Multi-Purpose Designs in Lightweight Cryptography

by

Morgan He

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The purpose of this thesis is to explore a number of techniques used in lightweight cryptography design and their applications in the hardware designs of two lightweight permutations called sLiSCP and sLiSCP-light. Most of current methods in lightweight cryptography are optimized around one functionality and is only useful for applications that require their specific design. We aimed to provide a design that can provide multiple functionalities. In this thesis, we focus and show the hash function and authenticated encryption of our design. We implemented two lightweight permutations designs of sLiSCP and sLiSCP-light in VHDL. During the verification of sLiSCP cipher, we discovered additional area that could be saved if we tweaked the design slightly. This would lead us to consider the design of sLiSCP-light which helps dramatically reduce area. Results of our designs of sLiSCP and sLiSCP-light satisfied the lightweight requirements, including hardware area, power, and throughput, for applications such as passive RFID tags. Lastly, we did tests on the randomness of Simeck and Simon Feistel structures. We wanted to observe the pseudorandom nature of structures similar to Simeck and Simon so we performed exhaustive tests on small instances of these structures to trace any trends in their behavior. We confirmed that Simon and Simeck were very consistent and provided acceptable pseudorandom results. For larger sizes, we expect similar results from Simon and Simeck.

Acknowledgements

I would like to thank Professor Guang Gong, who has supervised me for the past two years during my Master's program at the University of Waterloo. Her support, encouragement, and guidance throughout my academic career have been invaluable in my past accomplishments and future goals. She had an endless amount of professional knowledge that she would share with me, which was very inspiring. Her unwavering support in the difficult problems during my projects pushed me through to finish during those times when I thought I would give up. For all, that, I am truly grateful.

Also, I would like to thank Gangqiang Yang, who worked closely with me for the hardware designs and implementations. My largest contributions to our lightweight project was hardware based, and his guidance and knowledge was essential in the work that I did.

I would like to thank my team, who worked hard on the lightweight sLiSCP and sLiSCP-light permutations, including Riham AlTawy, Raghvendra Rohit, Kalikinkar Mandal, as well as Gangqiang Yang and Professor Guang Gong. Our team's work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the National Institute of Standards and Technology (NIST).

I would also like to thank my thesis readers, Professor Mark Aagaard and Professor Cathy Gebotys, for taking the time to read through my thesis. I take their comments and critique very seriously and think they are vital to the success of my work.

Lastly, I would like to thank all of the members of our Communication Security Lab (COMSEC) at the University of Waterloo. Our weekly seminars have broadened my views to different topics and opened my eyes to new ideas.

Thank you to all!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the age of digital communications, many of our technologies rely on security. With the increase in smaller and smaller devices, the security demands lightweight ciphers. In other words, we need cryptographic functions that are able to be performed on resource constrained devices.

Cryptography is the study of the methods in securing information. To keep the information secret, encryption is the process used to convert comprehensible information (i.e. plaintext) into incomprehensible information (i.e. ciphertext). Decryption is the process used to recover the plaintext from the ciphertext. Encryption and decryption constitute a pair of algorithms which is referred as a cipher. For example, Kasumi [1], also known as A5/3, is a block cipher in which encryption and decryption operations are identical with a reversal of the key schedule. It is used in the confidentiality and integrity algorithms for the third generation mobile phone system. It operates on blocks of 64 bits and outputs in block of 64 bits. As another example, WG [31] is a synchronous stream cipher that has been designed to produce a keystream with guaranteed randomness properties such as balance, long period, large and exact linear complexity, 3 level additive autocorrelation and ideal 2 level multiplicative autocorrelation. Also, it is resistant to data tradeoff attacks, algebraic attacks and correlation attacks.

In this thesis, we examine some existing lightweight designs for cryptoprimitives. Using this for background and comparison purposes, we present the sLiSCP/ sLiSCP-light families of lightweight permutations and their hardware implementations, as well as investigate the randomness of Feistel structures, involving Simon and Simeck derived functions. We were able to verify that our design satisfied NIST's lightweight property requirements with minimal hardware footprint as well as performed the necessary operations to ensure

the desired security levels. The most important requirement that we aim for, is the 2000 gate equivalent maximum, that can be devoted for low-cost RFID tags. While we try to minimize the power consumption and maximize the throughput in all of our designs, our main focus is on area. Each of our two designs were optimized on the commonly used hardware technologies such as CMOS 135nm and CMOS 65nm. We also explore serialized implementations and considered the throughput costs of such a method. Our results were measured using in gate equivalents (GE), which is the approximate area needed for a two-input drive-strength-one NAND gate. This is so that we can measure the complexity of the circuit, and deduce the silicon area from the manufacturing specifications.

Contributions include the hardware design and optimization of two lightweight permutation designs, sLiSCP and sLiSCP-light. Also included, are randomness tests on Simeck-like ciphers. The contributions of the thesis are summarized as follows.

1. **sLiSCP.** We provide two efficient parallel hardware implementations for the sLiSCP unified duplex sponge mode when using sLiSCP-192 (resp. sLiSCP-256) in CMOS 65 nm ASIC with area of 2289 (resp. 3039) GE and a throughput of 29.62 (resp. 44.44) kbps, and their areas in CMOS 130 nm are 2498 (resp. 3319) GE. The power consumption is 4.62 (resp. 5.88) μW. We have estimated the areas for half-serial and bit-serial implementation. Lastly, we made comparisons in hardware efficiency with existing lightweight cryptographic designs.

2. **sLiSCP-light.** For sLiSCP-light-192, we have parallel implementation hardware areas of 1820 (resp. 1892) GE in CMOS 65 nm (resp. 130 nm) ASIC. The areas for sLiSCP-light-256 are 2397 and 2500 GE in CMOS 65 nm and 130 nm ASIC, respectively. Overall, the unified duplex sponge mode of sLiSCP-light-192 which provides (authenticated) encryption and hashing functionalities, satisfies the area (1958 GE), power (3.97 μW), and throughput (44.4 kbps) requirements of passive RFID tags. For sLiSCP-light-192, we estimated that serialized 1-bit implementation would use 1572 (resp. 1669) GE and for sliscpl-256, it would use 2057 (resp. 2193) GE.

3. **Randomness test of Simeck-like structures.** We test the properties of a shift register, using a Simeck-like design. We look at the small instances of Simeck-like structures and search for the number of states that are in cycles that are larger than the square root of 2^n where $2n$ is the number of the bits in the internal states. We exhaustively search all shift parameters in our Simeck-like structure for smaller sizes. For our larger state sizes, we focus on the shift values of Simeck and Simon.

The remaining chapters of this thesis are organized as follows.

Chapter 2 discusses some of the existing lightweight tools that have been proposed by the cryptographic community. We look at the methods and purposes of the proposed tools as well as their relevance in lightweight applications.

Chapter 3 discusses the preliminary mathematics and background that are used in our new work. We explain LFSR states and hardware tools that are used in the following chapters.

Chapter 4 presents our new family of lightweight permutations called sLiSCP. We explain the overall design, the goals, and the applications and focuses on the hardware details and comparisons with other lightweight designs.

Chapter 5 shows a modified version of our work called sLiSCP-light. We explain the purpose of this modification and how it affects the hardware footprint, some of the changes in security level, and its hardware implementations for two instances (i.e. 192-bit internal states and 256-bit internal states).

Chapter 6 analyzes the randomness properties in the modified Feistel structure. We look at some of the examples from Simeck round functions and large versions created by extending the design.

Chapter 7 concludes the thesis and presents some future topics to explore.

Chapter 2

Literature Survey and Preliminaries

Lightweight cryptographic security has become an increasing topic of concerns for securing the Internet-of-Things (IOT). In this chapter, we review some of the background of lightweight cryptographic designs. We explore a survey of the existing designs used in lightweight cryptographic primitives.

The chapter contents are organized as follows. In Section 2.1, we explain linear and non-linear shift registers, which is an important tool that is used extensively in this thesis. Section 2.2 explores lightweight ciphers that are based around LFSRs in their structure. Section 2.3 looks at how the Feistel structure is used in certain lightweight primitives. We explain its uses in DES as well as Simon and Speck. Section 2.4 explains the substitution network is used in certain ciphers such as AES. In Section 2.5, we present the sponge structure, and its uses in ciphers such as Keccak. In Section 2.6, we look over some of the choices that were made when the existing lightweight permutations were designed. Lastly, in Section 2.7, we go over Simeck, which is function of our choice in our lightweight permutation.

2.1 Linear and Non-linear Feedback Shift Registers

We give a brief introduction about shift registers as an important tool for cryptographic functions. They are comprised of a series of flip-flops, each output of which connects to the input of the next, and all share the same clock. We use these shift registers to perform functions on an initial state. Figure 2.1 shows these shift registers. In this thesis, we use linear feedback shift registers (LFSR) and non-linear feedback shift registers (NLFSR).

LFSRs will have an input bits based on a linear function of its previous state, while NLFSRs have input bits based on a non-linear function of its previous state. These are used for their pseudorandom properties and have simple structures in implementation.

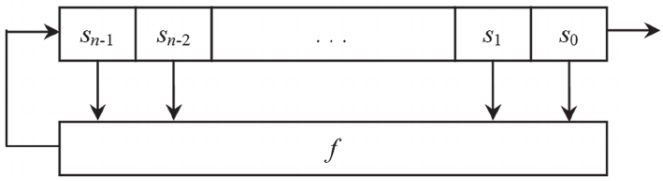

Figure 2.1: LFSR and NLFSR structure

2.2 Lightweight Primitives with LFSR Based Structures

Ciphers, including Grain [35], Trivium [24] and the lightweight parameters of WG [31] are included in the category of primitives that are structurally based on LFSRs. These designs use LFSRs to generate m-sequences and combine them using finite state machines over non-linear feedback shift registers.

Trivium was created to be a synchronous stream cipher designed to generate up to 2^{64} bits of key stream from an 80-bit secret key and an 80-bit initial value (IV). Its purpose was to simplify a stream cipher without sacrificing speed or flexibility. Trivium was submitted to the Profile II (hardware) of the eSTREAM competition [24] by its authors, Christophe De Cannire and Bart Preneel, and was selected as part of the portfolio for low area hardware ciphers (Profile 2) by the eSTREAM project.

Trivium generates up to 2^{64} bits of output from an 80-bit key and an 80-bit IV. The 288-bit internal state consisted of three shift registers of different lengths. At each round, one bit is shifted into the three shift registers using a non-linear combination of taps from that and one other register.Thus, one bit of output is produced. To initialize the cipher, the key and IV are written into two of the shift registers, with the remaining bits starting in a fixed pattern; the cipher state is then updated $4 \times 288 = 1152$ times, so that every bit of the internal state depends on every bit of the key and of the IV in a complex nonlinear way.

2.3 Feistel Structures

Originating from DES, these structures are a special type of nonlinear-feedback shift registers, called the Feistel structure. The basic structure involves a function performed on one half of the block, and then combined with the second half. The result is shifted down as in an NLFSR [29]. An example of these would be the ciphers DES [27], KATAN [26], SIMON, SPECK [11], and SIMECK [50]. A defining characteristic of this type of structure is that its encryption and decryption operations can be very similar or even identical. This makes implementation very efficient.

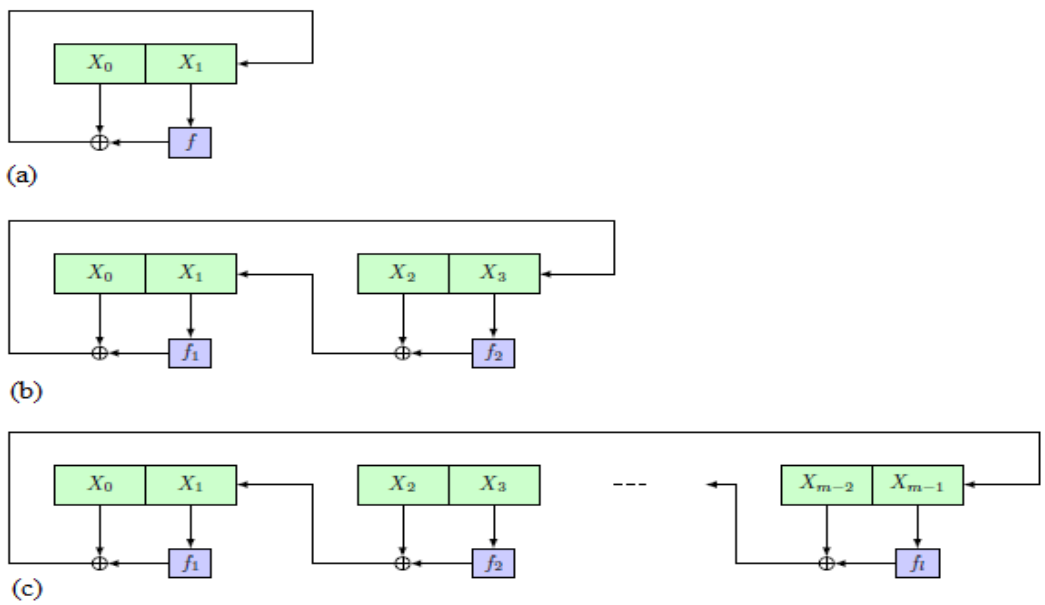

Figure 2.2: Feistel Structure using different lengths

2.3.1 Data Encryption Standard (DES)

The most well known Feistel structure would probably be used as DES or Data Encryption standard, with the structure shown in Figure 2.2-(a) [30] and detailed in Figure 2.3 [30]. IBM developed it in the 1970s and was based on the Feistel structure. The algorithm was submitted to the National Bureau of Standards (NBS), which would become the National Institute of Standards and Technology (NIST) in the future. After consultation with the National Security Agency (NSA), the NBS selected a modified version, which became the

official Federal Information Processing Standard (FIPS) for the United States in 1977 [48]. FIPS's overall structure used 16 identical stages of processing. The initial and final permutation were inverses.

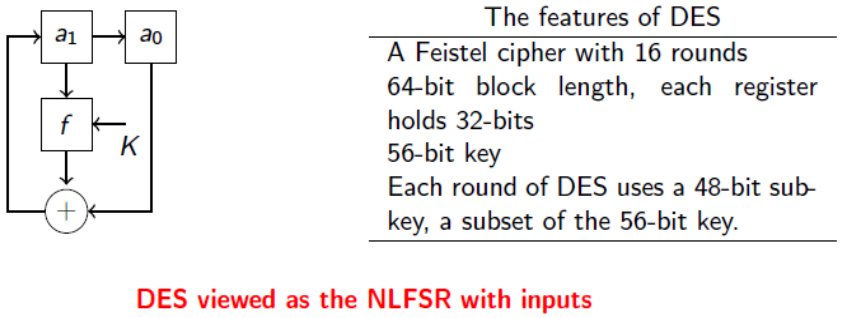

Figure 2.3: Digital Encryption Standard (DES) structure

2.3.2 Simon and Speck

The Simon block cipher is a balanced Feistel cipher with an n-bit word, and therefore the block length is $2n$ with the structure shown in Figure 2.2-(a). The key length is a multiple of n by 2, 3, or 4, which is the value m. A Simon cipher implementation is denoted as Simon$2n/nm$. The block component of the cipher is uniform between the Simon implementations; however, the key generation logic is dependent on the implementation of 2, 3 or 4 keys. This family of block ciphers was designed for lightweight purposes and has a focus on hardware optimization.

Speck supports a variety of block and key sizes. A block is always two words, but the words may be 16, 24, 32, 48 or 64 bits in size. The corresponding key is 2, 3 or 4 words. The round function consists of two rotations, adding the right word to the left word, XORing the key into the left word, then and XORing the left word to the right word. Speck was designed to be optimized in software.

2.4 Substitution and Permutation Networks (SPN)

The algorithms that use a substitution network apply several alternating layers of substitution boxes (S-boxes), the non-linear layer, followed by a linear layer which performs a

permutation, which shifts and mixes the output of the first layer. The advantages of this structure are the parallelisms so that computation can be very fast and efficient.

Some ciphers which use substitution and permutation networks include AES [44], Present [21], Piccolo [45], LED [34], EPCBC [51], KLEIN [32], PRINT [38], SKINNY [12], GIFT[10].

The AES (Advanced Encryption Standard) cipher [44] is very commonly used, and falls into this category. A block diagram of the round functions of AES are shown in Figure 2.4 [30]. AES is based on a design principle of a substitution/permutation network, a combination of both substitution and permutation, and is fast in both software and hardware. Unlike DES, AES does not use a Feistel network. AES is a variant of Rijndael [25], which has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. By contrast, the Rijndael specification is specified with block and key sizes that may be any multiple of 32 bits, with a minimum of 128 and a maximum of 256 bits.

AES operates on a 4×4 column-major order matrix of bytes, termed the state, although some versions of Rijndael have a larger block size and have additional columns in the state. The AES calculations are done in a particular finite field.

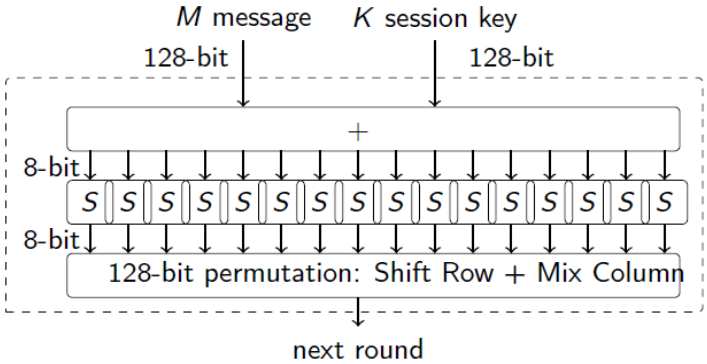

Figure 2.4: Advanced Encryption Standard (AES) structure

2.5 Sponge Construction

The sponge construction class of algorithms takes an input bit stream of any length and produces an output of any desired length. The defining characteristics of the sponge structure uses a fixed function that is repeatedly performed on the state memory, as shown

in Figure 2.5 [16]. The inputs are XORed over multiple rounds in the absorbing phase, while the output is calculated over multiple rounds in the squeezing phase.

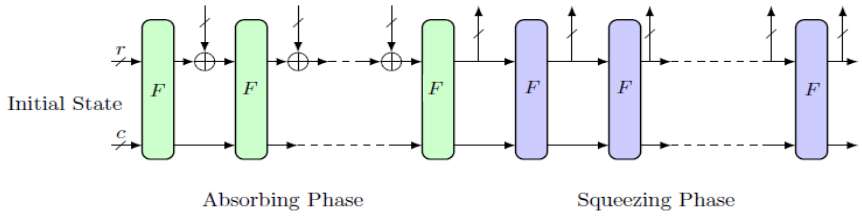

Figure 2.5: Sponge Construction Originally used in Keccak

Keccak is the most well known of these structures. The process involves where data is "absorbed" into the sponge, then the result is "squeezed" out. Message blocks are XORed into a part of the state during the absorbing phase, which is then transformed as a whole using a permutation function F. Then, output blocks are read from the same subset of the state during the squeeze phase, alternated with the state transformation function F. The size of the part of the state that is written and read is called the "rate" (denoted r), and the size of the part that is untouched by input/output is called the "capacity" (denoted c). The capacity determines the security of the scheme in bits. Thus, the maximum security level would be half the capacity in bits.

2.6 Lightweight Design for Cipher Systems

Ever since the introduction of the Sponge-based permutation dependent functions and the ability of such a construction to provide almost all of the major cryptographic functionalities, there has been a natural inclination towards designing cryptographic permutations. Starting from the Keccak family of permutations [16], and later Ascon [28], Norx [8], Simpira [47], and Gimli [13], all such proposals have the conventional aspect that designing a cryptographic permutation is better than a specific cryptographic primitive. While Ascon and Norx are used to instantiate a MonkeyDuplex sponge construction [17] that is optimized for authenticated encryption (AE) only, other proposals such as Simpira and Gimli focus mainly on the permutation design and only suggest section modes. What is clearly common among most of the permutation designs is that they either have sufficiently large state sizes ($\geq$ 320 bits) which directly translate to large hardware areas, or are optimized for software to make use of processor specific instructions (e.g., `Gimli` and `Simpira`). Lightweight AE schemes `Norx`-8 and `Norx`-16 with internal state sizes of 128 and 256 bits have lower

bounded estimated areas of 1368 and 2880 GE, respectively. However, both these `Norx` instances are specifically optimized for authenticated encryption that offer 80-bit and 96-bit security, respectively, and their security and instantiation for unkeyed modes are not investigated in the literature.

Because of NIST's lightweight cryptography project [42] that recognizes the apparent lack of cryptographic standards suitable for the whole spectrum of lightweight sections, several proposals have emerged. However, all of such proposals offer a single cryptographic functionality within the constrained hardware area (around 2000 GE [36]) dedicated for all security purposes. Examples of these algorithms are either block ciphers such as `Led` [34], `Present` [21], `Simon` and `Speck` [11], `Simeck` [50], `Skinny` [12], and `Gift` [10] or lightweight hash functions such as `Photon` [33], `Quark` [7], and `Spongent` [22].

2.7 Simeck

The Feistel structure that is used in our sponge construction consists of four blocks that will be introduced in the next two chapters. We opt to use the Simeck functions [50] as our round function due to its lightweight properties in hardware and its ability to be serialized. The state is split into two blocks. Several bitwise functions are performed on the first block, and then XORed with the second block. The bitwise function includes left shifting be 5 bits, XOR with the original, AND with a 1 bit left shifted version. The resulting block replaces the second block, while the original values in the second block are shifted into the first block. The full process and bitwise functions are shown in Figure 2.6. This process is repeated a number of times, depending on the size of the state [50].

Before the main rounds, the block is divided into two 32-bit halves and processed alternately; this crossing is known as the Feistel scheme. The Feistel structure ensures that decryption and encryption are very similar processes-the only difference is that the subkeys are applied in the reverse order when decrypting. The rest of the algorithm is identical. This greatly simplifies implementation, particularly in hardware, as there is no need for separate encryption and decryption algorithms.

The $\oplus$ symbol denotes the exclusive-OR (XOR) operation. The F-function scrambles half a block together with some of the key. The output from the F-function is then combined with the other half of the block, and the halves are swapped before the next round. After the final round, the halves are swapped; this is a feature of the Feistel structure which makes encryption and decryption similar processes.

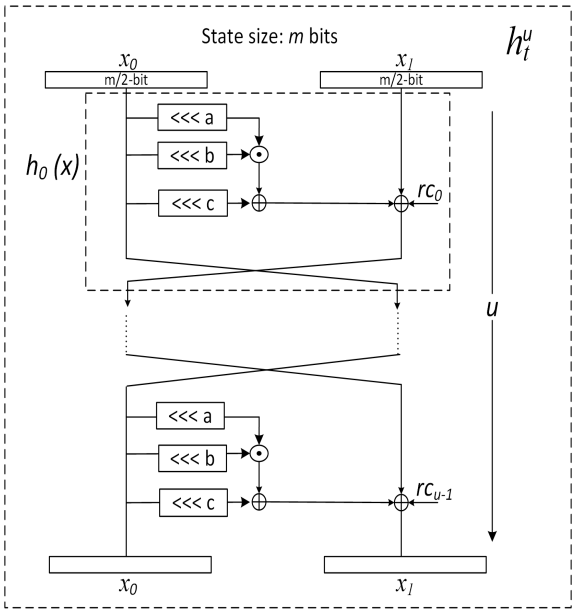

Figure 2.6: Simeck round function where $(a, b, c) = (5, 0, 1)$.

Chapter 3

sLiSCP: A Lightweight Cipher

In this chapter, we present the hardware implementation of sLiSCP which is a family of lightweight cryptographic permutations with the sole aim to provide a realistic lightweight cryptographic minimal design. Our design aims to provide multiple practical functionalities that are useful in lightweight applications today.

This Chapter is organized as follows. In Section 3.1, we introduce our lightweight permutation sLiSCP and give a brief overview. In Section 3.2, we explain the reasoning for our design and the advantages that we have over existing lightweight designs. In Section 3.3 we explain the structure of our permutation and our choices for parameters. We go in depth and explain our round function, type-2 Generalized Feistel Structure, modified Simeck, and our LFSR choice for our round constant. Section 3.4 presents the modes of operations for our permutation. We explain how sLiSCP is used in the unified duplex sponge as well as the Hash and AE modes. In Section 3.5, we go through the hardware implementation of this design and show our results. Section 3.6 goes through the hardware design choices and each process. In Section 3.5, we show the implementation results of sLiSCP. In addition, we discuss the hardware results of the Hash and AE modes.

3.1 sLiSCP

In SAC 2017, our group proposed the sLiSCP family of lightweight cryptographic permutations [2] was proposed specifically to address the limited hardware area which is dedicated for all security purposes in resource constrained devices. The name, sLiSCP, (pronounced ess - lisp) was an acronym for Simeck-based-permutations for Lightweight

Sponge Cryptographic Primitives. We wanted to highlight the sponge structure and the Simeck that was used in the design. We stressed that for such devices, it is desirable (if not only realistic) that a cryptographic design should provide low overhead for multiple cryptographic functionalities including (authenticated) encryption, hashing, and pseudorandom bit generation.

Hence, sLiSCP is proposed to be used in the unified sLiSCP duplex sponge construction to provide (authenticated) encryption and hashing functionalities. sLiSCP aims to provide an efficient and secure design for a sponge-specific permutation taking into perspective the relation between the state size and security parameters. sLiSCP offers two instances of the permutation with block sizes 192 and 256 bits with fully parallelized hardware areas of 2153 and 2833 GE in CMOS 65 nm ASIC, and 2318 and 3040 GE in CMOS 130 nm ASIC, respectively.

The sLiSCP family of permutations adopts two of the most efficient and extensively analyzed cryptographic structures, namely a 4-subblock Type-2 Generalized Feistel Structure (GFS) [43, 23] (see Figure 2.2), and a round-reduced unkeyed version of the Simeck encryption algorithm [50]. Specifically, the round function of Simeck is an independently parametrized version of the Simon round function [11] and has set a new record in terms of hardware efficiency and performance on almost all platforms. Moreover, Simeck, Simon and Simon-like variants have been extensively cryptanalyzed by the public cryptographic community [49, 39, 40, 41]. Simeck utilizes shift parameters that are more hardware efficient than those of Simon, and also in terms of bit diffusion, Simeck is better than other efficient shift parameters that are investigated in [39].

In this chapter, we present the design and the hardware optimization choices for our resulting implementations [46]. We implement the two instances of sLiSCP in the duplex unified sponge mode, and our parallel ASIC implementation results in CMOS 65 nm show that the areas of sLiSCP-192 and sLiSCP-256 are 2289 GEs and 3039 GEs with a throughput of 29.62 and 44.44 kbps, respectively, and their areas in CMOS 130 nm are 2498 GEs (resp. 3319).

The hardware section of our sLiSCP design is implemented using Application Specific Integrated Circuit (ASIC) implementations. We can benefit from the specific implementation of ASIC to reduce the area and power consumption as well as maximize the operating frequency.

3.2 Design Rationale

Our main objective for a minimal design is to provide multiple cryptographic functionalities, such as hashing modes and authenticated encryption modes. Using sLiSCP in the duplex sponge construction is an excellent choice as it offers a number of key features that enable the design of multiple cryptographic functionalities using the same hardware circuitry. In other words, both keyed primitives such as (authenticated) encryption and unkeyed primitives such as hash function and pseudo random bit generator are easily realized with minimum overhead. The sponge construction ensures provable security [15] when the underlying permutation is indistinguishable from a random function. Accordingly, the main challenge is to design a secure and hardware efficient permutation for resource constrained applications.

For an unkeyed sponge-based primitive with state size $b = r + c$, where r and c denote the rate and capacity, respectively, a bound of $2^{c/2}$ against generic attacks is proven [15], which sets a lower bound on the state size of the permutation. If the permutation is used to construct a hash function with output of size t, the permutation state size should be at least $(2t + r)$ bits.

For lightweight applications, a hash digest of 160 bits restricts the state size b to a minimum of 192 bits for a rate of 32 bits, which means that around 1000 GEs are reserved only for the state. While a substitution permutation network (SPN) based design requires a relatively small number of rounds to achieve the desired security, offers good throughput, and is simpler to cryptanalyze, the hardware implementation cost becomes expensive for a larger state size due to the large number of substitution boxes and their cost. To design a lightweight permutation, we opted for a non-SPN based design consisting of Type-2 GFS and a round-reduced Simeck-m (i.e. Simeck with m-bit state) as a round function, which are based on the Feistel network. Our design choices are motivated by design goals in security and hardware efficiency.

The sponge construction is well-investigated and has been cryptanalyzed and proven secure for different keyed and unkeyed applications including the SHA-3 winner Keccak [16]. Moreover, its security when instantiated with a specific permutation F relies on the resistance of F against distinguishing attacks and accordingly, we focus our cryptanalysis efforts to investigating sLiSCP against a wide variety of such attacks.

Both the implementations of Type-2 GFS and the Simeck-m round functions are extremely efficient in terms of the hardware footprint. To provide an average estimation on the GE count, we assume an ASIC $65\,nm$ technology that requires 2.5 GE for an XOR, 2 GE for an AND. Given the latter estimates, a 4-subblock Type-2 GFS using Simeck-48

mixing requires around $2 \times 48 \times 2.5 = 240$ GE and each Simeck-48 round function consumes around $24 \times (2.5 + 2 + 2.5) = 168$ GE, which sums to around 576 GEs for the parallel implementation of the permutation round function logic with a state size of $4 \times 48 = 192$ bits.

3.3 Specification of sLiSCP

sLiSCP uses a sponge structure to absorb inputs to provide multiple cryptographic functionalities. These include tools such as hash, encryption and authentication.

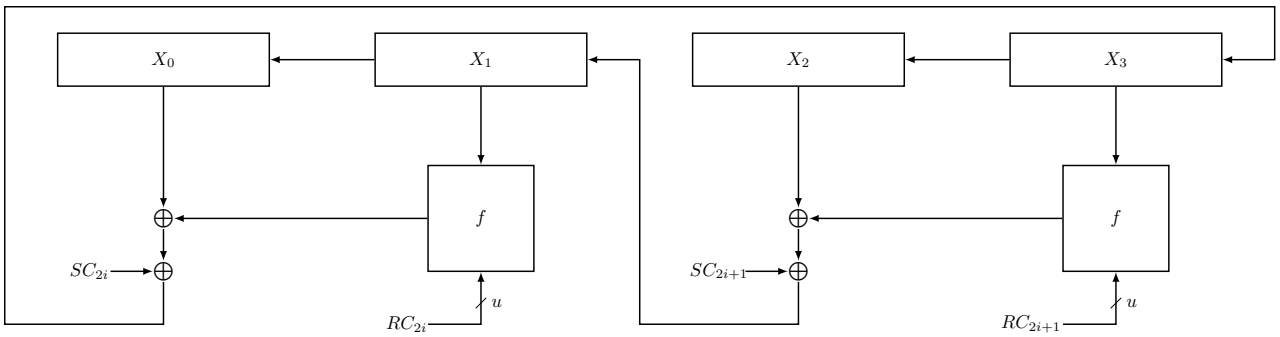

Figure 3.1: sLiSCP Round Function

3.3.1 Structure of sLiSCP

sLiSCP uses a Type 2 Generalized Feistel Structure (GFS), see Figure 3.2. This is similar to the Feistel structure presented previously in Figure 2.2-(b). The state consists of 4 blocks of 48 bits or 64 bits, respectively for the sLiSCP192 and sLiSCP256 sizes and 2 instances. Each pair of blocks use one Simeck^u-m round. This Simeck round function is slightly modified from the original Simeck family of ciphers. It is simplified and does not use a round key. Instead we generate round constants to add during each inner round of Simeck. This will be described in the next subsection. As shown in Figure 3.1, we use two instances of the round function in the generalized Feistel structure.

The internal round function used in the Feistel structure is chosen to be a modified Simeck function, due to its efficient implementation in hardware and its security. This allows the satisfaction of the lightweight requirements.

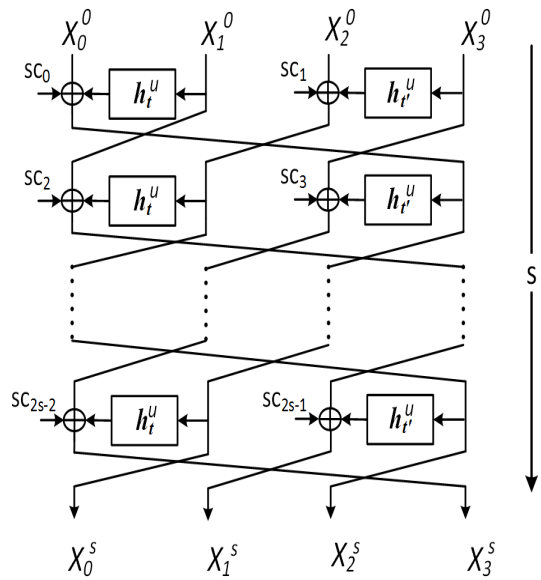

Figure 3.2: Type 2 Generalized Feistel Structure (GFS)

3.3.2 Round Function

In this subsection, we formally describe our sLiSCP permutation, which has two instances. The core algorithm of the sLiSCP permutation is built upon the Simeck cipher's round function and a 4-subblock Type-2 GFS construction. We present two lightweight instances of the sLiSCP permutation.

We use Simeck^u-m as a round function in the sLiSCP permutation. Simeck^u-m is derived from the Simeck cipher whose block length equal to m and round function is given by:

$$h_i(x) = R_i(x_0, x_1) = (x_1, f(x_0, x_1))$$

where $f(x_0, x_1) = (x_0 <<< 5) \odot (x_0 <<< 0) + (x_0 <<< 1) + x_1 + k_i$ $h_i : \mathbb{F}_2^m \to \mathbb{F}_2^m$, $<<<$ is a left cyclic shift operator, x_0 and x_1 are $\frac{m}{2}$-bit words and k_i is a $\frac{m}{2}$-bit round key added at the i-th round. We modify the round function as follows. Instead of adding a round key in $h_i, 0 \leq i \leq u-1$, we add a round constant rc_i in h_i where $rc_i = (C||t_i), C = 2^{\frac{m}{2}} - 2, t_i \in F_2$. This resulting round function is illustrated in Figure 2.6. This is the main Simeck function that is used throughout this thesis.

Let t be the integer representation of the u-tuple $(t_0, t_1, \cdots, t_{u-1})$. Simeck^u-$m$ is defined as

$$\text{Simeck}^u\text{-}m(x) = h_{u-1} \circ h_{u-2} \circ \cdots \circ h_0(x), x = x_0 \| x_1,$$

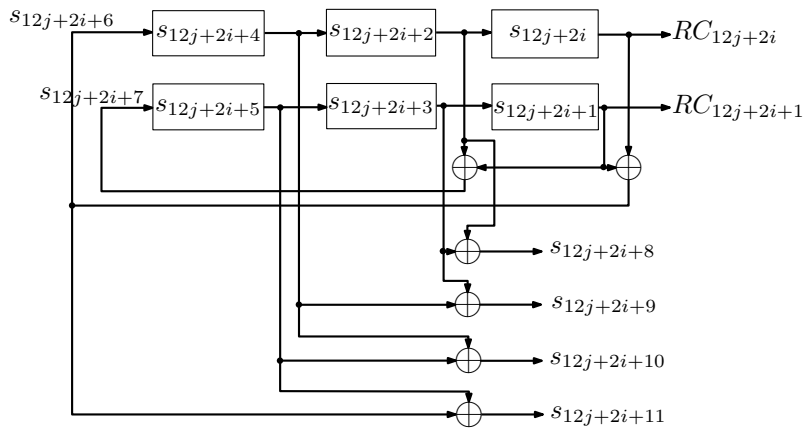

Figure 3.3: Degree 6 LFSR used to generate the Round Constants and Step Constants for sLiSCP-192

where the round constant rc_i is used in h_i at the i-th round. The round constants are generated using an LFSR described in Section 3.3.3. We refer to Simeck^u-m as h_t^u. We use the subscript t to uniquely instantiate h_t^u as $h_{t_1}^u$ and $h_{t_2}^u$, for $t_1 \neq t_2$, which are parametrized by different round constants, t_1 and t_2. We study the cryptographic properties of Simeck^u-m and present the bounds against differential and linear cryptanalysis for sLiSCP based on the minimum number of active S-boxes, see [3] for detailed cryptanalysis of the round function.

3.3.3 LFSR Design

The LFSR used in our sLiSCP-192 design is used for its simplicity. Since our Simeck function is used as a function for the GFS, the specific properties of the key generation used in original Simeck is not needed. Instead we use a simple LFSR to generate the constants that resemble the keys used in Simeck^u-m. Since our 4-block sLiSCP-192 round function would use two instances of the Simeck round function, our LFSR needs to produce two constants for each round function for each clock cycle. Then, we use a 6-stage LFSR. Illustrated in Figure 3.3, the LFSR produces two states in one clock cycle.

This uses 6 stages of LFSR with two bits output used in the sLiSCP-192 round function. The characteristic polynomial of the LFSR is represented as $p(x) = x^6 + x + 1$.

After the 6 rounds of the internal function, a 6-bit constant will be generated from the same parallel LFSR as the round constants with four extra XORs. During this step

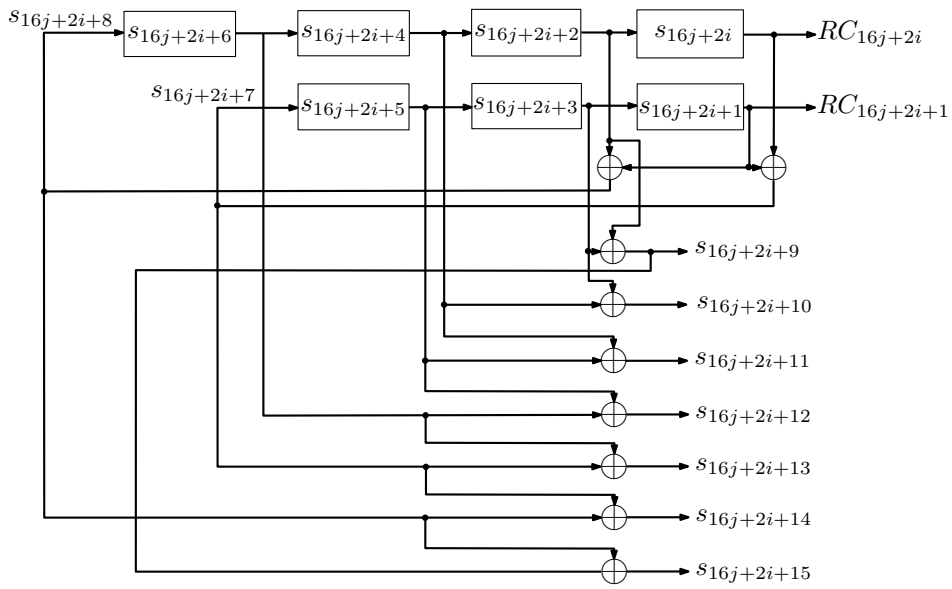

Figure 3.4: Degree 7 LFSR used to generate the Round Constants and Step Constants for sLiSCP-256

the states for the LFSR are $(s_{12j+10}, s_{12j+11}, s_{12j+12}, s_{12j+13}, s_{12j+14}, s_{12j+15})$. We assign $(s_{12j+10}, s_{12j+12}, s_{12j+14})$ to SC_{2j} and $(s_{12j+11}, s_{12j+13}, s_{12j+15})$ to SC_{2j+1}. For the other three values of SC_{2j} $(s_{12j+16}, s_{12j+18}, s_{12j+20})$ and SC_{2j+1} $(s_{12j+17}, s_{12j+19}, s_{12j+21}$), we use,

$s_{12j+16} = s_{12j+10} \oplus s_{12j+11}$

$s_{12j+17} = s_{12j+11} \oplus s_{12j+12}$

$s_{12j+18} = s_{12j+12} \oplus s_{12j+13}$

$s_{12j+19} = s_{12j+13} \oplus s_{12j+14}$

$s_{12j+20} = s_{12j+14} \oplus s_{12j+15}$

$s_{12j+21} = s_{12j+15} \oplus s_{12j+16}$.

Overall, our step constant would be generated as,

$$SC_{2j} = s_{12j+10}||s_{12j+12}||s_{12j+14}||s_{12j+16}||s_{12j+18}||s_{12j+20}$$
$$SC_{2j+1} = s_{12j+11}||s_{12j+13}||s_{12j+15}||s_{12j+17}||s_{12j+19}||s_{12j+21}$$

For our larger sLiSCP-256 design, we use a similar LFSR design. However, this LFSR is larger with 7 states. The characteristic polynomial of the LFSR is represented as $p(x) =$

x^7+x+1, shown in Figure 3.4. Similar to the previous LFSR, we generate 2 round constants each clock cycle and generate two step constants after 6 rounds. The step constants for this LFSR would be,

$$SC_{2j} = s_{16j+14}||s_{16j+16}||s_{16j+18}||s_{16j+20}||s_{16j+22}||s_{16j+24}||s_{16j+26}||s_{16j+28}$$
$$SC_{2j+1} = s_{16j+15}||s_{16j+17}||s_{16j+19}||s_{16j+21}||s_{16j+23}||s_{16j+25}||s_{16j+27}||s_{16j+29}$$

3.4 Modes of Operation

Because of the sponge design of sLiSCP, there are several modes that can be used. This section will describe how the sLiSCP modes are used, including the hash and authenticated encryption modes of sLiSCP.

3.4.1 The sLiSCP Mode

The utilized sponge mode modifies the keyed initialization and keyed finalization stages of the Ascon [28] and NORX [8] modes which make key recovery hard even if the internal state is recovered and also renders universal forgery with the knowledge of the internal state unattainable. The adopted modification makes the initialization and finalization stages more hardware efficient and adaptable to different primitive modes.

In particular, instead of initializing the state with the key, K, and then again XORing it with the permutation output that requires an extra $|K|$ XORs, we initialize the state with the key and then again absorb the key in the rate part during the initialization and finalization phases. We also use the domain separation technique as used in NORX because it runs for all rounds of all stages, and thus reduces the chances of side channel analysis and offers uniformity across different stages. The separation between the processing of different types of inputs is important to distinguish between the roles of the processed data. To this end, we only have one round function (See Fig. 3.5) that incorporates absorption, squeezing, and domain separation, and according to the fed inputs, we decide which stage and functionality to implement.

Initialization, absorbing, and squeezing.

Our sLiSCP permutation is based on Type-2 GFS where, apart of the permutation size, each subblock is either 48 or 64 bits. Since we use it in sponge-based modes, we need to

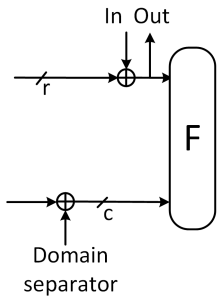

Figure 3.5: Unified round function which can be used in all stages of both keyed and unkeyed modes.

specify exactly from where the r-bit input is absorbed and the r-bit output is squeezed. For sLiSCP permutations, we consider the b-bit state as a series of four m-bit subblocks, X_0, X_1, X_2, X_3 (see Figure 3.2), where m is equal to 48 and 64 for sLiSCP-192 and sLiSCP-256, respectively. We divide the state S of sLiSCP as bytes, $S = (B_0, B_1, \cdots, B_{l-1})$ where $l = 24$ and 32, for sLiSCP-192 and sLiSCP-256, respectively. Moreover, each subblock X_i can be viewed as a series of $j = \frac{m}{8}$ bytes, $B_{ij+0}, B_{ij+1}, \cdots, B_{ij+(j-1)}$ arranged from left to right. In nonce-based keyed modes, initially, the state is loaded with the nonce and key bytes, denoted by NB_w, $w = 0, 1, \cdots, n$, and KB_z, $z = 0, 1, \cdots, k$, respectively and remaining bytes are set to zero. Also, when the hashing mode is employed, we load the state by a 3-byte IV and the remaining bytes are set to zero. Specifically, the first two IV bytes are assigned to the first two bytes of X_0, and the remaining IV byte is loaded in the first byte of X_2. All initialization public variables either nonce or IV are ceiling divided and loaded in the even indexed subblocks, X_0 and X_2, in an ascending byte order. The key is loaded in the odd indexed subblocks, X_1 and X_3 in the same manner, and if the key size is larger than half the state size, then remaining key bytes populate the remaining bytes in X_0 and X_2 equally and in an ascending order. For example, for sLiSCP-192/80, sLiSCP-192/112, and sLiSCP-256/128, the state is initialized as follows:

sLiSCP-192/80: $B_0 \leftarrow NB_0, B_1 \leftarrow NB_1, B_2 \leftarrow NB_2, B_3 \leftarrow NB_3, B_4 \leftarrow NB_4, B_{12} \leftarrow NB_5, B_{13} \leftarrow NB_6,$
$B_{14} \leftarrow NB_7, B_{15} \leftarrow NB_8, B_{16} \leftarrow NB_9, B_6 \leftarrow KB_0, B_7 \leftarrow KB_1, B_8 \leftarrow KB_2, B_9 \leftarrow KB_3,$
$B_{10} \leftarrow KB_4, B_{18} \leftarrow KB_5, B_{19} \leftarrow KB_6, B_{20} \leftarrow KB_7, B_{21} \leftarrow KB_8, B_{22} \leftarrow KB_9.$

sLiSCP-192/112: $B_0 \leftarrow NB_0, B_1 \leftarrow NB_1, B_2 \leftarrow NB_2, B_3 \leftarrow NB_3, B_4 \leftarrow NB_4, B_{12} \leftarrow NB_5, B_{13} \leftarrow NB_6,$
$B_{14} \leftarrow NB_7, B_{15} \leftarrow NB_8, B_{16} \leftarrow NB_9, B_6 \leftarrow KB_0, B_7 \leftarrow KB_1, B_8 \leftarrow KB_2, B_9 \leftarrow KB_3,$
$B_{10} \leftarrow KB_4, B_{11} \leftarrow KB_5, B_{18} \leftarrow KB_7, B_{19} \leftarrow KB_8, B_{20} \leftarrow KB_9, B_{21} \leftarrow KB_{10}, B_{22} \leftarrow KB_{11},$
$B_{23} \leftarrow KB_{12}, B_5 \leftarrow KB_6, B_{17} \leftarrow KB_{13}$

$$\begin{aligned}\textsf{sLiSCP-256/128:}\ & B_0 \leftarrow NB_0, B_1 \leftarrow NB_1, B_2 \leftarrow NB_2, B_3 \leftarrow NB_3, B_4 \leftarrow NB_4, B_5 \leftarrow NB_5, B_6 \leftarrow NB_6, B_7 \leftarrow NB_7,\\ & B_{16} \leftarrow NB_8, B_{17} \leftarrow NB_9, B_{18} \leftarrow NB_{10}, B_{19} \leftarrow NB_{11}, B_{20} \leftarrow NB_{12}, B_{21} \leftarrow NB_{13}, B_{22} \leftarrow NB_{14},\\ & B_{23} \leftarrow NB_{15}, B_8 \leftarrow KB_0, B_9 \leftarrow KB_1, B_{10} \leftarrow KB_2, B_{11} \leftarrow KB_3, B_{12} \leftarrow KB_4, B_{13} \leftarrow KB_5,\\ & B_{14} \leftarrow KB_6, B_{15} \leftarrow KB_7, B_{24} \leftarrow KB_8, B_{25} \leftarrow KB_9, B_{26} \leftarrow KB_{10}, B_{27} \leftarrow KB_{11}, B_{28} \leftarrow KB_{12},\\ & B_{29} \leftarrow KB_{13}, B_{30} \leftarrow KB_{14}, B_{31} \leftarrow KB_{15}\end{aligned}$$

In the sLiSCP modes, we use *initialize(x)* to denote the process of loading the state with x in the positions described above. As for absorbing and squeezing, we want the input bits to be processed by the Simeck^u-m box as soon as possible so we achieve better diffusion. Accordingly, choosing the right place for absorbing data determines how fast it is processed by the round function which is important since not all the subblocks in GFS constructions receive the same amount of processing at first.

The same $r/8$ bytes are used for absorbing and squeezing and they are denoted by the following state bytes:

$$\begin{aligned}\textsf{sLiSCP-192:}\ & B_6, B_7, B_{18}, B_{19}\\ \textsf{sLiSCP-256:}\ & B_8, B_9, B_{10}, B_{11},\\ & B_{24}, B_{25}, B_{26}, B_{27}.\end{aligned}$$

In the AE mode, the tag is extracted from the same byte positions which are used in the key initialization stage. Hence, the process of tag extraction from state S is denoted by $tagextract(S)$. We denote the rate and capacity parts of the state S by S_r and S_c, respectively, thus $S = (S_r, S_c)$. In what follows, we show how we use the unified round function depicted in Figure 3.5 to implement several functionalities using sLiSCP permutation in the sLiSCP mode.

3.4.2 Authenticated Encryption

An authenticated encryption algorithm takes as input a secret key K, a nonce N, a block header A (a.k.a, associated data) and a message M and outputs a ciphertext C with $|C|= |M|$, and an authentication tag T. Mathematically, an authenticated encryption $\mathcal{AE}$ mode is defined as

$$\mathcal{AE} : \{0,1\}^k \times \{0,1\}^n \times \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^t$$

with

$$\mathcal{AE}(K, N, A, M) = (C, T)$$

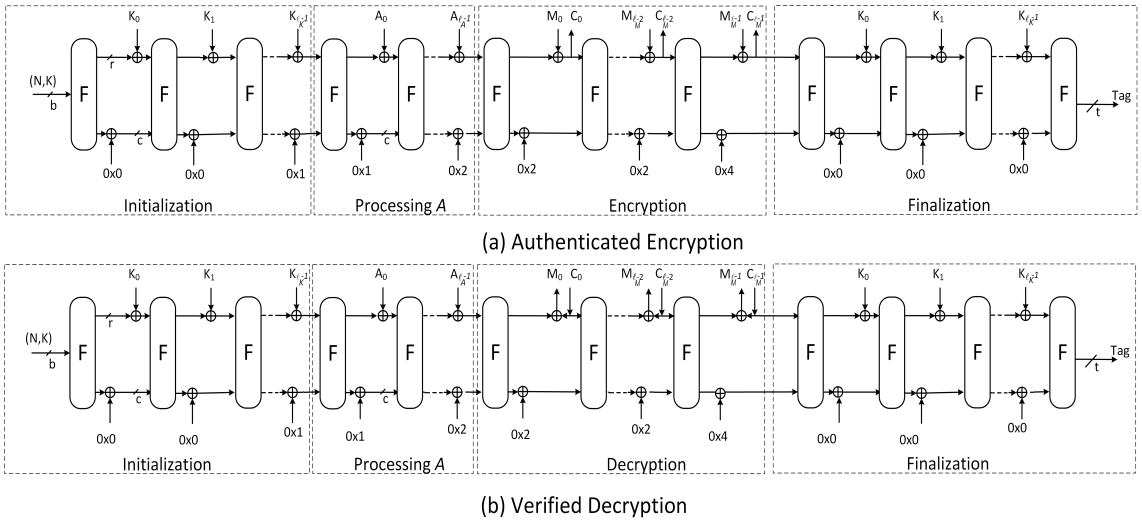

Figure 3.6: Sponge Construction for sLiSCP

where k is the bit length of K, n is the bit length of N.

We denote an instance of sLiSCP in a keyed mode by sLiSCP-b/k, where b and k denote the state size and the key length, respectively. In such a mode, we limit the number of processed bits per key to 2^a, which is known as the data usage limit [14]. Specifically, 2^a denotes the value that an implementation restricts the maximum message size (data queries) that can be processed per a given key such that one can attain bit security equal to 2^k when $c \geq k + a + 1$. Recommended parameters for sLiSCP when used in AE mode are listed in Table 3.1.

Table 3.1: Recommended parameter set for sLiSCP-b/k when used in authenticated encryption mode.

Algorithm	Key	Nonce	Tag	Block size r	Capacity c	Usage exponent a
sLiSCP-192/80	80	80	80	32	160	72
sLiSCP-192/112	112	80	112	32	160	40
sLiSCP-256/128	128	128	128	64	192	56

The depiction of the encryption and decryption processes using the sLiSCP sponge mode

is shown in Figure 3.6. We describe the padding rule and the algorithms of the AE below.

Padding: Padding is necessary when the length of the processed data is not a multiple of the rate r value and also to act as a delimiter between data of unknown lengths. Since the keys are of fixed length, we need to pad it by appending zeros only if its length is not a multiple of r bits such that the padded K is divided into ℓ_K r-bit blocks $K_0 \| K_1 \| \cdots \| K_{\ell_K - 1}$. Afterward, the padding rule (10^*) denoting a single 1 followed by required 0's is applied to the message M such that its length after padding is a multiple of r. Then the resulting padded message is divided into ℓ_M r-bit blocks $M_0 \| M_1 \| \cdots \| M_{\ell_M - 1}$. A similar procedure is carried out on the associated data A which results in ℓ_A r-bit blocks $A_0 \| A_1 \| \cdots \| A_{\ell_A - 1}$. In the case where no associated data is present, no processing is necessary. We summarize the padding rules for key, message and associated data below.

$$
\begin{aligned}
pad_r(K) &\rightarrow K \| 0^{r-(|K| \bmod r)}, \text{ if } |K| \bmod r \neq 0 \\
pad_r(M) &\rightarrow M \| 1 \| 0^{r-1-(|M| \bmod r)} \\
pad_r(A) &\rightarrow \begin{Bmatrix} A \| 1 \| 0^{r-1-(|A| \bmod r)} & if |A| > 0 \\ \phi & if |A| = 0 \end{Bmatrix}
\end{aligned}
$$

Initialization: The initial state S is loaded with the nonce N and key K as described in Section 3.4.1. Each r-bit key block K_i is absorbed by XORing it to the S_r part of the state and a one bit domain separator is XORed to the most significant bit in byte B_{23} and B_{31} for sLiSCP-192 and sLiSCP-256 with the absorption of the last key block $K_{\ell_K - 1}$, respectively. Afterward, the sLiSCP permutation is applied to the whole state. The initialization steps are described below.

$$
\begin{aligned}
S &\leftarrow F(initialize(N, K)) \\
S &\leftarrow F((S_r \oplus K_i), S_c),\ i = 0, \cdots, \ell_K - 2 \\
S &\leftarrow F((S_r \oplus K_i), (S_c \oplus 0^{c-1} \| 1)),\ i = \ell_K - 1
\end{aligned}
$$

Processing A: If there is associated data, each r-bit block A_i, $i = 0, \cdots, \ell_A - 1$ is XORed to the first S_r part of the internal state S and one-bit domain separator is XORed to the last byte of the states. Then, sLiSCP permutation is applied on the whole state.

$$
\begin{aligned}
S &\leftarrow F((S_r \oplus A_i), (S_c \oplus 0^{c-1} \| 1)),\ i = 0, \cdots, \ell_A - 2 \\
S &\leftarrow F((S_r \oplus A_i), (S_c \oplus 0^{c-2} \| 2)),\ i = \ell_A - 1
\end{aligned}
$$

Encryption: Similar to the processing of A, however, with a different domain separator, each message r-bit block M_i, $i = 0, \cdots, \ell_M - 1$ is XORed to the S_r part of the internal state S resulting in the corresponding ciphertext C_i which is then extracted from the S_r part of the state. After the computation of each C_i, the whole internal state S is permuted by F.

$$\begin{aligned} C_i &\leftarrow S_r \oplus M_i, \\ S &\leftarrow F(C_i, (S_c \oplus 0^{c-2} \| 2)) \text{ if } 0 \leq i < \ell_M - 2 \\ S &\leftarrow F(C_i, (S_c \oplus 0^{c-3} \| 4)) \text{ if } i = \ell_M - 1 \end{aligned}$$

To keep a minimal overhead, the last ciphertext block $C_{\ell_M - 1}$ is truncated so that its length is equal to that of the last unpadded message block $M_{\ell_M - 1}$ (i.e., $C_{\ell_M - 1} = \lfloor C_{\ell_M - 1} \rfloor_{(|M| \bmod r)}$).

Decryption: Each ciphertext r-bit block C_i, $i = 0, \cdots, \ell_M - 1$ is XORed to the S_r part of the internal state S to calculate the corresponding message block M_i, then the same C_i replaces the r-bit block S_r in the internal state, then the whole internal state S is transformed by the permutation F

$$\begin{aligned} M_i &\leftarrow S_r \oplus C_i \\ S &\leftarrow F(C_i, (S_c \oplus 0^{c-2} \| 2)),\ 0 \leq i < \ell_M - 2 \end{aligned}$$

The last message block $M_{\ell_M - 1}$ is calculated by XORing the ciphertext block $C_{\ell_M - 1}$ to the truncated S_r part of the state, then replacing the S_r part by $C_{\ell_M - 1} \| (\lceil S_r \rceil^{(r - |M| \bmod r)} \oplus (1 \| 0^{(r-1-|M| \bmod r)}))$.

$$\begin{aligned} M_{\ell_M - 1} &\leftarrow \lfloor S_r \rfloor_{(|M| \bmod r)} \oplus C_{\ell_M - 1} \\ S &\leftarrow F(C_{\ell_M - 1} \| (\lceil S_r \rceil^{(r - |M| \bmod r)} \oplus (1 \| 0^{(r-1-|M| \bmod r)})), (S_c \oplus 0^{c-3} \| 4)). \end{aligned}$$

Finalization: Finally, the ℓ_K key blocks are absorbed and the tag is extracted from the chosen bytes of the state as described earlier.

$$\begin{aligned} S &\leftarrow F((S_r \oplus K_i), S_c),\ i = 0, \cdots, \ell_K - 1 \\ T &\leftarrow tagextract(S). \end{aligned}$$

The decryption procedure returns the message blocks M_i ,$i = 0, 2, \cdots \ell_M - 1$, only if the calculated tag value is equal to the received tag value. The AE mode assumes nonce respecting adversary and we do not claim security in the event of nonce reuse, although, the initialization and finalization stages combined by the number of rounds used in the sLiSCP permutation tremendously reduces the effect of such attacks. We claim no security for reduced-round versions of the sLiSCP permutation operating in the sLiSCP mode. In summary, our security claims are given in Table 3.2 An authenticated encryption algorithm

Table 3.2: Security claims for sLiSCP operating in the sLiSCP AE mode where sLiSCP-b/k denotes sLiSCP with state size b and key size k.

Security property	sLiSCP-192/80	sLiSCP-192/112	sLiSCP-256/128
Data confidentiality	80	112	128
Data integrity	80	112	128
Associated data integrity	80	112	128
Nonce data integrity	80	112	128

can be easily used to provide either encryption or authentication only. More precisely, when using sLiSCP for encryption only, we run the algorithm as usual and stop after the last message block is encrypted and since we do not care about tag forgery, we can omit/ignore the finalization stage and the tag extraction stage. We also set to zero all the domain separation as we are only processing one domain of messages. For the MAC generation, we can ignore the initialization phase and directly load both the key and nonce in the state and start absorbing the message blocks directly after applying F once to the initialized state. This design decision is attributed to the fact that during the MAC generation, there is no leaked part of the state and the attacker has little control (only probabilistic) over the state which makes state recovery attacks harder than that in the case of authenticated encryption or encryption only. However, since we care about tag forgery, we need to maintain a strong keyed finalization stage, also, in this mode, we may zero all domain separator XORs because we are authenticating data apart of its role. Also, the adopted initialization and finalization stages are not efficient throughput wise when the processed message is short. However, in such a case, the ability of attacker to recover the internal state is reduced too, so when processing short messages we can directly initialize the state with the key and nonce, and skip the initialization phase.

3.4.3 Hash Computation

A hash function takes as input a message M, and a standard initialization vector IV, and then returns a fixed size output H, called hash or message digest. Formally, the hash mode is specified by

$$\mathcal{H} : \{0,1\}^* \times \{0,1\}^{iv} \rightarrow \{0,1\}^h$$

with $H = \mathcal{H}(M, IV)$ where iv is the length of the IV and h is the length of the hash. The depiction of the hashing process of the sLiSCP mode is shown in Fig. 3.7.

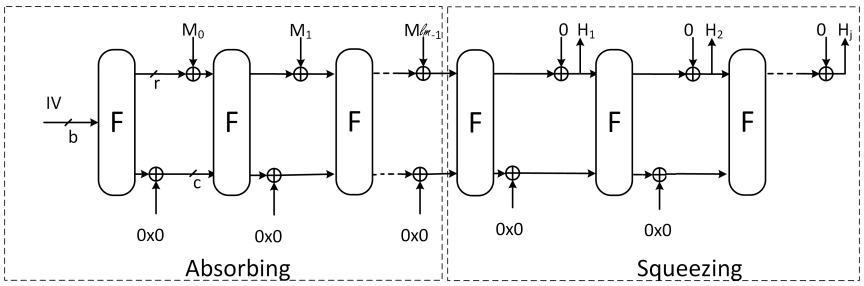

Figure 3.7: Hash computation of the sLiSCP mode of operation.

We adapt the sLiSCP mode such that it can be used to initially absorb the message blocks and then squeeze hash blocks to output the desired hash value. This is an unkeyed mode where we do not need an initialization or finalization stage. It has been shown [16] that inverting the squeezing phase falls in the category of "multiblock constrained-input constrained-output problem" which requires $2^{\min(h,b)-r}$ computations to recover the state before the squeezing phase. Once such an internal state is recovered one can launch a meet-in-the-middle attack with around $2^{c/2}$ computations to get a preimage of a given hash of length h [16, 33]. The latter condition reduces the generic preimage attack on the sponge-based hash functions from 2^h to $\min(2^{\min(h,b)}, \max(2^{\min(h,b)-r}, 2^{c/2}))$ where such a preimage security is usually dominated by $2^{\min(h,b)-r}$ and accordingly highly dependent on the squeezed bit rate. In [33], Guo *et al.* suggested using a flexible squeezing bit rate $r' < r$ that offers a trade-off between speeding up the hash computation and preimage security. More precisely, a smaller r' would make the time complexity of a preimage attack equal to $2^{h-r'}$ (assuming that the hash length is less than the state length) which is close to that of the expected generic one 2^h. On the other hand, if small inputs are hashed (e.g., electronic product code (EPC) data, which is a 96-bit string), small squeezing bit rate may make the computation too slow as one needs $\lceil h/r' \rceil - 1$ calls to the underlying permutation. Another solution to reach the expected generic preimage security is to run one more squeezing round

after one extracts the desired hash length h [33, 7], thus increasing the acquired output to $h + r'$ and in this case the complexity of the generic preimage attack is equal to :

$$\min(2^{\min(h+r',b)}, \max(2^{\min(h,b-r')}, 2^{c/2})) \geq 2^h \text{ when } c + r - r' \geq h.$$

We adopt a standard initialization vector that combines the parameters of a given sLiSCP instance. In particular, we use the same format used by Guo *et al.* in PHOTON such that for any instance the state is first initialized by $IV = h/2\|r\|r'$, where 8 bits are used to encode each of the used $h/2$, r, and r' sizes. The claimed security levels for the recommended parameters for sLiSCP in the hashing mode are given in Table 3.3.

Table 3.3: Recommended parameter set for sLiSCP-b when used in hashing mode and the associated bit securities.

Algorithm	IV	h	r	r'	c	collision	Sec. preimage	Primage
sLiSCP-192	$0x502020$	160	32	32	160	80	80	128
sLiSCP-256	$0x604040$	192	64	64	192	96	96	128
sLiSCP-256	$0x604020$	192	64	32	192	96	96	160

Initialization and Message Padding The state is first initialized with the IV and the padding rule (10^*) is applied to the input message M where a single 1 followed by enough $0s$ is appended to it such that its length after padding is a multiple of r bits. Then the resulted padded message is divided into ℓ_M r-bit blocks $M_0\|M_1\|\cdots\|M_{\ell_M-1}$. Accordingly, the message padding procedure is given by: $pad_r(M) \rightarrow M\|1\|0^{r-1-(|M| \bmod r)}$

Absorbing and Squeezing: Initially each message block is absorbed by XORing it to the S_r part of the state, then sLiSCP permutation is applied afterward. After absorbing all the message blocks, the h-bit output is extracted from the S_r part of the state r' bits at a time followed by the application of sLiSCP permutation until a total of $\lceil h/r' \rceil$ extractions are completed, then if the resulting extracted bits are more than the desired hash length, truncation is performed. Note that if $r' < r$, then its byte size is extracted from the same subblocks used in squeezing, X_1 and X_3, such that the first and second halves of the r' bytes are extracted from X_1 and X_3, respectively, in an ascending byte order.

$$Absorbing: \quad S \leftarrow F(S_r \oplus M_i, S_c) \text{ for } 0 \leq i \leq \ell_M - 1$$

$$
\begin{aligned}
Squeezing: \quad H_1 &\leftarrow S'_r \\
S &\leftarrow F(S),\ 2 \leq i \leq j,\ \text{for } j = \lceil h/r' \rceil \\
H_i &\leftarrow S'_r \\
H &\leftarrow \lfloor H_1 \| H_2 \| \cdots \| H_j \rfloor_h
\end{aligned}
$$

Reseedable Pseudo Random Bit Generator (PRBG). The hash construction can be used as a reseedable pseudo random bit generator [19] where initially the state is loaded by an all zero vector, and then, the initial seed is fed through a series of absorbing rounds. After the last absorbed rate part of the seed, the output stream is squeezed in r bits as needed. Also, because we are using a sponge duplex construction a new seed can be fed to the state while squeezing output at the same time, thus allowing the construction of a reseedable PRBG.

3.5 Implementation Options

In this section, we describe the hardware implementations of our lightweight sLiSCP design using ASIC.

3.5.1 Parallel

For optimization purposes, parallelism is frequently used. These techniques are useful to optimize the designs for achieving better results in terms of area, clock speed, and throughput. First, the Register Transfer Level (RTL) code is written using VHDL. Then the gate level netlist using logic synthesis tool is synthesized and verified. Finally, the area, power consumption, and clock speed are generated after the physical implementation.

The LFSR is chosen for each size of the design. For the 192-bit state size for sLiSCP, an LFSR of size 6 is used while an LFSR of size 7 is used for the 256-bit size of sLiSCP. The LFSR is designed using parallelism of 2 so that the next 2 states can be generated in 1 clock cycle.

3.5.2 ASIC Code Decisions

We implement our sLiSCP permutation using the parallel hardware architecture as shown in Figure 3.8. Each of the four m-bit subblocks of the registers are divided into two parts. In

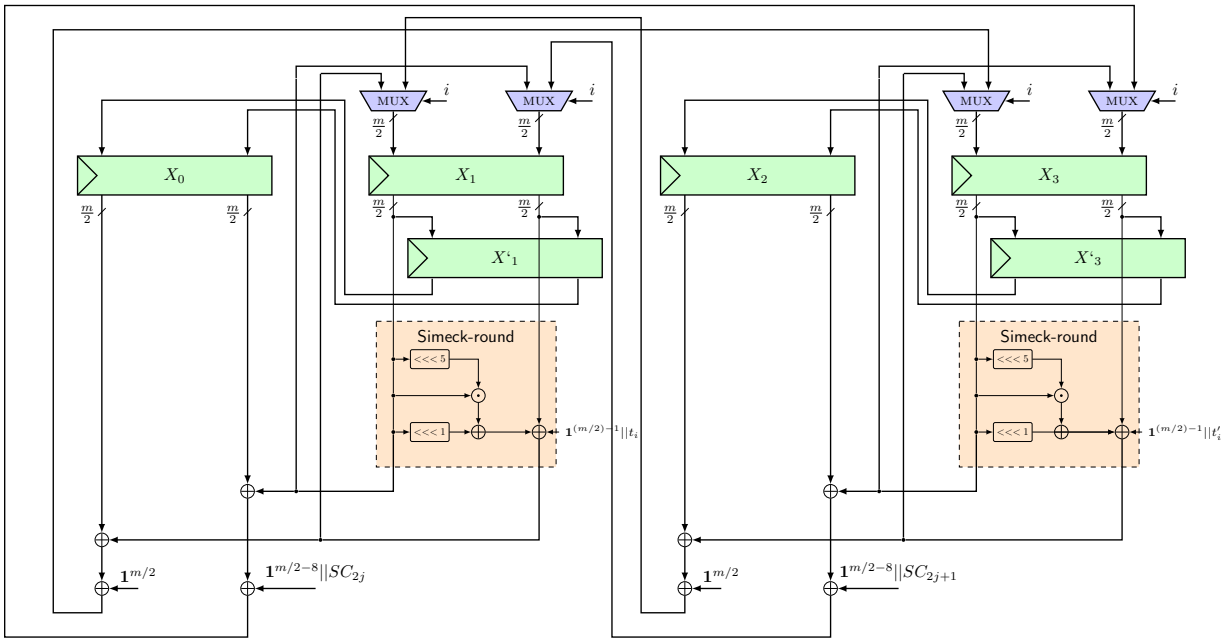

Figure 3.8: Hardware architecture of the sLiSCP permutation

order to control the internal rounds and the steps, two counters (i and j) are adopted where i ($0 \leq i \leq u$) controls the round function of Simeck and j ($0 \leq j \leq s$) controls the steps of the permutation. The output of Simeck round function (dashed box) on registers X_1 and X_3 is feedback to the left half of the registers during each clock cycle when $0 \leq i < u$, and at the same time the left half of the registers is shifted to the right half. The 1-bit round constant RC_{12j+2i} and $RC_{12j+2i+1}$ are first padded with $m/2 - 1$ bits '1's and then are added to the Simeck round function in each clock cycle and they are generated using the parallel LFSR as described in Section 3.3.3. The two extra registers X'_1 and X'_3 are used to store the initial values of registers X_1 and X_3 when i equals 0. At the last clock cycle, i equals u, so the permutation step begins.

During this clock cycle, the output of the Simeck round function based on register X_1 is first XORed with the left half of register X_0, and then is XORed with a constant of $m/2$ bits '1's. This new value is sent to the left half of the register X_3. Due to two different inputs for register X_3, a $m/2$ bits multiplexer is needed. Meanwhile, the left half of the register X_1 is XORed with the right half of the register X_0, and then is XORed with $m/2 - 6$ bits '1's padded with a 6-bit step constant SC_{2j}. The generated new value is shifted to the right half of the register X_3. A multiplexer in this case is needed as well.

The similar case for the output of the Simeck round function based on register X_3 XORed with register X_2, where the new results are sent to the register X_1. At the same time, the values of registers X_1' and X_3' are shifted into the registers X_0 and X_2 respectively. At the end of this clock cycle, all the registers are updated with a new value, and one step of permutation is finished, the counter j is increased by 1, and the counter i returns to 0. After s steps, one permutation is finished. Table 3.4 shows the number of discrete components in the sLiSCP permutation F, where XOR is 1-bit xor operation and MUX is 2 to 1 multiplexer.

Table 3.4: The number of discrete components in sLiSCP permutation

Permutation F	**Components**	**Numbers**	
		sLiSCP-192	sLiSCP-256
Step Function f	Registers	6×48	6×64
	XOR	108	140
	MUX	96	128
Simeck Round Function	AND	24	32
	XOR	49	65
LFSR Constants	Registers	6	7
	XOR	6	5

The authenticated encryption and hash modes of sLiSCP involve running the permutation multiple times in the sponge structure where r-bit input is absorbed using r XORs and r'-bit output is squeezed using r' XORs. The input for the authenticated encryption mode are key, nonce, associated data, and message, whereas there is only message for hash mode. In addition, a three bit domain separator is taken in the authenticated encryption mode, hence three more XORs are required.

We use the same ASIC design flow and metrics as described in Simeck [50]. Our implementation results are based on STMicroelectronics CMOS 65nm CORE65LPLVT library and the areas are obtained before the place and route phase in order to compare fairly with other lightweight candidates. To keep the consistency with other sponge based primitives, the throughput is computed at a frequency of 100 kHz using the following formula: $Throughput = \frac{r'}{(u*s)} * 100$ kbps. Our results for the hash and authenticated encryption modes of sLiSCP, using the same technology, are presented in Table 3.5 as well as a comparison with other lightweight hash functions and AE algorithms. We carry two implementations for the hash and AE modes in order to contrast the sLiSCP permutations with other dedicated designs. However, if a unified mode is used for both functionalities, then the consumed GE area will be dominated by that of the AE mode.

3.6 Hardware Code Discussions

Included in the Appendix A is the details of the hardware code for the cipher. Going through it, we explain much of the optimization that is done so that we obtain our desired hardware area.

3.6.1 Process 1

During the starting section we initialize the signal wires and registers that are used in our round function. We use 8 blocks, rather than 4 because it is more convenient this way when we separate and perform Simeck^u-m on the half blocks. We also use two temporary registers that hold the values in our second and fourth registers. This is because our implementation uses multiple clock cycles to perform the 6 rounds of Simeck. The multiple rounds allow us to drastically reduce the area for the Simeck^u-m function but this also means that we have to use several registers to hold our initial values of the registers. We also have several counter registers that count the round that we are on. These are very small compared to the large state registers that we are using and do not contribute much to our resulting area.

3.6.2 Process 2

In the second process as shown in Appendix A, we go through the second part of the process. We first initialize the values of the wires in the round function. We have a separate round function that takes the values from two 48 bit blocks or four 24 bit blocks and performs the Simeck^u-m function on them. It also takes in the input from our LFSR so that we can both add the round constant and the step constant at the end of the 6 rounds.

3.6.3 Process 3

In the third process we perform the permutation at the end of the 6 rounds of Simeck. We also use a counter to count out the 18 steps. This is done by using multiplexers to determine where the output of of shift register is input.

3.6.4 Process 4

While in the design, we split the state into 4 blocks, in the code, we further separate the state into 8 blocks. In the Simeck process, we feed the values of two of these blocks to perform our modified Simeck function. At the same time, we insert the values into the the other Simeck function.

In the hardware implementation of sLiSCP, we use 4 large registers to store the internal state. As each clock cycle performs one iteration of Simeck^u-m, most of the state will not change. Using multiplexers to control the shift of the large registers after the Simeck function finishes, we then perform the shifts after 6 rounds.

We also show the code for our Simeck^u-m function. The Simeck^u-m function is relatively simple, we separate the 48 bit block into two halves and perform two shifts on one half, one by 6 bits and one by 1 bit. This leaves us with an unshifted, a 6 bit shifted and 1 bit shifted sequence. We AND the 6 bit and unshifted sequence and then XOR it with the 1 bit shifted sequence. The result is then XOR with the right half of the register. We also add the round constant in this step. This result is then shifted into the original registers and we delete the original half block.

3.6.5 Process 5

In our implementation of the authentication and hash variations, we need to XOR the original initialized values of the state with 32 bits. This is done with multiplexers and XOR, which will add around 100 GE.

3.7 Results and Areas

After implementing our design and verifying the correctness of the hardware, we optimized the code to produce our acceptable results.

3.7.1 Hash Mode

Our implementation in CMOS 65nm shows that the area for the hash mode of sLiSCP-192 (resp. sLiSCP-256) is 2192 (resp. 2872) GEs with a throughput of 29.62 (resp. 44.44 kbps or 22.22 kbps depending on r') kbps. When compared with other primitives with similar

internal states, the area of sLiSCP-192 is comparable with that of Photon-160/36/36 and Spongent-160/160/16, only a few gates larger. However, the area of sLiSCP-192 is quite smaller than that of D-Quark, Keccak-f[40,160], Keccak-f[72,128]. In terms of throughput, sLiSCP-192 is better than Photon-160/36/36, D-Quark, and Spongent-160/160/16. The area of sLiSCP-256 is only 86 GEs larger than that of Photon-224/32/32 and is smaller than other primitives. The relevant throughput is only smaller than that of Spongent-160/160/80 and S-Quark.

3.7.2 AE Mode

For the authenticated encryption mode, the area of sLiSCP-192 (resp. sLiSCP-256) is 2202 (resp. 2882) GEs with a throughput of 29.62 (resp. 44.44) kbps. sLiSCP-256 has a GE area that is nearly equal to the estimated area of NORX-16. We note that serialized implementations of sLiSCP modes result in more savings in GE area and thus enable its adoption in highly constrained devices such as EPC tags. Overall, both the hash and authenticated encryption modes of sLiSCP are competitive with others in terms of area and throughput. As usual, in optimizing the critical path in ASIC implementation, the critical path does not pass through too many of the XOR gates.

3.8 Concluding Remarks

In this chapter, we covered the design of sLiSCP and its uses. We presented the encryption, decryption, hash and AE modes when they are used. We provide two efficient parallel hardware implementations for the sLiSCP unified duplex sponge mode. The hardware results of the design showed that it met the lightweight standards for cryptographic primitives.

Table 3.5: Parallel hardware implementation of sLiSCP modes and comparison with other lightweight hash and AE primitives. Throughput is given for a frequency of 100 kHz.

Hash function	**Parameters**				**Security(bits)**			**Process**	**Latency**	**Area**	**Throughput**
	r	c	r'	h	Pre	2nd Pre.	Coll.	(nm)	(Cycles)	(GEs)	(kbps)
sLiSCP-192	32	160	32	160	128	80	80	65	108	**2192**	29.62
Photon-160/36/36 [33]	36	160	36	160	124	80	80	180	180	2117	20.00
D-Quark [7]	16	160	16	176	160	80	80	180	88	2819	18.18
Spongent-160/160/16 [22]	16	160	16	160	144	80	80	130	90	2190	17.78
Keccak-f[40,160] [37]	40	160	40	200	160	160	80	130	18	4900	222.22
Keccak-f[72,128] [37]	72	128	72	200	128	128	64	130	18	4900	400.00
sLiSCP-256	64	192	64	192	128	96	96	65	144	**2872**	44.44
sLiSCP-256	64	192	32	192	160	96	96	65	144	**2872**	22.22
Photon-224/32/32 [33]	32	224	32	224	192	112	112	180	204	2786	15.69
Spongent-160/160/80 [22]	80	160	80	160	80	80	80	130	120	3139	66.67
Spongent-224/224/16 [22]	16	224	16	224	208	112	112	130	120	2903	13.33
Spongent-256/256/16 [22]	16	256	16	256	240	128	128	130	140	3281	11.43
S-Quark [7]	32	224	32	256	224	112	112	180	64	4640	50
AE algorithm				t	Con.	Int.					
sLiSCP-192/80	32	160	32	80	80	80	-	65	108	**2202**	29.62
sLiSCP-192/112	32	160	32	112	112	112	-	65	108	**2202**	29.62
sLiSCP-256/128	64	192	64	128	128	128	-	65	144	**2882**	44.44
Ketje-Jr [18]	16	184	16	96	96	96	-	-	-	4900	-
NORX-16 [9]	128	128	128	96	96	96	-	-	-	2880	-

r, c, r', h and t denote the input bitrate, capacity, output bitrate, digest length and tag size, respectively.
Confidentiality of plaintext.
Integrity of plaintext, associated data and nonce.
Considering it uses Keccak-200 as its underlying permutation, its area is atleast 4900 GEs.

Chapter 4

sLiSCP-light: A Modified Approach

In this chapter, we revisit the design approach of the sLiSCP family of permutations with the main aim of further reducing its hardware area. We examine the sLiSCP-light family of permutations.

The sLiSCP-light family of permutations tweaks the original Type-2 GFS design of sLiSCP. Our tweak gives up some of the desired features of the generic Feistel constructions which we do not make use of when the permutation is used in a sponge construction. Our adopted approach turns the Type-2 GFS into an elegant Partial SPN (PSPN) construction where the substitution layer updates half the state only and the permutation layer mixes the whole state resulting in a fully nonlinearly updated state after one step only (vs. half state as in sLiSCP). Iterated version of round-reduced unkeyed version of the Simeck encryption algorithm [50] are used as large Sboxes in the substitution layer. The round function of Simeck is an independently parametrized hardware efficient version of the Simon round function [11] and has set a new record in terms of hardware efficiency and performance in various platforms. Additionally, Simeck and Simon-like functions have received extensive attention from the cryptographic community in terms of their cryptanalysis [49, 39, 40, 41] and so far remains sufficiently secure.

This Chapter is organized as follows. In Section 4.1, we explain one of the issues that appeared during the design and implementation of sLiSCP. We explain, why it was a problem, how we hoped to fix it, and how it led us to our new design, sLiSCP-light. In Section 4.2, we show the step function of sLiSCP-light and how it differs from the previous sLiSCP design. In Section 4.3, we explain the hardware implementation of this design and show our results. We compare it to existing lightweight designs and also present a few other implementation methods, such as the half serial method and the 1-bit serialized

method.

4.1 Tweak Approach

In this section, we first identify an important aspect in the sLiSCP design where an extra hardware overhead is unjustified. Next, we explore and contrast new design options by which we can avoid the identified overhead and finally detail the tweaking approach which we have adopted in the new design of sLiSCP-light.

4.1.1 Extra Hardware Overhead of the sLiSCP Design

sLiSCP adopts a 4-subblock Type-2 GFS construction, which is like other generic Feistel constructions offers the following features: 1) no constraints on the bijectivity of F; and 2) low overhead for the inverse round function implementation. In the parallel round-based hardware implementation of sLiSCP, for a b-bit state, there are two registers of size $b/4$ bits each that are used as temporary storage. This half state of temporary storage is required because of the iterative nature of the two Simeck boxes. More precisely, the section of the two Simeck boxes requires the use of two extra $b/4$-bit registers to perform iterated updates on the input, and keep the intermediate and initial values of the registers at the same time. Initial values of such two registers are required to update the state of the following step through the linear cyclic shift permutation of Type-2 GFS. Since sLiSCP utilizes a u-round iterated unkeyed Feistel-based Simeck-m, where m is the block size, round function as the GFS F which is extremely hardware efficient (est. area of around 168 (resp.224) GE for Simeck-48 (resp. Simeck-64) boxes), an addition of around 400 (resp. 500) GE for temporary storage is not justified, especially for resource constrained devices.

4.1.2 Solution to Space Exploration

The new design of sLiSCP-light was triggered by the above observation and the need for finding an answer to the following question: *"how can we get rid of these two extra registers?"*. These two extra registers, which only function as temporary storage, use up around 400 GE for the 192-bit size and 500 GE for the 256-bit size for sLiSCP. We primarily wanted to keep the structure of the iterated Simeck-m box as the nonlinear component because it is extremely efficient in hardware as a large Sbox that encompasses both the nonlinear and linear/permutation mixing. Moreover, we can leverage the available

extensive cryptanalysis on Simeck and Simon-like functions to derive the cryptographic properties of such a large Sbox. Accordingly, we found that the following two solutions enable us to remedy the sLiSCP temporary storage problem.

- **Unrolled implementation.** For a u-round iterated Simeck-m box, a trivial solution would be to implement u sequential Simeck round function blocks so that no intermediate storage is required. More precisely, the output of one Simeck round is directly fed to the following one without storing it and by the end of the clock cycle, both the output of the u-rounds and initial state values are available for linear mixing and updating the state for the following step. However, we found out that although each round of Simeck-m costs around 168 (resp. 224) GE for Simeck-48 (resp. Simeck-64), the total hardware area for implementing u blocks of each round surpasses the hardware areas of the two registers which we are trying to save. Despite the much higher throughput, this increase in hardware size is also great enough that our design would no longer be considered lightweight. This method is not suitable for our purposes.

- **Tweaking the original design.** To discard the two temporary registers, we eliminate the need for storing the initial values of the odd indexed subblocks. Since the final values are the output of the u-round iterated Simeck-m box that is the only nonlinear component in the step function of the permutation, we cannot remove it. Accordingly,we opted for tweaking the original 4-subblock Type-2 GFS design so that the initial values of the odd indexed subblocks are not used in updating the even indexed subblocks in the next step. Our tweak is as follows. We use the nonlinearly updated values, by the Simeck-box, to update the even indexed subblocks in the next step. The new tweaked design can be interpreted as a Partial Substitution and Permutation Network (PSPN), which can also be viewed as a mix between Skipjack Rule A [20] and a Type-2 GFS function. In particular, we have a substitution layer that operates only on odd indexed subblocks (starting from index 0) followed by a permutation layer that mixes all the four subblocks resulting in a fully nonlinear-updated state.

4.2 Step Function of the Permutation

An s-step sLiSCP-light permutation takes an input of b bits from $\mathbb{F}_2^b$ and produces an output of b bits after applying the step function s times sequentially where $b = 4 \times m$ and m is an even positive integer. We denote by sLiSCP-light-b a b-bit sLiSCP-light permutation.

The state of the permutation is divided into 4 m-bit subblocks $(X_0^i, X_1^i, X_2^i, X_3^i)$, where i denotes the step number and $0 \leq i \leq s-1$. In each step, the state is updated by a sequence of three transformations: `SubstituteSubblocks` (SSb), `AddStepconstants` (ASc), and `MixSubblocks` (MSb), thus the step function is defined as

$$(X_0^{i+1}, X_1^{i+1}, X_2^{i+}, X_3^{i+1}) \leftarrow \mathsf{MSb} \circ \mathsf{ASc} \circ \mathsf{SSb}(X_0^i, X_1^i, X_2^i, X_3^i).$$

Our step function is shown in Figure 4.1.

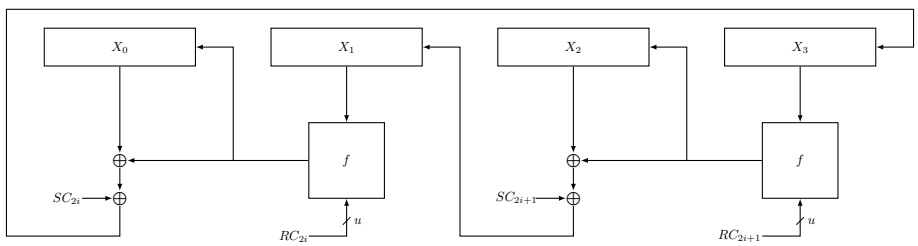

Figure 4.1: sLiSCP-light Step Function

4.2.1 `SubstituteSubblocks` (SSb)

This is a partial substitution layer of the SPN structure where the nonlinear operation is applied to the half of the state. It applies the u-round iterated unkeyed Simeck box (Simeck^u-m or h_u^t, as defined in Section 3.3.2) to the odd indexed subblocks only. The SSb transformation is defined as

$$\mathsf{SSb}(X_0^i, X_1^i, X_2^i, X_3^i) = (X_0^i, h_u^t(X_1^i), X_2^i, h_u^{t'}(X_3^i))$$

where h_u^t is the Simeck box applied on $m = \frac{b}{4}$ bits and t is a u-bit constant.

4.2.2 `AddStepconstants` (ASc)

In this layer, the step constants $\mathtt{SC_{2i}}$ and $\mathtt{SC_{2i+1}}$ are XORed with the two even indexed subblocks X_0 and X_2, respectively, $i = 0, 1, \ldots s-1$. Each $\mathtt{SC_j}$ is an m-bit constant of the form $\mathbf{1}^{m-6}||0^2||sc_j$ (resp. $\mathbf{1}^{m-8}||sc_j$) for Simeck^u-48 (resp. Simeck^u-64), where sc_j is 6 (resp. 8)-bit constant generated by an LFSR. The ASc transformation is given by

$$\mathsf{ASc}(X_0^i, h_u^t(X_1^i), X_2^i, h_u^{t'}(X_3^i)) = (X_0^i \oplus \mathtt{SC_{2i}}, h_u^t(X_1^i), X_2^i \oplus \mathtt{SC_{2i+1}}, h_u^{t'}(X_3^i)).$$

4.2.3 `MixSubblocks` (MSb)

This layer applies the linear transformation that is used in the Type-2 GFS to the subblocks of the state. More precisely, each even indexed subblock is replaced by the XOR of its initial value with its neighboring odd indexed subblock. Then a subblock cyclic left shift is applied. The MSb transformation is given by

$$(X_0^{i+1}, X_1^{i+1}, X_2^{i+1}, X_3^{i+1}) \leftarrow \mathsf{MSb}(X_0^i \oplus \mathtt{SC_{2i}}, h_u^t(X_1^i), X_2^i \oplus \mathtt{SC_{2i+1}}, h_u^{t'}(X_3^i)),$$

where

$$\begin{aligned} X_0^{i+1} &= h_u^t(X_1^i), & X_1^{i+1} &= X_2^i \oplus h_u^{t'}(X_3^i) \oplus \mathtt{SC_{2i+1}}, \\ X_2^{i+1} &= h_u^{t'}(X_3^i), & X_3^{i+1} &= X_0^i \oplus h_u^t(X_1^i) \oplus \mathtt{SC_{2i}}. \end{aligned}$$

The output of MSb has more bit diffusion than that of the original Type-2 GFS and more uniform degree distribution. In particular, after one step, all the components functions of all the subblocks have a degree equal to that of the Simeck^u-m box. Whereas in the Type-2 GFS, after one step, two even indexed subblocks have degree equals one as they are directly copied from the odd indexed subblocks. Moreover, if the output of the Simeck^u-m box has x bit diffusion, then two subblocks have x bit diffusion and the other two subblocks have $(x + 1)$ bit diffusion. In the case of Type-2 GFS, two subblocks have one bit diffusion and the remaining subblocks have $(x + 1)$ bit diffusion. Accordingly, the sLiSCP-light enhances the security of the whole permutation and hinders the extension of most of the distinguishers to more than 8 steps (vs. 9 steps in sLiSCP).

4.2.4 sLiSCP-light Permutation Instances

sLiSCP-light offers two lightweight instances, named sLiSCP-light-192 and sLiSCP-light-265, with state sizes 192 and 256 bits, respectively. Both instances adopt a PSPN step function that is iterated for $s = 12$ times. Simeck^u-48 and Simeck^u-64 boxes, where $u = 6$ and 8, are used as Sboxes in the SSb layer of sLiSCP-light-192 and sLiSCP-light-256, respectively. We keep the number of rounds u equal to 6 (resp. 8) for Simeck^u-48 (resp. Simeck^u-64) because it has been shown our previous work [3] that these parameters provide a good balance between the permutation throughput and differential and algebraic properties. We refer to one PSPN (resp. Simeck box) iteration by one step (resp. one round). Table 4.1 presents the recommended parameters for two lightweight instances of the sLiSCP-light permutation.

Table 4.1: Recommended parameter set for sLiSCP-light-192 and sLiSCP-light-256 permutations.

Permutation (b-bit)	Sbox size m	Rounds u	Steps s	Total # rounds ($u \cdot s$)
sLiSCP-light-192	48	6	12	72
sLiSCP-light-256	64	8	12	96

4.3 Implementations and Benchmarking

In this section, we provide the details of our ASIC hardware and bitsliced software implementations of both instances of the sLiSCP-light permutation. Moreover, we implement the hashing and authenticated encryption modes of sLiSCP-light in ASIC CMOS 65 nm and 130 nm technologies and provide a comparison with existing proposals in Table 3.5.

sLiSCP-light is highly hardware optimized and has very efficient ASIC implementations particularly because of its partial layers. More precisely, the Simeck^u-m boxes, step constant addition, and linear mixing are all applied on half of the state. Additionally, each Simeck^u-m box is itself a very efficient unkeyed Feistel round function. The datapath of the round-based ASIC parallel architecture implementation is depicted in Figure 4.2.

Table 4.2: Breakdown of the number of discrete components in both instances of sLiSCP-light, where XOR is 1-bit xor operation and MUX is 2-1 1-bit multiplexer.

Permutation block	**Discrete component**	sLiSCP-light-**192**	sLiSCP-light-**256**
State	Registers	4×48	4×64
	MUX	96	128
Simeck^u-m boxes	AND	2×24	2×32
	XOR	2×49	2×65
Add step constants	XOR	2×6	2×8
Mix Subblocks	XOR	2×48	2×64
LFSR	Registers	6	7
	XOR	6	9

Design flow and metrics. The Synopsys Design Compiler Version D-2010.03-SP4 is used to synthesize the RTL of the designs into netlist based on the STMicroelectronics

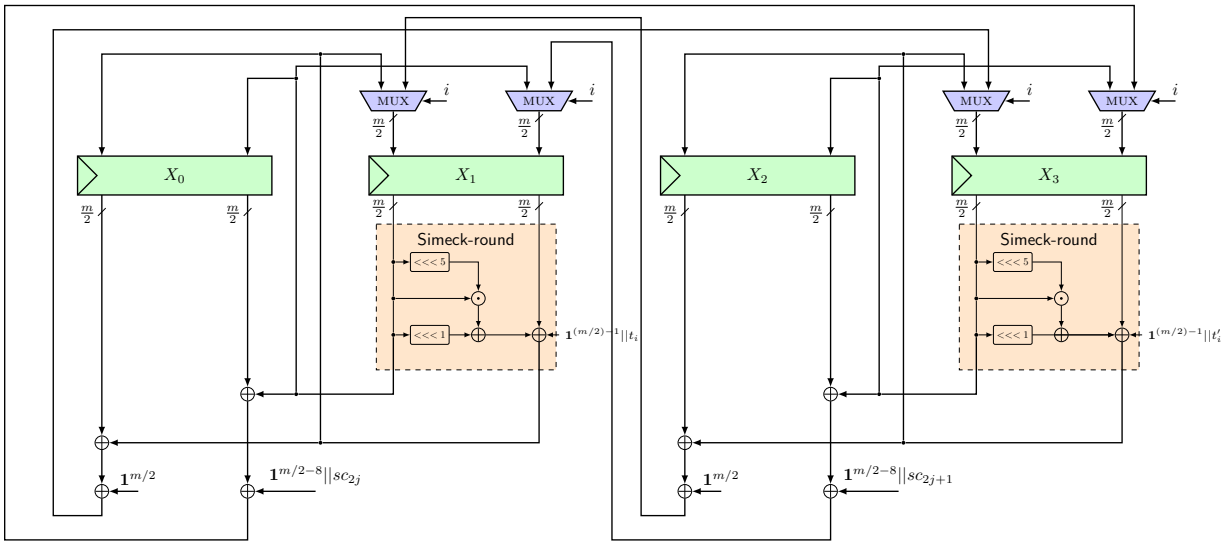

Figure 4.2: Parallel datapath of the sLiSCP-light permutation step function.

CMOS 65 *nm* CORE65LPLVT_1.20V and IBM CMOS 130 *nm* CMR8SF-LPVT Process SAGE v2.0 standard cell libraries with both having a typical 1.2V voltage. Cadence SoC Encounter v09.12-s159_1 is used to finalize the place and route phase in order to generate the layout of the designs. We use Mentor Graphics ModelSim SE 10.1a to conduct functional simulation of the designs and perform timing simulation by using the timing delay information generated from SoC Encounter. We provide the areas and power consumption of both sLiSCP-light instances after the logic synthesis.

We determine the power consumption based on the activity information generated from the timing simulation with a frequency of 100 kHz, and a duration time of 0.1s using SoC Encounter v09.12-s159_1. We specifically use 100 kHz clock frequency because it is widely used for benchmarking purpose in resource constrained applications and 0.1s is long enough to provide an accurate activity information for all the signals.

4.3.1 Description of the round-based implementation

Our round-based implementation executes one step of the permutation in u clock cycles, where $u = 6$ or 8, and requires the components as given in Table 4.2. As depicted in Figure 4.2, all four m-bit registers are divided into two parts to accommodate the Feistel execution of the Simeck^u-m boxes. Two counters i and j of 3 and 4 bits, respectively are

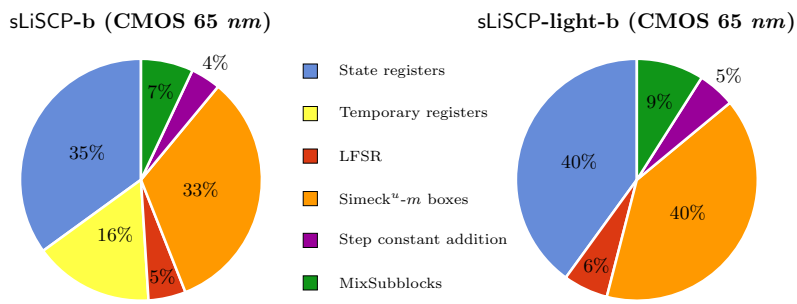

Figure 4.3: Breakdown of the area requirements of the two instances of sLiSCP-light components.

utilized, where i $(0 \leq i \leq u-1)$ controls the round function of Simeck and j $(0 \leq j \leq s-1)$ controls the permutation step function.

During each clock cycle when $0 \leq i < u-1$, we first XOR the right half of registers X_1 (resp. X_3) with $\mathbf{1}^{m/2-1}||t_i$ (resp. $\mathbf{1}^{m/2-1}||t'_i$) where t_i, t'_i are LFSR generated bits. Next, the right half output of the Simeck round function (dashed box) on registers X_1 and X_3 is fed back to the left half of the registers, and the left half of the registers is shifted to the right half. When i equals $u-1$, the left half of the register X_3 is replaced by the XORed value of the right half of register X_1, left half of register X_0 and $\mathbf{1}^{m/2}$. At the same time, the left half of the register X_1 is XORed with the right half of the register X_0, and then is XORed with $\mathbf{1}^{m/2-8}||sc_{2j}$.

In particular, for sLiSCP-light-192, the $(m/2-8)$ bits are first padded with two 0's followed by padding the 6-bit constant sc_{2j}. As with sLiSCP, our step constants and round constants are generated using LFSRs as shown in Figure 3.3 for sLiSCP-light-192 and Figure 3.4 for sLiSCP-light-256. The generated new value is then shifted to the right half of the register X_3. The same process takes place between X_2 and X_3 to update the value of X_1. At the same time, the values of registers X_1 and X_3 are shifted into the registers X_0 and X_2 respectively. Multiplexers are used at the inputs of X_1 and X_3 to make a selection between the output of the Simeck boxes when $i = u-1$ and the cyclically shifted registers. Finally, a new permutation step begins where j is incremented by 1 and i is reset to 0.

4.3.2 How Light is sLiSCP-light?

sLiSCP-light is specifically optimized for resource constrained applications. Table 4.3 shows our smallest ASIC implementations of both sLiSCP-light instances in CMOS 65 nm

and 130 *nm* technologies as well as those of other existing permutations. sLiSCP-light has the lowest area of 1820 (resp. 2397) GE for a 192- (resp. 256-) bit state in CMOS 65 *nm*. Although the state sizes of sLiSCP-light instances are close to the state size of Keccak-f[200], the areas of both instances of sLiSCP-light are significantly lower than that of Keccak-f[200] (est. 4540 GE). When compared with Norx-16 (est. area of 2496 GE), sLiSCP-light-256 has a slightly lower area in CMOS 65 *nm* and comparable area in CMOS 130 *nm*. For larger state sizes, the areas of the 256-bit state sLiSCP-light in CMOS 65 *nm* and 130 *nm* technologies are several magnitude smaller than that of Ascon and Gimli, which are mainly optimized for software platforms and/or specific processors.

Regarding the throughput, both 192 and 256-bit instances of sLiSCP-light have a throughput of 266.7 kbps, which is higher than that of Photon, Spongent, and Quark with a state of 176 bits. When sLiSCP-light instances are compared with Gimli and Ascon, the throughput of the sLiSCP-light instances is smaller as they have a larger number of rounds compared to Gimli and Ascon. However, such permutations are not considered suitable for lightweight applications due to their larger GE areas.

Application modes. We have also implemented the authenticated encryption and hashing modes using both instances of sLiSCP-light. We provide our implementation results in CMOS 65 *nm* and 130 *nm* technologies, and contrast the areas and throughput with other sponge-based primitives with similar or close state sizes and security parameters in Table 3.5 in the Appendix. One can clearly see that both instances of sLiSCP-light offer a competitive advantage over existing proposals in terms of both area and throughput. Specifically, our smallest implementation of the unified duplex sponge mode using sLiSCP-light-192 offers 80-bit and 112-bit security in the keyed modes depending on the data usage exponent [14], and 160-bit digest in the hashing mode. On top of that, such an implementation is realized with only 1958 GE and offers a throughput of 44.44 kbps which makes it suitable for providing almost all cryptographic functionalities of the most resource constrained devices such as passive RFID tags. When used in the sLiSCP duplex mode, a simpler initialization phase is adopted to offer competitive throughputs for the authenticated encryption of short messages.

4.3.3 Half Serial

After designing the sLiSCP-light cipher, we also looked into the possibility of a half parallel implementation. This process would perform one instance of Simeck round function. After the round function is finished, the hardware then proceeds to perform Simeck on the other half. The idea of this design would be to reduce the logic of two instances of Simeck to only

one instance. This design was implemented on sLiSCP-light with size 192. The resulting implementation had an area of 1899 GE, as opposed to the 1820 GE for our original parallel implementation. This design is actually worse than the fully parallel design and was because this hardware design would require extra multiplexers and area to choose between the two halves to perform the Simeck round function. Due to the similarities of the sLiSCP and sLiSCP-light this conclusion should hold true for both versions for all sizes. Pursuing a half-serial implementation would not reduce our hardware area as we hypothesized and would not be useful in our lightweight goals.

4.3.4 Estimates for 1-bit Serialized Implementations

Various serialization degrees has been demonstrated by the designers of the Simeck block cipher [50]. Accordingly, using the same methods adopted in [50], in what follows, we provide estimates for the areas of both sLiSCP-light instances when the Simeck^u-m boxes are serialized by degree 1.

The parallel Simeck^u-m implementation utilizes $m/2$ ANDs and $(2m+1)$ XORs which can be serialized using 1 AND, 3 XORs and 4 2-1 MUXs. Two of the MUXs are used to select the cyclic shift inputs, one is used to select the input of the registers, and another MUX is used to XOR the round constant.

When sLiSCP-light-192 is implemented in CMOS 65 nm technology that costs 2.25 GE per XOR, 2 GE per MUX and 1.25 GE per AND, then the serialized implementation of degree 1 of the Simeck^u-48 saves $(24\times1.25+49\times2.25)-(1\times1.25+3\times2.25+4\times2) = 124.25$ GE. Since there are two Simeck Sboxes, this results in 248 GE savings for sLiSCP-light-192. The respective saving for sLiSCP-light-256 is given by $(32\times1.25+65\times2.25-(1\times1.25+3\times2.25+4\times2))\times2 \approx 340$ GE. Thus, without considering other savings, the estimated areas for the 1-bit serialized implementations of sLiSCP-light-192 and sLiSCP-light-256 in CMOS 65nm ASIC can be obtained with at least 1572 and 2057 GE, respectively. Thus, the estimated areas for the 1-bit serialized implementations of sLiSCP-light-192 and sLiSCP-light-256 are 1572 and 2057 GE, respectively. For the CMOS 130 nm technology, the number of discrete components that we can save are the same. This technology requires 2 GE per XOR, 2.25 GE per MUX and 1.25 GE per AND. For sLiSCP-light-192, the serialized implementation saves around $(24\times1.25+49\times2-(1\times1.25+3\times2+4\times2.25))\times2 = 223.5$ GE. Similarly, we save around $(32\times1.25+65\times2-(1\times1.25+3\times2+4\times2.25))\times2 \approx 307.5$ GE for sLiSCP-light-256. Hence, an estimated area for sLiSCP-light-192 (resp. sLiSCP-light-256) can be at least achieved with 1669 (resp. 2193) GE in CMOS 130 nm ASIC.

It is also possible to serialize the entire step function. However, we need to include a

large number of multiplexers in the hardware design to perform Simeck on both X_1 and X_3 serially. This increase in GE is much larger than the area that is saved by performing one Simeck round function instead of two. Therefore, although the design can be completely serialized, it would not be worth it in terms of area and throughput.

4.4 Summary

sLiSCP-light fills the gap in the available cryptographic permutations with low GE area and offering acceptable security parameters for both keyed and unkeyed modes of sponge constructions. More precisely, the state size of the underlying permutation is bounded below by the sponge security parameters (i.e., capacity and rate), especially in hashing mode, a 160-bit digest with 32-bit hashing rate requires a permutations state of at least 192-bits. Both instances of the sLiSCP-light permutation are extremely hardware efficient to the point (e.g., 2000 GE), where the 192-bit sLiSCP-light costs only 1820 (resp. 1892) GE in CMOS 65 nm (resp. 130 nm) ASIC. Moreover, our smallest implementation of the fully parallelized unified sLiSCP duplex sponge mode using sLiSCP-light-192 has a cost of 1958 GE with a throughput of 44.44 kbps, and offers 80-bit and 112-bit security in the keyed modes depending on the data usage, and 160-bit digest in the hashing mode.

Permutation	**State size** (bits)	**Technology** (nm)	**Area** (GE)	**Cycles**	**Throughput** (kbps)	**Power** (μW)
sLiSCP	192	65	**2153**	108	177.77	4.62
		130	**2318**			7.44
	256	65	**2833**	144		5.88
		130	**3040**			8.75
sLiSCP-light	192	65	**1820**	72	266.66	3.97
		130	**1892**			5.05
	256	65	**2397**	96		4.77
		130	**2500**			7.27
Photon	196	180	1949	180	108.88	4.35
	256		2637	204	125.49	6.5
Spongent	176	130	2110	90	195.55	4.47
	240		2739	120	200.00	6.8
Quark	176	180	2739	88	200.00	4.76
	256		4480	64	400.00	8.39
Keccak	200	130	4540	18	1111.11	27.6
Norx-16	256	-	2496	-	-	-
Ascon	320	90	7080	12	26.66	43
Gimli	384	28	8097	24	1600.00	-
		180	5314		1600.00	-

Table 4.3: Parallel hardware implementation of sLiSCP, sLiSCP-light and comparison with other lightweight primitives. Throughput is given for a frequency of 100 kHz.

Chapter 5

Randomness Properties

In this chapter we explore the randomness properties of Simeck^u-m in sLiSCP and similarly structured designs using the Feistel structure. We look at how these could be possibly used as a pseudorandom number generator. We discuss the randomness testing of random number and pseudorandom number generators that may be used for many purposes including cryptographic, modeling and simulation applications.

5.1 Processing Methods

5.1.1 Tests

Our testing involved using small versions of Simeck-like functions with different shift values (see Figure 5.1). We would have an internal state equal to the ones used in Simeck, and sLiSCP. However, in our tests, we use smaller internal states to look more exhaustively. This would not accurately represent the randomness properties of the larger states but we may see some similarities and trends. In these tests, we omit the constant k_i to look more closely at how the shift values of (a, b, c) affect properties. We exhaustively test out each possible starting state value and record the number of times Simeck is performed on the state until it cycles and reaches its initial state. Since Simeck uses AND, XOR, and shift operations only, the 0 state always outcomes to itself. The length of this cycle will always be 1. However, we looked at the lengths of the other cycles and made some observations on how long each of them were.

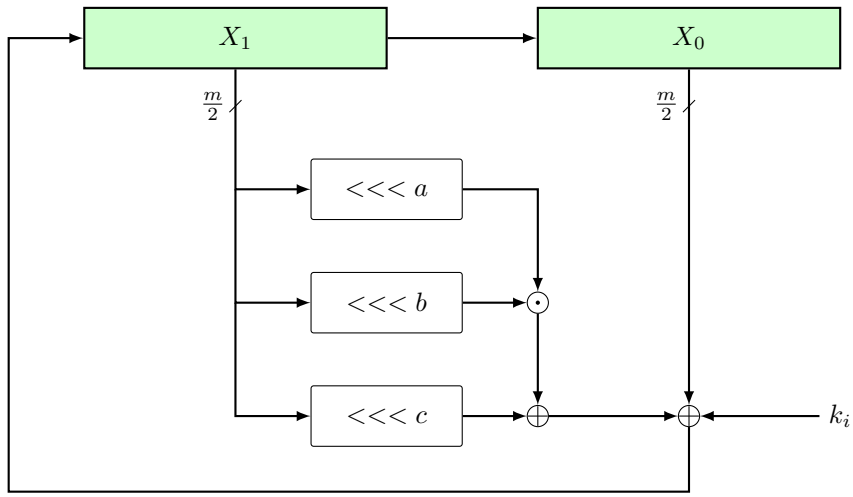

Figure 5.1: Simeck using shift values of (a, b, c)

5.1.2 Acceptable Cycles

For our tests we look at each of the cycles in the Simeck states. Since we perform Simeck on an initial state and repeat this process many times, eventually the internal state of the cipher will cycle around and shift back into the initial state. This is because both Simeck and Simon use a Nonlinear Feedback Shift Register (NLFSR) structure. Based on these properties, we would like our cipher to have states where it takes a large amount of repeated Simeck/Simon operations for it to cycle around and output the original state. We define the acceptable size for this cycle to be the square root of 2^m where m is the number of bits in the state.

5.2 Shift Values and Periods

Based on our findings, we can see that the shift values used in Simeck and Simon seem adequate for many of the purposes in randomness testing.

Using the Simeck shift values of $(a, b, c) = (5, 0, 1)$ for $m = 16$ we provided detailed results for one instance in Table 5.2 as an example. After finding the cycles, we recorded the size of the cycles of each initial state. With a size of $m = 16$, an acceptable state would be in a cycle of length greater than $2^8 = 256$.

We look through our results to compare the ratio of states that are in a cyclic sequence

that has length greater than $2^{m/2}$ where m is the state size, which we previously noted were states in acceptable cycles. After running exhaustive search on several values of m, we obtain that the highest percentage of the acceptable states usually peaks at around 60-70%. We compared these to the shift values suggested by Simon and Simeck, they seem to produce acceptable states at only around 5-15% lower than the maximum percentage for different state sizes. For most of the shift values that did produce the maximum percentage was not consistent, in that for larger or smaller m values, their percentage dropped dramatically. Although, the Simon and Simeck shift values did not always produce the highest percentage of acceptable states, the consistency for them to provide high percentages were the most useful conclusion in these tests. This suggests that the shift values for Simon and Simeck should provide acceptable pseudorandom values for larger values of m.

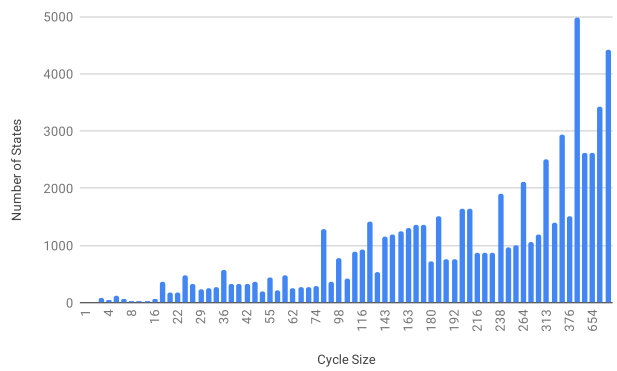

Figure 5.2: Randomness tests of the Simeck structure for $m = 16$, $(a, b, c) = (5, 0, 1)$

Our code for the testing involves doing an exhaustive test of all of the starting states. For each starting state, the program repeatedly performs Simeck until its original state is reached. The program records the number of Simeck operations that were performed, which would correspond to the size of the cycle. Since each of the intermediate states would belong to this cycle, the program will ignore these states when testing initial values later. The program continues to test other initial states until all 2^m states have been recorded. See Appendix B for exact testing procedure.

For our tests, we started with just a single Feistel structure, as in Figure 2.2-(a). Once we tested the randomness features for Simeck for that, we increased the number of blocks to cover the structures as in Figure 2.2-(b) and Figure 2.2-(c). The tests all proved that the acceptable ranges were above 60% for the Simon and Simeck shift values.

It is important that we test the shift values of each design separately. We look at the Simeck implementation first. We use the same round function and notation as Figure 2.6. The (a, b, c) values are varied for each test while also considering the random constant that is inserted after each round. For larger values of m, such as for $m \geq 8$, we simply used the shift (a, b, c) for Simeck and Simon, which were $(5, 0, 1)$ and $(8, 1, 2)$, respectively. We display the results in Table 5.1.

Table 5.1: Randomness tests of the Simeck and Simon for larger values of m.

Shift Values $(5, 0, 1)$		**Shift Values** $(8, 1, 2)$	
Size m	**Acceptable States (%)**	**Size** m	**Acceptable States (%)**
16	46.9	16	57.0
18	65.3	18	54.5
20	47.2	20	43.4

5.3 Conclusions

The main point of this chapter was to observe some of the properties of Simeck and Simon in small sizes. In practical applications, the sizes would be much larger, but with these small sizes, we are able to exhaustively search for every possible state. From there, we can see some possible relations for the shift values when using the Feistel structure as in Simeck and Simon. While some shift amounts provided the highest number of acceptable states for one value of m, they were not always consistent when m was changed. On the other hand, we observed that Simon and Simeck usually had a very high amount of acceptable states. They were close to but not the highest. However, their consistency would be very important for larger sizes.

Chapter 6

Conclusions and Future Work

In this thesis, we presented some previous lightweight designs, their uses, and their strengths. sLiSCP and sLiSCP-light is discussed and their hardware specifics are presented. We showed that both versions of sLiSCP and sLiSCP-light have very acceptable hardware area while also performing multiple functionalities that should be required in future applications.

6.1 Conclusions

Both sLiSCP and sLiSCP-light have very promising hardware footprints for their uses. The advantage of our designs compared to others with similar hardware footprints are mainly that we provided multiple functionalities, such as hashing and authenticated encryption. The compact nature of our design, which allows authenticated encryption modes and hashing modes should be very useful in lightweight applications.

Our hardware area is acceptable with the 2000 GE requirement for lightweight designs set by NIST and our power consumption is low as well. Our sLiSCP-light design is an improvement over our sLiSCP design in hardware, and although the security features are promising, our modified Feistel structure has not been studied to the degree of the original Feistel structure. That would be the main difference between sLiSCP and sLiSCP-light for practical purposes. Even with these differences, our permutations are very hardware efficient, with sLiSCP-light-192 only costing 1820 GE with CMOS 65nm technology. Our parallel implementation is also an indication that permutation can have a reasonable throughput and does not sacrifice all of its speed for efficiency.

To summarize the thesis, the necessary optimization of the two permutation designs for lightweight applications the instances of 192-bit and 256-bit sLiSCP and sLiSCP-light were performed. The main factor, hardware area of 2289 GE for our sLiSCP-192 in CMOS 65 nm was comparable to the other lightweight ciphers developed. Our sLiSCP-light was even more efficient, with sLiSCP-light-192 achieving an area of 1820 GE in CMOS 65 nm technology.

6.2 Future Work

One of the major options for future work involves implementing the permutations completely in serial. This may reduce the area even further. In addition, since Simeck^u-m is able to be serialized, there should be no hardware issues in this task. Due to these findings, it would be a very useful project and would strengthen the uses for sLiSCP and sLiSCP-light even further.

There is also much more future work to be done in the randomness properties of Feistel Structure. Although some of it was explored in Chapter 6, a more in-depth study would be very useful. Larger state values can be tested as well as many other shift values. We also do not limit ourselves to the simple Feistel structure of Simeck/Simon and so we can further look at Type-2 GFS structures as in sLiSCP.

Publications Related to Thesis

I am part of the design for the cipher systems sLiSCP and sLiSCP-light for hardware implementations. Those full research results have been published in the following three articles. For each of the three publications, I was the first author in the implementation group of sLiSCP and sLiSCP-light(hardware and software).

1. R. Altawy, R. Rohit, M. He, K. Mandal, G. Yang and G. Gong. sLiSCP: Simeck-based Permutations for Lightweight Sponge Cryptographic Primitives, *Selected Areas in Cryptography SAC 2017*, August 16-18, 2017, Revised Selected Papers, pp.129-150.

2. R. Altawy, R. Rohit, M. He, K. Mandal, G. Yang and G. Gong. Towards a Cryptographic Minimal Design: The sLiSCP Family of Permutations, *IEEE Transactions on Computers*, Vol. 67, Issue 9, pp. 1341 1358, 2018.

3. R. Altawy, R. Rohit, M. He, K. Mandal, G. Yang and G. Gong. sLiSCP-light: Towards Lighter Sponge-specific Cryptographic Permutations, *ACM Transactions on Embedded Computing Systems*, Vol. 17, Issue 4, Article No. 81, pp. 1 26, 2018.

References

[1] 3rd Generation Partnership Project. Specification of the 3GPP Confidentiality and Integrity Algorithms - Document 2: KASUMI Specification (Release 6). Technical Report 3GPP TS 35.202 V6.1.0 (2005-09), September 2005.

[2] Riham AlTawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. sLiSCP: Simeck-based permutations for lightweight sponge cryptographic primitives. Cryptology ePrint Archive, Report 2017/747, 2017. https://eprint.iacr.org/2017/747.

[3] Riham AlTawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. sLiSCP: Simeck-based permutations for lightweight sponge cryptographic primitives. In *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, pages 129–150, 2017.

[4] Riham AlTawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. sLiSCP: Simeck-based permutations for lightweight sponge cryptographic primitives. In *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, pages 129–150, 2017.

[5] Riham Altawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. sLiSCP-light: Towards hardware optimized sponge-specific cryptographic permutations. *ACM Trans. Embed. Comput. Syst.*, 17(4):81:1–81:26, August 2018.

[6] Riham AlTawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. Towards a cryptographic minimal design: The sLiSCP family of permutations. *IEEE Transactions on Computers*, 67(9):1341–1358, Sept 2018.

[7] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. *Journal of Cryptology*, 26(2):313–339, Apr 2013.

[8] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. Norx: Parallel and scalable AEAD. In *19th European Symposium on Research in Computer Security - Volume 8713*, ESORICS 2014, pages 19–36, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[9] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. NORX8 and NORX16: Authenticated encryption for low-end systems. *IACR Cryptology ePrint Archive*, 2015:1154, 2015.

[10] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu F Sasaki, Siang Meng Sim, and Yosuke Todo. Gift: A small present - towards reaching the limit of lightweight encryption. In *CHES*, 2017.

[11] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 175:1–175:6, New York, NY, USA, 2015. ACM.

[12] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The skinny family of block ciphers and its low-latency variant mantis. In *Proceedings, Part II, of the 36th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016 - Volume 9815*, pages 123–153, Berlin, Heidelberg, 2016. Springer-Verlag.

[13] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, Franois Xavier Standaert, Yosuke Todo, and Benoît Viguier. GIMLI: A cross-platform permutation. In *Cryptographic Hardware and Embedded Systems CHES 2017 - 19th International Conference, Proceedings*, volume 10529 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 299–320. Springer Verlag, 2017.

[14] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the security of the keyed sponge construction. In *Symmetric Key Encryption Workshop*.

[15] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *EUROCRYPT*, pages 181–197, 2008.

[16] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak specifications. *Submission to NIST (Round 2)*, 2009.

[17] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Permutation-based encryption, authentication and authenticated encryption. In *DIAC*, 2012.

[18] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Caesar submission: Ketje v2. 2014.

[19] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge-based pseudo-random number generators. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'10, pages 33–47, Berlin, Heidelberg, 2010. Springer-Verlag.

[20] Céline Blondeau, Andrey Bogdanov, and Meiqin Wang. On the (in)equivalence of impossible differential and zero-correlation distinguishers for Feistel- and skipjack-type ciphers. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security*, pages 271–288, Cham, 2014. Springer International Publishing.

[21] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[22] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. Spongent: A lightweight hash function. In *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'11, pages 312–325, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] Andrey Bogdanov and Kyoji Shibutani. Generalized Feistel networks revisited. *Des. Codes Cryptography*, 66(1-3):75–97, January 2013.

[24] Christophe De Canniere and Bart Preneel. Trivium specifications. *eSTREAM, ECRYPT Stream Cipher Project*, 2006.

[25] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer-Verlag, Berlin, Heidelberg, 2002.

[26] Christophe De Cannière, Orr Dunkelman, and Miroslav Knežević. KATAN and KTANTAN — a family of small and efficient hardware-oriented block ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 272–288, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[27] W. Diffie and M. E. Hellman. Special feature exhaustive cryptanalysis of the nbs data encryption standard. *Computer*, 10(6):74–84, June 1977.

[28] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2, submission to the CAESAR competition. 2016.

[29] Solomon Golomb. *Shift Register Sequences*. Aegean Park Press, Berlin, Heidelberg, 1982.

[30] Guang Gong. Lecture notes of ece 710 topic 21: Communication security, 2017.

[31] Guang Gong and A. M. Youssef. Cryptographic properties of the Welch-Gong transformation sequence generators. *IEEE Transactions on Information Theory*, 48(11):2837–2846, Nov 2002.

[32] Zheng Gong, Svetla Nikova, and Yee Wei Law. Klein: A new family of lightweight block ciphers. In Ari Juels and Christof Paar, editors, *RFID. Security and Privacy*, pages 1–18, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[33] Jian Guo, Thomas Peyrin, and Axel Poschmann. The photon family of lightweight hash functions. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 222–239, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[34] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The LED block cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 326–341, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[35] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. *The Grain Family of Stream Ciphers*, pages 179–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[36] Ari Juels and Stephen A. Weis. Authenticating pervasive devices with human protocols. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 293–308, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[37] Elif Bilge Kavun and Tolga Yalcin. A lightweight implementation of keccak hash function for radio-frequency identification applications. In Siddika Berna Ors Yalcin, editor, *Radio Frequency Identification: Security and Privacy Issues*, pages 258–269, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[38] Lars Knudsen, Gregor Leander, Axel Poschmann, and Matthew J. B. Robshaw. Printcipher: A block cipher for ic-printing. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 16–32, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[39] Stefan Kölbl, Gregor Leander, and Tyge Tiessen. Observations on the simon block cipher family. In *Links Among Impossible Differential, Integral and Zero Correlation Linear Cryptanalysis*, volume 9215, pages 161–185. 08 2015.

[40] Kota Kondo, Yu Sasaki, and Tetsu Iwata. On the design rationale of Simon block cipher: Integral attacks and impossible differential attacks against Simon variants. In *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, pages 518–536, 2016.

[41] Zhengbin Liu, Yongqiang Li, and Mingsheng Wang. Optimal differential trails in simon-like ciphers. *IACR Trans. Symmetric Cryptol.*, 2017(1):358–379, 2017.

[42] Kerry A. McKay, Lawrence E. Bassham, Meltem Sonmez Turan, and Nicky W. Mouha. Report on lightweight cryptography (NISTIR8114).

[43] Kaisa Nyberg. Generalized Feistel networks. In *Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '96, pages 91–104, Berlin, Heidelberg, 1996. Springer-Verlag.

[44] Information Technology Laboratory (National Institute of Standards and Technology). *Announcing the Advanced Encryption Standard (AES).* Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, MD, 2001.

[45] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: An ultra-lightweight blockcipher. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 342–357, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[46] Tomoyasu Suzaki and Kazuhiko Minematsu. Improving the generalized Feistel. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption*, pages 19–39, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[47] Ivan Tjuawinata, Tao Huang, and Hongjun Wu. Cryptanalysis of Simpira v2. In Josef Pieprzyk and Suriadi Suriadi, editors, *Information Security and Privacy*, pages 384–401, Cham, 2017. Springer International Publishing.

[48] US Department of Commerce/National Bureau of Standards. FIPS 46, Data Encryption Standard. Technical report, 1977.

[49] Qingju Wang, Zhiqiang Liu, Kerem Varıcı, Yu Sasaki, Vincent Rijmen, and Yosuke Todo. Cryptanalysis of reduced-round SIMON32 and SIMON48. In Willi Meier and Debdeep Mukhopadhyay, editors, *Progress in Cryptology – INDOCRYPT 2014*, pages 143–160, Cham, 2014. Springer International Publishing.

[50] Gangqiang Yang, Bo Zhu, Valentin Suder, Mark D. Aagaard, and Guang Gong. The Simeck family of lightweight block ciphers. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 307–329, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[51] Huihui Yap, Khoongming Khoo, Axel Poschmann, and Matt Henricksen. Epcbc - a block cipher suitable for electronic product code encryption. In Dongdai Lin, Gene Tsudik, and Xiaoyun Wang, editors, *Cryptology and Network Security*, pages 76–97, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

Appendix A

sLiSCP Implementation

Process 1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.simeck_pkg.all;

entity dp is
port (
reset,
clk              : in std_logic;
data_block       : in block_elem;
cont_irounds     : in unsigned(reg_rounds 1 downto 0 );
cont_steps       : in unsigned(reg_steps  1 downto 0 );
encoding_message : out block_elem
);
end entity;

architecture main of dp is

signal message_block  :  block_elem;

signal comp18: std_logic;

signal round_message_o : word_elem;
```

```
signal round_message_i1,
round_message_i2,
round_message_i3 : word_elem;
signal round_message_oo : word_elem;
signal round_message_ii1,
round_message_ii2,
round_message_ii3 : word_elem;

signal d_message_block0,
d_message_block1,
d_message_block2,
d_message_block3,
d_message_block4,
d_message_block5,
d_message_block6,
d_message_block7   : word_elem;

signal message_block2_back,
message_block3_back,
message_block6_back,
message_block7_back        : word_elem;

signal message_block0_feedback,
message_block1_feedback,
message_block4_feedback,
message_block5_feedback  : word_elem;

signal rc1_tmp, rc2_tmp  : std_logic;
signal RC1_ext_tmp, RC2_ext_tmp  : std_logic_vector(5 downto 0);
signal lfsr_tmp          : std_logic_vector(5 downto 0);
signal lfsr_tmp_out      : std_logic_vector(5 downto 0);
```

Process 2

```
begin
wait until rising_edge(clk);
if reset ='1' then
```

```
lfsr_tmp <= "111111";
else
lfsr_tmp <= lfsr_tmp_out;
end if;
end process;

lfsr_para2_1:  entity work.lfsr_para2
port map
( lfsr_in    =>  lfsr_tmp
,lfsr_out   =>  lfsr_tmp_out
,rc1        =>  rc1_tmp
,rc2        =>  rc2_tmp
,RC1_ext        =>  RC1_ext_tmp
,RC2_ext        =>  RC2_ext_tmp
);

roundfun1 :  entity work.roundfun
port map
(  a_i   => round_message_i1
,b_i   => round_message_i2
,c_i   => round_message_i3
,d_o   => round_message_o
);

roundfun2 :  entity work.roundfun
port map
(  a_i   => round_message_ii1
,b_i   => round_message_ii2
,c_i   => round_message_ii3
,d_o   => round_message_oo
);
```

Process 3

```
comp18<= '1' when cont_steps < 18 else '0';
round_message_i1 <= message_block(2);
round_message_i2 <= message_block(3);
round_message_i3 <= cont_round (word_sz 2 downto 0) & rc1_tmp;
```

```
round_message_ii1 <= message_block(6);
round_message_ii2 <= message_block(7);
round_message_ii3 <= cont_round (word_sz 2 downto 0) & rc2_tmp;

d_message_block0     <= message_block2_back;
d_message_block1     <= message_block3_back;

d_message_block2     <= round_message_o when cont_irounds < 5 else
message_block4_feedback;
d_message_block3     <= message_block (2) when cont_irounds < 5 else
message_block5_feedback ;

d_message_block4     <= message_block6_back;
d_message_block5     <= message_block7_back;

d_message_block6     <= round_message_oo when cont_irounds < 5 else
message_block0_feedback;
d_message_block7     <= message_block (6) when cont_irounds < 5 else
message_block1_feedback ;

message_block4_feedback  <=  message_block (4) xor round_message_oo
xor cont_round(word_sz 1 downto 0);
message_block5_feedback  <=  message_block (5) xor message_block
(6) xor (cont_round(word_sz  9 downto 0)& "00" & RC2_ext_tmp);


message_block0_feedback  <=  message_block (0) xor round_message_o
xor cont_round(word_sz 1 downto 0);
message_block1_feedback  <=  message_block (1) xor message_block
(2) xor (cont_round(word_sz  9 downto 0)& "00" & RC1_ext_tmp);
```

Process 4

```
process
begin
wait until rising_edge (clk);
if cont_irounds =0  then
message_block2_back <= message_block(2);
```

```
message_block3_back <= message_block(3);
message_block6_back <= message_block(6);
message_block7_back <= message_block(7);
end if;

end process;

process
begin
wait until rising_edge (clk);
if reset ='1' then
message_block  <= data_block;
else
if cont_irounds = 5 and comp18='1' then
message_block (0) <= d_message_block0;
message_block (1) <= d_message_block1;

message_block (4) <= d_message_block4;
message_block (5) <= d_message_block5;
end if;

if comp18='1' then
message_block (2) <= d_message_block2;
message_block (3) <= d_message_block3;

message_block (6) <= d_message_block6;
message_block (7) <= d_message_block7;
end if;
end if;

end process;

encoding_message <= message_block;
end architecture;
```

Process 5

```
library ieee;
```

```
use ieee.std_logic_1164.all;
use work.simeck_pkg.all;
use work.util.all;

entity roundfun is
port (
a_i,
b_i    : in word_elem;
c_i    : in word_elem;
d_o    : out word_elem
);
end entity;

architecture main of roundfun is

signal a1, a2: word_elem;


begin

a1 <= a_i(word_sz  6 downto 0) & a_i(word_sz 1  downto word_sz  5);

a2 <= a_i(word_sz 2 downto 0) & a_i(word_sz  1);

d_o <= (a_i and a1 ) xor b_i xor a2  xor c_i;

end architecture;
```

Appendix B

Simeck Randomness Cycles

```
period=0;
for(int i=0; i<count 2;i++){

for(int j=i+1;j<count 1;j++){
if (results[i]==results[j]&& results[i+1]==results[j+1]){
rep = 1;
period=j i;
break;
}
}
if (rep==1){
break;
}
}
```