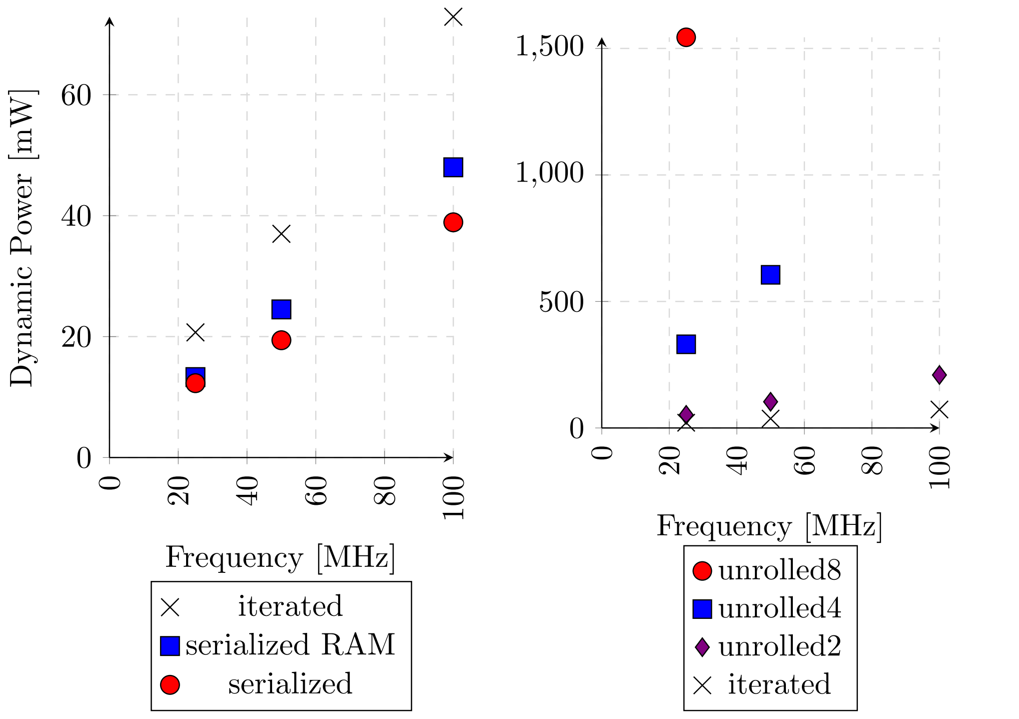


Figure 5.12: The dynamic power consumption of the Joltik-BC-192 implementations

The energy consumption is then calculated by combining the power consumption and the amount of clock cycles that are needed to process a block. The energy consumption for the three measured frequencies is then averaged. This energy consumption is determined during encryption and decryption, which is displayed in Figure 5.13. Because of scaling issues, the unrolled8 implementations are not displayed on the graph. They had a value of 3.745 J/Gb and 4.825 J/Gb for encryption, and 4.060 and 5.090 J/Gb for decryption (for respectively Joltik-BC-128 and Joltik-BC-192). This is more than 10 times as inefficient as the iterated implementation.

A difference between encryption and decryption is visible, especially for the iterated and the serialized implementations that use distributed ram. There is also a big difference between the two versions of Joltik, despite that they only differ in the key schedule. The energy consumed during encryption and decryption is averaged to obtain the average energy consumption of the implementations.

In the top part of Figure 5.14, the energy consumption is compared to the area of the implementations. And in the bottom part, the energy consumption is compared to the throughput if the implementations were to be run at their maximum frequency.

In Joltik-BC-192 it is visible that the unrolled4 and unrolled8 implementations, are visibly worse than the rest. Just like the two serialized implementations (with the exception that the serialized ram implementation is a negligible amount smaller than the iterated implementation). It should be noted that the area measurements are quite inaccurate because if the synthesis is ran multiple times, slightly different

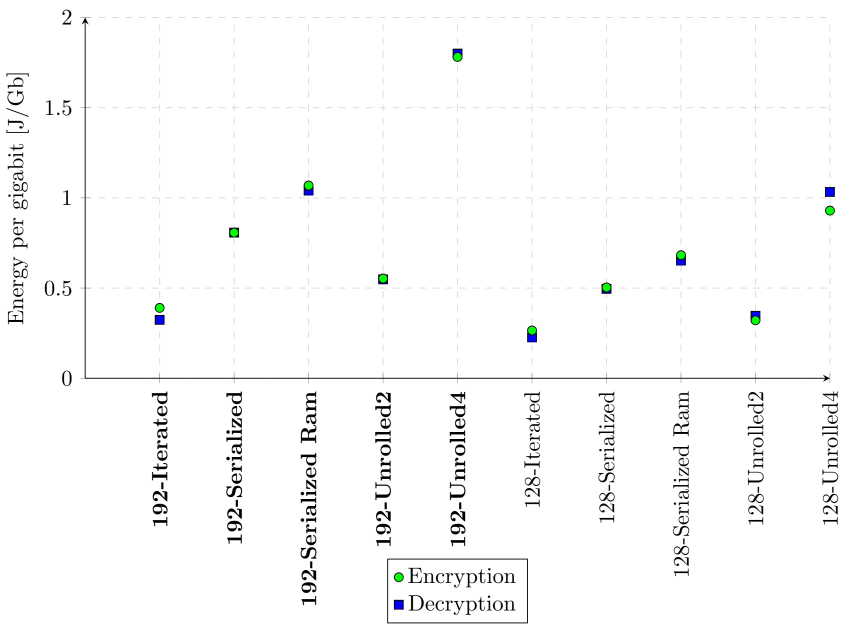


Figure 5.13: The energy consumption of the Joltik-BC implementations

results will be obtained every time. The numbers are therefor just an indication of what size the implementation will be.

5.6 Interpretation of the Results

It is visible from the measurement results that Joltik does not have much potential to be unrolled or serialized. Any unrolling beyond two gives worse results in both energy, area and maximum throughput. An explanation is that a round has complicated wiring (mainly because of the shift and inverse shift functions), and when combining multiple rounds, more glitches are induced that cause slower timing and higher power consumption.

The results seem to confirm this, since the smaller Joltik-BC-128 responds better to unrolling. The proportional increase of the required energy, and lower maximum clock frequency for each unrolled implementation, is smaller than in the case of the bigger Joltik-BC-192. When a round is unrolled 8 times, and even the Joltik-BC-128 implementation becomes too complex, the results worsen drastically.

Joltik also does not benefit much from serialization. Only a minuscule area reduction is achieved, at the cost of more than double the energy consumption and a four times lower throughput. Using distributed ram doesn't seem to make much of a difference either. This is in contrast with the large area reduction that was achieved in AES with a similar method (see Section 3.2.1).

An explanation is that Joltik has small logic already in the iterated implementation. For example it has 4-bit S-boxes as opposed to the 8-bit S-boxes of AES, which are a lot bigger in hardware. The serialization overhead has a bigger effect

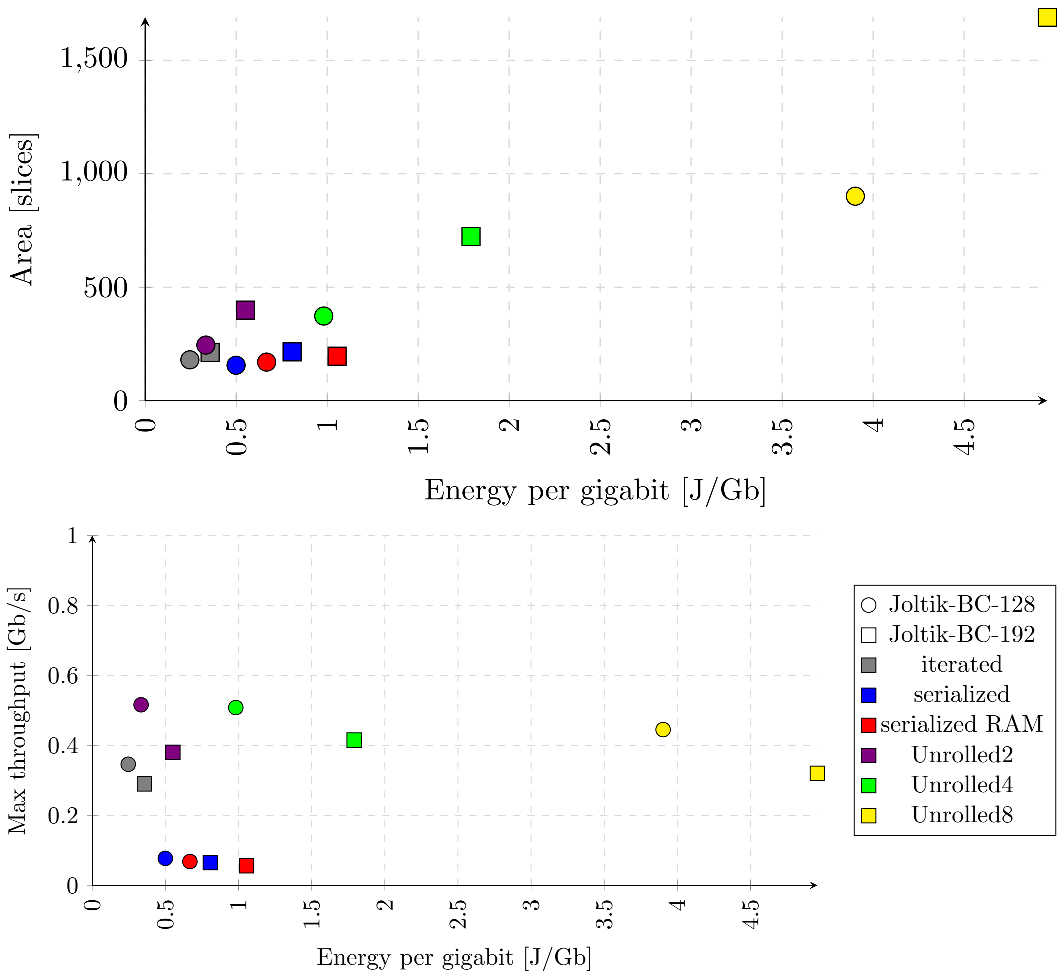


Figure 5.14: The area and maximum throughput versus the energy consumption of the Joltik-BC implementations

than the achieved logic reduction. Further, it is possible the number of used registers and LUTs have been reduced, but because the implementation is so small this is not visible in the number of used slices.

The number of used LUTs and registers of the serialized implementations are listed in Table 5.4. In the first serialized implementation, the number of registers increases because of the additional registers to hold the temporary state. Unfortunately the number of LUTs don't change much, the overhead from the complicated wiring and additional multiplexers outweigh the benefits of the reduced logic.

In the serialized implementation with distributed ram it is visible that the number of registers is highly reduced, and the number of LUTs stays the same or even decreases. So using the ram had an effect on the area. However the energy consumption increased, and since the area effect is not visible in the number of slices, the implementation does not have many benefits over the iterated implementation.

Implementation	Joltik-BC-196		Joltik-BC-128	
	LUT's	Registers	LUT's	Registers
Iterated	683	297	555	205
Serialized	685	329	457	254
SerializedRam	604	121	444	109

Table 5.4: The number of LUTs and Registers used in certain Joltik-BC implementations

In Table 5.5, the area, maximum throughput and energy consumption of the implementations that offer a relevant trade-off are listed. At the cost of a 87% increase in area and a 54% increase in energy, Joltik-BC-192 can be made 31 % faster. Joltik-BC-128 can be made 49% faster at the cost of a 36% area and 37 % energy increase. This is a trade-off that might be useful in some applications.

Implementation	Area (Slices)	LUTs	Registers	Energy (J/Gb)	max throughput (Gb/s)
BC-192-Iterated	213	267	683	0.357	0.290
BC-192-Unrolled2	399	272	1539	0.551	0.380
BC-128-Iterated	180	205	555	0.246	0.346
BC-128-Serialized	156	254	457	0.500	0.077
BC-128-Unrolled2	245	201	863	0.338	0.516

Table 5.5: The measurement results for the best Joltik-BC implementations

A benefit of Joltik that should be noted, is that it has no initialization phase and a really small finalization phase, with a length equivalent to only one data block. So the listed results will be achieved for almost any message length (a message of 0.5 Kb already achieves 90% of this maximum performance).

5.7 Conclusion

All the implementations of Joltik and their performance were discussed in this chapter. Six implementations were made of both Joltik-BC-128 and Joltik-BC-192, which are used in $Joltik - 64 - 64$ and $Joltik - 128 - 64$ respectively. An iterated implementation, two serialized implementations (one with distributed ram and one without), and three unrolled implementations with a unrolling factor of 2, 4 and 8.

Only some of these implementations offered a relevant improvement or trade-off, but even then this was pretty minor. The iterated implementation still performs the best in most of the cases. Some possible explanations for the results were also discussed. The next two chapters will present the implementations and their results for the next two ciphers, and in Chapter 8 the three ciphers will be compared with each other.

Chapter 6

The Implementation of Morus

In this chapter, the implementations of Morus and their performance are discussed. The implementation approach for these implementations was detailed in Chapter 4. The area, energy, and throughput of the implementations is measured and compared to each other. In the previous chapter, this was done for the Joltik implementations, and in the next chapter it will be done with respect to Ascon.

In *Section 1*, the specifications of the Morus cipher are explained in more depth. This is important to understand the implementations, which are presented though Sections 2 to 4. In *Section 2*, the iterated implementation is described, which is also the reference implementation that the other implementations modify. In *Section 3*, the serialized implementation is described, and in *Section 4*, the implementation where all five rounds of Morus are unrolled is discussed.

In *Section 5*, the most important measuring results are shown. The more detailed results can be found in Appendix B for reference. In *Section 6*, these results are discussed.

6.1 The Structure of Morus

Morus has three different recommended parameters, one of which is implemented in this thesis, namely *Morus-640* (see Section 2.3). Morus uses a custom *state update function* to update its state every time a new block of data is processed.

In this section the specification of Morus is presented. First, the global structure of Morus-640 is explained, followed by the structure of the state update function.

6.1.1 Morus-640

A diagram of an encryption with Morus-640 is given in Figure 6.1. Confidentiality is achieved by XORing the internal state with the plaintext to generate the ciphertext (the *output generator* in Figure 6.1). And authenticity and integrity are achieved by injecting the plaintext in the state update function.

The internal state consists of five times 128-bit, for a total of 640 bits. The only other storage needed is a 128-bit register to hold the input during the finalization.

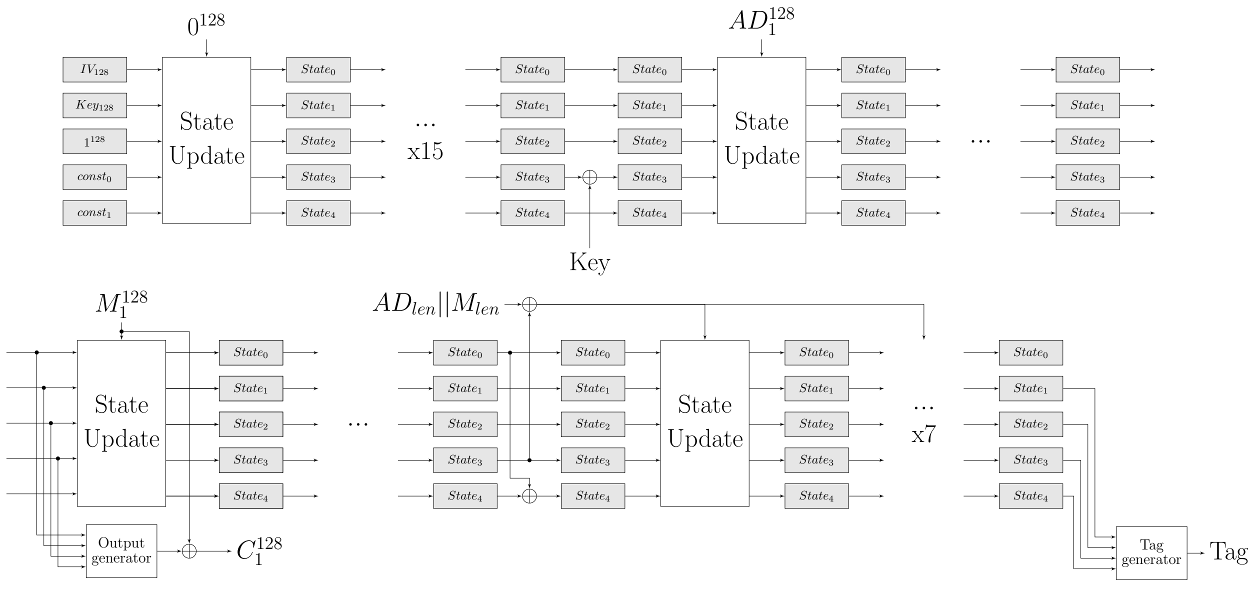


Figure 6.1: High level diagram of Morus-640

After initialization, the AD and plaintext are inserted in the state update function in 128-bit blocks. If the last block is not the full 128 bit, it is padded with zeros.

The structure of the decryption mode is similar. The ciphertext is processed in the same way as the plaintext, but this plaintext is then used as input to the state update function. So an additional 128-bit register is needed to hold it (or the register that is used during the finalization can be reused).

To achieve 128-bit security with a good margin, the initialization and finalization stage have to be quite long, requiring 16 and 8 times the effort of the encryption of a single 128-bit block respectively.

6.1.2 The State Update Function

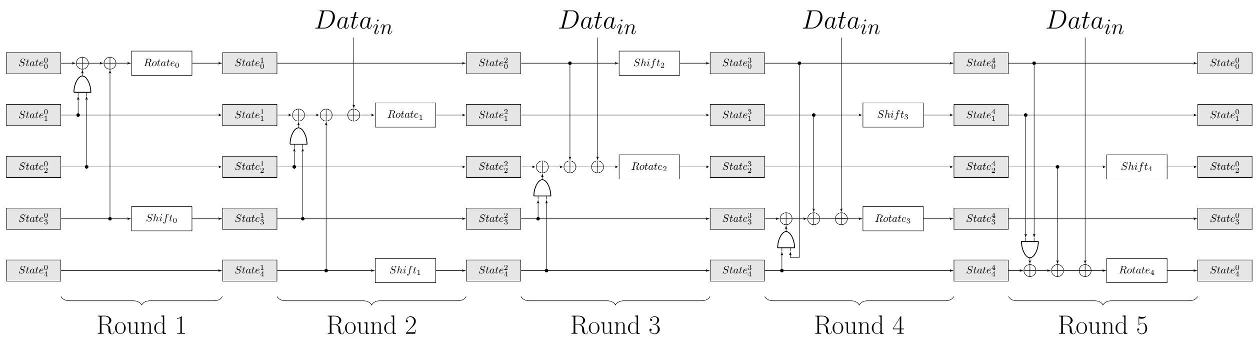


Figure 6.2: The five rounds of the round update schedule used in Morus-640

The state update function is a dedicated function designed for Morus, partially based on a type 3 Feistel scheme [19]. It consists of five rounds, that have a short critical path and only use *XOR*, *shift* and *AND* operations. The full state update function is shown in Figure 6.2.

The rotation function splits the 128-bit substate up in parts of 32-bit, and shifts these by an amount that is dependent on the current round. The shift function, shifts the whole 128-bit substate by a certain amount. The constants can be found in the specification paper of Morus [23]. The critical path of one round consists of an AND-gate, two XOR-gates and a bit rotation, which suggest that full unrolling of the five rounds could give good results (this is done in Section 6.4).

6.2 The Iterated Implementation

In this section, the iterated implementation of Morus is described. Just like in the case of Joltik (see Section 5.2), the iterated implementation is also the reference implementation. It determines the base structure for the other implementations of Morus in this thesis.

Like all three ciphers in this thesis, Morus is implemented with the GMU hardware API. The developer only has to implement the CipherCore, which contains the cryptographic cipher. A separate kernel in this CipherCore contains the state update function, and a small datapath and control FSM in the CipherCore link the kernel with the GMU API.

The kernel is the part of the implementation that is further optimized for energy, and the part that will be adapted in the other implementations of Morus. Similar to the case of Joltik, and as described in chapter 4, this will be the part that consumes the majority of energy, and its round based structure can be used to make unrolled and serialized implementations to analyze the area-energy trade-off.

The GMU API works the same way as in the case of Joltik, and will therefor not be repeated in this section (see Section 5.2.1). First, an explanation of how the kernel is linked to the GMU API is provided, followed by a description of the iterated implementation of the kernel.

6.2.1 The CipherCore

In contrast to the block cipher-based algorithm Joltik, Morus' internal state needs to be stored between calls to the state update function. Additional small operations are done on this state during some stages of execution. For example after initialization, a part of the internal state is XORed with the key (see Section 6.1.1).

This internal state is stored in the kernel itself, so to support these slight differences between the stages, the kernel has five modes in which it can operate (this is clarified in Section 6.2.2). It also has a port for the block size, which is of relevance if the last block of the decryption does not fit in one 128-bit block. And necessary data inputs are also supplied to the kernel, e.g. the key, public nonce, message length,... They are directly linked to the GMU API. This keeps the datapath of the CipherCore really simple. It is shown in Figure 6.3.

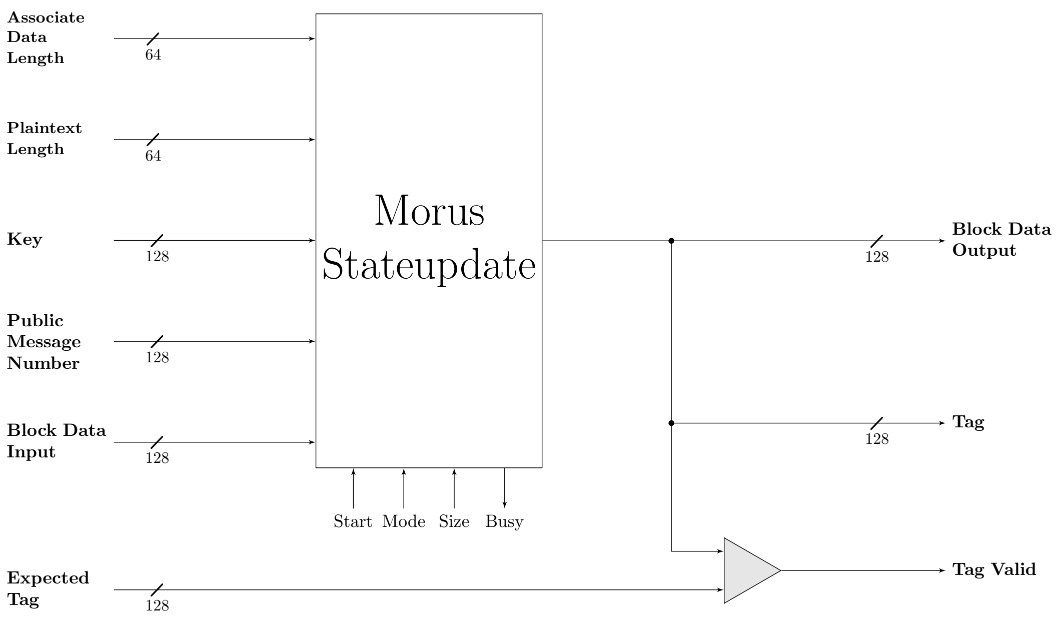


Figure 6.3: The Datapath of the CipherCore of Morus

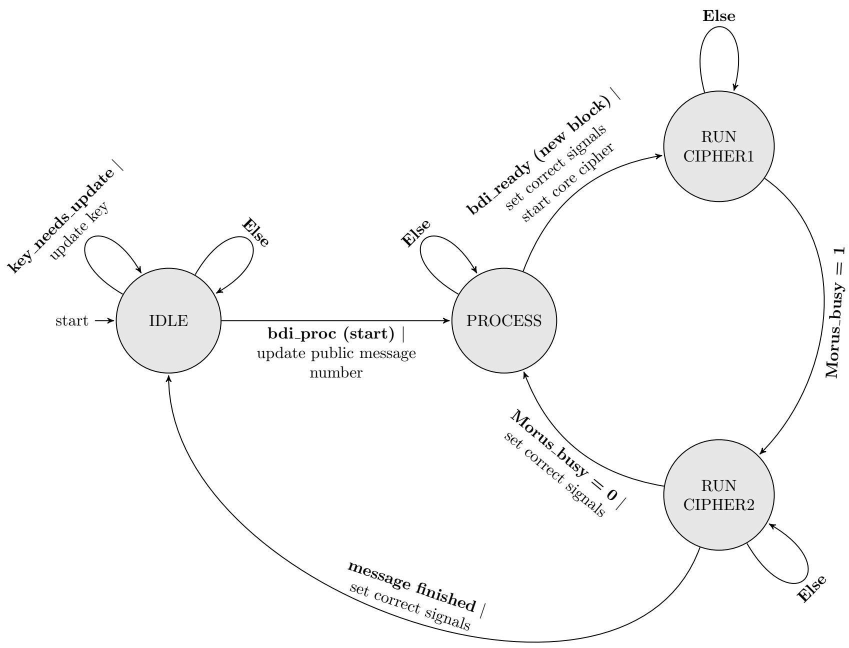


Figure 6.4: The Finite State Machine of the CipherCore of Morus

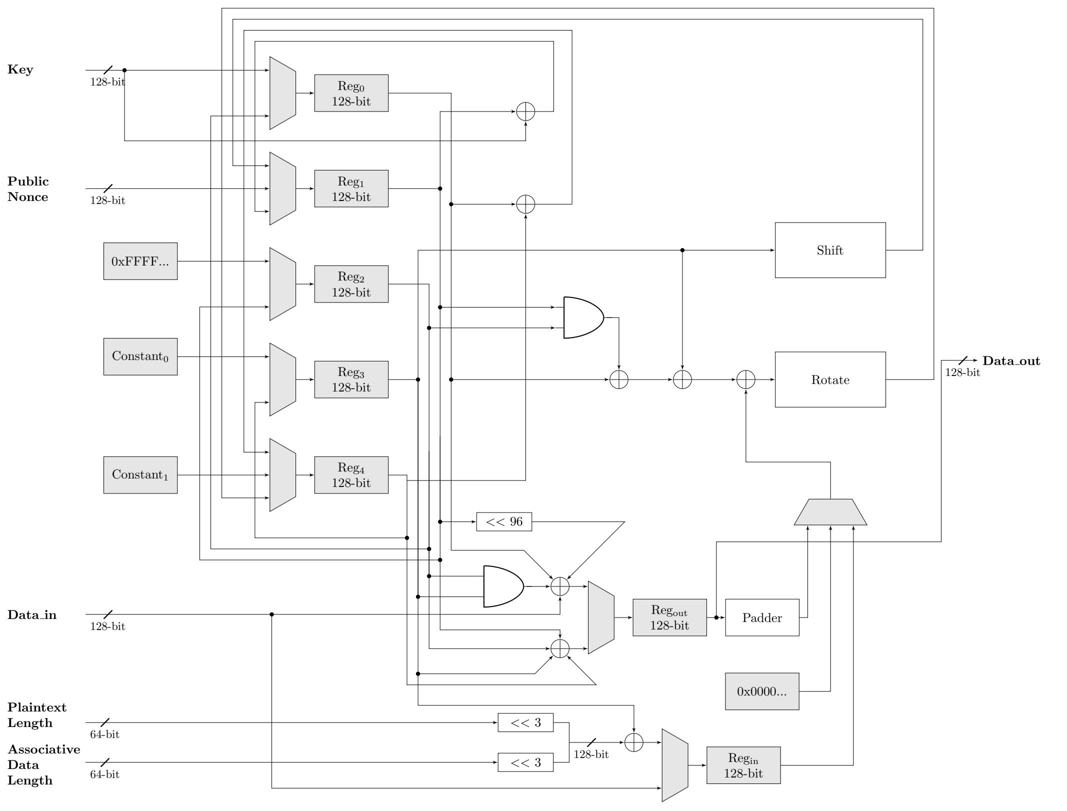


Figure 6.5: The datapath of the iterated implementation of Morus

The controller is relatively simple as well. It translates the control signals of the GMU API, to those of the kernel. It is shown in Figure 6.4. It works the same as the controller of the Joltik CipherCore, and was already explained in that section (Section 5.2.2). A handshake is used for communication with kernel, using the its *start* and *busy* signal.

6.2.2 The Kernel

The Kernel contains the state update function. In Section 6.1.2, an overview of this state update is given. This hardware implementation consists of a datapath and a FSM. In Figure 6.3 the in- and outputs of the kernel are visible. The datapath and control FSM of the iterated implementation of the kernel are shown in Figures 6.5 and 6.6.

The round update schedule applies simple operations with four of the five internal state registers. These are visible in the middle-right of Figure 6.5. To avoid having to multiplex the state registers to these operations, the results are stored in different registers for each round instead. If the actual storage would be compared to Figure 6.2, Register 0 would hold state 0 in the first round, but the result would then be written in Register 4, so when the next round starts, the *round logic* has the same inputs.

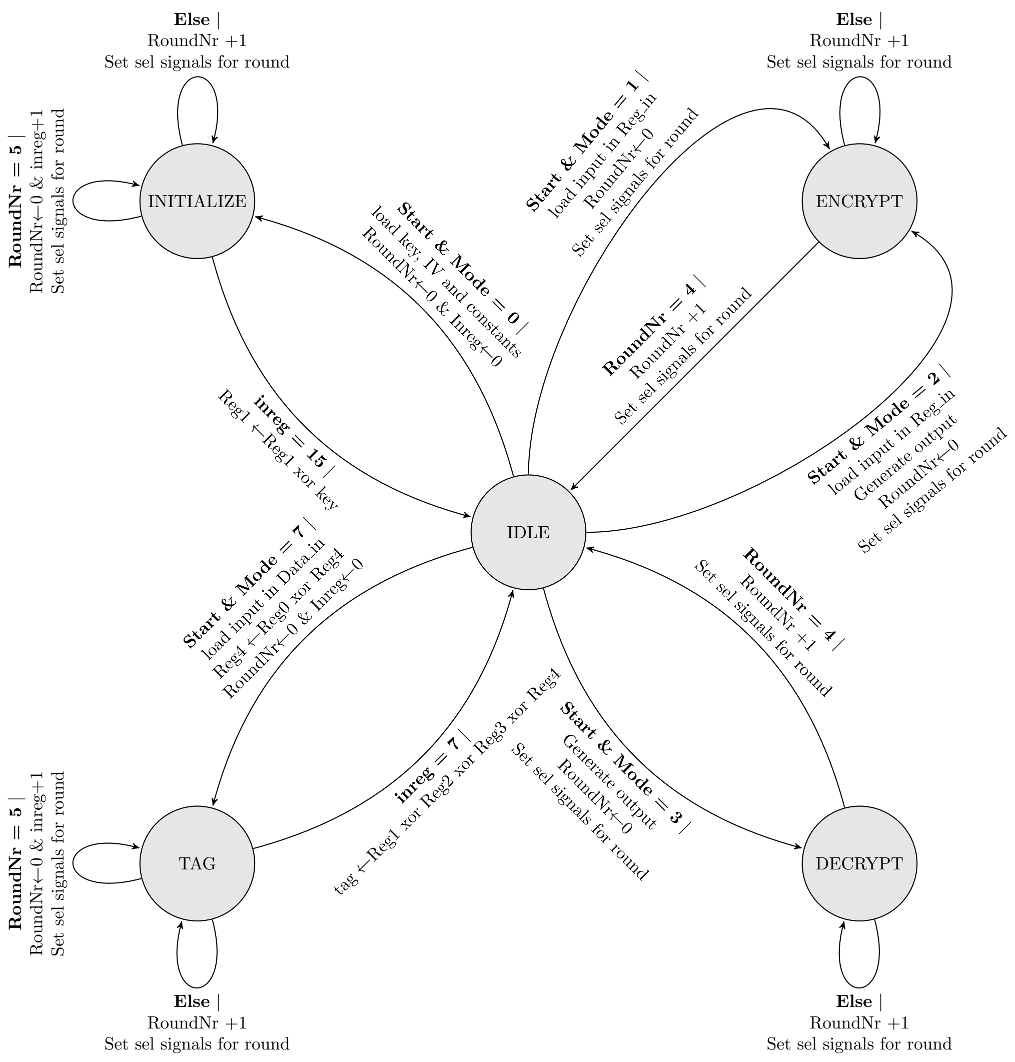


Figure 6.6: The FSM of the iterated implementation of Morus

To support all modes of execution, the hardware implementation of the kernel has some additional hardware apart from the logic needed for the state update (for example to XOR with the key after initialization and to generate the output from the state). These are the two XORs on the top of Figure 6.5.

There is a register to hold the output, and a register to hold the temporary value needed during the generation of the tag. The registers are called Reg_{out} and Reg_{in} respectively on Figure 6.5. All logic at these two registers is there to either generate the plain- or ciphertext from the internal state, generate the final tag, or generate the temporary value for the tag (shown respectively from top to bottom on Figure 6.5).

The FSM (shown in Figure 6.6) controls all the multiplexers, and enables the correct registers, based on the execution mode (initialization, associated data, encryption, decryption or finalization) and the current round. During initialization and finalization, the rounds have to be repeated multiple times. The iterated implementation executes at the speed of one round per clock cycle, so processing one block takes five clock cycles.

6.3 The Serialized Implementation

In this section the serialized implementation of Morus is described. Like in all implementations, only the kernel is changed compared to the iterated implementation.

One round is serialized by a factor four, and thus takes four clock cycles to execute instead of one. The aim of this implementation is to reduce the resources (chip-area) taken by the implementation, at the cost of a slower execution and higher energy consumption per block. However as was seen in Joltik (in Section 5.5), a good result is not guaranteed.

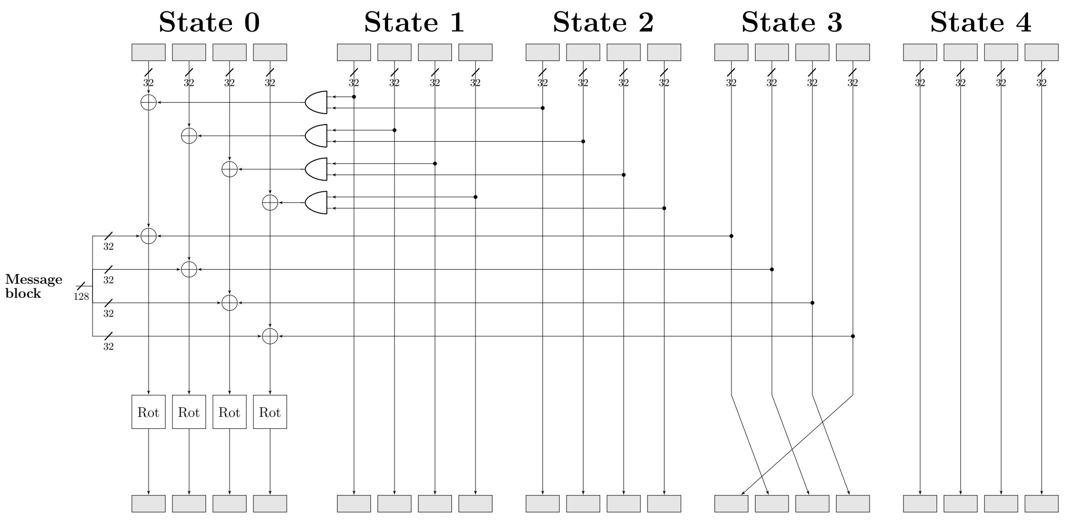


Figure 6.7: The structure of a round of Morus-640

In Figure 6.7, the structure of a round is shown. Because the rotation works on 32-bit parts of the state, and the shift operation shifts in multiples of 32-bit, no additional registers are needed to store intermediary values if a round is split into four parts. If it would be split in more than four parts, some registers would be required to store intermediary values of the rotation and shift operation.

In practice this means that every 128-bit state register from the iterated implementation, is split into four 32-bit registers. A multiplexer selects which of those four are connected to the round logic every cycle. These multiplexers create some additional overhead, so the area increase has to be offset by the reduction of round logic for there to be any improvement.

The variable shift in each round (see Figure 6.2), shifts with either 32, 64 or 96 bits. Because these are multiples of 32-bits, it can be achieved by simply selecting a different sub-state register as needed (this mixing is also visible in Figure 6.7). The controller keeps track of what sub-register has to be selected in each round in state 3.

All these changes are applied to the iterated implementation, and the external functionality is kept the same, with the exception that one state update takes four times as much cycles now. When this kernel is placed in the CipherCore it therefor works as intended.

6.4 The Unrolled Implementations

The short critical path of the state update indicates that good performance could be achieved if multiple rounds are unrolled. Since five is a prime number, the implementation can not be unrolled with a factor smaller than five without a lot of overhead. However, because the critical path is so short, it would be expected that an unrolling of five would give the best results anyway. This implementation of the kernel is discussed in this section.

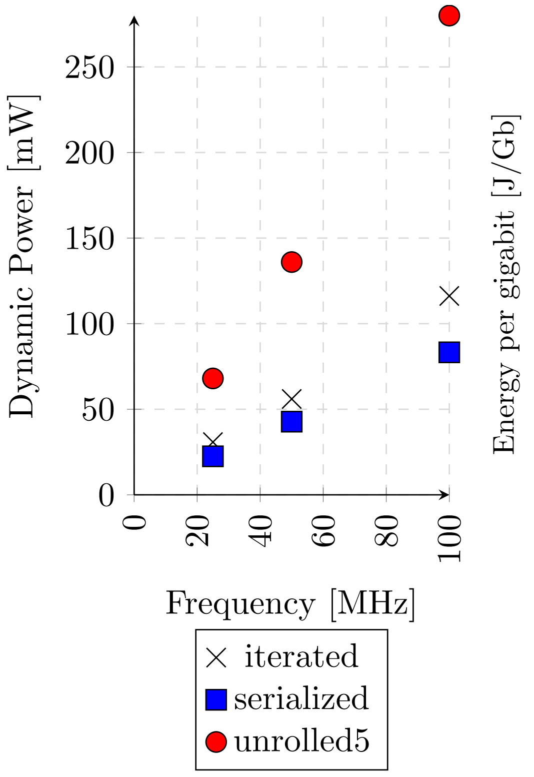
When all five rounds are unrolled, the critical path doesn't become five times as long, but less than three times. One AND-gate, 6 XOR gates and three bit rotations, instead of an AND-gate, 2 XOR gates and one bit rotation. This can be deduced from Figure 6.2. On FPGA the critical path will be reduced even more because the synthesizer can combine multiple operations in a single LUT.

This is then applied in a straightforward way to the datapath of the iterated kernel, so all extra functionality is kept. The implementation can process one 128-bit block in each clock cycle. However the GMU API is limited to a lower speed. So when used with the API, the speed is limited to 3 clock cycles per block if 128-bit input and output ports were to be used. This overhead is a big disadvantage of the API and one of the reasons why all measurements are done on the Kernel, and not the CipherCore or GMU API (see Section 4.2).

6.5 The Measurements Results

The measurement results of the three implementations of the Morus kernel are presented in this section. Just as in the case of Joltik, the implementations are measured with the setup described in Section 4.5. The maximum frequency and area of the implementations are determined by synthesizing the implementations with a wrapper at different clock speed constraints. These results are presented in this section, but a more detailed description can be found in Appendix B.

The dynamic power consumption during encryption with the implementations at the three measured frequencies is shown on the left in Figure 6.8. For reference, the static power is around 200 mW. The dynamic power follows the frequency linearly, which confirms that what is measured is indeed the dynamic power. When the energy per block is determined and averaged over the frequencies, the results on the right of Figure 6.8 are obtained. The energy consumed during encryption is very similar to that during decryption. This is expected because the operations are almost exactly the same (see Section 6.1.2).



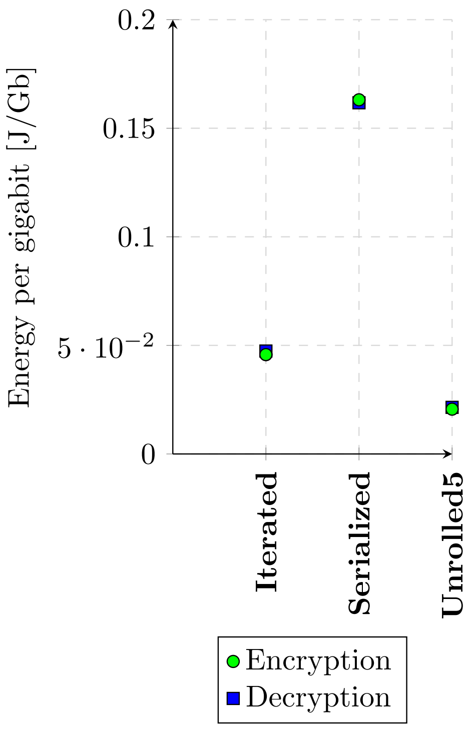


Figure 6.8: The dynamic power consumption and energy consumption of the Morus implementations

In Figure 6.9, the required energy of the implementations is compared to respectively the area and the throughput. There is a clear trade-off visible between the three implementations, where speed and energy efficiency come at the cost of area.

In contrast to Joltik and to a lesser degree Ascon, Morus has a big initialization and finalization stage. This means that the previous energy efficiency and throughput are obtained only for large messages. This is shown in Figure 6.10. At message lengths of 20-30 kB, all implementations are within 10 % of their maximum throughput and minimum energy consumption.

6.6 Interpretation of the Results

From the measurement results it is visible that Morus is very suited to unrolling. Unrolling the full five rounds leads to a power consumption of about 2.5 times the power consumption of the iterated implementation, but at five times the speed, approximately halving the energy use per data block. Unfortunately Morus only has five rounds, and no other unrolling factors can be tested without making the implementation incompatible with the used API.

This result was expected because of the really short critical path of Morus. Unrolling reduces the influence of the overhead of the registers and routing on the

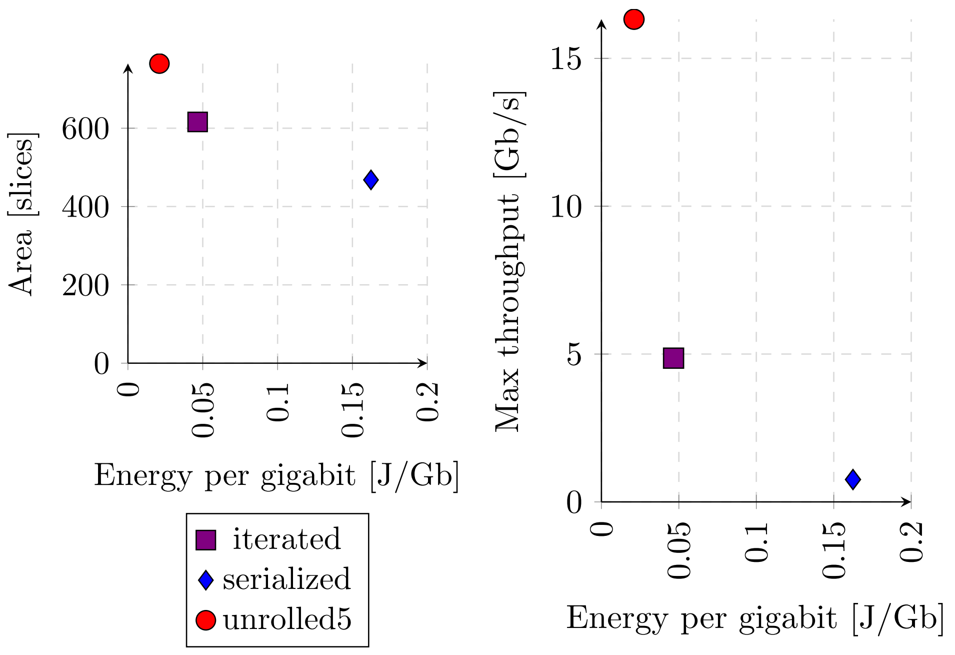


Figure 6.9: The area and maximum throughput versus the energy consumption of the Morus implementations

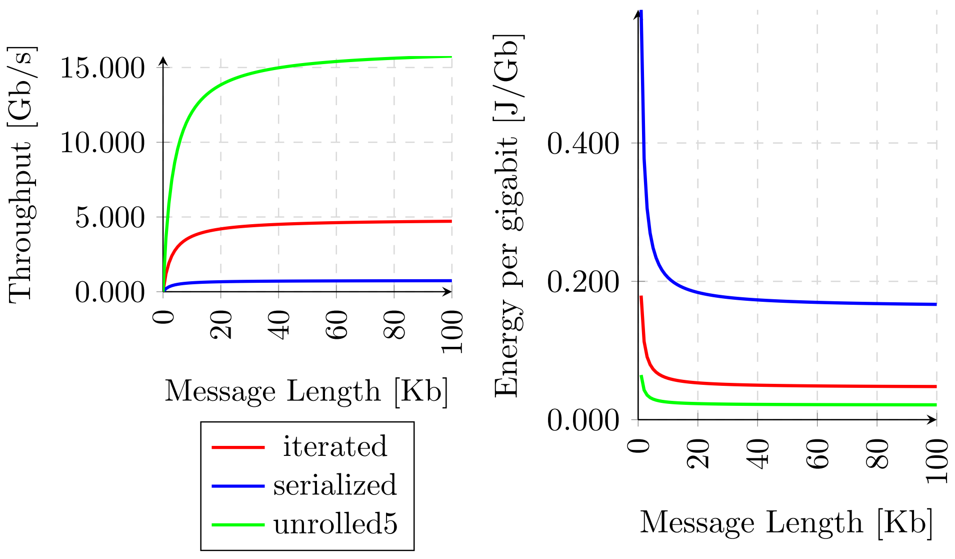


Figure 6.10: The throughput and energy consumption in function of the message length for the Morus implementations

Implementation	**Area (Slices)**	**LUTs**	**Registers**	**Energy (J/Gb)**	**max throughput (Gb/s)**
Serialized	468	1407	955	0.162	0.756
Iterated	616	1965	944	0.047	4.861
Unrolled5	765	2618	945	0.021	16.322

Table 6.1: The measurement results for the best Morus implementations

total energy, maximum clock speed and area. The path is probably also short enough so glitches don't make the power rise drastically, as was the case in Joltik.

Serializing Morus leads to a small reduction in area of about 25% compared to the iterated implementation. However the power consumption drops less than 20%, with the implementation taking four times as long to complete a block. This leads to a much larger overall energy consumption. The area reduction is not coming close to the theoretical reduction of 75% by serializing with a factor four. This could be due to the overhead of the multiplexers that connect the state registers to the round logic, and because the number of registers is not reduced when serializing.

In Table 6.1 the area, maximum throughput, and energy consumption of the three implementations is listed. All the implementations offer some trade-off compared to each other. The serialized implementation is 25 % smaller than the iterated implementation, but at the cost of a 3.5 times larger energy consumption and a 6.5 times smaller maximum throughput. The unrolled implementation is more than twice as energy-efficient and 3.5 times as fast as the iterated implementation, but at the cost of a 25 % larger area.

A large disadvantage of Morus is the long initialization and finalization stage. Message lengths of 20-30 kB are needed before the energy consumption and throughput reaches the values in Table 6.1. If the aim is to encrypt short messages, the cipher will be a lot less efficient and fast. For example, a 1 kB message is around four times as slow and energy inefficient than the values listed in the table.

6.7 Conclusion

All the implementations of Morus and their performance were discussed in this chapter. Only three implementations were made, but they all offer a relevant trade-off. An iterated implementation, serialized implementation and unrolled implementation (a serialization with factor 4 and an unrolling with factor 5).

The serialized implementation has lower speed and worse energy consumption but the benefit of smaller area. The unrolled implementation offers high speed and low energy consumption at the cost of higher area. And the iterated implementation lies in between. The previous chapter analyzed the implementations of Joltik, and the next will discuss the implementations of Ascon. In Chapter 8, the three ciphers will be compared with each other.

Chapter 7

The Implementation of Ascon

In this chapter, the implementations of Ascon and their performance are discussed. Ascon is the last of the three ciphers that still needs to be discussed. The implementation approach was detailed in Chapter 4. The area, energy, and throughput of the implementations is measured and compared to each other.

In *Section 1*, the specifications of the Ascon cipher are explained in more depth. This is important to understand the implementations, which are presented though Sections 2 to 4. In *Section 2*, the iterated implementation is described, which is also the reference implementation that the other implementations modify. In *Section 3*, the serialized implementation is described, and in *Section 4*, the implementations with multiple unrolled rounds are discussed.

In *Section 5*, the most important measuring results are shown. The more detailed results can be found in Appendix C for reference. In *Section 6*, these results are discussed.

7.1 The Structure of Ascon

Ascon is based on duplex sponge modes. It has two recommended sets of parameters, which only differ slightly, and both are implemented in this thesis (see Section 2.4). The main part of Ascon is the permutation that permutes the state after the initialization, finalization and each data block. In this section the global structure of Ascon is explained first, followed by a more detailed explanation of the permutation function.

7.1.1 Ascon-128

A diagram of an encryption with Ascon-128a is shown in Figure 7.1. The AD and plaintext are XORed with a part of the output of the permutation function, and this output is then injected in the next iteration of the permutation function. In this way, confidentiality, authenticity and integrity are achieved at once.

The internal state consists of five 64-bit sub-states for a total size of 320 bit. No other storage is needed, except optionally to hold the output blocks. This would

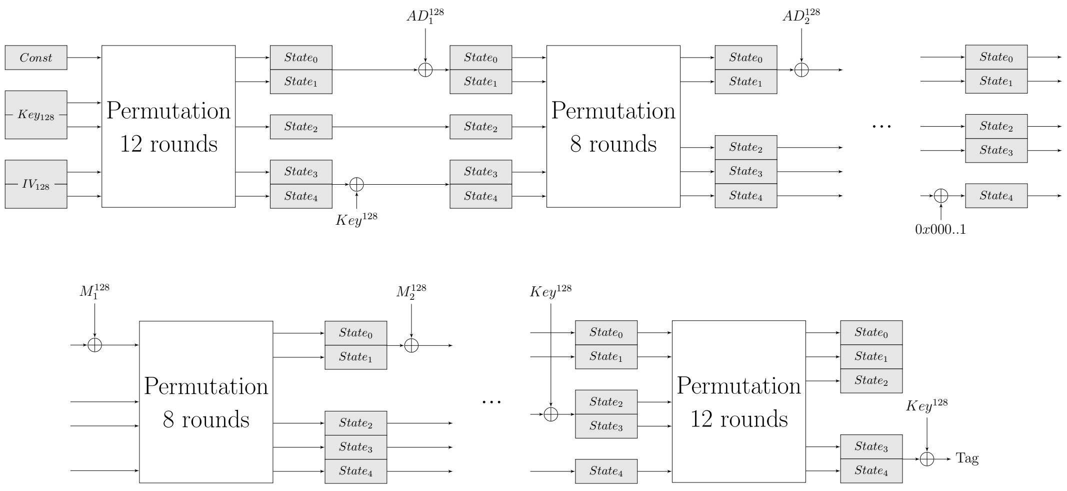


Figure 7.1: High level diagram of Ascon-128a

Name	**key-size**	**public message number size**	**tag size**	**Data block size**	**Rounds begin and end**	**Rounds datablock**
$Ascon-128a$	128	128	128	128	12	8
$Ascon-128$	128	128	128	64	12	6

Table 7.1: The sizes of the key, tweak, public number and block number for the analyzed versions of Ascon

make the communication more convenient (the output would be available for more than one clock cycle). After the initialization, the AD and plaintext are processed in 128-bit blocks (in Ascon-128a). Ascon-128a and Ascon-128 only differ slightly. The block size of Ascon-128 is 64 bit instead of 128 bit, and the number of rounds of the permutation function differs. This is shown in Table 7.1.

The AD and plaintext are padded with a single '1' bit followed by zeros, until the total length is a multiple of the block length. In contrast with Joltik, this padding happens even if the message would originally fit in full blocks. E.g if the message length is 256 bit, two blocks will be filled and a third will be added that contains just '0x8000...'. This will play a role in the implementations because the GMU API does not support this padding mode, so additional logic will be required in the CipherCore or kernel.

The structure of the decryption mode is similar. The only difference is that the ciphertext block is directly inserted in the permutation function, and the output of the XOR between this block and the state yields the plaintext. So very little additional logic is needed to make an implementation perform both encryption and decryption.

While there is an initialization and finalization stage, the overhead of these is not so big, their equivalent is only about three blocks of data (compared to 24 blocks in Morus-640). This makes Ascon more suitable for short messages than Morus.

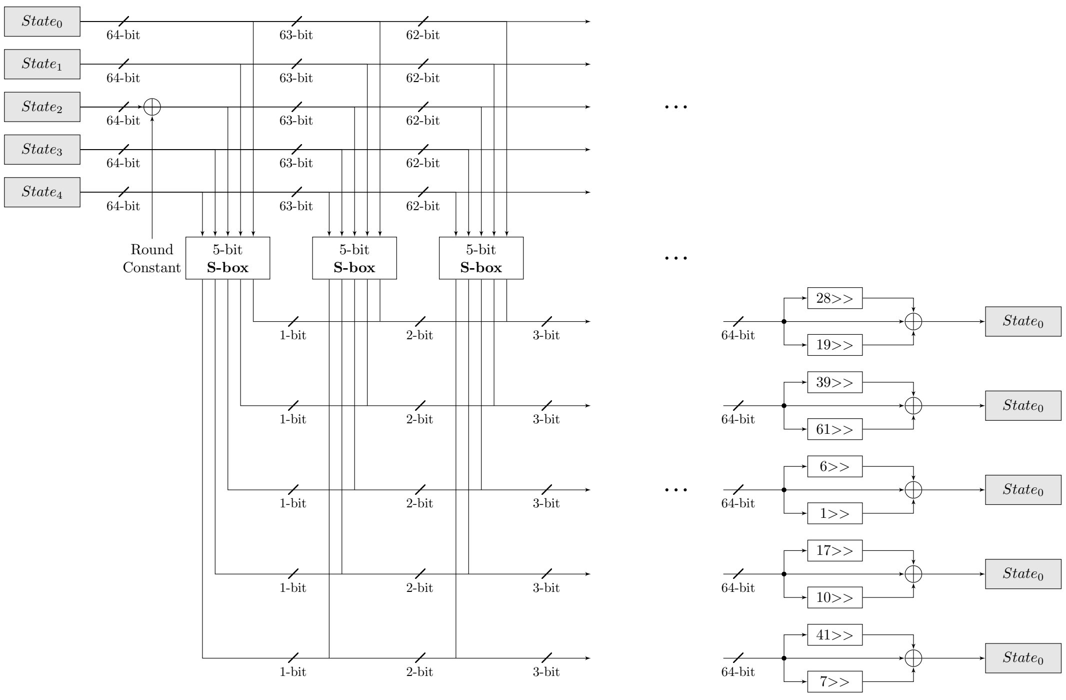


Figure 7.2: One round of the permutation used in the Ascon cipher

7.1.2 The Permutation Function

One round of the permutation of the Ascon cipher is shown in Figure 7.2. A round does not consist of many steps and should have a moderately short critical path (two XORs, a S-box and a shift operation). Between 6 and 12 of these rounds are repeated in one permutation (see Section 7.1.1).

A round consists of three operations on a 320-bit state. First an *addition of roundconstants* is done to a small part of the state, followed by 64 5-bit S-boxes, and in the end a *diffusion layer* that XORs the state parts with two shifted versions of themselves. The specifications of the S-box can be found in the submission paper of Ascon [24].

7.2 The Iterated Implementation

In this section the iterated implementation of Ascon is described. Just as with the implementations of Joltik and Morus, this implementation is the reference implementation for the other implementations of Ascon. It lays the base structure that the other implementations adapt.

The same API is used as in the other two ciphers, and its structure will not be repeated in this section. It can be found it Section 5.2.1. The developer only has to implement the CipherCore, which contains the cryptographic cipher. A separate kernel in this CipherCore contains the permutation function of Ascon, and a small datapath and control FSM in the CipherCore link the kernel with the GMU API.

First the structure of the CipherCore will be discussed, followed by a description of the iterated implementation of the kernel. Just as in Joltik and Morus, this kernel will consume the majority of the energy and area, and it can easily be incorporated in to other designs (see Section 4.3).

7.2.1 The CipherCore

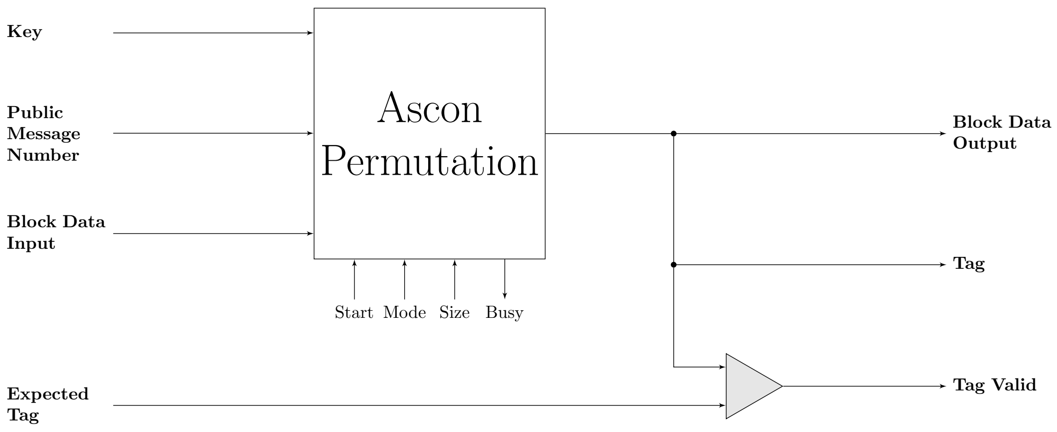


Figure 7.3: The datapath of the CipherCore of Ascon

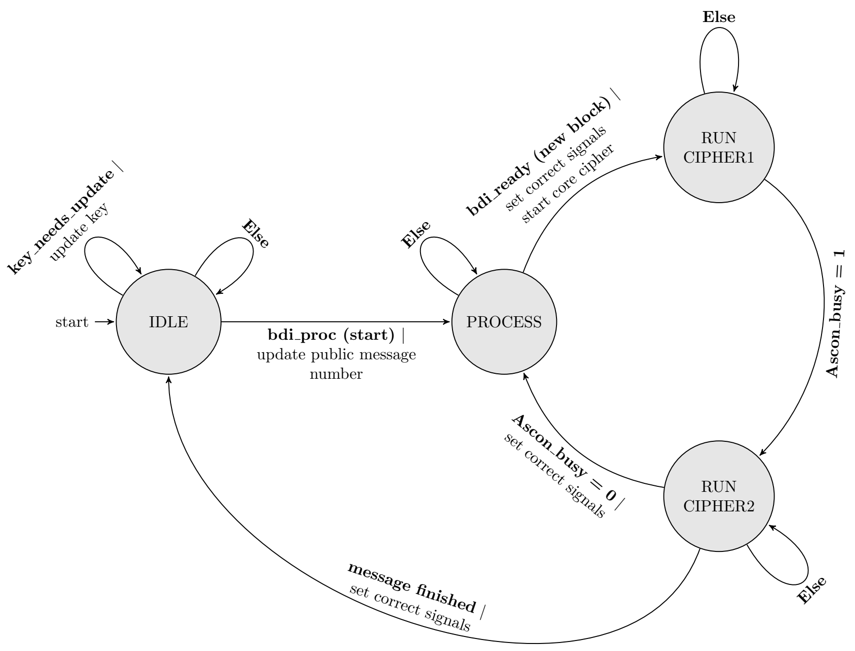

IDLE
start
key_needs_update / update key
Else
bdi_proc (start) / update public message number
PROCESS
Else
bdi_ready (new block) / set correct signals start core cipher
RUN CIPHER1
Else
Ascon_busy = 1
RUN CIPHER2
Else
Ascon_busy = 0 / set correct signals
message finished / set correct signals

Figure 7.4: The Finite state machine of the CipherCore of Ascon

Just as in Morus, the internal state needs to be stored between the permutations. To keep the majority of the functionality in the kernel, this state is stored inside the kernel. Some small additional operations need to be done on the state, like XORing part of the state with the key after initialization. Since the state is stored inside of the kernel, the logic for this needs to be placed there as well. The kernel has an external *mode* port so the CipherCore can tell it which of these special operations need to be done (more on this in Section 7.2.2).

Because all this functionality is inside the kernel, the logic that is needed outside of the kernel, to link it with the GMU API, is quite small. The necessary data inputs are supplied straight from the API to the kernel, and the kernel has four ports to communicate with the controller. The controller can specify the mode and the current block length (this is needed during the decryption of the last block), and a start and busy signal are used for the communication through a handshake signal.

The datapath and the controller used to control the kernel are shown in Figures 7.3 and 7.4. The controller works the same as the controller of the Joltik CipherCore, and was already explained in that section (Section 5.2.2).

7.2.2 The Kernel

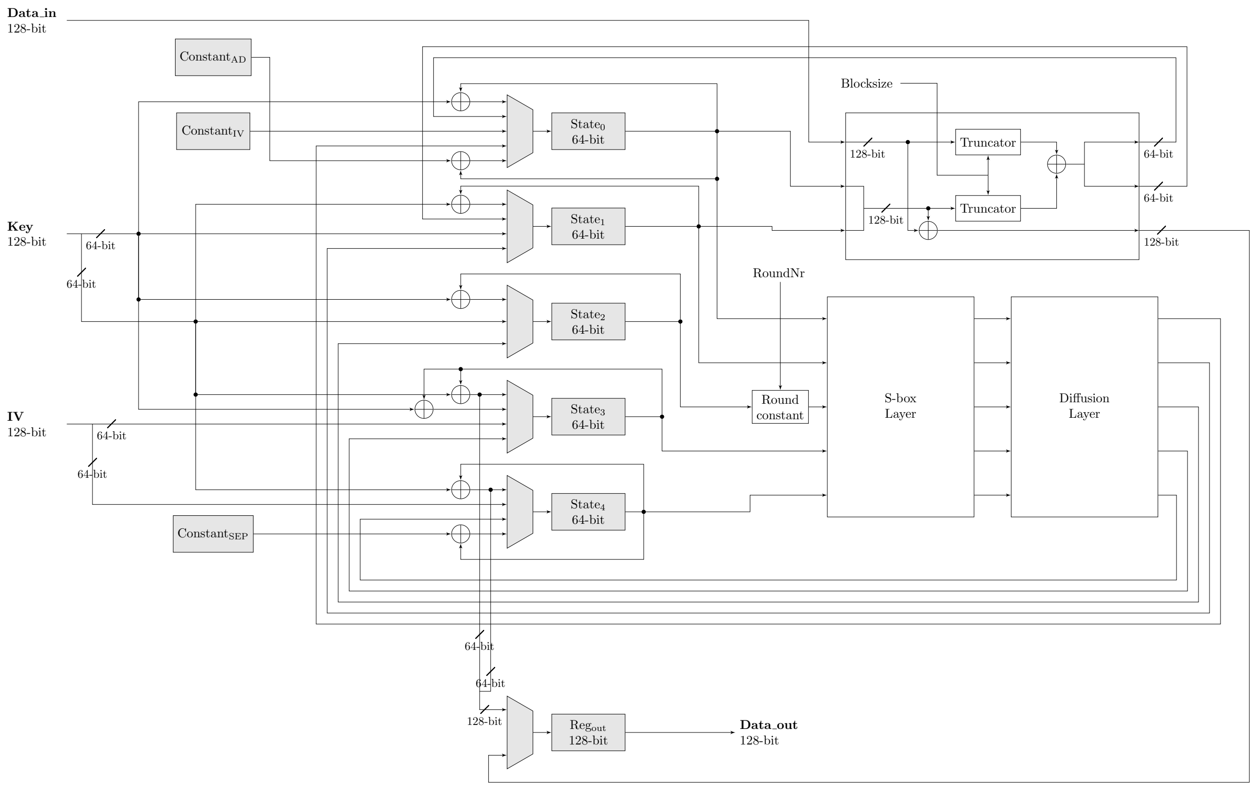


Figure 7.5: The datapath of the iterated implementation of Ascon

The main part of the kernel is the permutation function. To support all modes of operation, the hardware implementations also contain additional logic to for example XOR part of the state with the key (these additional operations are visible in Figure 7.1). There are a fair amount of these additional operations because the special padding of Ascon is not supported by the GMU API (see Section 7.1.1).

The datapath of the iterated implementation of the kernel is shown in Figure 7.5. Left of the state registers the additional logic is visible, on the bottom right the logic for one round of the permutation is visible, and the top right shows the logic to generate the output. The *truncators* are needed when the last block during decryption is not a full block. Since it is an iterated implementation, the logic for one round of the permutation is present and the cipher is executed at the speed of one round per clock cycle.

The S-box and diffusion layer are implemented in LUTs (see Section 7.1.2 for their specifics). An additional register is used to hold the output so the communication with the kernel is more flexible. The kernel for the Ascon-128a version is shown on the figures. The Ascon-128 version has a smaller block size of only 64 bit, so its datapath looks slightly different.

The kernel can operate in nine modes, which include initialization, finalization, associated data, encryption and decryption. But also some additional modes to handle the special cases when the block needs to be padded. The FSM of the controller is shown in Figure 7.6. It stays fairly simple, because most of these modes are only slight modifications of the standard modes.

7.3 The Serialized Implementations

Ascon can be serialized to a high factor without much overhead (see Section 3.2.2 in the literature review). To have a comparable implementation compared to the serialized implementations of Joltik and Morus, the serialized implementation of Ascon serializes its S-boxes with a factor 4. This section contains the description of this implementation.

The main parts of a round of the Ascon permutation, are the S-boxes and linear diffusion layer. Unfortunately this second part can only be computed after the first part is done. If a round is serialized, the S-boxes and linear diffusion layer will have to be calculated in separate stages. In this implementation the S-boxes are serialized with a factor 4, and the linear diffusion layer is calculated separately for each state register.

This leads to the fairly complicated structure shown in Figure 7.7. Compared to the iterated implementation no additional registers are needed, but some additional multiplexers are. First the S-box layer is executed in four clock cycles. 16 bits from each 64-bit state are selected, put through the S-boxes and written back to the same part of the state. Then in five clock cycles, each 64-bit state part is put through the linear diffusion layer. Three multiplexers select the correct state part and shifting value.

Now, one round takes nine clock cycles. The logic from Figure 7.7 is fit in the iterated datapath from Figure 7.5 so all the additional required functionality is still

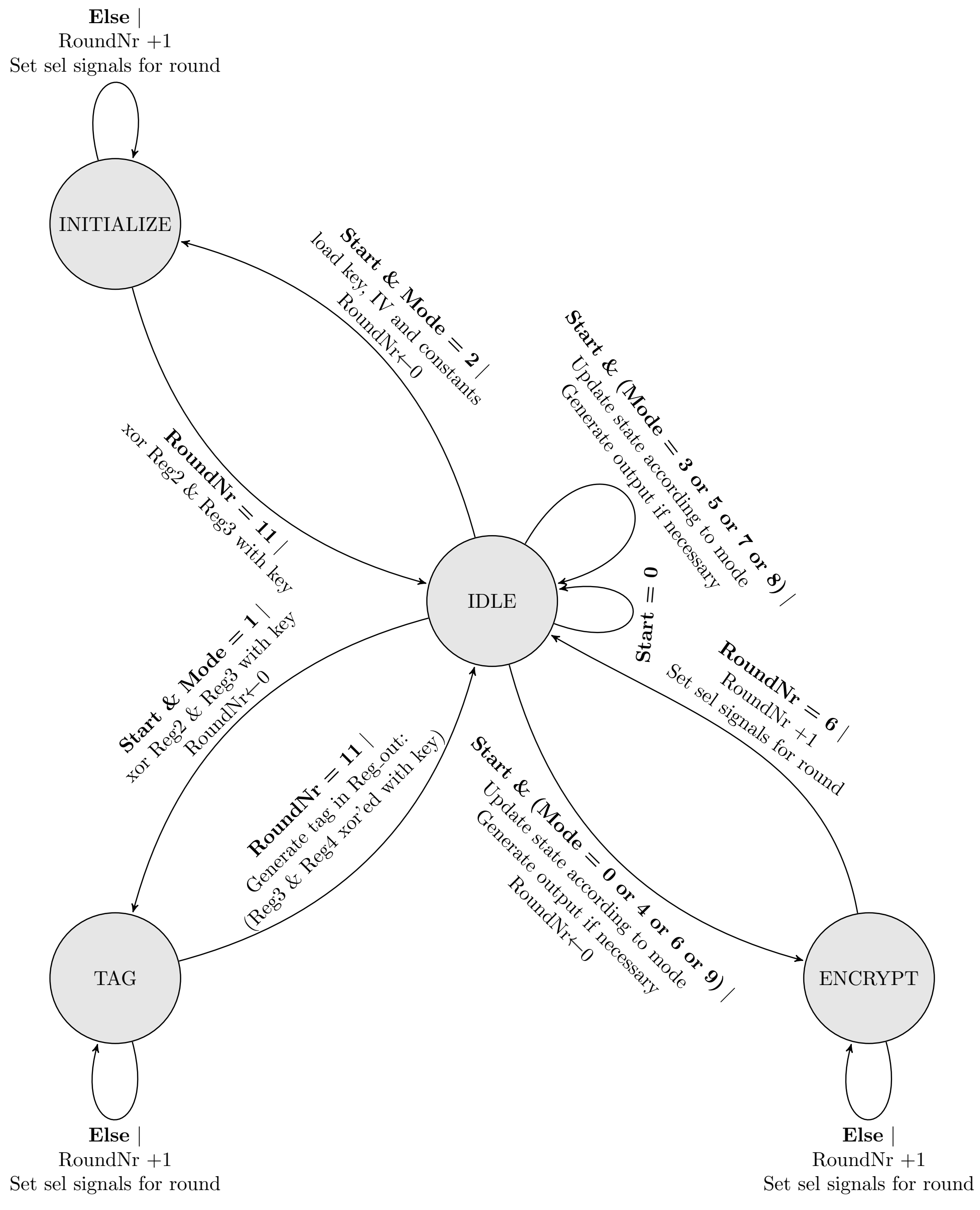

Else |
RoundNr +1
Set sel signals for round
INITIALIZE
Start & Mode = 2 |
load key, IV and constants
RoundNr←0
RoundNr = 11 |
xor Reg2 & Reg3 with key
Start & (Mode = 3 or 5 or 7 or 8) |
Update state according to mode
Generate output if necessary
Start = 0
IDLE
Start & Mode = 1 |
xor Reg2 & Reg3 with key
RoundNr←0
RoundNr = 6 |
RoundNr +1
Set sel signals for round
RoundNr = 11 |
Generate tag in Reg_out:
(Reg3 & Reg4 xor'ed with key)
Start & (Mode = 0 or 4 or 6 or 9) |
Update state according to mode
Generate output if necessary
RoundNr←0
TAG
ENCRYPT
Else |
RoundNr +1
Set sel signals for round
Else |
RoundNr +1
Set sel signals for round

Figure 7.6: The Finite state machine of the iterated implementation of Ascon

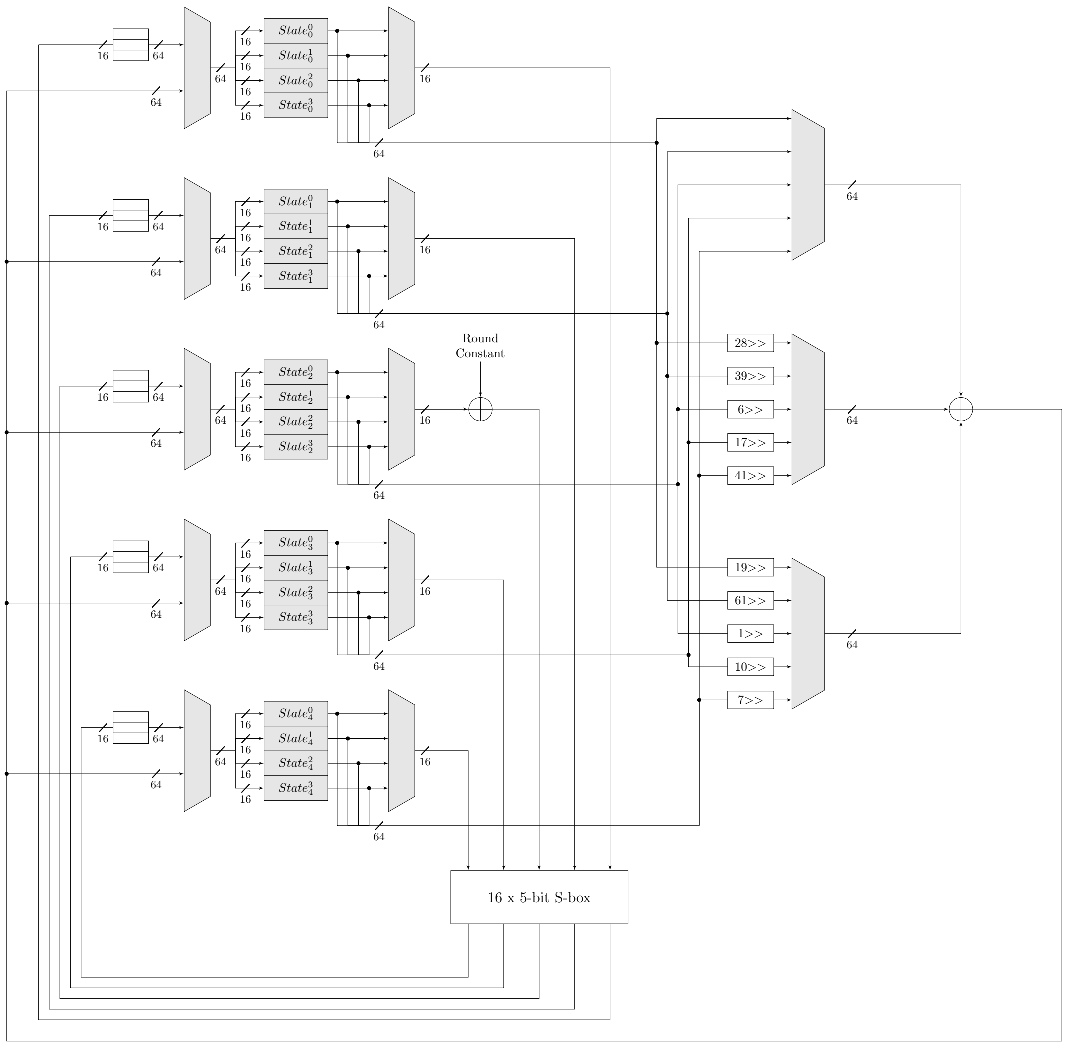


Figure 7.7: The structure of the serialized implementation of Ascon

present. Externally the serialized kernel functions exactly the same as the iterated kernel, and differs only in the number of clock cycles used for a state update.

7.4 The Unrolled Implementations

Ascon does not have as much potential for unrolling as Morus, but its critical path is still shorter than that of Joltik. The relative performance of the unrolled Ascon implementation is expected to lie in between the two other ciphers. In Section 3.2.2 of the literature review, the unrolling performance of Ascon was analyzed on ASIC. The unrolled implementations performed slightly better energy-wise than the iterated implementation, but the difference was small. It is possible different results will be obtained on FPGA.

All unrolling factors that still allow the initialization and normal operation to be executed in a whole number of cycles, are implemented. This is shown in Table 7.2. The implementations are fairly straightforward: the S-box layer, linear diffusion layer and round constant addition are copied and placed after each other. As many times, as the unrolling factor that is used.

Name	**Rounds begin and end**	**Rounds datablock**	**unrolling factors implemented**
$Ascon-128a$	12	8	2,4
$Ascon-128$	12	6	2,3,6

Table 7.2: All the unrolled implementations for Ascon-128a and Ascon-128

7.5 The Measurements Results

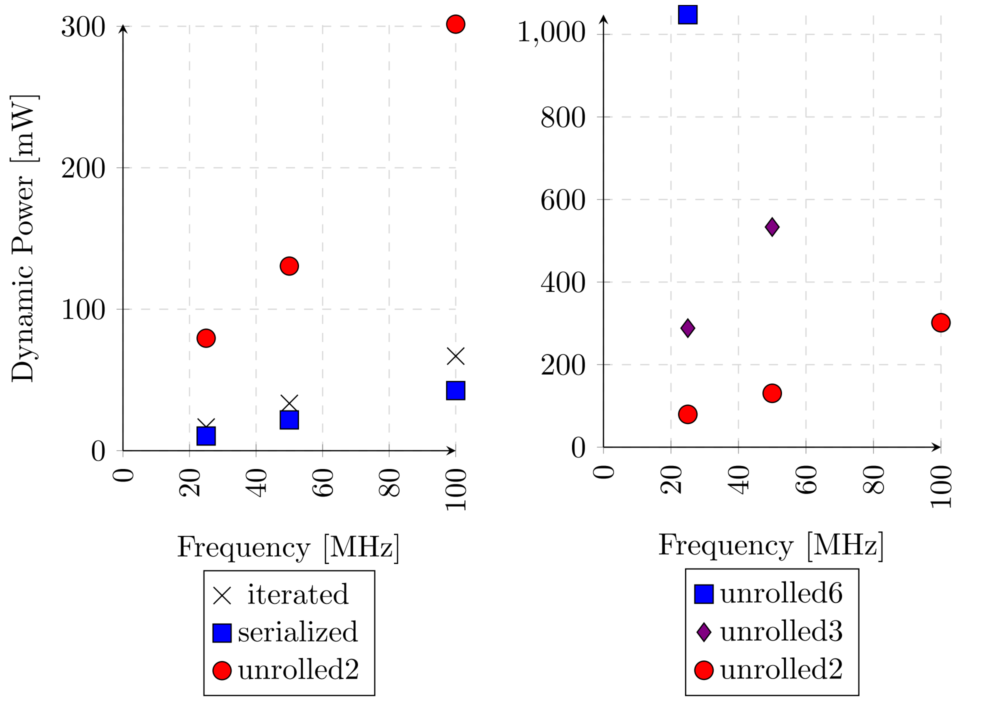


Figure 7.8: The dynamic power consumption of the Ascon-128 implementations

Four implementations were made of Ascon-128a, and five implementations of Ascon-128. The same measuring setup as for Joltik and Morus is used, which is described in Section 4.5. The maximum frequency and area of the implementations are determined by synthesizing the implementations with a wrapper at different clock speed constraints. These results are presented in this section, but a more detailed description can be found in Appendix C.

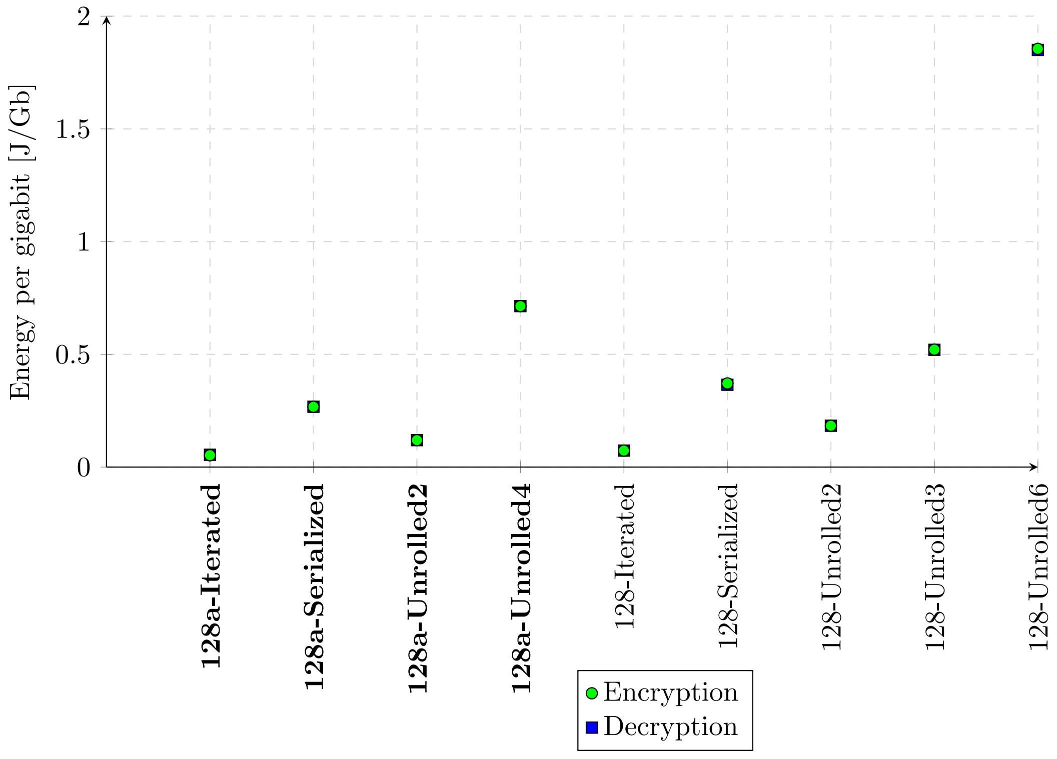


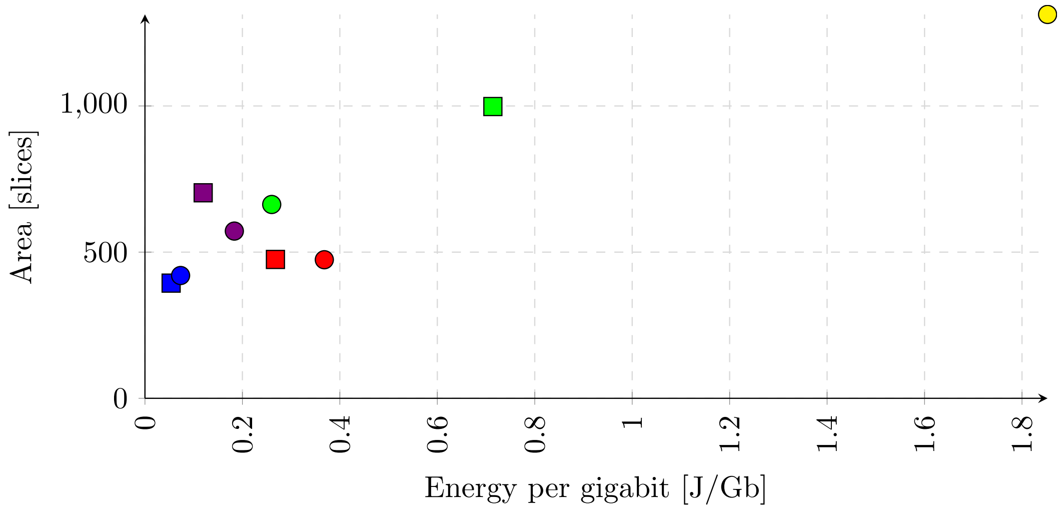
Figure 7.9: The energy consumption of the Ascon implementations

The dynamic power consumption of the Ascon-128 implementations is shown in Figure 7.8. The power consumption of the Ascon-128a implementations have a similar pattern. The power follows linearly with the frequency. This means that the energy consumption per data block of one implementation is the same at all frequencies. When these are averaged over the frequencies, the results in Figure 7.9 are obtained. The energy spent for an encryption and decryption are the same within the margin of error. This is expected because the operations during encryption and decryption are almost exactly the same (see Section 7.1.2).

When the energy consumption is compared to the area and maximum throughput of the implementations, the results shown in Figure 7.10 are obtained. Just as in Joltik, too much unrolling gives objectively worse results. Any unrolling beyond two leads to a slower and less energy-efficient implementation. In Ascon-128 an unrolling of two already gives worse results than the iterated implementation. Serialization also gives worse results on these two factors.

Energy wise it seems that Ascon-128 performs worse than Ascon-128a, despite being slightly smaller and having less rounds. The reason is that it only processes 64 bit instead of 128 bit at a time, which halves the performance when the energy is measured per bit.

Ascon has an initialization and finalization overhead that lays in between that of Morus and Joltik. This overhead is equivalent to around 3 to 4 data blocks, both in



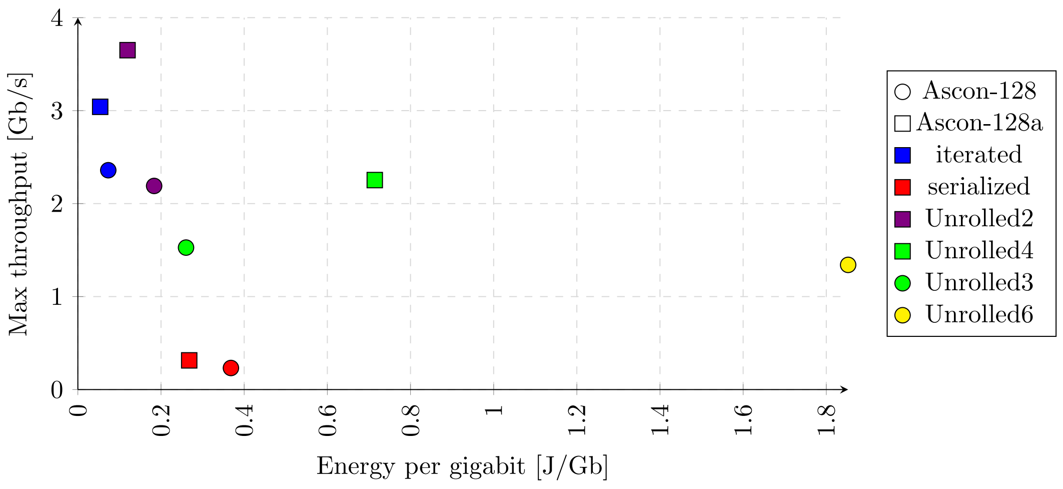


Figure 7.10: The area and maximum throughput plotted vs the energy consumption of the Ascon implementations

throughput and energy consumption.

7.6 Interpretation of the Results

From the measurement results it is visible that Ascon gives bad results when unrolled. Unrolling with a factor 2 slightly improves the maximal throughput of Ascon-128a, but at the cost of a larger area and energy consumption. All other unrolling factors for Ascon-128 or Ascon-128a give worse results on all three analyzed factors compared to the iterated implementation. These results get exponentially worse when the unrolling factor is increased. E.g the energy consumption of unrolling Ascon-128a six times is 3.5 times as worse, compared to unrolling it three times.

On ASIC the energy consumption improved when unrolling Ascon, although only by a small amount (see Section 3.2.2 in the literature review). Because the logic is implemented in LUTs and connected through a routing matrix, this leads to a vastly longer critical path on the FPGA. So, the power consumption due to glitches has a bigger effect than in the ASIC implementations. This long critical path also leads to a lower maximum frequency, which causes a lower maximum throughput, despite that processing a block takes less clock cycles.

Serialization does not offer any benefits either. The serialized implementations have a higher energy consumption, larger area and lower maximal throughput than the iterated implementations. The area is 13% and 20% higher for respectively Ascon-128 and Ascon-128a. In the ASIC implementations (see Section 3.2.2), the area of serialized implementation amounted only to 35% of that of the iterated implementation. This large difference with the results on FPGA is due to the higher serialization factor. The implementation on ASIC serialized the S-box with a factor 64, instead of 4, and fully serialized the linear diffusion layer as well. One round took 512 cycles in their implementation.

It seems the overhead of the serialization in the FPGA implementation, outweighs the benefits of the reduced logic. If the S-boxes would be serialized with a higher factor, shift registers could be used to store the state. This would eliminate the multiplexers after the state registers in Figure 7.7, and lead to a reduction of the amount of registers. This setup could give better area results than the one used.

Implementation	**Area (Slices)**	**LUTs**	**Registers**	**Energy (J/Gb)**	**max throughput (Gb/s)**
128a-Iterated	394	1427	495	0.054	3.041
128a-Unrolled2	703	2341	495	0.119	3.650
128-Iterated	420	1297	474	0.073	2.359

Table 7.3: The measurement results for the best Ascon implementations

In Table 7.3, the area, maximum throughput and energy consumption of the implementations that offer relevant trade-offs are listed. Only the unrolling with factor 2 of Ascon-128a offers any benefit over the iterated implementation. A small 20% improvement in maximal throughput at the cost of 78% more area and a 2.2 times worse energy consumption.

The initialization and finalization overhead are quite small. A message of 1.5-3.5 kB will already reach 90% of the speed and energy consumption values in Table 7.3. This overhead in Ascon will be compared to that of Joltik and Morus in the next chapter, along with all the other results.

7.7 Conclusion

All the implementations of Ascon and their performance were discussed in this chapter. Four implementations were made of Ascon-128a, and five implementations were made of Ascon-128. One iterated and serialized implementation for each of these versions, and respectively two and three unrolled implementations.

Almost all the serialized and unrolled implementations gave worse results in area, energy consumption and maximum throughput than the iterated implementations. Only the unrolled implementation with factor two of Ascon-128a gave a slightly higher throughput. Some possible explanations for these results were discussed.

All ciphers and their implementations have now been presented, measured and discussed. In the next chapter the ciphers will be compared with each other, and the claims of their authors will be evaluated. The strengths and weaknesses of each cipher will be identified as well.

Chapter 8

Comparison of the Ciphers

In the past three chapters, the implementations of the three ciphers have been treated. They were presented, measured, and their results were discussed. In this chapter, the results from the three ciphers are combined. It is important to keep in mind that all measurements are done on a Spartan 6 FPGA. The same conclusions can not necessarily be applied to hardware implementations on ASICs.

In *Section 1*, conclusions are formed about how the ciphers performed on energy, area and speed. And *Section 2* analyzes the effectiveness of the optimizations. Certain optimizations gave better results than others, depending on the type of cipher. This could be useful information during the optimization of other ciphers. The exact measurement results of the implementations can be found in Appendix A, B and C.

8.1 The Performance of the Ciphers

In this section, the ciphers are compared on the three important factors of energy, area and speed. In low-energy applications like wireless sensors, the energy consumption and chip area are the most important design factors. Those applications often do not require continuous transfer of data, making speed less of an issue. This trade-off is discussed in the first part of this section. The second part analyzes the more classical trade-off between area and speed. The claims of the authors of the ciphers in regard to these two factors are discussed.

8.1.1 The Trade-off Between Energy and Area

Speed is not an issue in many applications, but energy efficiency and chip area is. To identify the best candidates for such applications, the implementations of each cipher that offer some benefit on these two criteria are compared. This is shown in Figure 8.1. It is important to note that $Joltik - 64 - 64$, which uses Joltik-BC-128, only offers 64 bit of confidentiality. The rest of the ciphers offer 128 bit of confidentiality. If the application requires 128 bit, the light green results in Figures 8.1 and 8.2 should be ignored.

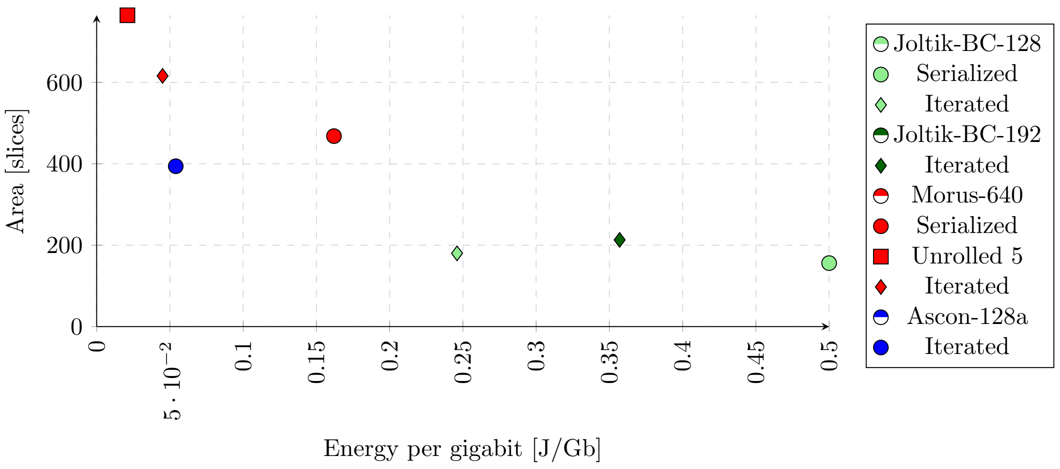


Figure 8.1: Energy versus area comparison of the best implementations of Joltik, Morus and Ascon

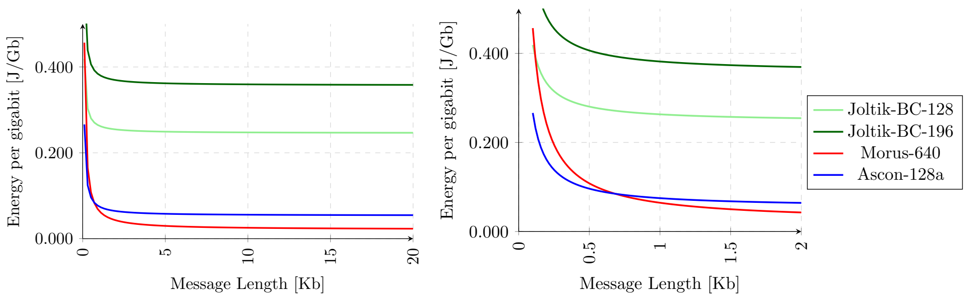


Figure 8.2: Energy efficiency in function of the message length for the most efficient implementations of Joltik, Morus and Ascon

It is visible that all ciphers have their benefits. Joltik is the smallest and least efficient cipher. Morus is the largest and most energy efficient cipher, and Ascon lies in between Joltik and Morus. The Joltik-BC-128 kernel performs better than Joltik-BC-192, but offers a lower confidentiality security.

Because of the overhead necessary for the initialization or finalization of the ciphers, the energy consumption per bit also depends on the length of a message. The values in Figure 8.1 apply only in the case where the message is sufficiently long. In Figure 8.2 the energy consumption for the most efficient implementations of each cipher are shown in function of the message length. Ascon-128a is more efficient than Morus-640 if the message is under five data blocks long (0.64 Kb). Despite their really short overhead, the Joltik implementations never beat Morus or Ascon in energy efficiency.

8.1.2 The Trade-off Between Speed and Area

In Section 2.1, it was determined what ciphers would be analyzed in the thesis. The focus of the thesis was on ciphers with potential to be used in low-energy applications. Because very little research and claims were made about the energy consumption of the ciphers in the competition, a large part of the actual decision was based on the claimed light-weight and complexity of the ciphers in hardware. In Figure 8.3 the speed and area of the cipher implementations are compared.

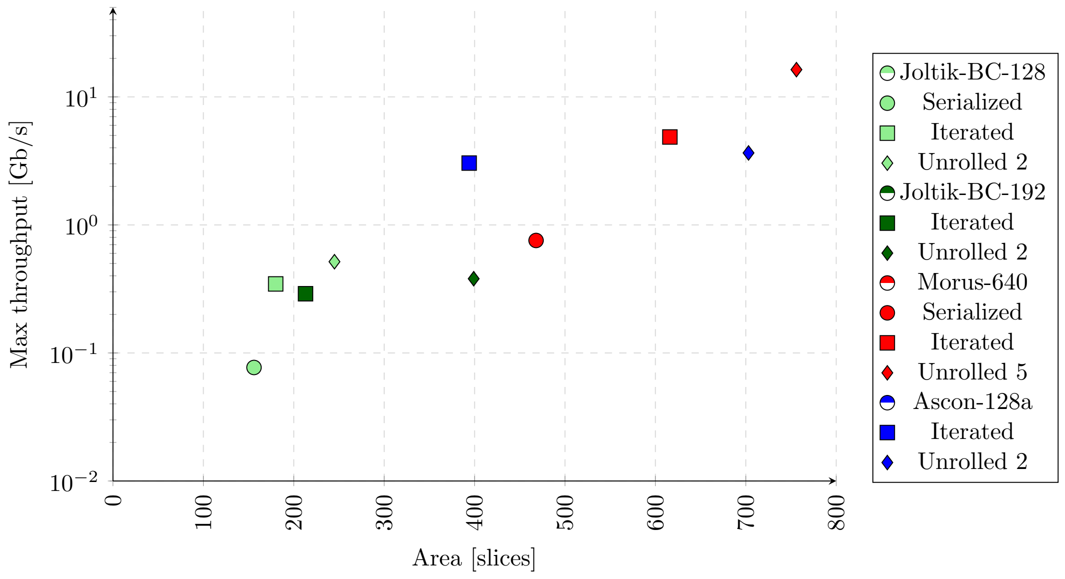


Figure 8.3: The maximum throughput in function of the area for relevant implementations of Joltik, Morus and Ascon

In the submission paper of Joltik the authors claimed that "Joltik has a low area footprint" [21]. This is confirmed in the results. Joltik is by far the smallest cipher of the three. Only 156 slices are necessary for the serialized implementation of Joltik-BC-128. It fits easily on the smallest Spartan 6 FPGA, which has only 600 slices [55], with plenty of slices left over for some small additional logic. Joltik will fit on older and cheaper models as well.

The authors of Morus claimed that Morus is designed to be fast in hardware [23]. This is also confirmed in the results. The fully unrolled implementation can reach high speeds of 16 Gb/s on the Spartan 6. Several copies of the implementation can also be placed on the same chip to reach even higher speeds by encrypting multiple messages at once.

The authors of Ascon wrote that its main features are that it is "lightweight in hardware, while still being reasonable fast". As well as allow efficient implementation of side-channel resistance features [24]. The side-channel resistance was not analyzed in this thesis, but the trade-off between its area and speed is visible in the results. The area and the speed of the iterated implementation of Ascon-128a lies in between the areas and speed of the Joltik and Morus implementations.

8.2 Global Optimizations on the Ciphers

Several implementations have been made of the three ciphers. They differ from the iterated implementations on an architectural level. They unroll or serialize multiple rounds of the ciphers. No pipelining or parallelization has been tried, which is something that could potentially be done in further research.

In this section, the potential of these improvements is discussed. The results of the improved implementations are compared to the respective iterated implementations of the ciphers. Possible causes of the results are identified, and some generalized conclusions are made. In the first part of this section, the potential of the *unrolling* of round based ciphers is discussed, and in the second part the *serialization* is discussed.

8.2.1 Unrolling

In Figure 8.4, the relative critical path delay and relative energy consumption of all the unrolled implementations are shown. Point $(1, 1)$ represents the iterated implementations. It is important to note that less cycles are needed per block in the unrolled implementations, and that this is already taken in to account in the relative energy consumption. The relative power consumption is a lot higher than the relative energy consumption. The numbers above the data points represent the unrolling factor of that implementation.

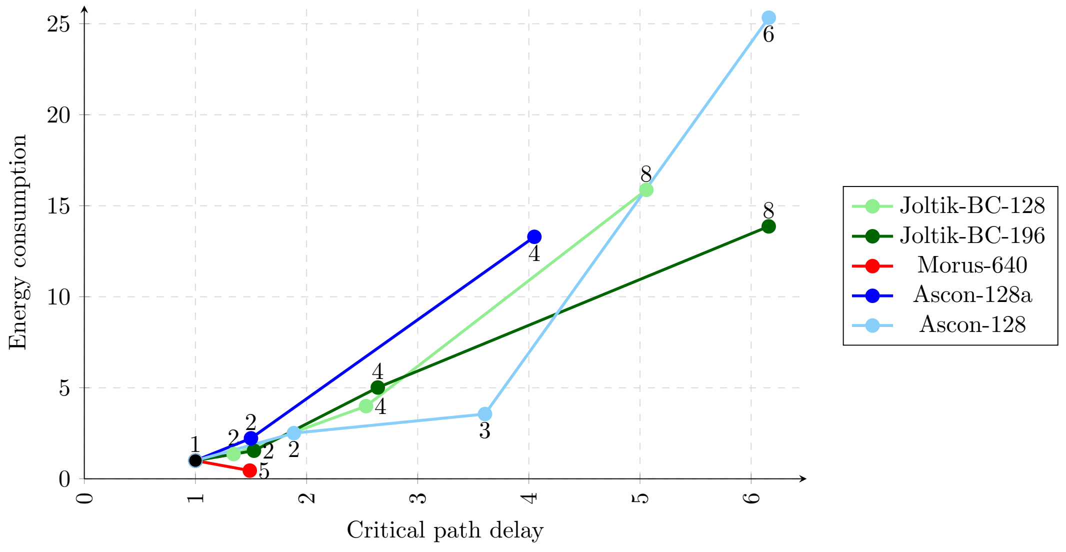


Figure 8.4: The relative change in critical path delay versus the relative change in the energy consumption of the unrolled implementations

It is visible that Morus-640 is the only cipher that responded well to unrolling. The energy consumption approximately halved, and the critical path only got about 1.5 times longer when a round was unrolled five times. For the other two ciphers the results got worse for each higher unrolling factor. A very short critical path seems to be a necessity to achieve a better energy performance through unrolling on FPGA. Once the critical path becomes too long, the energy consumption rises drastically.

When an unrolled implementation of Ascon is made on ASIC, the pattern is different (these ASIC results are discussed in Section 3.2.2 of the literature review). The energy consumption slightly dropped with each unrolling factor (they unrolled up to a factor of six). While even higher factors will probably eventually also worsen the energy consumption on ASIC, this turning point seems to be further away than in the FPGA implementations.

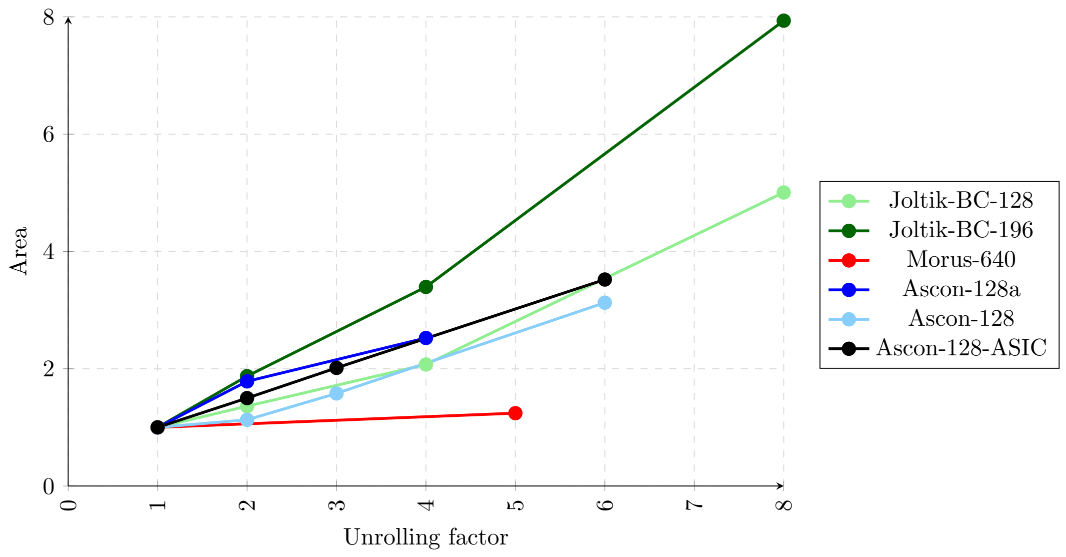


Figure 8.5: The relative area increase versus the unrolling factor of the unrolled implementations

In Figure 8.5, the relative area increase is compared to the unrolling factor. The ASIC measurements for Ascon-128 are also included [37]. The results here do correspond to the ASIC results. It is again visible that Morus responds very well to unrolling. The area only increases 25% when its rounds are unrolled with a factor five. For the other two ciphers the area increases faster. At first it increases slower than the unrolling factor, until a certain point where the area increases faster than the unrolling factor. The implementation of Joltik-BC-196 with an unrolling factor of 8 is more than twice as big than the implementation with an unrolling factor of 4.

This rapid increase of the area is not a logical result, since doubling the logic should only at worst double the area (and not increase it even more). This rapid increase is not due to overfilling of the FPGA, since less than 20% of the area was in use. A reason could be that the synthesis tool has more trouble optimizing the implementation because it is bigger. The synthesis settings for the implementations were more or less left in the standard mode, so perhaps better settings could result in more logical results.

8.2.2 Serialization

The area reduction from serialization is highly dependent on the implementation. The structure of an iterated implementation is fairly straightforward because it just uses the structure of a round. In serialized implementations several structures can be used, sometimes with extra logic or registers for temporary values. Because only a part of the state-bits are needed each round, more efficient storage than just registers can be used (such as ram).

The magnitude in which a cipher benefits from serialization is further dependent on the structure of its rounds. In some ciphers almost no extra register and logic is needed to serialize them, and some ciphers need extra logic and registers even for small amounts of serialization. Sometimes, even if the number of LUTs and register that are used, are reduced, it does not automatically lead to a reduction of the number of slices. This was the case in one of the serialized implementations of Joltik (see Section 5.6).

Not enough focus of this thesis was on the serialized implementations to make any big conclusions about the serialization of the ciphers. The individual results for the serialized implementations are discussed in the chapter of each respective cipher.

8.3 Conclusion

This chapter combined the performance results of the three ciphers. For low-energy applications like wireless sensors, the area and energy-efficiency are the most important criteria. In classical applications mostly the speed and area are important. The used architectural optimizations were examined, and their benefits for particular ciphers generalized.

Each cipher has its advantages for low-energy applications. Joltik is the smallest but least energy-efficient cipher. Morus is the largest and most efficient cipher. Ascon lies in between. In the classical comparison between speed and area, the results are similar. Joltik is the smallest and slowest cipher, Morus the largest and fastest, and Ascon lies in between. This corresponds to the claims the respective authors made in the submission paper of their cipher.

The impact of unrolling was very positive for Morus because of its very short critical path. However it had a negative effect on the energy consumption and speed of the two ciphers with a longer critical path (Joltik and Ascon). A longer critical path leads to more glitches, which have a big effect on the energy consumption. The results of the Ascon implementations were compared with ASIC implementations of Ascon, and it was found that the energy rise due to a too long critical path did not yet happen there at an unrolling factor of six.

Area-wise, the area rises slowly with the unrolling factor. A doubling of the round logic, does not lead to a doubling of area until a certain point. After which the area does rise faster than then unrolling factor. A possible reason is that the synthesis tool has more trouble optimizing when the implementation is that big, and better synthesis settings could lead to better results.

Not enough focus was on serialization to make meaningful global conclusions. The results are highly dependent on the structure of the ciphers themselves, and the way the serialization is implemented. The individual serialization for each cipher is discussed in the implementation chapter of each respective cipher.

Chapter 9

Conclusion

In the introduction of this thesis, the need for small, robust and energy-efficient encryption was stated. This requires ciphers that are small, robust and energy efficient. Three diverse authenticated ciphers from the CAESAR competition were chosen and several optimized implementations of them were made for FPGA. These are ciphers that were designed to be light-weight ciphers in hardware. The aim was to research the following three elements:

1. The trade-off between the energy consumption, area and speed for each of the three ciphers
2. The general performance of each cipher
3. A possible generalization of the results, to find effective optimization strategies for certain type of ciphers

The most important empirical findings and conclusions in regard to these three questions are described in *Section 1*. *Section 2* discusses the limitations of the research. Finally, *Section 3* places the thesis in existing research and discusses potential further research.

9.1 Empirical Findings

The specifics of the implementations and their performance can be found in the respective chapters of the ciphers. Chapter 5 for Joltik, Chapter 6 for Morus and Chapter 7 for Ascon. The chapters also include a detailed description of the ciphers themselves. In Chapter 8, the ciphers are compared and cross-cipher conclusions are made. This section will synthesize the findings to answer the three main objectives of the research.

Only the kernel of the implementations was measured. The kernel is the core of the implementation that contains the most active and important part of the cipher, without any overhead of the external interface. It consumes the majority of energy and area, and can be implemented in other designs if different external communication is required. The starting point of each cipher was an iterated implementation, which was then serialized and unrolled in various degrees in the other implementations.

9.1.1 The Trade-off between Energy, Speed and Area for Joltik, Morus and Ascon

Unrolling Joltik beyond a factor two, did not yield any benefits in either speed, area or energy efficiency. The critical path became too long, which caused the power consumption to rise too much to save any energy through the execution in less clock cycles. When Joltik was unrolled with a factor two, it caused the energy consumption and area to rise, but the maximum throughput was also increased.

The serialized implementation of Joltik had a slightly lower area at the cost of a double energy consumption and a four times as low maximum throughput. A serialized implementation where the registers were replaced with distributed ram further reduced the LUTs and registers needed, but this was not translated in to a lower number of FPGA slices. The smaller Joltik-BC-128 responded better to the optimizations than the larger Joltik-BC-196.

Fully unrolling all five rounds of Morus gave good results. The energy consumption halved and the maximum speed increased four times, at the cost of only a 25% larger area. The serialized implementation reduced the area by 25%, but at the cost of 3.5 times the energy and a 6.5 times slower maximum speed. All three implementations of Morus had their benefits, depending on how large of a priority the smaller area is.

Ascon responded a bit worse to unrolling and serializing than Joltik. An unrolling with factor two still gave a slight benefit in maximum speed, but at the cost of an almost double energy consumption and area. The serialization did not improve the area. The serialization serialized the S-box layer of Ascon by a factor four. It is possible to serialize Ascon with a higher factor without creating much extra overhead. It would also give the opportunity to use shift registers to more efficiently store the state. It is expected that this would cause an area reduction, as was the case with a similar ASIC implementation [37]. Ascon-128a performed better than Ascon-128.

9.1.2 General Performance of the Ciphers

For applications constrained by low energy and area, each of the ciphers offered some benefit. Joltik was the smallest and least efficient cipher. Morus was the largest and most efficient cipher. And Ascon lied between those two, although for short messages of under 0.5 Kb Ascon was more efficient than Morus.

A similar result was obtained when the classical trade-off between area and speed was analyzed. Joltik was the smallest and slowest cipher, Morus the largest and fastest cipher, and Ascon lied in between. This corresponded to the claims of the respective authors of the ciphers. Joltik was claimed to be very light-weight in hardware, Morus to be fast, and Ascon to offer a trade-off between a small area and a reasonable speed.

9.1.3 Effective Optimization Strategies

The response of the ciphers to unrolling seemed to be very dependent on their internal structure. A short critical path equaled a very good response to unrolling (as was the case with Morus). A longer critical path quickly led to a higher energy consumption, that rose higher the more the cipher was unrolled. When unrolled ASIC implementations [37] were compared with those on FPGA, it seemed that this critical path could be a lot longer on ASIC before the unrolling had a negative effect on the energy consumption.

Area-wise unrolling led to a slow increase in area, which was less steep in the Morus cipher because of the short critical path. The FPGA implementations followed a similar pattern in area as the unrolled ASIC implementations. Only at high unrolling factors the area rose faster than would be expected (double the logic lead to more than double the area). This could be due to less optimal settings of the synthesis tool for larger designs.

Global conclusions about the response of ciphers to serialization were hard to make. The benefits of serialization depended highly on how this serialization was implemented. Not enough of those implementations have been made in this thesis to draw conclusions. The implementations that have been made do not show much consistency in their results.

9.2 Limitations

The study analyzed the hardware performance of three ciphers that showed potential to be used in energy constrained applications. As a consequence of the variability of hardware and hardware implementations, some limitations were encountered.

Results can differ greatly on different hardware, especially between ASIC and FPGA large differences exist. Comparing the results of this thesis to implementations on other platforms, or even to the same platform with other settings and versions of synthesis software will not be so accurate.

The implementations themselves are optimized, but there is always some room for improvement. Especially in the synthesis software more effective settings could be used for better results. Not many settings have been tried, and no extra constraints aside from clock, input and output constraints have been supplied to improve the results.

However special care was taken that all implementations followed the same structure. The same hardware platform and environment factors were used during measurement. The same synthesis software and settings were used. And the optimization strategy was also the same for all the implementations. So for their comparison in this thesis, most of those influencing factors have been eliminated and the comparisons can be expected to be accurate.

9.3 Further Direction of Research

The energy and area consumption of hardware implementations of three ciphers from the CAESAR competition were analyzed in this thesis. Since low-energy applications are becoming more and more common, energy efficiency in hardware is an important factor in the evaluation of a cipher. Unfortunately relatively little research is available about the energy efficiency of cryptographic ciphers in hardware, and there are many obstacles in comparing implementations on different platforms.

Not all possible implementations of the ciphers were explored in this thesis. Mainly parallelization and pipelining might yield interesting results. There are also other hardware oriented ciphers in the CAESAR competition, whose energy usage can be analyzed as well. There is a lot of potential work left in this area.

As a final note I would like to conclude that all three ciphers are worthy candidates in the second round of the CAESAR competition. They have their own strengths and weaknesses when implemented on FPGA, and appropriate applications can be found for all three ciphers.

Appendices

Appendix A

Measurement details for the Joltik implementations

This appendix contains the detailed measurement results of the Joltik implementations.

The energy measurements for the Joltik-BC-192 and Joltik-BC-128 implementations are listed in Tables A.1 and A.2 respectively. Most columns are self-explanatory. The *real power column* is the dynamic power weighted by the *activity* of the measuring setup. The measuring setup waits 3 cycles between each subsequent encryption or decryption. Thus the real dynamic power that the design uses is higher than the measured one. The proportion of these *wasted* cycles was determined by simulation in Modelsim.

The area was measured by a different setup. Joltik-BC was wrapped in a a simple *serial-in, parallel-out* wrapper and synthesized for the same Spartan 6 that was used for the measurements (xc6slx45 csg324-3). The clock was constrained, and the synthetisation and implementation steps were repeated multiple times until the maximum clock speed was attained.

These results are shown in Tables A.3 and A.4. The *real register* column lists the synthethised registers minus the registers used for the wrapper. So it contains the real number of registers used in the implementations. The average energy is the average of the 100 and 50 MHz measurements (since these are the most accurate, as explained in Section 5.5), which is then averaged over encryption and decryption.

To determine the energy consumption, information about the number of cycles that the implementations use to encrypt or decrypt a block are needed. This information also makes it possible to calculate the time and energy that the message overhead requires (in the case of Joltik, it consists of the single encryption at the end to generate the tag). These values are displayed in Tables A.5 and A.6. This overhead is expressed relative to the average time and energy needed to process a single block of data (64-bit in Joltik).

Version	Frequency (Mhz)	Mode	Avg. Static Power (Watt)	Avg. Total Power (Watt)	Avg. Dynamic Power (Watt)	Real Power (Watt)	Energy per block (J/64-bit)	Energy per gigabit (J/Gb)
Iterated	100	Encrypt	0.190	0.257	0.067	0.073	2.48E-08	0.387
Iterated	50	Encrypt	0.188	0.222	0.034	0.037	2.52E-08	0.393
Iterated	25	Encrypt	0.186	0.205	0.019	0.021	2.81E-08	0.439
Iterated	100	Decrypt	0.190	0.246	0.056	0.061	2.07E-08	0.324
Iterated	50	Decrypt	0.188	0.216	0.028	0.030	2.07E-08	0.324
Iterated	25	Decrypt	0.186	0.203	0.017	0.019	2.52E-08	0.393
Serialized4	100	Encrypt	0.190	0.228	0.038	0.039	5.17E-08	0.808
Serialized4	50	Encrypt	0.189	0.208	0.019	0.019	5.17E-08	0.807
Serialized4	25	Encrypt	0.187	0.199	0.012	0.013	6.53E-08	1.020
Serialized4	100	Decrypt	0.190	0.226	0.036	0.037	4.90E-08	0.765
Serialized4	50	Decrypt	0.189	0.209	0.020	0.020	5.44E-08	0.850
Serialized4	25	Decrypt	0.187	0.197	0.010	0.010	5.44E-08	0.850
Serialized4_ram	100	Encrypt	0.191	0.238	0.047	0.048	6.77E-08	1.058
Serialized4_ram	50	Encrypt	0.189	0.213	0.024	0.025	6.91E-08	1.080
Serialized4_ram	25	Encrypt	0.187	0.200	0.013	0.013	7.49E-08	1.170
Serialized4_ram	100	Decrypt	0.191	0.237	0.046	0.047	6.81E-08	1.064
Serialized4_ram	50	Decrypt	0.189	0.211	0.022	0.022	6.51E-08	1.018
Serialized4_ram	25	Decrypt	0.187	0.199	0.012	0.012	7.10E-08	1.110
Unrolled2	100	Encrypt	0.203	0.381	0.178	0.209	3.56E-08	0.556
Unrolled2	50	Encrypt	0.201	0.289	0.088	0.104	3.52E-08	0.550
Unrolled2	25	Encrypt	0.198	0.242	0.044	0.052	3.52E-08	0.550
Unrolled2	100	Decrypt	0.203	0.378	0.175	0.206	3.50E-08	0.547
Unrolled2	50	Decrypt	0.201	0.289	0.088	0.104	3.52E-08	0.550
Unrolled2	25	Decrypt	0.198	0.243	0.045	0.053	3.60E-08	0.563
Unrolled4	50	Encrypt	0.199	0.653	0.454	0.605	1.09E-07	1.703
Unrolled4	25	Encrypt	0.191	0.439	0.248	0.331	1.19E-07	1.860
Unrolled4	50	Decrypt	0.199	0.665	0.466	0.621	1.12E-07	1.748
Unrolled4	25	Decrypt	0.191	0.438	0.247	0.329	1.19E-07	1.853
Unrolled8	25	Encrypt	0.206	1.171	0.965	1.544	3.09E-07	4.825
Unrolled8	25	Decrypt	0.206	1.224	1.018	1.629	3.26E-07	5.090

Table A.1: Measurement results for the Joltik-BC-192 implementations on a Digilent ADEPT setup

Version	Frequency (Mhz)	Mode	Avg. Static Power (Watt)	Avg. Total Power (Watt)	Avg. Dynamic Power (Watt)	Real Power (Watt)	Energy per block (J/64-bit)	Energy per gigabit (J/Gb)
Iterated	100	Encrypt	0.201	0.256	0.055	0.061	1.60E-08	0.249
Iterated	50	Encrypt	0.196	0.227	0.031	0.035	1.80E-08	0.281
Iterated	25	Encrypt	0.195	0.210	0.015	0.017	1.74E-08	0.272
Iterated	100	Decrypt	0.201	0.249	0.048	0.054	1.39E-08	0.218
Iterated	50	Decrypt	0.196	0.222	0.026	0.029	1.51E-08	0.236
Iterated	25	Decrypt	0.195	0.207	0.012	0.013	1.39E-08	0.218
Serialized4	100	Encrypt	0.182	0.212	0.030	0.031	3.12E-08	0.488
Serialized4	50	Encrypt	0.183	0.199	0.016	0.016	3.33E-08	0.520
Serialized4	25	Encrypt	0.182	0.190	0.008	0.008	3.33E-08	0.520
Serialized4	100	Decrypt	0.182	0.213	0.031	0.032	3.22E-08	0.504
Serialized4	50	Decrypt	0.183	0.198	0.015	0.015	3.12E-08	0.488
Serialized4	25	Decrypt	0.182	0.190	0.008	0.008	3.33E-08	0.520
Serialized4_ram	100	Encrypt	0.186	0.222	0.036	0.037	4.03E-08	0.630
Serialized4_ram	50	Encrypt	0.185	0.206	0.021	0.022	4.70E-08	0.735
Serialized4_ram	25	Encrypt	0.183	0.194	0.011	0.011	4.93E-08	0.770
Serialized4_ram	100	Decrypt	0.186	0.220	0.034	0.035	3.94E-08	0.616
Serialized4_ram	50	Decrypt	0.185	0.204	0.019	0.020	4.41E-08	0.689
Serialized4_ram	25	Decrypt	0.183	0.193	0.010	0.010	4.64E-08	0.725
Unrolled2	100	Encrypt	0.188	0.309	0.121	0.149	1.94E-08	0.303
Unrolled2	50	Encrypt	0.184	0.252	0.068	0.084	2.18E-08	0.340
Unrolled2	25	Encrypt	0.183	0.217	0.034	0.042	2.18E-08	0.340
Unrolled2	100	Decrypt	0.188	0.319	0.131	0.161	2.10E-08	0.328
Unrolled2	50	Decrypt	0.184	0.257	0.073	0.090	2.34E-08	0.365
Unrolled2	25	Decrypt	0.183	0.218	0.035	0.043	2.24E-08	0.350
Unrolled4	50	Encrypt	0.188	0.487	0.299	0.427	5.98E-08	0.934
Unrolled4	25	Encrypt	0.182	0.330	0.148	0.211	5.92E-08	0.925
Unrolled4	50	Decrypt	0.188	0.521	0.333	0.476	6.66E-08	1.041
Unrolled4	25	Decrypt	0.182	0.346	0.164	0.234	6.56E-08	1.025
Unrolled8	25	Encrypt	0.186	1.042	0.856	1.498	2.40E-07	3.745
Unrolled8	25	Decrypt	0.192	1.120	0.928	1.624	2.60E-07	4.060

Table A.2: Measurement results for the Joltik-BC-128 implementations on a Digilent ADEPT setup

Version	Max Frequency (Mhz)	Slices	LUT's	Register	Real registers	Max throughput (Gb/s)	Energy per gigabit (J/Gb)
Iterated	154	213	683	591	267	0.290	0.357
Serialized	135	215	685	653	329	0.065	0.808
SerializedRam	123	196	604	445	121	0.056	1.055
Unrolled2	101	399	1359	596	272	0.380	0.551
Unrolled4	58	723	2592	598	274	0.415	1.791
Unrolled8	25	1690	5384	636	312	0.320	4.958

Table A.3: Synthesizing with wrapper on xc6slx45 csg324-3 for the Joltik-BC-192 implementations

Version	Max Frequency (Mhz)	Slices	LUT's	Register	Real registers	Max throughput (Gb/s)	Energy per gigabit (J/Gb)
Iterated	141	180	555	464	205	0.346	0.246
Serialized	122	156	457	513	254	0.077	0.500
SerializedRam	115	170	444	368	109	0.068	0.668
Unrolled2	105	245	863	460	201	0.516	0.334
Unrolled4	56	373	1382	459	200	0.508	0.981
Unrolled8	28	901	2503	458	199	0.445	3.903

Table A.4: Synthesizing with wrapper on xc6slx45 csg324-3 for the Joltik-BC-128 implementations

Version	Cycles for encryption	Cycles for decryption	Cycles for message overhead	Time ratio overhead/message block	Energy ratio overhead/message block
Iterated	34	34	31	1.00	1.10
Serialized	133	133	133	1.00	1.00
SerializedRam	141	145	141	0.99	1.01
Unrolled2	17	17	17	1.00	1.00
Unrolled4	9	9	9	1.00	1.00
Unrolled8	5	5	5	1.00	0.97

Table A.5: The amount of clock cycles the operations take in the Joltik-BC-196 implementations

Version	Cycles for encryption	Cycles for decryption	Cycles for message overhead	Time ratio overhead/message block	Energy ratio overhead/message block
Iterated	26	26	26	1.00	1.08
Serialized	101	101	101	1.00	1.00
SerializedRam	109	113	109	0.99	1.01
Unrolled2	13	13	13	1.00	1.00
Unrolled4	7	7	7	1.00	0.96
Unrolled8	4	4	4	1.00	0.96

Table A.6: The amount of clock cycles the operations take in the Joltik-BC-128 implementations

Appendix B

Measurement details for the Morus implementations

This appendix contains the detailed the measurement results of the Morus implementations.

The energy measurements for Morus-640 implementations are listed in Table B.1. The columns have the same meaning as in the Joltik-BC measurements (see Appendix A). An extra measurement is done in Morus, the *empty* measurement, which measures only the initialization and finalization stages. The *energy per block* value of these measurements is the total energy spent during one initialization and finalization.

The results of the area measurements are displayed in Table B.2. The same setup was used as in Joltik, and the columns have the same meaning (see Appendix A). The clock cycles needed to encrypt a block of data, and the relative overhead of the initialization and finalization are listed in Table B.3.

Version	Frequency (Mhz)	Mode	Avg. Static Power (Watt)	Avg. Total Power (Watt)	Avg. Dynamic Power (Watt)	Real Power (Watt)	Energy per block (J/64-bit)	Energy per gigabit (J/Gb)
Iterated	100	Encr	0.202	0.285	0.083	0.116	5.81E-09	0.0454
Iterated	50	Encr	0.195	0.235	0.040	0.056	5.60E-09	0.0438
Iterated	25	Encr	0.192	0.214	0.022	0.031	6.16E-09	0.0481
Iterated	100	Decr	0.202	0.288	0.086	0.120	6.02E-09	0.0470
Iterated	50	Decr	0.195	0.236	0.041	0.057	5.74E-09	0.0448
Iterated	25	Decr	0.192	0.215	0.023	0.032	6.44E-09	0.0503
Iterated	100	Empty	0.202	0.300	0.098	0.104	1.27E-07	n.a.
Iterated	50	Empty	0.195	0.243	0.048	0.051	1.24E-07	n.a.
Iterated	25	Empty	0.192	0.219	0.027	0.029	1.40E-07	n.a.
Serialized4	100	Encr	0.197	0.271	0.074	0.083	2.00E-08	0.1561
Serialized4	50	Encr	0.191	0.229	0.038	0.043	2.05E-08	0.1603
Serialized4	25	Encr	0.189	0.209	0.020	0.023	2.16E-08	0.1688
Serialized4	100	Decr	0.197	0.273	0.076	0.086	2.05E-08	0.1603
Serialized4	50	Decr	0.191	0.229	0.038	0.043	2.05E-08	0.1603
Serialized4	25	Decr	0.189	0.209	0.020	0.023	2.16E-08	0.1688
Serialized4	100	Empty	0.197	0.280	0.083	0.084	4.18E-07	n.a.
Serialized4	50	Empty	0.191	0.233	0.042	0.043	4.23E-07	n.a.
Serialized4	25	Empty	0.189	0.211	0.022	0.022	4.43E-07	n.a.
Unrolled5	100	Encr	0.209	0.279	0.070	0.280	2.80E-09	0.0219
Unrolled5	50	Encr	0.203	0.237	0.034	0.136	2.72E-09	0.0213
Unrolled5	25	Encr	0.200	0.217	0.017	0.068	2.72E-09	0.0213
Unrolled5	100	Decr	0.209	0.277	0.068	0.272	2.72E-09	0.0213
Unrolled5	50	Decr	0.203	0.236	0.033	0.132	2.64E-09	0.0206
Unrolled5	25	Decr	0.200	0.216	0.016	0.064	2.56E-09	0.0200
Unrolled5	100	Empty	0.209	0.334	0.125	0.159	4.47E-08	n.a.
Unrolled5	50	Empty	0.203	0.265	0.062	0.079	4.43E-08	n.a.
Unrolled5	25	Empty	0.200	0.231	0.031	0.040	4.43E-08	n.a.

Table B.1: Measurement results for the Morus-640 implementations on a Digilent ADEPT setup

Version	Max Frequency (Mhz)	Slices	LUT's	Register	Real registers	Max throughput (Gb/s)	Energy per gigabit (J/Gb)
Iterated	190	616	1965	1593	944	4.861	0.0466
Serialized	142	468	1407	1604	955	0.756	0.1624
Unrolled5	128	765	2618	1594	945	16.322	0.0210

Table B.2: Synthesizing with wrapper on xc6slx45 csg324-3 for the Morus-640 implementations

Version	Cycles for encryption	Cycles for decryption	Cycles for message overhead	Time ratio overhead/message block	Energy ratio overhead/message block
Iterated	5	5	122	24.4	22.3
Serialized	24	24	496	20.7	20.7
Unrolled5	1	1	28	28.0	16.2

Table B.3: The amount of clock cycles the operations take in the Morus-640 implementations

Appendix C

Measurement details for the Ascon implementations

This appendix contains the detailed the measurement results of the Ascon implementations.

The energy measurements for the Ascon-128a and Ascon-128 implementations are listed in Tables C.1 and C.2 respectively. The columns have the same meaning as in the Morus-640 measurements (see Appendix B).

The results of the area measurements are displayed in Tables C.3 and C.4. The same setup was used as in Joltik, and the columns have the same meaning (see Appendix A). The clock cycles needed to encrypt a block of data, and the relative overhead of the initialization and finalization stages are listen in Tables C.5 and C.6.

Version	Frequency (Mhz)	Mode	Avg. Static Power (Watt)	Avg. Total Power (Watt)	Avg. Dynamic Power (Watt)	Real Power (Watt)	Energy per block (J/64-bit)	Energy per gigabit (J/Gb)
Iterated	100	Encr	0.196	0.258	0.062	0.076	6.82E-09	0.053
Iterated	50	Encr	0.202	0.233	0.031	0.038	6.82E-09	0.053
Iterated	25	Encr	0.195	0.210	0.015	0.018	6.60E-09	0.052
Iterated	100	Decr	0.198	0.261	0.063	0.077	6.93E-09	0.054
Iterated	50	Decr	0.202	0.234	0.032	0.039	7.04E-09	0.055
Iterated	25	Decr	0.196	0.212	0.016	0.020	7.04E-09	0.055
Iterated	100	Empty	0.201	0.263	0.062	0.072	2.02E-08	n.a.
Iterated	50	Empty	0.203	0.234	0.031	0.036	2.02E-08	n.a.
Iterated	25	Empty	0.196	0.213	0.017	0.020	2.21E-08	n.a.
Serialized4	100	Encr	0.190	0.235	0.045	0.046	3.38E-08	0.264
Serialized4	50	Encr	0.185	0.207	0.022	0.023	3.30E-08	0.258
Serialized4	25	Encr	0.182	0.194	0.012	0.012	3.60E-08	0.281
Serialized4	100	Decr	0.190	0.235	0.045	0.046	3.38E-08	0.264
Serialized4	50	Decr	0.185	0.207	0.022	0.023	3.30E-08	0.258
Serialized4	25	Decr	0.182	0.194	0.012	0.012	3.60E-08	0.281
Serialized4	100	Empty	0.190	0.237	0.047	0.048	1.06E-07	n.a.
Serialized4	50	Empty	0.185	0.207	0.022	0.022	9.88E-08	n.a.
Serialized4	25	Empty	0.182	0.194	0.012	0.012	1.08E-07	n.a.
Unrolled2	100	Encr	0.209	0.412	0.203	0.284	1.42E-08	0.111
Unrolled2	50	Encr	0.183	0.301	0.118	0.165	1.65E-08	0.129
Unrolled2	25	Encr	0.190	0.244	0.054	0.076	1.51E-08	0.118
Unrolled2	100	Decr	0.206	0.408	0.202	0.283	1.41E-08	0.110
Unrolled2	50	Decr	0.184	0.303	0.119	0.167	1.67E-08	0.130
Unrolled2	25	Decr	0.191	0.245	0.054	0.076	1.51E-08	0.118
Unrolled2	100	Empty	0.209	0.414	0.205	0.264	4.22E-08	n.a.
Unrolled2	50	Empty	0.184	0.306	0.122	0.157	5.02E-08	n.a.
Unrolled2	25	Empty	0.191	0.246	0.055	0.071	4.53E-08	n.a.
Unrolled4	25	Encr	0.196	0.653	0.457	0.762	9.14E-08	0.714
Unrolled4	25	Decr	0.198	0.655	0.457	0.762	9.14E-08	0.714
Unrolled4	25	Empty	0.199	0.650	0.451	0.659	2.64E-07	n.a.

Table C.1: Measurement results for the Ascon-128a implementations on a Digilent ADEPT setup

Version	Frequency (Mhz)	Mode	Avg. Static Power (Watt)	Avg. Total Power (Watt)	Avg. Dynamic Power (Watt)	Real Power (Watt)	Energy per block (J/64-bit)	Energy per gigabit (J/Gb)
Iterated	100	Encr	0.193	0.245	0.052	0.067	4.68E-09	0.073
Iterated	50	Encr	0.179	0.205	0.026	0.033	4.68E-09	0.073
Iterated	25	Encr	0.174	0.187	0.013	0.017	4.68E-09	0.073
Iterated	100	Decr	0.194	0.246	0.052	0.067	4.68E-09	0.073
Iterated	50	Decr	0.181	0.207	0.026	0.033	4.68E-09	0.073
Iterated	25	Decr	0.176	0.189	0.013	0.017	4.68E-09	0.073
Iterated	100	Empty	0.190	0.246	0.056	0.066	1.84E-08	n.a.
Iterated	50	Empty	0.182	0.211	0.029	0.034	1.90E-08	n.a.
Iterated	25	Empty	0.177	0.192	0.015	0.018	1.97E-08	n.a.
Serialized4	100	Encr	0.200	0.241	0.041	0.042	2.34E-08	0.365
Serialized4	50	Encr	0.193	0.214	0.021	0.022	2.39E-08	0.374
Serialized4	25	Encr	0.178	0.188	0.010	0.010	2.28E-08	0.356
Serialized4	100	Decr	0.201	0.242	0.041	0.042	2.34E-08	0.365
Serialized4	50	Decr	0.194	0.216	0.022	0.023	2.51E-08	0.392
Serialized4	25	Decr	0.179	0.189	0.010	0.010	2.28E-08	0.356
Serialized4	100	Empty	0.202	0.245	0.043	0.044	9.67E-08	n.a.
Serialized4	50	Empty	0.195	0.217	0.022	0.022	9.89E-08	n.a.
Serialized4	25	Empty	0.180	0.191	0.011	0.011	9.89E-08	n.a.
Unrolled2	100	Encr	0.187	0.388	0.201	0.302	1.21E-08	0.188
Unrolled2	50	Encr	0.189	0.276	0.087	0.131	1.04E-08	0.163
Unrolled2	25	Encr	0.190	0.243	0.053	0.080	1.27E-08	0.199
Unrolled2	100	Decr	0.192	0.393	0.201	0.302	1.21E-08	0.188
Unrolled2	50	Decr	0.192	0.279	0.087	0.131	1.04E-08	0.163
Unrolled2	25	Decr	0.192	0.245	0.053	0.080	1.27E-08	0.199
Unrolled2	100	Empty	0.192	0.422	0.230	0.299	4.78E-08	n.a.
Unrolled2	50	Empty	0.193	0.288	0.095	0.124	3.95E-08	n.a.
Unrolled2	25	Empty	0.192	0.253	0.061	0.079	5.08E-08	n.a.
Unrolled3	50	Encr	0.186	0.506	0.320	0.533	3.20E-08	0.500
Unrolled3	25	Encr	0.188	0.361	0.173	0.288	3.46E-08	0.541
Unrolled3	50	Decr	0.187	0.510	0.323	0.538	3.23E-08	0.505
Unrolled3	25	Decr	0.189	0.361	0.172	0.287	3.44E-08	0.538
Unrolled3	50	Empty	0.190	0.552	0.362	0.507	1.22E-07	n.a.
Unrolled3	25	Empty	0.189	0.378	0.189	0.265	1.27E-07	n.a.
Unrolled6	25	Encr	0.191	0.931	0.740	1.480	1.18E-07	1.850
Unrolled6	25	Decr	0.196	0.938	0.742	1.484	1.19E-07	1.855
Unrolled6	25	Empty	0.198	0.851	0.653	1.045	3.34E-07	n.a.

Table C.2: Measurement results for the Ascon-128 implementations on a Digilent ADEPT setup

Version	Max Frequency (Mhz)	Slices	LUT's	Register	Real registers	Max throughput (Gb/s)	Energy per gigabit (J/Gb)
Iterated	214	394	1427	1017	495	3.041	0.0537
Serialized	179	475	1806	1068	546	0.314	0.2676
Unrolled2	143	703	2341	1017	495	3.650	0.1195
Unrolled4	53	998	1297	999	477	2.253	0.7141

Table C.3: Synthesizing with wrapper on xc6slx45 csg324-3 for the Ascon-128a implementations

Version	Max Frequency (Mhz)	Slices	LUT's	Register	Real registers	Max throughput (Gb/s)	Energy per gigabit (J/Gb)
Iterated	258	420	1297	961	474	2.359	0.073
Serialized	199	474	1517	990	503	0.232	0.368
Unrolled2	137	572	1881	936	449	2.190	0.183
Unrolled3	72	663	2466	938	451	1.527	0.260
Unrolled6	42	1313	4371	930	443	1.341	1.853

Table C.4: Synthesizing with wrapper on xc6slx45 csg324-3 for the Ascon-128 implementations

Version	Cycles for encryption	Cycles for decryption	Cycles for message overhead	Time ratio overhead/message block	Energy ratio overhead/message block
Iterated	9	9	28	3.11	3.09
Serialized	73	73	220	3.01	3.04
Unrolled2	5	5	16	3.20	3.00
Unrolled4	3	3	10	3.33	2.88

Table C.5: The amount of clock cycles the operations take in the Ascon-128a implementations

Version	Cycles for encryption	Cycles for decryption	Cycles for message overhead	Time ratio overhead/message block	Energy ratio overhead/message block
Iterated	7	7	28	4.00	4.06
Serialized	55	55	220	4.00	4.20
Unrolled2	4	4	16	4.00	3.91
Unrolled3	3	3	12	4.00	3.74
Unrolled6	2	2	8	4.00	2.82

Table C.6: The amount of clock cycles the operations take in the Ascon-128 implementations

Bibliography

[1] V. T. Hoang and P. Rogaway, "On generalized feistel networks." Cryptology ePrint Archive, Report 2010/301, 2010. http://eprint.iacr.org/.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche1, "Duplexing the sponge: single-pass authenticated encryption and other applications." Second SHA-3 candidate conference, Santa Barbara, CA, 2010. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/presentations/DAEMEN_SpongeDuplexSantaBarbaraSlides.pdf.

[3] M. Riley, B. Elgin, D. Lawrence, and C. Matlack, "Missed alarms and 40 million stolen credit card numbers: How target blew it." bloomberg, 2014. http://www.bloomberg.com/news/articles/2014-03-13/target-missed-warnings-in-epic-hack-of-credit-card-data.

[4] A. Hern, "Could a simple mistake be how the nsa was able to crack so much encryption?." The Guardian, 2015. https://www.theguardian.com/technology/2015/oct/15/nsa-crack-encryption-software-reusing-passwords.

[5] D. Evans, "The internet of things, how the next evolution of the internet is changing everything." Cisco, 2011. https://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf/.

[6] K. Rose, S. Eldridge, and L. Chapin, "The internet of things: An overview, understanding the issues and challenges of a more connected world." The Internet Society (ISOC), 2015. http://www.internetsociety.org/sites/default/files/ISOC-IoT-Overview-20151221-en.pdf/.

[7] Wikipedia, "Authenticated encryption." https://en.wikipedia.org/wiki/Authenticated_encryption.

[8] D. Kravets, "Uk prime minister wants backdoors into messaging apps or he'll ban them." arstechnica, 2015. http://arstechnica.com/tech-policy/2015/01/uk-prime-minister-wants-backdoors-into-messaging-apps-or-hell-ban-them/.

[9] J. J. Grimmett, "Encryption export controls (crs report no. rl30273)," 2001. http://www.au.af.mil/au/awc/awcgate/crs/rl30273.pdf/.

[10] “Announcing development of a federal information processing standard for advanced encryption standard.” National Institute of Standards and Technology,Docket No. 960924272-6272-01,RIN 0693-ZA13, 1997. http://csrc.nist.gov/archive/aes/pre-round1/aes_9701.txt/.

[11] S. Babbage, C. D. Canniere, A. Canteaut, C. Cid, H. Gilbert, T. Johansson, M. Parker, B. Preneel, V. Rijmen, and M. Robshaw, “The estream portfolio (rev. 1).” ECRYPT Network of Excellence, 2008. http://www.ecrypt.eu.org/.

[12] “Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (sha-3) family.” National Institute of Standards and Technology,Docket No. 070911510-7512-01, 2007. https://www.gpo.gov/fdsys/pkg/FR-2007-11-02/html/E7-21581.htm/.

[13] X. Wang and H. Yu, “How to break md5 and other hash functions.” EUROCRYPT’05 Proceedings of the 24th annual international conference on Theory and Applications of Cryptographic Techniques, pp. 19-35, 2005.

[14] V. Rijmen and E. Oswald, “Update on sha-1.” Cryptology ePrint Archive, Report 2005/010, 2005. http://eprint.iacr.org/.

[15] “Caesar competion call draft.” CAESAR commitee, 2014. http://competitions.cr.yp.to/caesar-call-5.html/.

[16] “Caesar commitee members.” CAESAR commitee, 2015. http://competitions.cr.yp.to/caesar-committee.html/.

[17] M. Bellare, P. Rogaway, and D. Wagner, “Eax: A conventional authenticated-encryption mode.” Cryptology ePrint Archive, Report 2003/069, 2003. http://eprint.iacr.org/.

[18] F. Abed, C. Forler, and S. Lucks, “General overview of the authenticated schemes for the first round of the caesar competition.” Cryptology ePrint Archive, Report 2014/792, 2014. http://eprint.iacr.org/.

[19] Y. Zheng, T. Matsumoto, and H. Imai, “On the construction of block ciphers provably secure and not relying on any unproved hypotheses.” Advances in Cryptology - CRYPTO ’89, 9th Annual International Cryptology Conference, 1989. http://eprint.iacr.org/.

[20] Wikipedia, “Block cipher mode of operation.” https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

[21] J. Jean, I. Nikolić, and T. Peyrin, “Joltik v1,” 2014. http://competitions.cr.yp.to/round1/joltikv1.pdf/.

[22] J. Jean, I. Nikolić, and T. Peyrin, “Tweaks and keys for block ciphers: the tweakey framework.” Cryptology ePrint Archive, Report 2014/831, 2014. http://eprint.iacr.org/.

[23] H. Wu and T. Huang, "The authenticated cipher morus (v1)," 2015. http://competitions.cr.yp.to/round2/morusv11.pdf/.

[24] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.1," 2015. http://competitions.cr.yp.to/round2/asconv11.pdf/.

[25] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Permutation-based encryption, authentication and authenticated encryption." DIAC – Directions in Authenticated Ciphers, 2012. http://keccak.noekeon.org/KeccakDIAC2012.pdf/.

[26] S. Choi, R. Scrofano, and V. K. Prasanna, "Energy-efficient design of kernel applications for fpgas through domain-specific modeling." 5th annual Military and Aerospace Programmable Logic Devices, 2002.

[27] V. K. Prasanna, "Energy-efficient computations on fpgas." J. Supercomput., vol. 32, no. 2, pp. 139–162, 2005.

[28] A. Raghunathan, N. K. Jha, , and S. Dey, "High-level power analysis and optimization." Kluwer Academic Publishers, 1998.

[29] J. Ou and V. K. Prasanna, "A methodology for energy efficient fpga designs using malleable algorithms." 14th International Conference, FPL 2004, pp. 729-739, 2004.

[30] J. Ou and V. K. Prasanna, "Xilinx power estimator user guide." Xilinx, 2015. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug440-xilinx-power-estimator.pdf.

[31] "Fips 197, advanced encryption standard." National Institute of Standards and Technology, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[32] P. Hämäläinen, M. Hännikäinen, and T. D. Hämäläinen, "Efficient hardware implementation of security processing for ieee 802.15.4 wireless networks." 48th Midwest Symposium on Circuits and Systems, pp. 484-487, 2005.

[33] P. Chodowiec and K. Gaj, "Very compact fpga implementation of the aes algorithm." 5th International Workshop, Cologne, Germany, pp.319-333, 2003.

[34] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, "A compact rijndael hardware architecture with s-box optimization." Advances in Cryptology — ASIACRYPT, pp. 239-254, 2001.

[35] P. Hämäläinen, T. Alho, M. Hännikäinen, and T. D. Hännikäinen, "Design and implementation of low-area and low-power aes encryption hardware core." 9th EUROMICRO Conference on Digital System Design, 2006.

[36] X. Zhang and K. K. Parhi, "Implementation approaches for the advanced encryption standard algorithm." IEEE Circuits and Systems Magazine (Volume 2, Issue: 4), pp. 24-46, 2003.

[37] H. Groß, E. Wenger, C. Dobraunig, and C. Ehrenhöfer, "Suit up!—made-to-measure hardware implementations of ascon." Digital System Design (DSD), Euromicro Conference, pp. 645-652, 2015.

[38] "Xilinx power estimator user guide." Xilinx. http://www.xilinx.com/video/hardware/power-optimization-using-vivado.html.

[39] N. Grover and M. K. Soni, "Reduction of power consumption in fpgas - an overview." I.J. Information Engineering and Electronic Business, pp. 50-69, 2012.

[40] C. Cernazanu-Glavan, S. Fedeac, A. Amaricai-Boncalo, and M. Marcu, "Energy profiling of fpga designs." IEEE International Symposium on Robotic and Sensors Environments (ROSE), pp. 118-123, 2014.

[41] "Spartan-6 fpga configurable logic block user guide." Xilinx, 2010. http://www.xilinx.com/support/documentation/user_guides/ug384.pdf.

[42] D. Meidanis, K. Georgopoulos, and I. Papaefstathiou, "Fpga power consumption measurements and estimations under different implementation parameters." International Conference on Field-Programmable Technology (FPT), pp. 1-6, 2011.

[43] "Xilinx power tools tutorial for spartan-6 and virtex-6 fpgas." Xilinx, 2010. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ug733.pdf.

[44] R. Jevtic and C. Carreras, "Power measurement methodology for fpga devices." IEEE Transaction on Instrumentation and Measurement, Vol. 60, No. 1, 2011.

[45] "Power measurements on igloo2 evaluation kit." Microsemi, 2014. http://www.microsemi.com/document-portal/doc_view/132967-ac411-power-measurements-on-igloo2-evaluation-kit.

[46] "Sakura-g specifications." Morita Tech Co., LTD., 2013.

[47] "Modelsim pe student edition." Mentor Graphics, 2015. https://www.mentor.com/company/higher_ed/modelsim-student-edition/.

[48] "Ise design suite." Xilinx, 2013. http://www.xilinx.com/products/design-tools/ise-design-suite.html.

[49] "Atlys board reference manual." Xilinx, 2013. http://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPAtlys/documentation/Atlys_rm.pdf.

[50] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, M. U. Sharif, and K. Gaj, "Gmu hardware api for authenticated ciphers." Cryptology ePrint Archive, Report 2015/669, 2015. http://eprint.iacr.org/.

[51] "Brutus framework with reference c-codes of the ciphers in the caesar competition." GitHub, 2015. https://github.com/mjosaarinen/brutus.

[52] Wikipedia, "Field-programmable gate array." https://en.wikipedia.org/wiki/Field-programmable_gate_array.

[53] P. Clarke, "Xilinx launches spartan-6, virtex-6 fpgas." EETimes, 2009. http://www.eetimes.com/document.asp?doc_id=1311532.

[54] "Spartan-6 fpga family." www.xilinx.com, 2015. http://www.xilinx.com/products/silicon-devices/fpga/spartan-6.html.

[55] "Spartan-6 family overview." Xilinx, 2011. http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf.

[56] "Synthesis and simulation design guide." Xilinx, 2009. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/sim.pdf.

[57] "Targeting and retargeting guide for spartan-6 fpgas." Xilinx, 2010. http://www.xilinx.com/support/documentation/white_papers/wp309.pdf.

[58] "Sakura." SAKURA website, 2016. http://satoh.cs.uec.ac.jp/SAKURA/.

[59] Wikipedia, "Rs-232." https://en.wikipedia.org/wiki/RS-232.

[60] "Agilent 6000 series oscilloscope, user guide." Agilent Technologies, 2006. http://cp.literature.agilent.com/litweb/pdf/54684-97011.pdf.

[61] Wikipedia, "Advanced encryption standard." https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[62] Wikipedia, "Mds matrix." https://en.wikipedia.org/wiki/MDS_matrix.

[63] "Spartan-6 libraries guide for hdl designs." Xilinx, 2009. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/spartan6_hdl.pdf.