Tampere University

Maria Matache

STANDARDIZING LIGHTWEIGHT CRYPTOGRAPHY

Faculty of Information Technology and Communication Sciences
Bachelor of Science Thesis
April 2020

ABSTRACT

Maria Matache: Standardizing Lightweight Cryptography
Bachelor of Science Thesis
Tampere University
International Degree Programme in Science and Engineering, BSc (Tech)
Major: Information and Communications Technology
Examiner: Dr. Billy Brumley
April 2020

In this thesis work, we present an overview of the NIST call for lightweight algorithms and the current trend in cryptography. Moreover, we illustrate a C implementation of one of the submitted candidates, Gimli-24-Cipher. The algorithm has been computed in accordance with the provided pseudocode in the *Gimli 20190927 [13]* official NIST submission documentation. We outline the algorithm and demonstrate its flexibility and versatility in the scenario of the C programming language. We analyze its particularities and we study its hierarchical structure based on a mode of operation built upon an underlying primitive. We conclude that Gimli-24-cipher is a robust, universal and strong lightweight algorithm. We therefore motivate the usage of the algorithm in the context of the future TIE-31106 Cryptography Engineering course curriculum at Tampere University.

Keywords: thesis, lightweight cryptography, computer science

Table of Contents

List of Abbreviations

NIST	National Institute of Standards and Technology, U.S. Department of Commerce
AES	Advanced encryption standard
DES	Data encryption standard
AEAD	Authenticated Encryption with Associated Data
SP Networks	Substitution Permutation Networks
AD	Associated Data
uint32	unsigned integer with 32 bits
uint8	unsigned integer with 8 bits

Section 1: Introduction

1.1 Introducing Lightweight Cryptography

Cryptography is the art and science of secret communication. Being one of the oldest scientific disciplines, it was presumably introduced in the ancient Egypt more than 4000 years ago with the purpose of providing secrecy of private information. All along history, ciphers enabled protection of undisclosed data and modern cryptography developed out of the necessity of digital confidentiality.

Today we are witnessing a rapid development of new technologies prone to cyber-attacks. The past decade offered many new communication devices that have unique requirements and limitations when it comes to securing private data stored on them.

Modern computing environments such as IoT, sensor networks or new medical devices have something in common. Due to their special hardware design these technologies only use a small part of their processing power for security purposes. The current standard cryptographic encryptions such as AES or DES are unable to provide protection for these constrained platforms.

Innovative cryptographic methods are intensively researched in order to overcome these newly emerged limitations. A recent branch of cryptography, called “Lightweight Cryptography (LWC)”, arose out of the need of algorithms, designed for latest technological developments, that consume little power of the device they secure.

However, these algorithms differ in many ways and as opposed to their classical counterparts, no standard has yet been chosen. Further in this thesis I will investigate the latest trends of lightweight cryptography and I will discuss in detail an encryption technique that has been proposed as a suitable standard lightweight algorithm.

Moreover, as part of the “Cryptography Engineering” course at Tampere University, students have to implement two assignments in the C language. These are structured such that there is a low-level primitive and a mode of operation on top of it. The purpose of my study is to search for such a suitable project assignment for the next year’s cryptography course, that is in line with this requirement and is up to date to modern security trends, thus spiking students’ interest in this fascinating relatively young discipline.

1.2 NIST competition

In order to standardize lightweight cryptography, the U.S. National Institute of Standards and Technology (NIST) has launched on August 27, 2018 a competition for everyone to propose suitable lightweight algorithms. They listed a document, *Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process* [12], regarding the requirements and the criteria for assessing the submissions. The competition is currently in the second selection phase and 32 out of the initially 57 candidates were considered for further participation in the next round. In this paper I will present, analyze and implement one of them in detail. The candidate I chose for deeper examination is Gimli-24-cipher, an AEAD built on top of a permutation of 24 iterations.

Gimli is an authenticated encryption family based on a 384-bits permutation, which was designed to fit a wide range of computing environments, including classical CPUs as well as smaller constrained devices. Being accessible, high performant and universal, Gimli would be a suitable choice as a project assignment of the future cryptography course. Later in this paper I will go through the design characteristics of this encryption and I will present its implementation in the programming language C.

In this thesis I am investigating the properties of lightweight cryptography, specifically the peculiarity of Gimli. In particular, this thesis intends to demonstrate Gimli's functionality, as well as the potential it has as a standard lightweight authenticated encryption scheme. Second section outlines the essential theory behind cryptography basics, authenticated encryption, as well as the sponge and duplex constructs and other cryptographic concepts. Third section does an overview of the NIST competition and the fourth chapter goes into the details of Gimli's family of algorithms and its features, as well as the description of the C implementation and its results. The thesis is then concluded in Section 5.

Section 2: Theoretical Background

Let us begin by discussing the theory behind the cryptographic basics of the concepts presented in the next chapters.

For the past decades through the incredibly fast developing technology and telecommunications industry we were given instant access to large amounts of information. As a consequence of this, the process of copying and altering data became inevitably easy. Cryptography is the collection of techniques crucial in the digital world to ensure originality, security and integrity of information. That is why this discipline has eventually four main goals: *confidentiality, data integrity, authentication* and *non-repudiation.*

Confidentiality refers to secrecy of information against all who are not authorized to possess it. This is achieved through mathematical models which mask the content. Data integrity means guaranteeing that a piece of information was not altered and that it preserved its original form. Authentication enables a sender and a receiver to validate the identity of one another. Lastly, non-repudiation prevents one from erroneously negating involvement in an illicit digital action one partook in.

These basic principles combined are the ultimate goal of cryptography and all theory of this discipline was developed to address these criteria. All NIST candidates that participated in the call for lightweight algorithms are ciphers that try to meet these targets. I will further describe the theoretical background to get a better understanding of the later portrayed encryption algorithm. This thesis describes a lightweight primitive that provides security and prevents malicious digital activity in resource constrained devices, for which well-established classical cryptographic standards are not suitable.

2.1 Basic concepts

Encryption functions are a method of converting a plaintext P, usually consisting of binary strings into an apparently random ciphertext C, that loses connection to the initial plaintext message. This encryption function E is a bijection and thus has an inverse, the decryption process D, that reverts a ciphertext back into its original message format. The elements of the key set K are the parameters of the encryption transformations which determine the output ciphertext.

The purpose of having key elements and not just simple algorithmic encodings, is that the encryption and decryption elements should be public information. This way it is not needed to rewrite the whole functions every time a new ciphertext is desired. Instead, there is a so-called flowing key schedule that changes with every new bit of information and thus produces distinctive ciphertexts while keeping the same algorithmic approach. Generally, all security protocols can be divided into unkeyed encryptions, symmetric key encryptions and public key encryptions. This division is based on the nature of the used key to encrypt, since it can be non-existent, public or private.

The goal of encrypting messages is to give a sender the opportunity to transmit secret information to someone else that is authorized to possess that knowledge. The sender and receiver usually share a secret or a public key used for encryption and decryption purposes. Moreover, the sender and the receiver want to protect the integrity of the message, which means they want to secure the data and not let it be altered by external malicious activity. For this reason, there are authentication methods to prevent prohibited interventions and criminal digital attacks. AEAD signalizes erroneous message transmission and rejects the ciphertext before it is decrypted.

The NIST lightweight algorithmic candidates are built upon either symmetric key or unkeyed encryptions that use authenticated methods to safeguard the validity of the enciphered message, as well as the origin of the sender, determined by a given tag input parameter to the decryption function.

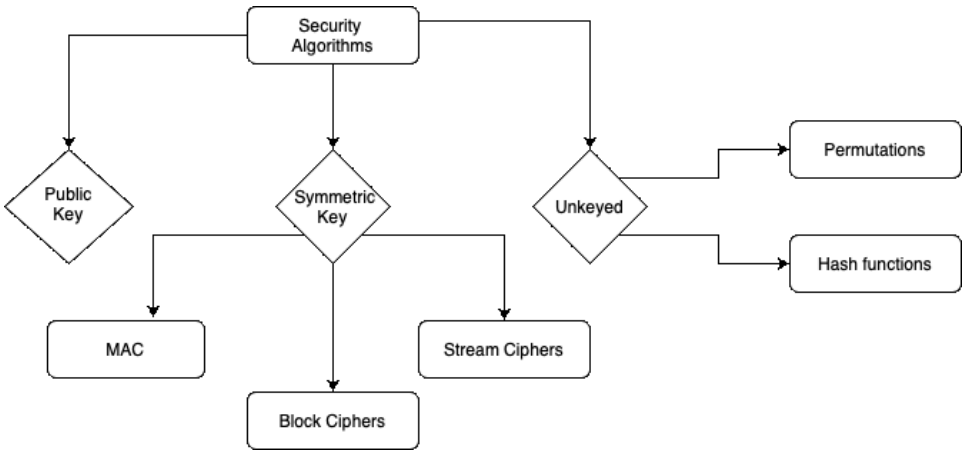

Figure 2.1: Security primitives diagram

All candidates of the NIST competition are based on an underlying security primitive and a mode of operation on top of it. Further concepts and notions used in the later discussed cipher are outlined in the following subsections.

2.2 Underlying cryptographic primitive

There exist several low-level function types that undertake the role of the foundation of a more complex cryptographic scheme. The purpose of this thesis is to find a suitable project assignment that is structured as follows: a low-level building block that is part of a larger mode of operation. From this point of view, all NIST candidates have this design rationale. One requirement of the NIST competition is that the mode of operation used should be an authenticated encryption scheme. Let us discuss the further notions that lay at the core of the lightweight cryptography AEAD schemes. There are several types of low-level primitives used by the candidates and this thesis focuses on permutations.

2.2.1 Symmetric-key cryptography

Symmetric-key encryption is a cryptographic primitive that has the property of having a secret key as tool to both encrypt and decrypt a message. Mathematically we can

describe $e, d \in K$ as two keys that are usually equal for both E as well as D functions, thus they are called “symmetric”. The encryption algorithm itself can be public and as long as the key is kept secret, the encrypted message should not be susceptible to any type of external attack. The sender needs to privately share the secret key with the receiver, in order for the latter to decrypt the sent message without the risk of a third party intercepting their communication.

2.2.2 Permutations and SP networks

Permutations are mathematical bijective functions commonly used in cryptography. They are usually mappings from a domain to a codomain set and in this case the codomain is the set itself. Permutations are so to say arrangements of order in linear manner within a given data set. Defined as:

$$p: S \rightarrow S, \tag{2.3} [3]$$

permutations are the core of most cryptographic constructs.

A substitution-permutation network, short SPN, is a sequence of mathematical procedures, in which a block of binary data is taken, and several substitutions and permutations are performed on it for many rounds and through several layers, with the scope of achieving diffusion and confusion. Moreover, SP-boxes (substitution permutation boxes) are part of many primitive ciphers, since they tend to achieve a high level of security by performing operations in many iterations.

The S-boxes and P-boxes work in a bitwise manner and are designed such that the operations they use, offer an efficient method to perform fast calculations at hardware level. Such common bitwise operations are also found in the SP-box of Gimli. These are XOR, AND and shifting. SP-boxes are hence particularly useful in lightweight cryptography, since they use little power consumption and are computationally very efficient.

2.3 Hash functions

Hash functions are cryptographic primitives of significant importance related to data origin authentication and identification. Hash functions map binary variables of arbitrary length to strings of fixed length called *hash values.* These functions are frequently called one-way functions, since it is almost impossible to find their inverse in a short amount of time.

One of the most common applications of hash functions is the data authentication of the received ciphertext. The way it works is very straightforward. The sender encrypts a message with a construct that uses a hash function. The obtained output signed tag is then sent along the ciphertext to the receiver. The receiver has access to the secret valid tag which is then compared to the obtained hash value from executing the decryption. If they coincide then the message is identified and validated, otherwise it means it was altered and it will be disregarded.

2.4 Authenticated Encryption

Authenticated Encryption is a cryptographic technique that guarantees confidentiality and authenticity of the encoded data. Moreover, authenticated encryption with associated data (AEAD), is a variation of AE that enables the receiver to check the integrity in both plaintexts as well as ciphertexts. AEAD uses associated data as parameter inside the function and thus blends the AD into the ciphertext. The achieved result is that the apparently correct but invalid ciphertexts can be detected by the AEAD decryption function and rejected. If an attacker tries to break the cipher by sending apparently accurate ciphertexts but does not know the tag that resulted from processing the data associated with it, the AEAD scheme will ignore it and reject it.

In order for AEAD to ensure confidentiality and integrity, the encryption scheme it uses has four input parameters: a secret key, a nonce, the associated data and the plaintext message. It can be mathematically described as:

$$E(K, N, AD, P,) \rightarrow C, T, \tag{2.4}$$

where the four inputs are the key K, the nonce N, the associated data AD and the plaintext D. As output the function returns the ciphertext C and the authentication tag T, that is used in the decryption process to verify integrity of the ciphertext. Since we are talking about bijective functions, the decryption is the inverse mapping of the encryption and can be mathematically modelled as:

$$D(C, T) \rightarrow P \; or \; error, \tag{2.5}$$

In the decryption process there is one output, the plaintext P or an error if the verification of the tag did not succeed.

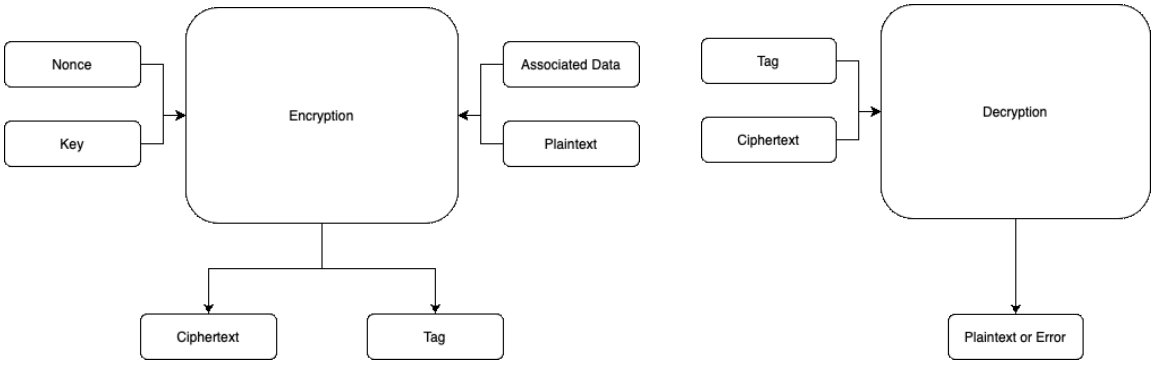

Figure 2.2: A diagram of the encryption and decryption process of AEAD

2.5 Modes of operation

A mode of operation is the higher-level algorithm that lies above a lower-level cryptographic primitive.

There are many different modes of operation that can be combined with all types of cryptographic primitives. In this thesis we focus on permutations as underlying operation. One interesting and important mode of operation for this primitive is the duplex construct of sponge functions.

2.5.1 Sponge functions – the duplex construction

A sponge function is a family of cryptographic techniques that takes an input of arbitrary n-bit length and returns an output variable of another arbitrary length. This type of construction usually has some kind of primitive underneath, in our case a permutation algorithm, that operates on a fixed number of bits of the initial input parameter. A sponge function works on a state of $b = r + c$ [15] bits, where r is the size of each block in which the plaintext is split and on which the permutation operates. The c bits are not impacted by the block operations and they only show the level of security that is achievable by the sponge construction. The idea is to divide the message to be encrypted (or the associated data) into r-bits blocks and then continue to the next two phases:

1. The “absorption” step takes each block of size r and adds them to the first r-bits of the state and then performs a lower level primitive on the whole state, for example a permutation.
2. The “squeezing” part happens after all blocks have been processed and it returns an arbitrary number of output blocks of r-bits size.

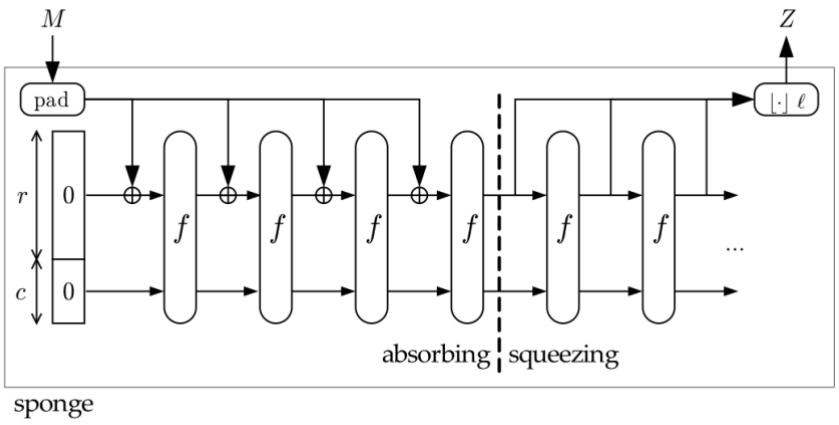

Figure 2.3: The sponge construction [6]

The duplex construction is a variation of the sponge functionality. It also can use a permutation as underlying primitive, but unlike the sponge construct it also does extra

steps between its calls. Using the previous mentioned definitions, the duplex construct works as follows:

1. The first step, same as the absorption, takes each block of size r (depicted in the figure below as σ) and adds them to the first r-bits of the state and then performs a lower level primitive on the whole state, for example a permutation.
2. Next it takes the first r-bits of the state and outputs them as a block Z that is dependent on all the previous returned outputs.

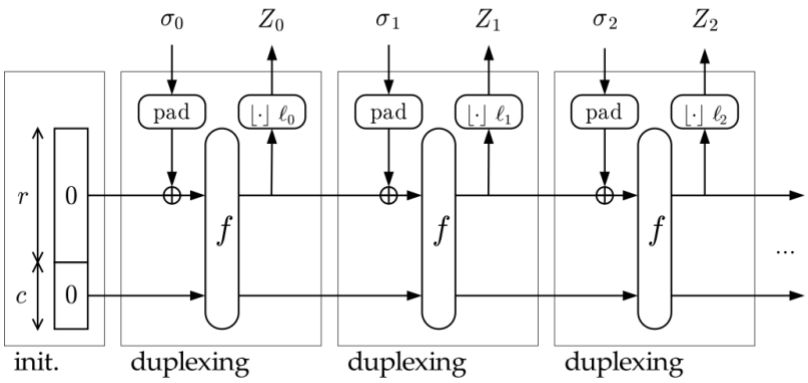

Figure 2.4: The duplex construction [6]

The duplexing construct is an intermediary mode of use that has an underlying primitive and a higher-level mode of operation, such as authenticated encryption with associated data. The way the parameter inputs, the r-bits blocks, and the outputs, the Z blocks, of the duplexing function correlate to the AEAD inputs (key, nonce, AD and message) and the AEAD returned values (ciphertext, tag) is determined by the AEAD scheme used on top of the duplexing construct.

Section 3: NIST overview

NIST published a document [12] with the requirements for the participation in the competition for a lightweight cryptographic standard. In order for a candidate to qualify for submission, the algorithm must be a family of authenticated encryption schemes. Hash functions may be submitted as well. The AEAD schemes must follow certain rules in order to be considered valid for the competition:

- The key must be fixed sixed and of minimum 128 bits long
- The nonce must be fixed sixed and of minimum 96 bits long
- The plaintext, AD and the ciphertext have variable sizes
- The tag must be of minimum 64 bits long

Regarding the security of the authenticated encryption schemes submitted, the algorithms must provide secrecy, reliability and integrity of the plaintext and the ciphertext.

Three main types of primitives have been chosen by the second round NIST candidates in their submissions and these are permutations, stream ciphers and block ciphers. Around half of all the candidates used permutation schemes as underlying primitive. ACE, Ascon, DryGASCON, Gimli-24-Cipher, ISAP, KNOT, ORANGE, Oribatida, PHOTON-Beetle, SPRAKLE, Xoodyak, SPIX, SpoC, Subterranean 2.0 and WAGE are all constructed on top of a permutation function.

The next preferred primitive was the block cipher. SUNDAE-GIFT and GIFT-COFB both are based on one of the most performant lightweight cryptographic protocols, the block cipher GIFT-128. The least popular choice was the tweakable block cipher as in ForkAE, SKINNY-AEAD and Lotus-AEAD.

All in all, permutations are dominant in the choice of the candidates. Furthermore, Gimli-24-Cipher promises to be a standard for all sorts of devices. This makes it an attractive option for situations when complex communication systems with elements of different power consumption need to be safeguarded from digital attackers. I chose to implement and discuss Gimli-24-cipher using the programming language C. The following section focuses on Gimli's algorithm family.

Section 4: Implementation of Gimli-24-cipher in C

4.1 Overview of Gimli-24-Cipher

Gimli-24-cipher is an AEAD scheme based on a 384-bits permutation that aims at becoming a standard suitable for many types of computing platforms, that include both resource-constrained devices as well as powerful computers. The purpose is to have one algorithm that works well in the context of several environments and interconnected computational systems.

Gimli NIST candidate is a family of authenticated encryptions that consists of a cipher with 256-bits key, 128-bits nonce and 128-bits tag. The Gimli family also includes a hash function that is the foundation behind the AEAD cipher. These two are constructed upon an underlying 384-bits permutation. Further we will analyze the design rationale of the algorithm and then we will go deeper into the particularities of the code in the C language.

4.1.1 Gimli permutation

The underlying primitive of the Gimli-24-cipher is a sequence of 24 rounds of permutations, substitutions and bitwise operations. The permutation algorithm can be divided into three main layers, the non-linear, the linear and the constant addition.

First of all, we can visualize the 384-bits state as an array of type uint32 with 12 words. The state can be seen as both a 3×4 2D uint32 array as well as a 1D uint32 array with 12 sequential elements. In the official description of the Gimli permutation there is a representation of the state as a double array of size 3×4, with 3 rows and 4 columns. The given representation follows by denoting the first row with words of type x_i with $i \rightarrow 0,1,2,3$, the second row with y_i with $i \rightarrow 0,1,2,3$, and the third row with z_i with $i \rightarrow 0,1,2,3$. In total we have 12 words of each 4 bytes and if we think about how the 384-bits state is divided, we can say that each of the three rows is a sequence of four 4-byte words.

The permutation's non-linear layer, the SP-box, is applied to x_i, y_i, z_i respectively, which in terms of a 2D array means that the permutations are performed column-wise. This creates a natural vectorization of the bitwise operations.

In the following figure there is a representation of how I stored the state in the form of an array of type unsigned integer of 32 bits, since the 12 words each have 4 bytes. The reason why I firstly stored the four $x's$ and then four $y's$ and then the last four $z's$ is because of how the authenticated cipher initializes the state. The first row of the state as a double array is represented by four words of type x and is initialized with the 16 bytes of the nonce N. The second row of the state is illustrated by four words of type y and it takes the value of the first 16 bytes of the key K. Lastly, the third row of the double array is represented by words of type z and is initialized with the next 16 remaining bytes of the key K.

x_0	x_1	x_2	x_3	y_0	y_1	y_2	y_3	z_0	z_1	z_2	z_3

Table 4.1: A visualization of the 1D uint32 state[12] storing elements as 4-bytes words sequentially.

As we can notice in the table below each row represents a sequence of four 4-bytes words. The 1D array of 12 words is thus the concatenation of the three rows of the below illustrated 2D array, one after the other.

$x_{0..3}$	$s_{0,0} = N_0 \dots N_3$	$s_{0,1} = N_4 \dots N_7$	$s_{0,2} = N_8 \dots N_{11}$	$s_{0,3} = N_{12} \dots N_{15}$
$y_{0..3}$	$s_{1,0} = K_0 \dots K_3$	$s_{1,,1} = K_4 \dots K_7$	$s_{1,2} = K_8 \dots K_{11}$	$s_{1,3} = K_{12} \dots K_{15}$
$z_{0..3}$	$s_{2,0} = K_{16} \dots K_{19}$	$s_{2,1} = K_{20} \dots K_{23}$	$s_{2,2} = K_{24} \dots K_{27}$	$s_{2,3} = K_{28} \dots K_{31}$

Table 4.2: A visualization of the 2D uint32 array s[12] that stores twelve 4-bytes words into x_i, y_i, z_i with $i \rightarrow 0,1,2,3$. The state is initialized with the first 16 bytes of the nonce N and with the following 32 bytes of the key K.

The Gimli permutation takes as parameter a strict 48-byte input and executes the non-linear layer, the SP-box, sequentially column by column. Each column represents a triple of 4-byte words x_i, y_i, z_i *with* $i \rightarrow 0,1,2,3$. The SP-box transforms them and then Gimli moves on to the next phase, the linear level, where the first row of words is permuted depending on the round number.

The Gimli permutation processes at linear level four words at a time, by interchanging their position. The first four 4-bytes words of the state are transformed by two functions, the Small Swap and the Big Swap. The linear layer permutes these words as shown below:

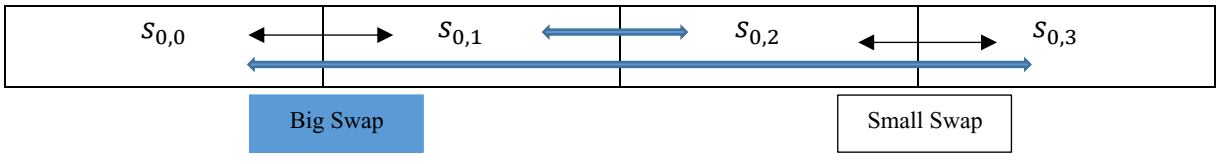

Table 4.3: The linear layer of Small Swap and Big Swap

With each of the 24 Gimli rounds, the algorithm interchanges the order of the first four words depending on the round number respectively. Small Swap happens every four rounds starting from the first one and big swap occurs every four rounds starting from the third iteration.

4.1.2 Gimli duplex construct and AEAD mode of operation

Gimli-24-cipher is a lightweight algorithm that is hierarchically structured. The lowest level is an underlying permutation, followed by a duplex construct and lastly wrapped under an AEAD scheme.

The duplexing in Gimli's case happens as follows: The 384-bits/48-bytes state is initialized according to *Table 4.2*. Next, the AEAD divides the associated data into blocks of 128 bits or 16 bytes. Following the duplexing construction concept, the next phase is the absorption of the 128-bits block into the first 128-bits/16-bytes of the state. Afterwards, the Gimli permutation is called on the whole state. The output block is then fed to the next duplexing call until all 128-bits blocks have been absorbed. The AEAD continues by processing the last AD block of n-bits, $n \leq 16$ by XORing some values in the new state and calling the permutation one more time.

The subsequent step is the handling of the message plaintext. This is done in a similar fashion to the procedure done to the AD. Lastly, the squeezing phase of the duplex construct occurs, and the last duplexing function returns the last 16-byte output block that is copied as the first 16 bytes of the ciphertext. This stands as the authenticating tag used by the decryption function to validate integrity of the ciphertext.

The AEAD encryption scheme executes its procedures on 1-byte words, byte by byte, arranged in blocks of 16 bytes. The absorption function is XORing the first 16 bytes of the state with 16-bytes blocks of the AD and of the message. The Gimli permutation, however, computes on 4-byte words. Because of this, I defined Gimli in C to have as input an array of uint32 type with 12 words, but the AEAD construct to work on an array of uint8 type with 48 words. In order to call the permutation on the uint8 type array we need some functionality to do a type conversion between the two array types.

Based on the aforementioned things, the AEAD happens as follows: We take the first 16-bytes stored in the uint8 state, we XOR each element byte-by-byte with the block of 16-bytes of the message/AD and then we use a function that transfers the uint8 state into an uint32 state. In other words, to do the Gimli permutation on the uint8 state we are concatenating its 1-byte elements into 4-byte words. Afterwards we revert the output of the permutation back to the uint8 type by splitting all 4-bytes words into 1-byte elements and we feed the new uint8 array to the next duplexing call.

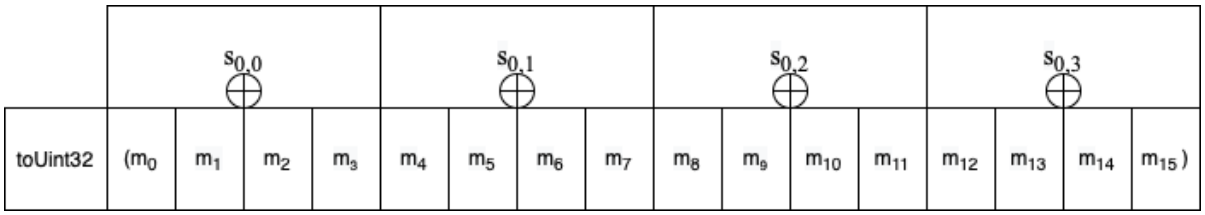

Figure 4.1: This figure illustrates the concept behind the absorption function of the first four 4-byte words of the state array uint32 s[12].

toUint8	$s_{0,0}$				$s_{0,1}$				$s_{0,2}$				$s_{0,3}$			
	byte1	byte2	byte3	byte4	byte5	byte6	byte7	byte8	byte9	byte10	byte11	byte12	byte13	byte14	byte15	byte16

Figure 4.2: This figure illustrates the concept behind the conversion of the uint32 s[12] into an array of type uint8 state[48].

Algorithm 5 The GIMLI AEAD encryption process.

Input: $M \in \{0,1\}^*, A \in \{0,1\}^*, N \in \mathbb{F}_{256}^{16}, K \in \mathbb{F}_{256}^{32}$
Output: GIMLI-CIPHER-ENCRYPT$(M, A, N, K) = C \in \{0,1\}^*, T \in \mathbb{F}_{256}^{32}$

Initialization
$\mathbf{s} \leftarrow \mathbf{0}$
for i from 0 to 3 **do**
  $s_{0,i} \leftarrow \textbf{touint32}(N_{4i}|| \ldots ||N_{4i+3})$
  $s_{1,i} \leftarrow \textbf{touint32}(K_{4i}|| \ldots ||K_{4i+3})$
  $s_{2,i} \leftarrow \textbf{touint32}(K_{16+4i}|| \ldots ||K_{16+4i+3})$
end for
$\mathbf{s} \leftarrow$ GIMLI$(\mathbf{s})$
Processing AD
$a_1, \ldots, a_s \leftarrow \textbf{pad}(A)$
for i from 1 to s **do**
  if $i == s$ **then**
    $s_{2,3} \leftarrow s_{2,3} \oplus$ 0x01000000
  end if
  $\mathbf{s} \leftarrow \textbf{absorb}(\mathbf{s}, a_i)$
end for
Processing Plaintext
$m_1, \ldots, m_t \leftarrow \textbf{pad}(M)$
for i from 1 to t **do**
  $k_i \leftarrow \textbf{squeeze}(\mathbf{s})$
  $c_i \leftarrow k_i \oplus m_i$
  if $i == s$ **then**
    $s_{2,3} \leftarrow s_{2,3} \oplus$ 0x01000000
  end if
  $\mathbf{s} \leftarrow \textbf{absorb}(\mathbf{s}, m_i)$
end for
$C \leftarrow c_1|| \ldots ||c_t$
$T \leftarrow \textbf{squeeze}(\mathbf{s})$
return C, T

Figure 4.3: Pseudocode of the Gimli AEAD [10].

4.1.3 Gimli C implementation step by step

4.1.3.1 Gimli-24 permutation

Let us begin by looking under the hood of the Gimli permutation algorithm in C. First of all, as mentioned before, Gimli takes as input a 32-bits unsigned integer type array, with 12 words stored sequentially as presented in *Table 4.1*. The NIST submission of Gimli uses a permutation primitive with 24 rounds. According to the presented pseudocode, we start iterating at $i = 24$, since the number of the round is relevant for the operations performed in that round.

4.1.3.2 The non-linear layer

The first thing done by Gimli is extracting the x_i, y_i, z_i for each column from 0 to 3. Each x_i, y_i, z_i pack of the 1D array goes through an SP-box. x_i, y_i, z_i with column numbers $i \rightarrow 0,1,2,3$ are stored in my code in the uint32 word_array32 [12] at positions $i, i+4, i+8$ such as highlighted in the table below.

x_0	x_1	x_2	x_3	y_0	y_1	y_2	y_3	z_0	z_1	z_2	z_3

Table 4.4: x_i, y_i, z_i extracted by the SP-box

x and y are given the values of the first two words of the respective column index of the uint32 word_array32 [12], and are afterwards rotated by 24 and 9 bits respectively. The last word of the processed column, z, is initialized as simply the last word of the state's column. I wrote an uint32 leftRotate() function that takes as parameters a 4-byte word and an integer number that represents the number of bits by which the word should to be rotated. This function returns the new shifted value. It follows the common approach of cyclic rotations, (word << n)|(word >> (32 - n)), where word is a 4-byte word from the state array and n is the number of bits by which the word is rotated.

Next, after the initial setting, the SP-box moves to the following step that does bitwise operations of XOR, shifting, AND and swapping between x, y, z and then updates the values of the words in the state array accordingly to the given pseudocode presented in *Gimli 20190927 [13]*.

4.1.3.3 The linear layer

The linear part of the Gimli permutation consists of two operations, the small and the big swap. The small swap happens every four rounds starting from the first round. If $round \bmod 4$ is 0, the small swap occurs. The big swap occurs every four rounds but starting from round number 3, thus $round \bmod 4$ must be 2. The first operation swaps the first and the second word and then the third and the fourth. The latter operation swaps the first and the third and then the second and the fourth.

4.1.3.4 The add constant layer

If the round is a multiple of 4 then we XOR the constant 0x9e377900 with round number and the first word of the state. In the following listing I present the C code version of the above explained algorithm.

```
void gimli(uint32_t word_array[12])
{
    for(int i = 24; i >= 1; i--)
    {
        for(int j = 0; j <= 3; ++j)
        {
            uint32_t x = leftRotate(word_array[j], 24);
            uint32_t y = leftRotate(word_array[j+4], 9);
            uint32_t z = word_array[j+8];

            word_array[j+8] = x ^ (z << 1) ^ ((y&z) << 2);
            word_array[j+4] = y ^ x ^ ((x|z) << 1);
            word_array[j] = z ^ y ^ ((x&y) << 3);

        }
//Small Swap every 4 rounds starting from the first round
        if (i%4 == 0)
        {
            uint32_t aux = word_array[0];
            word_array[0] = word_array[1];
            word_array[1] = aux;

            aux = word_array[2];
            word_array[2] = word_array[3];
            word_array[3] = aux;
        }
//Big Swap every 4 rounds starting from the third round
        if(i%4 == 2){
            uint32_t aux = word_array[0];
            word_array[0] = word_array[2];
            word_array[2] = aux;

            aux = word_array[1];
            word_array[1] = word_array[3];
            word_array[3] = aux;
        }
        //XOR the round constant
        if(i%4 == 0){
            word_array[0] = word_array[0] ^ (((uint32_t)0x9e377900)
     | ((uint32_t)i) );
        }
    }
}
```

Listing 1: A C implementation of Gimli permutation with 24 rounds

4.1.3.5 The type conversion in C

As defined before, the Gimli-24-cipher is an authenticated encryption that is built upon a duplex mode of operation. The AEAD scheme initializes an array of 48 bytes, uint8 word_array, and each of its 4 bytes sequentially represents a 32-bits word of the uint32 word_array32. Thus, before being able to apply the Gimli-24 permutation, we need to have the initial state converted into an uint32 type array. This in fact means that we want to concatenate each byte of the uint8 word_array into 4-byte words. In the specification of Gimli-24-cipher it is mentioned that we encode the words in little endian form. To encode the array in little endian format we use left shifting and OR for concatenation as presented in the code below:

```
1  void toUint32(uint8_t *state, uint32_t *new_state){
2      for (int i=0; i<=11; i++) {
3          new_state[i] =state[4*i+3]<<24)|(state[4*i+2]<<16)|(state
       [4*i+1]<<8)|(state[4*i]);
4    }
5  }
```

Listing 2: A function that converts an array of type uint8 into an array of type uint32 in little endian form

The same thing needs to be done in reverse mode, e.g. we need to split the 4-byte elements of the uint32 array into four 1-byte words. Using the same logic as before, the code below splits the words:

```
1  void toUint8(uint32_t *state, uint8_t *new_state){
2
3      for (int i=0; i<= 11; ++i) {
4          new_state[4*i] = state[i];
5          new_state[4*i+1] = state[i]>>8;
6          new_state[4*i+2] = state[i]>>16;
7          new_state[4*i+3] = state[i]>>24;
8
9      }
10 }
```

Listing 3: A function that converts an array of type uint32 into an array of type uint8 in little endian form

4.1.3.6 The AEAD and duplex construct in C

Finally, let us examine the authenticated encryption with associated data of the Gimli cipher. The AEAD scheme is the mode of operation that has an underlying permutation primitive.

When initializing the uint8 state array in the beginning of the authenticated encryption we do it so accordingly to *Table 4.2*. We can make use of the C language function memcpy(), that takes a destination pointer, a source pointer and the number of bytes

to be copied inside the destination variable. In this case we want as mentioned before, the first 16 bytes of the nonce and the next 32 of the key.

```
memcpy(word_array, nonce, 16);
memcpy(word_array + 16, key, 32);
```

Listing 4: Initialization of the uint8 word_array according to Table 4.2

The next step in this lightweight algorithm, is the processing of associated data. According to the notions presented in the theoretical background section, the AD is a necessary part in authenticating the integrity of both the plaintext as well as the ciphertext. The absorption of the duplexing construct happens as follows: The AD is XORed into the first four words of the state in blocks of 16 bytes and then Gimli permutation is called, with the help of the type conversion functions. Since the associate data is handled in divisions of each 16 bytes, the last remaining n-bit block will be equal or smaller to 16, $n \leq 16$. This last block is XORed into the first n bytes of the word array. Last two operations represent the XORing of 1 at index of the length of the AD and at position 47. Following that, Gimli is called one more time.

```
while (adlen >= 16){
        for(i = 0; i <= 15; i++){
            word_array[i] ^= ad[i];
            }
      //perform Gimli permutation
        toUint32(word_array, word_array32);
        gimli(word_array32);
        toUint8(word_array32,word_array);
        ad += 16;
        adlen -= 16;

    }

    //process last AD block that has less than 16 bytes
    for(i = 0; i < adlen; i++){
        word_array[i] ^= ad[i];
    }

//processing of the last bytes
word_array[adlen] ^= 1;
word_array[47] ^= 1;
toUint32(word_array, word_array32);
gimli(word_array32);
toUint8(word_array32,word_array);
```

Listing 5: The processing of associated data

After the processing of the associated data, the Gimli AEAD scheme moves on to the processing of the message plaintext and the construction of the ciphertext. The message encryption procedure is similar to the previously explained AD code. The technique includes now the creation of the ciphertext by XORing the state words with the message 16-bytes block and calling the Gimli permutation after each block was XORed. The last block is managed similarly to the last AD block.

```
while (messagelen >= 16) {
        for(i = 0; i<= 15; i++){
            cipher[i] = word_array[i] ^ message[i];
            word_array[i] ^= message[i];
            }

        message += 16;
        cipher += 16;
        messagelen -= 16;

    }

    //process last message block that has less than 16 bytes

    for(i=0;i < messagelen;i++){
        c[i] = word_array[i] ^ m[i];
        word_array[i] ^= m[i];
    }
//processing of the last bytes
word_array[messagelen] ^= 1;
word_array[47] ^= 1;
toUint32(word_array, word_array32);
gimli(word_array32);
toUint8(word_array32,word_array);
```

Listing 6: The processing of the plaintext

Lastly, the ciphertext takes the first 16-bytes of the resulted word array. This represents the squeezing phase of the duplex construct. It is also the returned tag that will be checked by the decryption algorithm to prove the validity of a ciphertext and identify the sender. In the formation of the ciphertext, implicitly the tag, all four parameters, the AD, the message, the key, the nonce play a crucial role. This makes the ciphertext secured and both confidentiality as well as data integrity are preserved.

```
for(i=0; i<= 15;i++){
        c[i] = word_array[i];
    }
```

Listing 7: Final version of the ciphertext

4.2 Test vectors and results

In this section we can see some test vectors of the Gimli-24-cipher algorithm in different cases and their produced ciphertexts.

```
Key = 000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
Nonce = 000102030405060708090A0B0C0D0E0F
PT =
AD =
CT = 14DA9BB7120BF58B985A8E00FDEBA15B
```

Figure 4.2.1: Encrypted ciphertext CT after applying Gimli-24-cipher with given key, nonce but no plaintext nor AD

```
Key = 000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
Nonce = 000102030405060708090A0B0C0D0E0F
PT =
AD = 00010203040506
CT = DE61093D8570D64FCDB8C89B76B29E24
```

Figure 4.2.2: Encrypted ciphertext CT after applying Gimli-24-cipher with given key, nonce, AD but no plaintext

```
Key = 000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
Nonce = 000102030405060708090A0B0C0D0E0F
PT = 000102
AD =
CT = 7F8A2C65CABBEE8A9A9A959CEC122483E4E496
```

Figure 4.2.3: Encrypted ciphertext CT after applying Gimli-24-cipher with given key, nonce, plaintext but no AD

```
Key = 000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
Nonce = 000102030405060708090A0B0C0D0E0F
PT = 000102
AD = 000102030405060708090A0B0C0D0E0F10111213141516
CT = 55865A89409F6C8C101C46E2C1204EFD681A69
```

Figure 4.2.3: Encrypted ciphertext CT after applying Gimli-24-cipher with given key, nonce, plaintext and AD

4.3 Gimli's particularities

4.3.1 Vectorization and parallelization

We can say that an algorithm is parallelizable if it is possible to execute multiple computations simultaneously. Gimli's duplexed AEAD requires the output of previous iterations, which makes the mode of operation unparallelizable. However, the Gimli primitive permutation performs the operations of the non-linear layer, the SP-box, in parallel, since the nonlinear functions never require the result of the previous computations. This makes Gimli's permutation performant and creates fast diffusion.

Vectorization makes the hardware implementation more efficient and is extremely useful in resource-constrained environments. Since Gimli's SP-box works in a column-wise direction, the permutation itself vectorizes the operations.

4.3.2 Duplex construct

One characteristic of Gimli-24-cipher is that it is built upon an unkeyed primitive, a permutation, and on top of that there is a mode of operation, hierarchically structured. This AEAD scheme takes advantage of the duplex construct by using operations that work as the sponge functionalities of absorption and squeezing. This approach is very beneficial in securing the underlying permutation. The purpose of duplexing is in fact a higher level of security of the encrypted data. Since the permutation is an unkeyed encryption in order for it to be safe to be used as underlying function for the authenticated encryption, the duplex construct is used. This makes Gimli-24-cipher a performant lightweight permutation-based algorithm that provides high security of the encrypted data.

4.3.3 Universality of Gimli-24-cipher

What makes Gimli special, is its compactness and high security in lightweight platforms. However, Gimli was designed to be a good fit for many types of hardware. The idea was to create one universal, standard encryption that can be used by lightweight, powerful and hybrid computational systems. This bridges the gap between classical standards and lightweight algorithms, since only this single cipher can be used in complex systems and their communication channels.

5. Conclusions

The field of lightweight cryptography is rich with theories worthy of scientific investigation. The digital world we live in right now is more than ever prone to cyberattacks. Technology is developing at a very high pace and information security is the only source of protection of data stored in modern tech environments. Classical standard ciphers are not appropriate for many devices that have limited power consumption and other resource limitations. NIST launched a call for algorithms competition with the hope to establish one standard algorithm for lightweight platforms.

In this thesis, we have scratched the surface of this new emerging discipline. Utilizing the basic principles of cryptography we have analyzed and implemented the NIST candidate Gimli-24-cipher, a versatile computational lightweight framework suitable for encrypting data in a wide range of computational systems including resource-constrained devices.

In order to demonstrate the functionality incorporated in Gimli, we have analyzed its algorithmic particularities and we have implemented the C version of the underlying primitive, Gimli permutation, and of the mode of operation, an AEAD scheme built upon a duplex construct.

The main purpose of the thesis was to show the potential of the Gimli-24-cipher as an appropriate project assignment for the Cryptography Engineering course at Tampere University. Gimli is a lightweight modern cryptographic algorithm that is designed to follow the structure of a high-level mode of operation built upon an underlying low-level primitive. The compactness, the symmetry and its universality make Gimli a strong candidate among lightweight encryptions. This makes it suitable to be utilized as part of the course future arrangements.

REFERENCES

[1] H. Delfs and H. Knebl, “Symmetric-key cryptography,” in *Information Security and Cryptography*, 2015.

2] J. Ikbal, “An introduction to cryptography,” in *Information Security Management Handbook, Sixth Edition*, 2007.

3] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. 1996.

[4] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. 2014.

[5] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “On the security of the keyed sponge construction,” *SKEW*, 2011.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “Duplexing the sponge: Single-pass authenticated encryption and other applications,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012, doi: 10.1007/978-3-642-28496-0_19.

[7] M. Bellare and C. Namprempre, “Authenticated encryption: Relations among notions and analysis of the generic composition paradigm,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2000, doi: 10.1007/3-540-44448-3_41.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “Sponge Functions,” *ECRYPT hash Work.*, 2007.

[9] J. Balasch *et al.*, “Compact implementation and performance evaluation of hash functions in attiny devices,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013, doi: 10.1007/978-3-642-37288-9_11.

[10] D. J. Bernstein *et al.*, “GIMLI: A cross-platform permutation,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017, doi: 10.1007/978-3-319-66787-4_15.

[11] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varici, and I. Verbauwhede, “SPONGENT: The design space of lightweight cryptographic hashing,” *IEEE Trans. Comput.*, 2013, doi: 10.1109/TC.2012.196.

[12] National Institute of Standards and Technology. "Submission requirements and eval- uation criteria for the lightweight cryptography standardization process", August 2018.https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/ documents/final-lwc-submission-requirements-august2018.pdf.

[13] Daniel J. Bernstein, Stefan Kolbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, Fran´cois-Xavier Standaert, Yosuke Todo, and Benoˆıt Viguier. "Gimli 20190927", Submission to NIST Lightweight Cryptography competition, 2019

[14] Eline Bovy. Bachelor's Thesis. Comparison of the second-round candidates of the NIST lightweight cryptography competition, 2020

[15] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, "Cryptographic sponge functions", 2011